Licencia

UOC
www.uoc.edu

Estudios de Informática y Multimedia

Proyecto Fin de Carrera

Diseño e implementación de un *Framework de Presentación*

Curso 2012/13

Nombre: Daniel Rodríguez Simó

Username: drodriguezsi

Tutor : Óscar Escudero Sánchez

Agradecimientos

Quiero agradecer el apoyo a Isabel, mi mujer, de cara a todo el tiempo invertido para poder sacar adelante estos años de estudio y esfuerzo. Y en general a toda mi familia, especialmente a mis padres Eugenia y Juan Pedro, porque sin ellos y sin la educación que se han esforzado en darme, esto nunca habría sido posible y a Pilar y Eugenia, mis abuelas, que aunque no se encuentren entre nosotros, siempre algo de ellas que nos acompaña en el día a día.

Por último agradecer a mi tutor Óscar su apoyo y orientación de cara a la consecución de este objetivo y al buen desarrollo de este Proyecto Fin de Carrera. A todos vosotros, Gracias!

Descripción General

El presente proyecto se centra en el estudio y elaboración de un marco de trabajo basado en un Framework de Presentación, dedicado al desarrollo de aplicaciones web bajo la plataforma J2EE. El Framework en cuestión recibe el nombre de "***FCUOC***", el cual se ha construido siguiendo las características principales del patrón MVC (Modelo-Vista-Controlador).

Previo a la construcción del tal Framework, se realizará un "*overview*" de los Frameworks actuales del mercado, destacando las características más relevantes de tres de ellos, en concreto los Frameworks de *Struts*, *Tapestry* y más actuales como *Grails*, extrayendo aquellos detalles y patrones que nos sean de mayor utilidad de cara a la construcción de nuestro modelo y finalizando con una comparativa que nos arroje las diferencias, ventajas y desventajas que nos ofrezcan unos Frameworks con respecto a otros.

Conocidos los objetivos y las herramientas y patrones como punto de partida, se realizará un análisis y diseño tanto del Framework como de la aplicación que hará uso del mismo. Tras esta fase y definidos todos los diagramas correspondientes al modelo de datos, casos de uso, etc. que representarán nuestro sistema se procederá a la implementación.

La implementación se ha realizado utilizando tecnologías estándar, abiertas y *Open Source*, donde se servirá del API de Java 1.6 y MySQL como tecnología de Base de Datos. Por otro lado, la aplicación se desplegará en un Servidor de Aplicaciones JBoss y contaremos con herramientas para realizar compilaciones y despliegues de forma automática, mediante Ant.

Índice General

Índice de Contenidos

Índice de Figuras

1. Introducción

1.1 Descripción del Proyecto

El presente proyecto se va a encargar de realizar una implementación de un Framework para la capa de presentación, dentro de las conocidas arquitecturas Modelo/Vista/Controlador (MVC).

Para lograr ese objetivo la atención se centrará, por tanto, en el estado del arte de los Frameworks J2EE dedicados a dar soporte al patrón MVC y más en concreto de los actuales ***Frameworks de Presentación.***

Para ello se estudiará y evaluarán las diferentes tecnologías existentes en la actualidad aplicadas para este fin, se comentarán los aspectos más relevantes de cada una de ellas y sacando conclusiones de las ventajas/desventajas y en definitiva las características más representativas de todas ellas.

Una vez conocidas tales tecnologías, se realizará un diseño y la correspondiente implementación de tal Framework de presentación. Tal Framework implementará una pequeña aplicación del ***Club Ciclista UOC (CUOC),*** que será un Gestor/Backoffice que permitirá registrarnos como *administrador* y gestionar los socios dados de alta en la aplicación (y con posibilidad de ampliación, por ejemplo, introduciendo diferentes "*roles*" que permitiesen unas operaciones u otras).

Permitirá, por tanto, con el rol de "administrador", registrarnos inicialmente y editar los datos, perfil, listar todos los socios, y dar de alta y baja a éstos. Otras ampliaciones (indicadas en la aplicación) podrían ser dar de alta competiciones y asignar o borrar socios a cada una de ellas.

1.2 Objetivos

Objetivos Técnicos:

Fijados como objetivo los ***Frameworks*** de presentación, se estudiarán las principales características que nos ofrece este tipo de arquitecturas, analizando su estructura y funcionalidades básicas, que nos ayudarán a entender y encauzar nuestro posterior desarrollo.

Tras realizar una introducción a la tecnología propuesta y previa a las fases de análisis e implementación del Framework fijado, se estudiarán tres alternativas actuales del mercado, donde se analizarán las características más relevantes de cada una de ellas, y destacando las ventajas/inconvenientes de unas respecto a otras.

Dentro del estudio del mercado actual de Frameworks de presentación (*MVC*) basados en Java, se pueden encontrar una gran diversidad, entre los que se podrían destacar: *Struts*, *Turbine*, *Tapestry*, *Apache Cocoon*, *JPublish*, *JSF*, *vRaptor*, *Chrysalis*, *Spring*, etc.

De todos ellos se han seleccionado tres de los más representativos a tener en cuenta, que serán los que se sometan a análisis. Estos serán los siguientes:

- *Struts*
- *Tapestry*
- *Grails*

De cada uno de los mencionados, se extraerán las características más esenciales además de información de los patrones J2EE utilizados por éstos, en los que posteriormente nos basaremos y servirán de referencia de cara a la construcción de nuestro propio Framework.

El nuevo Framework deberá tener en cuenta, a grandes rasgos, las características esenciales de este tipo de arquitecturas MVC, diferenciando claramente tres componentes:

- Implementación de un ***controlador*** que se encargue de procesar todas las peticiones, y en función de éstas, gestionar de un modo u otro su solución.

- Implementación del ***modelo***, con toda la lógica necesaria y servicios para llevar a cabo las peticiones enviadas.

- Desarrollo de la ***vista*** del patrón MVC, donde se serializará los diferentes datos e información, frutos de un procesado previo de la capa de negocio.

De la mano del estudio de los diferentes Frameworks y de la implementación de uno propio, se adquirirán conocimientos sobre los ***patrones de diseño*** MVC y la forma de implementarlos, identificando las fases por las que pasan las peticiones de los usuarios.

En cuanto a la ***tecnología*** que empleada para la implementación de nuestra aplicación será Java J2EE con el *JDK* 1.6. Tal aplicación se instalará en un Servidor de Aplicaciones *JBoss* 6.1.0 y empleará como Base de Datos *MySQL*. También para la realización de todo el proceso de compilación y generación de los entregables se hará uso de la herramienta *Ant*, versión 1.8.4

Objetivos Funcionales:

La aplicación ***Club Ciclista UOC (CUOC)*** hará uso de nuestro nuevo Framework ***FCUOC*** y cubrirá varias funcionalidades bien diferenciadas, que según el perfil/rol con el que se interactuará. En la aplicación real se ha gestionado un único perfil que será el de administrador y toda la gestión de usuarios pero se presenta la posibilidad de gestionar por un lado los diferentes perfiles y por otro la gestión de competiciones como mejoras/evoluciones de la aplicación. Para el caso de tener ambas características adicionales también implementadas, el funcionamiento de la aplicación sería el siguiente:

1. *Perfil Usuario/Miembro:*

Este perfil englobará todos aquellos individuos que deseen formar parte del club ciclista, a los que les ofrecerá la posibilidad de registrarse introduciendo sus datos personales.
Una vez registrados y posteriormente aceptados (por el administrador del sistema), podrán logarse siempre que deseen, pudiendo acceder a las siguientes opciones:

- Editar sus datos personales (aquellos que introdujo en el momento del registro).
- Poder apuntarse a cualquiera de las carreras que el club organice o de las que forme parte.

2. ***Perfil Administrador:***

Este perfil estará íntimamente ligado a la administración del Club Ciclista, por lo que no tendrá acceso ningún otro individuo ajeno al club.
Con el usuario y contraseña correspondientes, el administrador podrá realizar las siguientes acciones:

- Dar de alta solicitudes de nuevos miembros, pudiendo visualizar y editar (si fuese necesarios) cualquiera de sus datos personales.
- Gestionar las competiciones de las que el club formará parte (bien sea organizadas por el club como por terceros) y publicarlas, para que todos los miembros del club tengan acceso y puedan apuntarse si así lo desean.

En términos generales, se tratará de realizar un interfaz lo más sencillo y usable posible, de cara a facilitar al usuario el acceso a las diferentes opciones y campos disponibles. También se realizarán las comprobaciones y validaciones necesarias en los diversos formularios, a fin de evitar inconsistencias en el sistema, provenientes de la introducción de datos incorrectos.

Otras posibles funciones extras que podrían implementarse podría ser la posibilidad de añadir más validaciones (hacer más robusta la introducción de datos), que el usuario pudiese subir archivos (como por ejemplo, la foto de perfil), añadir funcionalidades/validaciones en la capa cliente mediante JQuery, etc.

1.3 Plan de Trabajo

Todos los puntos anteriormente comentados se pueden resumir y organizar de forma esquemática en los siguientes hitos a ir alcanzando a lo largo del proyecto. Tales hitos se harán corresponder con cada una de las entregas del proyecto y podrían ser los siguientes:

- **Fase1 → *Propuesta del proyecto*** (*Pec1*):
  - ✓ Plan de Trabajo, fijando objetivos generales y específicos.
  - ✓ Definición de las subtareas a realizar.
  - ✓ Distribución de Tiempos para tales tareas.
  - ✓ Inicio de la memoria.

- **Fase 2 → *Análisis y Diseño*** (*Pec2*):
  - ✓ Estudio de la tecnología actual a emplear.
  - ✓ Visión general de los Frameworks de presentación:
    - Análisis del Framework 1 (*Struts*).
    - Análisis del Framework 2 (*Tapestry*).
    - Análisis del Framework 3 (*Grails*).
  - ✓ Realización de comparativa y conclusiones del estudio.
  - ✓ Continuación de la memoria hasta la presente fase.

- **Fase 3 → *Implementación*** (*Pec3*):
  - ✓ Análisis y Diseño de la aplicación.
  - ✓ Implementación:
    - Implementación del Framework.
    - Aplicación J2EE Contenedora.
  - ✓ Continuación de la memoria hasta la presente fase.

- **Fase 4 → *Entrega Final:***
  - ✓ Elaboración y realización de juegos de Pruebas.
  - ✓ Finalización de la memoria de todo el proyecto.
  - ✓ Elaboración de una presentación del trabajo realizado.

1.3.1 Planificación

Vista la subdivisión general realizada para las diferentes tareas, con un ***Diagrama de Gantt*** se mostrará la planificación de tiempos estimada para cada una de ellas, creando una serie de hitos más importantes que se harán corresponder, como ya se ha comentado, con cada una de las entregas estipuladas en el plan docente, que serán las entregas de las diferentes *PECs* y la *Entrega Final.*

Al resto de "subtareas" ubicadas entre los diferentes hitos se les ha asignado tiempos puramente estimatorios, que podrán variar levemente a medida que transcurra el proyecto.

Tal diagrama tendrá el siguiente aspecto:

NOTA: Se ha subdividido el diagrama general en cuatro partes, para poder visualizar mejor cada una de las tareas de cada etapa.

- Planificación para el hito 1 (***Pec1***): Fecha de Entrega: 3 de Octubre del 2012.

Fig.1.1: *Diagrama Gantt de planificación de "Propuesta Proyecto"*

- Planificación para el hito 2 (***Pec2***): Fecha de Entrega: 8 de Noviembre del 2012.

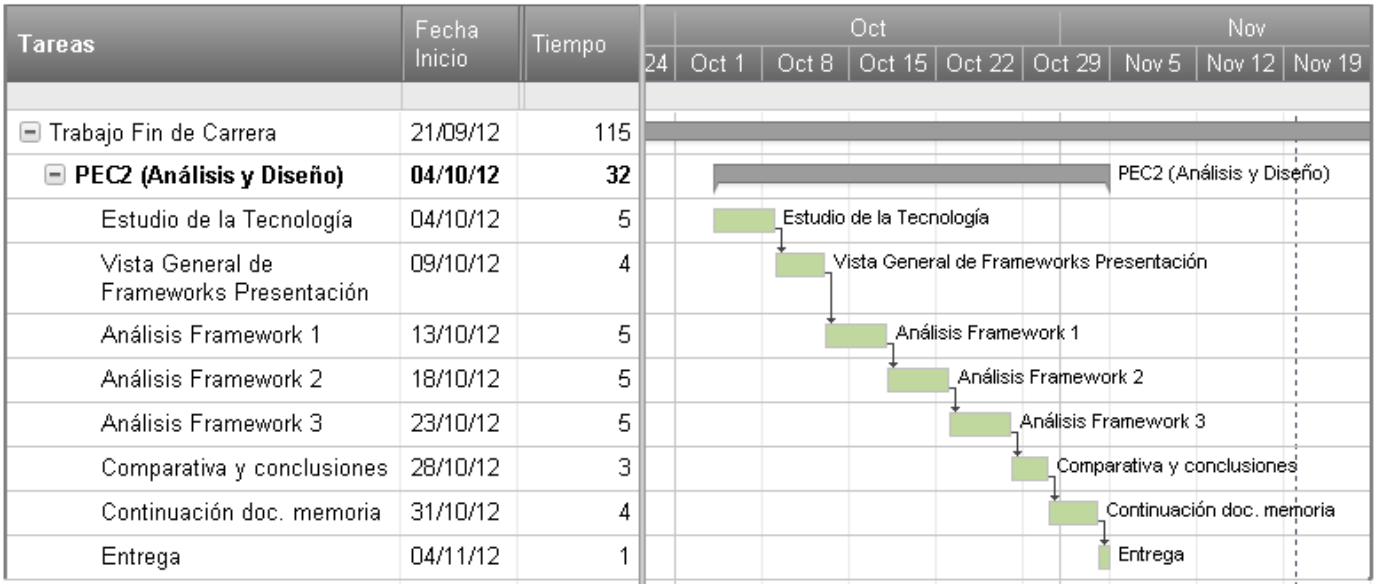


Fig.1.2: *Diagrama Gantt de planificación de "Análisis y Diseño"*

- Planificación para el hito 3 (***Pec3***):
    - Fecha de Entrega: 17 de Diciembre del 2012.

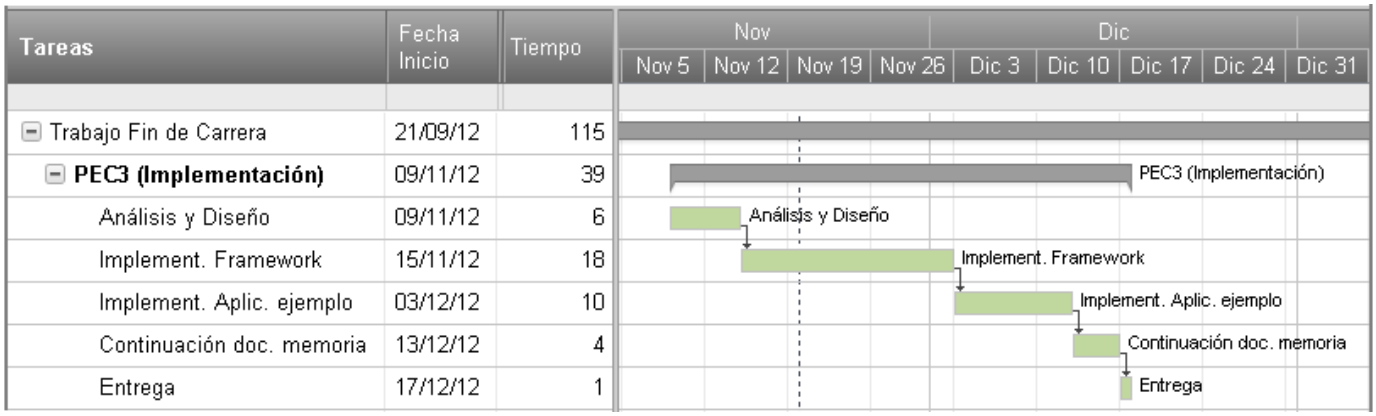


Fig.1.3: *Diagrama Gantt de planificación de "Implementación"*

- Planificación para el hito 4 (***Entrega Final***):
    - Fecha de Entrega: 14 de Enero del 2013.

Fig.1.4: *Diagrama Gantt de planificación de "Entrega Final"*

1.4 Resultados Obtenidos

Se ha obtenido como producto el Framework de presentación objetivo de este proyecto, empaquetado con todas las clases, ficheros de configuración, y la aplicación de ejemplo de uso. Los productos obtenidos y que acompañarán al proyecto serán los siguientes:

- ✓ Memoria y Documentación:

Memoria Final (formato .pdf) con toda la documentación acerca del plan de trabajo, estudio del mercado y estudio e implementación del Framework y la Aplicación que lo incluye. También se incluirán como anexos, entre otros, los documentos necesarios que expliquen la correcta instalación y ejecución del producto desarrollado.

- ✓ Implementación del Framework FCUOC
- ✓ Implementación de la Aplicación de Ejemplo Club Ciclista UOC.
- ✓ Presentación (*PowerPoint*) explicando los puntos más relevantes de todo el Proyecto.

2. Entorno de Trabajo J2EE

2.1 Principales Características

Antes de nada se realizará una breve introducción a la plataforma J2EE:

¿Qué se entiende por J2EE?

J2EE es una plataforma de programación dirigida a desarrollar y ejecutar software de aplicaciones en el lenguaje de desarrollo JAVA. J2EE nos permite emplear arquitecturas de N capas distribuidas, y se apoya en su totalidad en componentes de software modulares que se ejecutan sobre un servidor de aplicaciones.

El estándar J2EE define detalladamente un conjunto de servicios que un servidor de aplicaciones ha de contemplar, de la mano de una API estándar para acceder a tales servicios.
Se debe tener en cuenta que J2EE no es un producto sino una especificación, en base a esta especificación nacen numerosas implementaciones.

Entrando más en detalle. La especificación J2EE define tres tipos de componentes básicos que pueden emplear de cara al desarrollo:

- ***Servlets:***

Los Servlets nos proporcionan un método para implementar programas del lado del servidor. El uso común para el que están dedicados es para la generación de páginas web dinámicas.

- ***JSP (Java Server Pages):***

Los JSP permiten a los diseñadores web elaborar páginas web interactivas sin entrar en detalles del lenguaje java. A diferencia del HTML, los JSP permiten fragmentos de código Java incrustados en la
página web.

- ***EJB (Enterprise Java Beans)***

De los tres componentes descritos, son los más relevantes de cara a tener en cuenta.
De un EJB (que es más en concreto una clase java) se pueden destacar entre otras varias características, como por ejemplo:

- ✓ Son distribuidos.
- ✓ Usan transacciones.
- ✓ Multihilos.
- ✓ Persistentes

Se ha de tener en cuenta que muchas de las características de los EJBs son proporcionadas por los distintos servidores de aplicación.

A grandes rasgos, entre las diferentes características que posee la plataforma J2EE que ya se ha mencionado (además de las que se estudiarán a continuación con respecto a la arquitectura en capas), se puede observar que las aplicaciones implementadas siguiendo este estándar tendrán las siguientes ***ventajas***:

- ✓ Alta productividad en el desarrollo de las distintas tecnologías J2EE y gran facilidad para la integración de aplicaciones corporativas y con sistemas ya existentes.

- ✓ Mayor escalabilidad al describir las características básicas de transacciones y desarrollando distintos tipos de componentes de aplicación J2EE con modelos flexibles de seguridad.

- ✓ Mayor posibilidad de elección de las diferentes plataformas de desarrollo y producción.

- ✓ Uso de herramientas y software libre que agiliza la implementación de software y que permiten el funcionamiento en los distintos módulos de ejecución.

2.2 Arquitectura MVC

2.2.1.1 Características Principales y Evolución

La especificación de J2EE define su arquitectura basándose en los conceptos de *capas*, *containers*, *componentes*, *servicios* y las características de cada uno de éstos.

A lo largo de la evolución de J2EE, se han concebido diferentes arquitecturas a la hora del diseño de aplicaciones, donde a grandes rasgos se pueden diferenciar varias etapas:

- ✓ **Arquitectura de dos capas:**

Inicialmente, una aplicación web estándar se planteaba bajo una arquitectura de dos capas, teniendo por un lado la vista y el negocio y en la otra capa el acceso a datos:

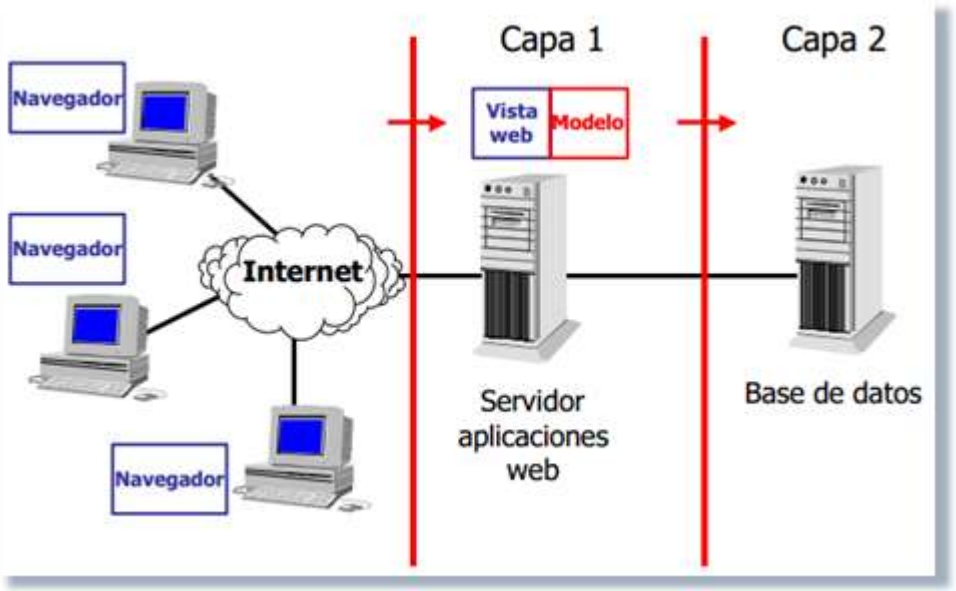


Fig.2.1: *Esquema arquitectura web en dos capas*

Los problemas de mezclar la lógica de negocio y la presentación, entre otras, es que era una solución ***no escalable***, por lo que en caso de una numerosa llegada de petición, la aplicación podría llegar a "morir de éxito".

Otro problema existente es el hecho de tener íntimamente ligada la capa de la vista, de tal forma que se limitaba la posibilidad de poder implementar otras vistas, por ejemplo, si quisiéramos desarrollar otro tipo de vistas para un usuario administrador.

Del mismo modo, si quisiéramos hacer actualizaciones o nuevas implementaciones por el lado del modelo, se vería la aplicación altamente afectada debido al ***alto acoplamiento*** entre las capas. El mantenimiento por consiguiente también se dificulta.

Soluciones posteriores pasaron por mantener la misma arquitectura pero replicar la Capa1 (vista+modelo) N veces, con lo que no quedaba bien solventado el problema.

- ✓ **Arquitectura de tres capas:**

Evoluciones posteriores que solventarían en cierta medida tales problemas, irían de la mano de una definición de la arquitectura en tres capas, que representamos de forma muy clara en el siguiente diagrama:

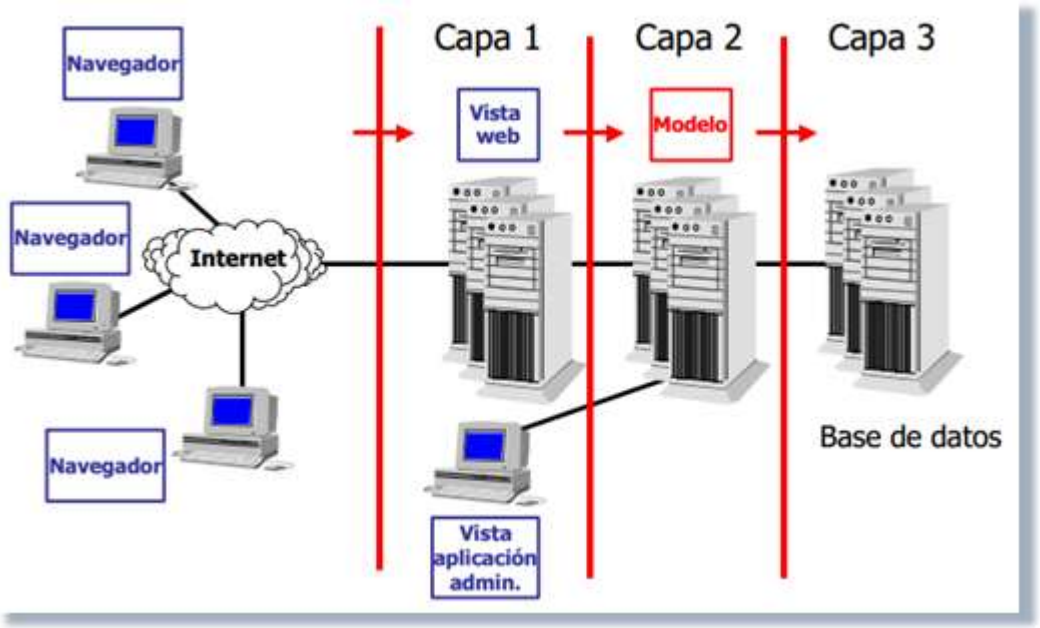


Fig.2.2: *Esquema arquitectura web en tres capas*

Con esta solución de separar la lógica de presentación y la lógica de negocio en 2 capas, se solventa el problema de la escalabilidad y tolerancia a fallos.

Queda claro que las arquitecturas en dos capas no son las más aconsejables, pero cabe decir que en la actualidad aún son numerosas las aplicaciones web que siguen empleando este método (replicando por ejemplo la capa web y empleando una máquina más potente para la base de datos), o en aplicaciones para intranets (donde no se llega muchas veces si a replicar la capa web).

2.2.2 Arquitectura MCV: Capas

Las aplicaciones J2EE son divididas en diversas capas: la capa cliente, la capa web, la capa de negocio y la capa de datos:

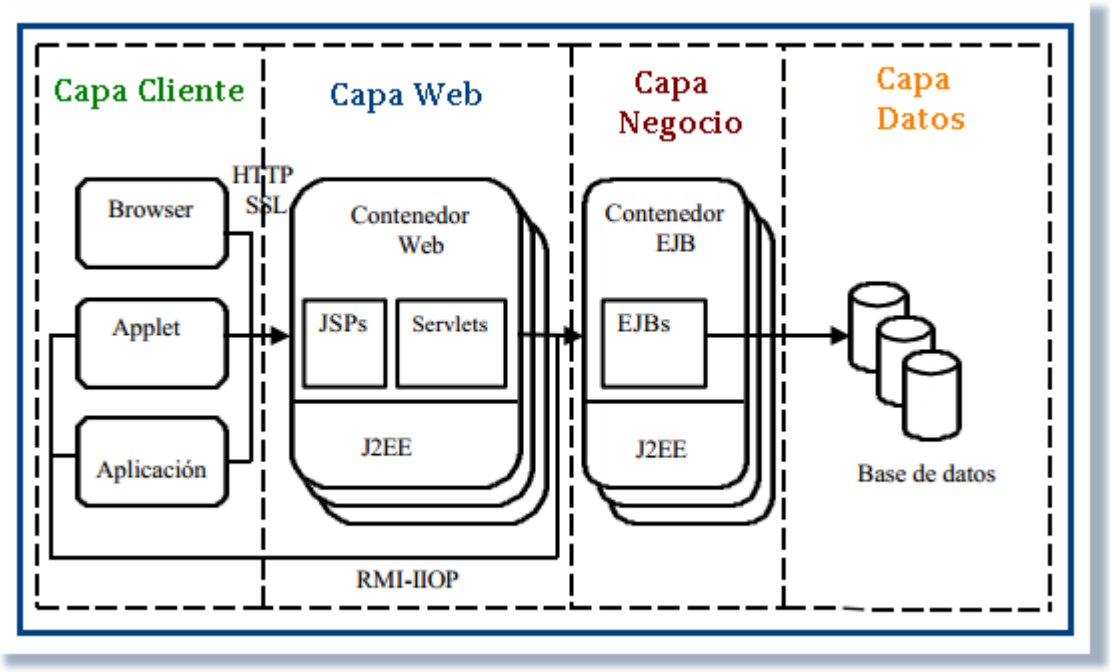


Fig.2.3: *Esquema Arquitectura Web J2EE en cuatro capas*

2.2.2.1 Capa Web

Esta capa corresponde básicamente a lo que encontramos en la máquina de cada cliente. Es la interfaz gráfica del sistema y se encarga de interactuar con el usuario. J2EE tiene soporte para diferentes tipos de clientes incluyendo clientes HTML, applets Java y aplicaciones Java.

2.2.2.2 Capa de Presentación

Se encuentra ubicada en el servidor web de aplicaciones y contiene toda la lógica de presentación que se emplea para generar una respuesta al cliente. "Conectada" con la anterior capa, recibe los datos del usuario desde la capa cliente y basado en éstos genera una respuesta apropiada a la solicitud. J2EE utiliza en esta capa las componentes *Java Servlets* y *JavaServer Pages* para crear los datos que se enviarán al cliente.

2.2.2.3 Capa de Negocio

Esta capa se encontrará en el servidor de aplicaciones y contiene el núcleo de la lógica de negocio de la aplicación. La capa de negocio no proporcionará las interfaces necesarias para utilizar el servicio de los componentes del negocio.

Las componentes del negocio por otro lado, interactuarán con la capa de datos.

Pueden proporcionar un Framework que es necesario para desarrollar una aplicación multicapa. Proporcionan servicios tales como cómputo distribuido, multihilo, seguridad y persistencia.

2.2.2.4 Capa de Datos

La capa de datos es responsable del sistema de información de nuestra aplicación (o de nuestra empresa) incluyendo bases de datos, sistema de procesamiento datos, sistemas *legados* y sistemas de planificación de recursos. En esta capa se puede encontrar el punto donde las aplicaciones J2EE se podrán integrar con otros sistemas no J2EE.

2.3 Patrones de Diseño

2.3.1 ¿Qué se entiende por un Patrón de Diseño?

En términos generales, la elaboración de patrones de diseño es la herramienta para la búsqueda de soluciones a problemas comunes en el desarrollo de software y de otros ámbitos relacionados con el diseño de interfaces.

Es por tanto, un diseño específico para un problema, pero general como para poder adecuarse a futuros requisitos y problemas.

Las ***características básicas*** que han de seguir todos los patrones son las siguientes:

- Evitar resolver cada problema partiendo de cero, por lo que el grado de reutilización ha de ser elevado, partiendo de soluciones que han sido útiles en el pasado y sabiendo que el patrón diseñado será aplicable a problemas de diseño en diferentes circunstancias.
- El grado de efectividad del patrón se medirá en la medida en que sea capaz de resolver problemas similares en ocasiones anteriores.
- Los patrones de diseño son guías, no reglas rigurosas.

2.3.2 Patrón Modelo Vista Controlador (MVC)

El conocido patrón Modelo – Vista - Controlador (*MVC*) es en sí una arquitectura de software donde se separan 3 componentes bien diferenciados en una aplicación:

- La interfaz de usuario.
- Lógica de negocio.
- Los datos.

✓ **Modelo**:

- Representa la parte fuerte de la aplicación, es decir, todo aquello con lo que el sistema opera. Se encarga de ejecutar o de llevar a cabo los servicios o eventos dictados por el controlador e interactuar si es necesario con la bases de datos (capa de datos) aplicando para ello persistencia.
- Representa los datos del programa. Gestiona los datos y controla sus transformaciones.
- No posee relación ni conocimiento de la Vista ni del Controlador, sino que será el sistema quien mantenga las relaciones pertinentes entre el modelo y la vista, y de avisar a la vista cuando cambie el modelo.

✓ **Vista**:

- Interfaz que se genera y que llega al usuario, a través del cual interactúa y envía peticiones.
- En sí la vista se encarga de generar una representación visual del Modelo y muestra los datos y la información al usuario. Interactúa con el Modelo a través de una referencia al mismo.

✓ **Controlador**:

- Es en sí el interceptor o interceptores (en el caso de existir N controladores) de las diferentes peticiones y se ubica entre los dos anteriores (Modelo y los Datos), recibiendo las peticiones por parte del usuario e inicializando/invocando los servicios o eventos correspondientes a dicha petición.
- Con los datos obtenidos, devuelve un modelo a la capa de vista que posteriormente se encargará de hacer llegar al usuario.
- Entrará en acción cuando se realice algún cambio, ya sea en la información del Modelo o por alteraciones de la Vista. Interactúa con el Modelo a través de una referencia al mismo.

Estos tres componentes se unen mediante un patrón *Observer*, que tiene como misión informar cuando un objeto cambia de estado, todas sus dependencias sean notificadas y actualizadas.
A continuación se muestra un gráfico que explica de forma muy general una vista de proceso que muestra las relaciones entre las 3 capas de la arquitectura MVC bajo J2EE:

Fig.2.4: *Ciclo de vida de una petición (MVC)*

¿Qué ventajas conlleva el uso de MVC?

Entre las ventajas del empleo de esta arquitectura, podríamos destacar principalmente la siguiente, y es que es capaz de separar la lógica del negocio de la interfaz de usuario, con lo que se consigue:

- **I.** Facilitar la evolución por separado de ambos componentes (modular y poco acoplado).
- **II.** Incrementar la flexibilidad y aumentar su reutilización.
- **III.** Proporcionar un software mantenible.

2.3.3 Patrones de Diseño J2EE

A continuación se muestran los patrones que presenta J2EE para cada una de las diferentes capas:

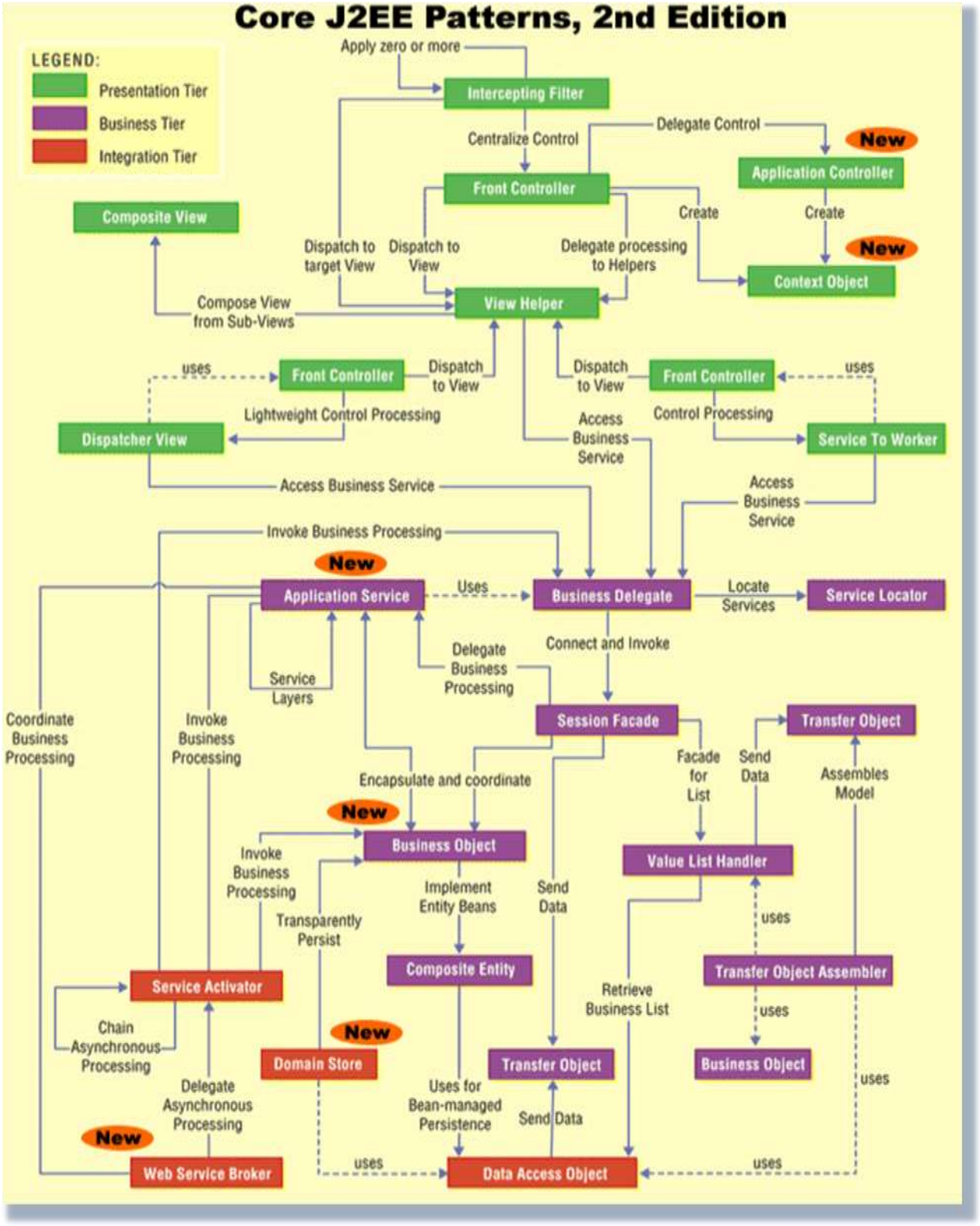


Fig.2.5: *Esquema de Patrones de Diseño J2EE*

A continuación vamos a dar una visión general de la lista de patrones:

Capa Web

a) Intercepting Filter:

Facilita el pre-procesamiento y post-procesamiento de una petición. *Intecepting Filter* se encuentra entre el cliente y el servidor y capta y modifica las peticiones y la posterior respuesta. También permite realizar otras funciones como compresión, logging, inserción de componentes desacoplados en la respuesta.

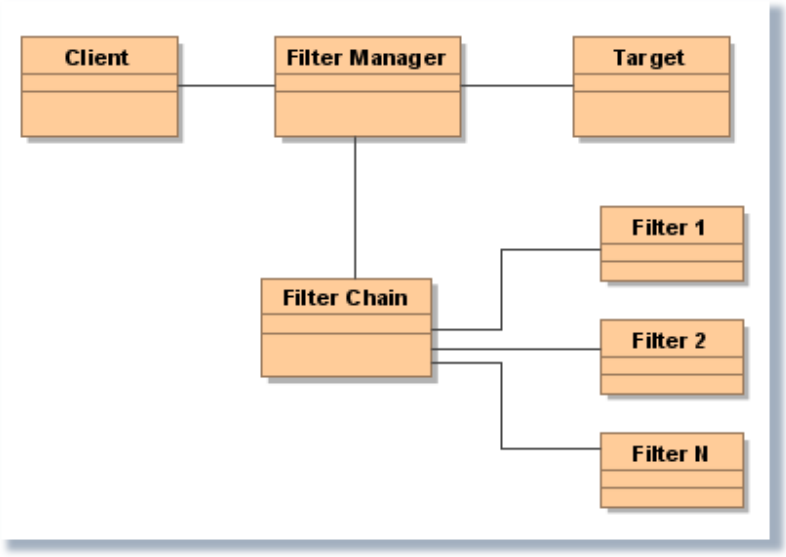


Fig.2.6: *Arquitectura del Patrón Intercepting Filter*

b) Front Controller:

Proporciona un control centralizado del manejo de la petición. Nos servirá por tanto como punto de entrada a nuestra aplicación, donde a partir de ella delega las peticiones al resto de clases.

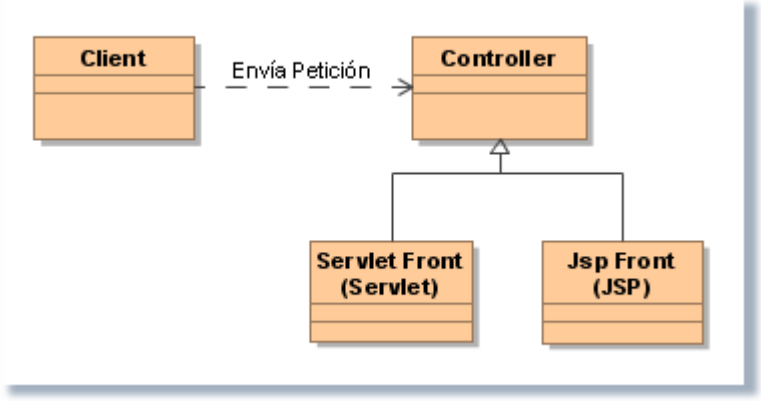


Fig.2.7: *Arquitectura del Patrón Front Controller*

c) Context Object:

Encapsula el estado en una forma de protocolo independiente para ser compartido a través de la aplicación. El modo de encapsulación será vía API, donde el mecanismo con el que se va a obtener la información y el uso de la tecnología será totalmente independiente.
Este patrón se empleará en nuestro Framework y su representación es la siguiente:

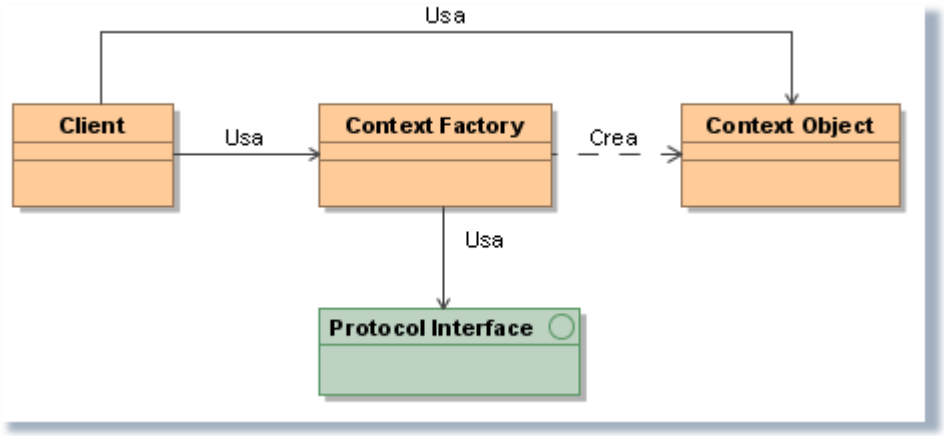


Fig.2.8: *Arquitectura del Patrón Context Object*

d) **Application Controller:**

Centraliza y modulariza el manejo de la acción y vista. En función de las peticiones recibidas, es el encargado de decidir qué acciones de negocio se van a realizar y qué vistas se van a emplear en la respuesta.

e) **View Helper:**

Separa el procesamiento de la lógica de la vista. Básicamente se encarga de que no se mezcle el código con la implementación de la vista. Las formas de conseguir tal objetivo pasarán por emplear Tags, Custom Tags, etc.

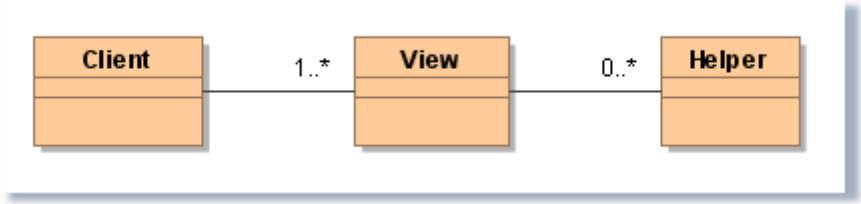


Fig.2.9: *Arquitectura del Patrón View Helper*

f) **Composite View:**

Crea una vista compuesta de las sub-vistas separando el contenido y la capa de gestión. Esto será útil cuando queramos crear una vista "compleja" compuesta de vistas simples (cabecera/pie, tablas de opciones, migas de pan, etc.)

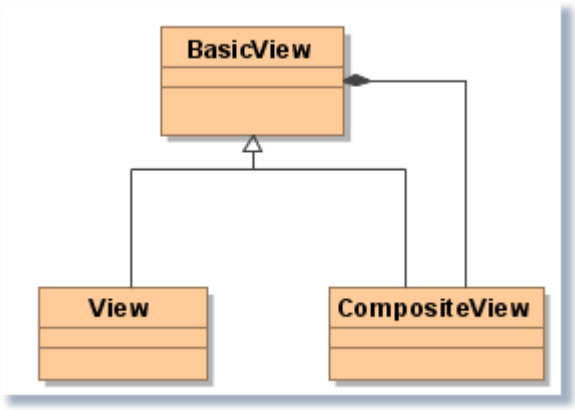


Fig.2.10: *Arquitectura del Patrón Composite View*

g) **Service to Worker:**

Llama al procesamiento de la lógica de negocio previo al procesamiento de la vista. Se encarga de centralizar la ejecución de la petición, en el cual invoca la lógica de negocio necesaria recuperando la información devuelta (y almacenándola en el modelo) y acto seguido dirigiéndose a la vista creando una respuesta dinámica según los datos guardados en el modelo (haciendo uso del patrón *View Helper*).

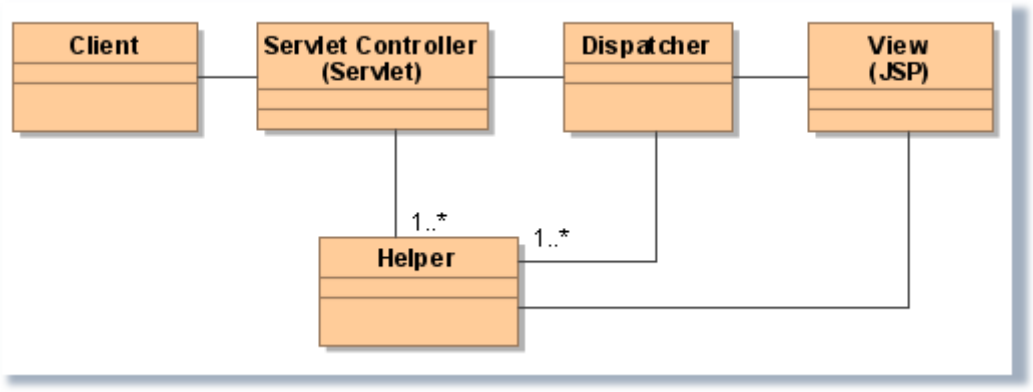


Fig.2.11: *Arquitectura del Patrón Service to Worker*

h) **Dispatcher View:**

Llama al procesamiento de la vista previo al procesamiento de la lógica de negocio. En este caso el orden es diferente, donde el control lo toma la vista y ésta es la que llama al negocio.

Se dará en los casos normalmente en los que el modelo ya está creado, pero como contrapartida conlleva mayor mantenimiento pues envuelve más lógica además de llamadas al negocio.

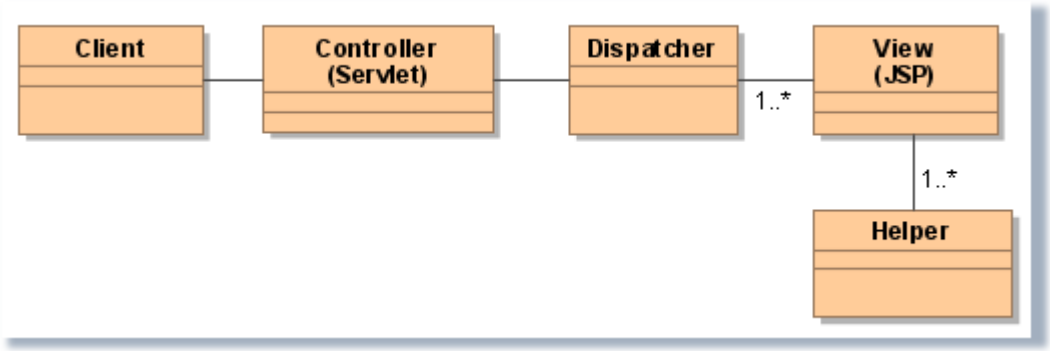


Fig.2.12: *Arquitectura del Patrón Dispatcher View*

Capa de Negocio

a) **Bussiness Delegate:**

Encapsula el acceso al servicio de la lógica de negocio. Sirve de punto intermedio entre la capa de presentación y la de negocio y se encarga entre otras cosas de, proporcionar un acceso común a la capa de negocio, conseguir separar físicamente entre ambas capas, actuar como caché ante numerosas peticiones, etc.

b) **Service Locator:**

Centraliza la búsqueda de la lógica para los servicios de negocio y los componentes. Realiza una búsqueda del servicio solicitado de forma transparente a donde esté implementado, indicando el modo de invocar a tal servicio.

c) **Session Facade:**

Expone los servicios de grano grueso a clientes remotos, de tal forma que se comportará como una interfaz centralizada de peticiones a métodos del negocio, evitando que se expongan los servicios de forma independiente, consiguiendo de este modo reducir el acoplamiento entre capas y haciendo transparente al cliente las posibles relaciones entre los diferentes componentes del servicio y del negocio.

d) **Application Service**

Añade comportamiento para proporcionar una capa de servicio uniforme. Con esto se logra sacar la lógica de negocio de los objetos de fachada, consiguiendo centralizar servicios de negocio determinados en un objeto que los englobe. Mediante tal patrón también se conseguirá centralizar la lógica de los diferentes casos de uso en un objeto determinado, sin mezclarlo con los objetos generales del negocio.

e) **Bussiness Object**

Encapsula y diferencia los datos del negocio de la lógica. Se suelen implementar como Beans de entidad y se encargarán por tanto de encapsular los datos, la lógica y su persistencia.

f) **Composite Entity**

Implementa objetos de negocio persistentes usando Beans de entidad. Además dará la posibilidad de agrupar objetos con respecto a un objeto principal (en función de la relación que guarden).

g) **Transfer Object:**

Transporta datos de diferentes tipos a través de una determinada capa. Tales datos servirán tanto para lectura como para escritura e irán almacenados en un único objeto, consiguiendo de este modo reducir el tráfico de objetos entre capas.

h) **Transfer Object Assembler:**

Ensambla un objeto de transferencia (Transfer Object) para múltiples fuentes de datos. El ensamblador como tal, podrá construir un Transfer Object compuesto de más Transfer Object.

i) **Value List Handler:**

Gestiona la búsqueda e iteración de grandes conjuntos de datos. Tal búsqueda la hará haciendo uso de un DAO (Data Access Object), guardando los resultados de tal forma que el cliente pueda acceder a ellos (mediante un Iterator).

Capa de Datos:

a) **Data Access Object:**

Abstrae y encapsula el acceso al almacén de persistencia. Se podrá acceder por tanto a los datos mediante un API independiente de la tecnología usada y abstrayéndonos del sistema de persistencia empleado.

b) **Service Activator:**

Proporciona acceso asíncrono a uno o más servicios. Permite mediante servicio de mensajes, recibir y gestionar peticiones asíncronas, que después transformará en llamadas a servicios del negocio.

c) **Domain Store:**

Proporciona persistencia (a través de un Framework de persistencia) de forma transparente para los objetos de negocio.

d) **Web Service Broker:**

Expone uno o más servicios usando XML y protocolos Web.

2.3.4 Otros patrones

Además de los ya estudiados patrones de Diseño J2EE, existen otra serie de patrones para la programación orientada a objetos, que podríamos clasificar en tres tipos:

2.3.4.1 Patrones de Creación

Características Principales:

- ✓ Abstraen el mecanismo de instanciación
- ✓ Colaboran en el sentido de que el sistema sea independiente de cómo se crean, componen y representan los objetos.
- ✓ Permiten configurar el sistema en tiempo de compilación (estáticamente) o de ejecución (dinámicamente).

- ✓ Flexibilizan: Qué se crea / Cómo se crea / Quien lo crea / Cuándo se crea.

Tipos:

a) Singleton:

Aseguran que una clase posee una estancia única y da un punto de acceso global a dicha estancia.

b) Abstract Factory:

Proporciona una interfaz para implementar familias de objetos relacionados o que dependan entre sí, sin llegar a detallar sus clases concretas. También denominado: "*Kit".*

c) Factory Method:

Define una interfaz para crear un objeto, pero delega en las subclases que decidan qué clase instanciar. Permite además que una clase delegue en sus subclases la creación de objetos. También denominado: *"Virtual Constructor".*

2.3.4.2 Patrones de Estructurales:

Características Principales:

- ✓ Establecen cómo se componen clases y objetos para formar estructuras mayores que implementen nuevas funcionalidades.
- ✓ Los patrones de clase usan la herencia para componer interfaces o implementaciones.
- ✓ Los patrones de objeto, definen formas de componer objetos para implementar nuevas funcionalidades.

Tipos:

a) Composite:

Se encargan de componer objetos en estructuras con forma de árbol para representar jerarquías.

b) Facade:

Proporcionan una interfaz unificada para un conjunto de interfaces de un subsistema.

Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

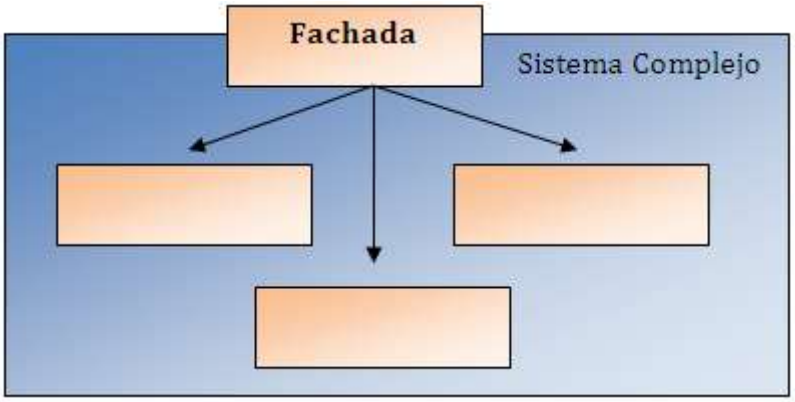


Fig.2.13: *Arquitectura del Patrón Facade*

c) ***Proxy:***

Proporcionan una representación o alternativa de otro objeto para gestionar y controlar el acceso del mismo. También denominado: *"Surrogate".*

d) ***Adapter:***

Transforma el interfaz de una clase en otro interfaz que espera el cliente. El Adapter nos permitirá trabajar con clases juntas que de otra forma no sería posible por tener interfaces incompatibles. También denominado: *"Wrapper".*

2.3.4.3 Patrones de Comportamiento:

Características Principales:

- ✓ Se basan en algoritmos y en asignación de responsabilidades entre objetos.
- ✓ Especifican tanto patrones de clases y objetos, como patrones de comunicación entre ellos.
- ✓ Se caracterizan por tener un flujo de control complejo, difícil de seguir en tiempo de ejecución.
- ✓ Permiten que de cara al diseño sólo nos tengamos que preocupar de cómo interconectar objetos.

Tipos:

a) ***Iterator:***

Proporciona un mecanismo de acceso a los elementos de un contenedor de forma secuencial sin exponer su representación interna. También denominado: *"Cursor".*

b) *Observer:*

Especifica una dependencia de 1 a N entre objetos, de tal modo que cuando un objeto cambia de estado se notifica y se actualizan automáticamente todos los objetos que dependen de él. También denominado: *"Dependents", "Publish-Subscribe".*

c) *Template Method:*

Encargado de definir el esqueleto de un algoritmo, delegando en las subclases alguno de sus pasos. Permite que las subclases cambien pasos de un algoritmo sin cambiar su estructura.

d) *State:*

Permitir que un objeto varíe su comportamiento cada vez que cambie su estado interno. También denominado: *"Objets for state".*

e) *Strategy:*

Especifica una familia de algoritmos, encapsula cada uno de ellos, y los hace intercambiables entre sí. También denominado: *"Policy".*

f) *Comand:*

Encapsula peticiones a objetos. Permite parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de peticiones, así como deshacer las peticiones. También denominado: *"Action", "Transaction".*

g) *Chain of Responsability:*

Evita acoplar el emisor de una petición a su receptor, dando a más de un objeto la posibilidad de responder a la petición.

Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.

A continuación, vistos todos los tipos de patrones, mostramos un diagrama resumen para comprobar la clasificación que se hace para los diferentes tipos:

		Objetivo		
		Creación	***Estructurales***	***De comportamiento***
Contexto	***Clase***	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Fig.2.14: *Esquema de Patrones de Diseño J2EE*

3. Frameworks de Trabajo

3.1 Estado del Arte

3.1.1 Introducción y características

Actualmente se puede definir un marco de desarrollo o técnicamente conocido como Framework, un a implementación compuesta de un conjunto de componentes y librerías empleados para desarrollar una estructura estándar de una aplicación, que, sumado a una metodología de uso y su correspondiente documentación nos permitirá diseñar, analizar, implementar y desplegar aplicaciones de forma estándar, de alta calidad y rapidez.

También se pueden definir los Framework como soluciones completas que contemplan herramientas de apoyo a la construcción (ambiente de trabajo o desarrollo) y motores de ejecución (ambiente de ejecución).

En los tiempos que corren, se pueden contar con numerosos Frameworks disponibles en el mercado para el desarrollo de aplicaciones. Una gran parte de este conjunto son comerciales

implementados por empresas y otros tantos son de código abierto, implementados por comunidades Open Source.

Dentro de la tecnología que estamos estudiando Java, en el ámbito específico de aplicaciones Web se tienen numerosos Framework, como Struts1, Struts2, Java Server Faces, Spring, etc. Estos Frameworks en la práctica son como ya se ha mencionado, conjuntos de librerías (API's) para desarrollar aplicaciones Web, más una serie de librerías para su ejecución (o motor), más un conjunto de herramientas para facilitar esta tarea (debuggers, ambientes de desarrollo como Eclipse, etc.).

Fig.3.1: *Frameworks del Mercado Actual*

3.1.2 Principales objetivos

Uno de los principales objetivos de un Framework es promover la reutilización del código, a fin de reducir los tiempos de desarrollo de los proyectos software. Como ya se ha mencionado, se cuenta con numerosos Frameworks en el mercado, cada uno dedicado a diferentes objetivos, bien sea al desarrollo de aplicaciones Web, implementación de aplicaciones multiplataforma, para lenguajes de programación concretos, para sistemas operativos, etc.

Un Framework determina en sí la estructura general de una aplicación, la arquitectura de que está compuesta, la composición entre las diferentes clases y objetos y las relaciones entre los mismos.

Un Framework además recolectará aquellas decisiones sobre el diseño que son comunes a su dominio de aplicación. Por este motivo, nos permitirá no sólo reutilizar código si también el diseño planteado con el consiguiente ahorro de recursos y tiempo dedicados para sacar adelante la aplicación.

3.1.3 Ventajas del uso de Frameworks

Entre las principales ventajas que nos ofrece el empleo de Frameworks podríamos destacar las siguientes:

- ✓ *Proporciona una arquitectura consistente entre aplicaciones:*

  Al usar Frameworks, las aplicaciones generadas comparten una arquitectura común. Esto promueve que sea más fácil de aprender, implementar y mantener.

  A nivel de programación no se deberá invertir gran parte del tiempo en buscar las clases necesarias, interconectarlas o descubrir los métodos que contienen. Los Frameworks ocultan toda esta complejidad dando un alto nivel de abstracción y transparencia de cara al desarrollador.

  Ofrecerá por tanto generadores de código y plantillas arquitectónicas, desarrolladas con las últimas versiones de componentes y recogiendo todas las mejores prácticas de los mismos.

- ✓ *Reduce los tiempos de implementación/desarrollo:*

  Los proyectos de desarrollo ya no tendrán que resolver los múltiples problemas asociados a las aplicaciones web. Los Frameworks reducirán además no sólo la codificación en sí, sino la puesta en marcha, ya que proporcionan subsistemas que se sabe que ya funcionan. En definitiva, proporcionan código que no se tendrá que reescribir y será más mantenible.

- ✓ *Reduce los riesgos del proyecto:*

  Con un modelo de programación complejo, el riesgo de cometer fallos en los inicios de un proyecto es alto. Un Framework de desarrollo de aplicaciones reduce significativamente este riesgo al proporcionar una base fiable y suficientemente probada. Gran parte de esa base es el conocimiento adquirido, reutilizable en todos los proyectos.

3.1.4 Frameworks del mercado a analizar

De cara a la elección/diseño de los mejores componentes de nuestro Framework, vamos a realizar un estudio previo del estado del arte en cuanto a Frameworks de presentación se refiere. Como ya se ha mencionado, existen en la actualidad innumerables Frameworks basados en JAVA, bien sean libres como de ámbito comercial. A continuación se mostrará un listado de los principales Frameworks basados en JAVA de la actualidad (como enlace de referencia rápida, se han añadido páginas de la *Wikipedia*):

Frameworks para Aplic.Web basadas en JAVA				
Apache Struts	AppFuse	Flexive	GWT	Grails
Vaadin	ItsNat	JavaServer Faces	Jspx	Makumba
OpenXava	Play	Eclipse RAP	RIFE	Seam
Spring	Stripes	Tapestry	WebWork	WaveMaker
Wicket	ZK	ICEfaces		

Fig.3.2: *Frameworks en la actualidad basados en Java*

Y una vez hecho un overview de las principales tecnologías, se centrará nuestro foco en tres de ellas: ***Struts2***, ***Tapestry*** y ***Grails***.

¿Por qué escogemos estos Frameworks?

Se ha escogido *Struts2* porque desde hace años (desde su predecesor *Struts1*) ha sido un referente en el mercado de los Frameworks de presentación y a día de hoy sigue siendo ampliamente utilizado en grandes desarrollos y a todos los niveles, y por otro lado *Tapestry* y *Grail*s, porque junto con otros Frameworks más, son grandes desarrollos que están entrando en el mercado con mucha fuerza, y aunque no tienen el poder de los Frameworks ya asentados, a juzgar por el gran seguimiento que tienen (JIRA, foros, redes sociales), prometen ser buenas alternativas a tener en cuenta para futuros proyectos.

A nivel personal, los he escogido por "empaparme" más sobre sus características, partiendo de un Framework bien conocido como es el caso de *Struts2* y ampliando conocimiento con estas dos tecnologías, que a día de hoy prácticamente las desconozco.

De todos ellos se estudiarán diversos puntos de cara a extraer toda la información posible para el posterior diseño de nuestro Framework de presentación. Se extraen por tanto:

- Características principales.
- Arquitectura Modelo Vista Controlador que implementa.
- Principales componentes.
- Ciclo de vida de una petición
- Configuración específica.

Una vez analizados todos estos puntos y previo al diseño de nuestro proprio Framework se elaborará una tabla descriptiva que confronte las diferencias más reseñables de las tres tecnologías.

3.2 Struts

3.2.1 Introducción

Struts es el Framework de presentación (software libre) de Apache orientado a la implementación y desarrollo de aplicaciones web basadas en plataforma J2EE. Struts implementa una arquitectura Modelo Vista Controlador basada en "*Servlet Filters*".

Struts2 se desarrollaba como parte del proyecto Jakarta de la Apache Software Foundation, pero actualmente es un proyecto independiente conocido como Apache Struts.

Struts2 proporcionará por lo tanto a los desarrolladores una infraestructura unificada sobre la que se pueden basar las aplicaciones web proporcionando soluciones de calidad, en un entono abierto, cooperativo, eficaz y apropiado tanto para desarrolladores independientes como para grandes equipos de de desarrollo.

3.2.2 Principales Características

Entre las diferentes características que nos presenta tal tecnología, se pueden destacar las siguientes:

- ✓ Principalmente es Framework Modelo Vista Controlador orientado a acciones.
- ✓ Arquitectura con un bajo acoplamiento y muy flexible, compuesta de plugins e interceptores que gestionarán las diferentes peticiones tratándolas con una acción o conjunto de acciones definidas.
- ✓ Proporciona un Framework de validación que nos permitirá desacoplar las reglas de validación del código de las acciones. Permite también usar cualquier clase Java normal (POJO) como una acción.
- ✓ Los componentes de la interfaz de usuario estarán predefinidos a través de una librería de etiquetas.
- ✓ El controlador ya se encuentra implementado por Struts2, aunque si fuera necesario se puede heredar y ampliar o modificar
- ✓ Define un mapa de navegación mediante un fichero de configuración o de anotaciones Java.
- ✓ Soporta diferentes tecnologías para la vista, como son los JSPs, plantillas Velocity, XSLT, Ajax, etc. gracias a plugins que permiten definir nuevos tipos de vistas.
- ✓ Se integra fácilmente con otras tecnologías como Hibernate, Spring, SiteMesh, JSTL, etc.
- ✓ Posee un motor de inyección de dependencias que permite inyectar unos componentes dentro de otros. El motor usado por defecto es Spring.

3.2.3 Arquitectura MVC

Con respecto a la arquitectura MVC que nos propone Struts2, se pueden destacar la siguiente estructura:

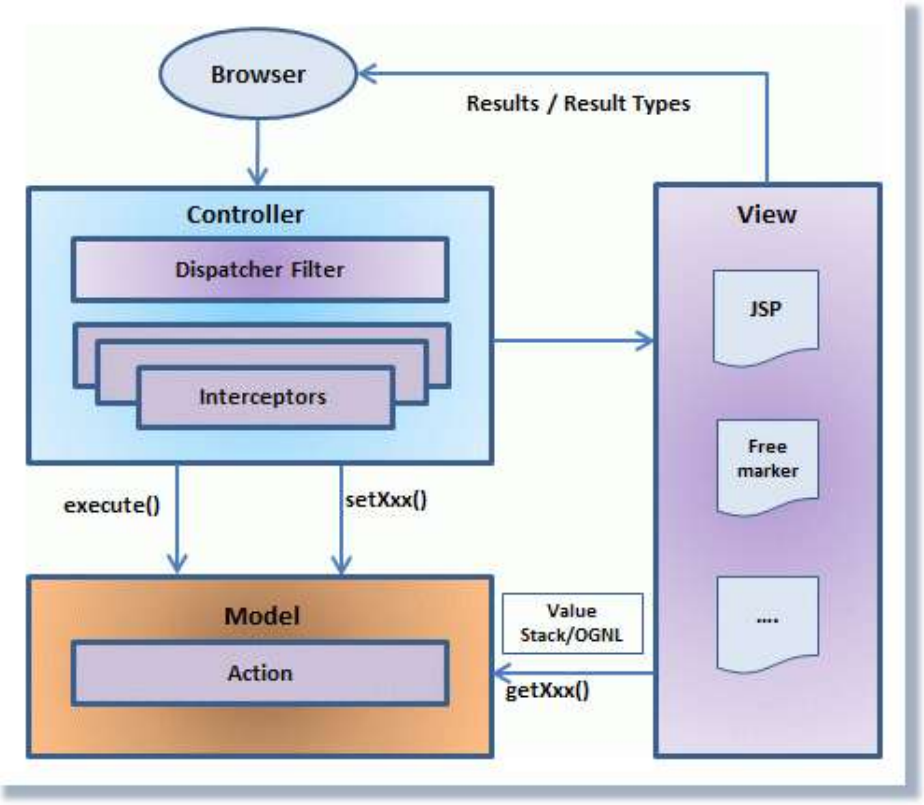


Fig.3.3: *Arquitectura MVC de Struts2*

El diagrama superior representa tal arquitectura a alto nivel para Struts2. El controlador se implementa con un servlet Dispatcher Filter además de diferentes interceptores, el modelo se basa en Actions y la vista como una combinación del resultado de Types y Results. La pila de valores y el OGNL proporcionan un thread común, enlazando y habilitando la integración entre los otros componentes.

- ✓ ***Controlador:***

  Implementado por el FilterDispatcher, es un Servlet Dispatcher Filter además de diferentes interceptores , que procesarán las peticiones entrantes y, en función de estas peticiones decidirán qué acción/es son las que deben de ejecutarse.

- ✓ ***Modelo:***

  El modelo se implementa mediante Actions que serán clases Java. Una acción (patrón ***Command***) se invocará en función de la petición recibida, para que ejecute una determinada lógica de negocio y devuelva finalmente al cliente el resultado de tal proceso.

En función del resultado devuelto (String) por el Action, el Framework sabrá a qué Vista deberá enviar la salida. El Action contiene la lógica de negocio, pero también servirá de Transfer Object para devolver los resultados a la Vista.

- ***Vista:***

  Esta capa se implementará mediante un objeto Result que será en la mayoría de los casos JSP, plantillas de Velocity, etc. pero como ya se ha mencionado al principio, podría soportar hasta plantillas XSLT. La vista resultante se decidirá en función del resultado de la acción ejecutada. En este punto se podrán montar vistas compuestas mediante el patrón ***Composite View***.

3.2.4 Ciclo de vida de una petición

Con el siguiente diagrama se puede representar muy claramente el flujo de estados por los que va pasando una petición:

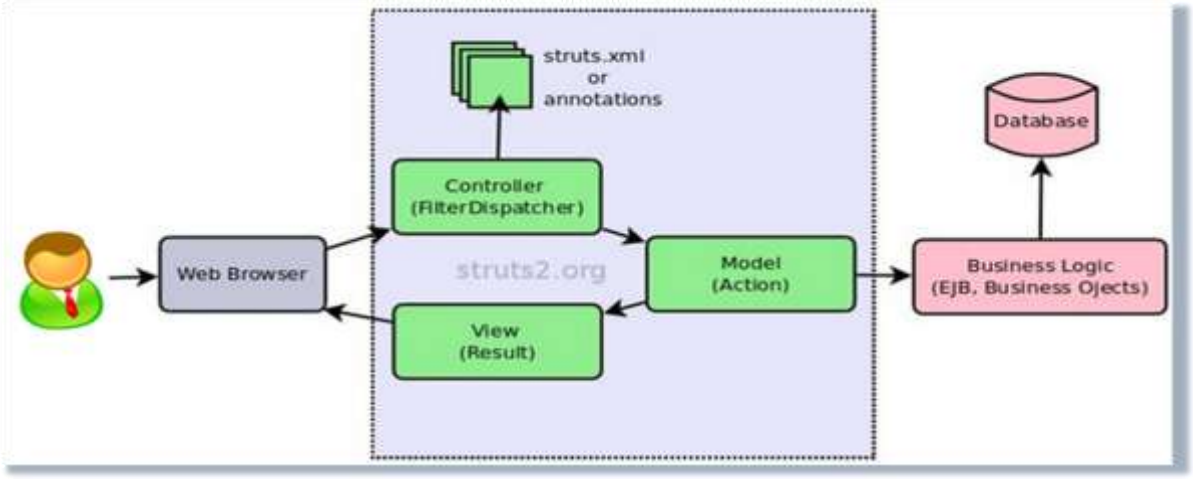


Fig.3.4: *Ciclo de vida de una petición con Struts2*

1. El usuario genera una petición del navegador accediendo mediante una URL a la aplicación.

2. La petición siempre llegará al controlador *FilterDispatcher* (este será configurado en el web.xml de todas las aplicaciones Struts2).

3. El *FilterDispatcher* buscará la clase *Action* a llamar en el fichero struts.xml. Alternativamente puede adivinarla usando convenciones. Según la URL invocada (*Action Mapping*), se instanciará una subclase de Action entre las configuradas en el fichero struts-config.xml.

El código del Action será el que haga los llamamientos al Modelo, bien directamente, bien a través del ***patrón*** de diseño ***Business Delegate***, que a fin de cuentas será como una interfaz restringida del modelo orientada a las necesidades concretas de la aplicación.

4. Se aplicarán los Interceptores definidos. Existirán diferentes interceptores que se pueden configurar para que ejecuten diferentes funcionalidades, workflows, validaciones, subida de ficheros, etc.

5. Se ejecutará la/s clases Action. Éstas especificarán la vista a mostrar usando anotaciones o pueden especificarse directamente en el fichero struts.xml. De todos modos, Struts2 sabe qué Vista (Result) será invocada mostrando los datos de vuelta al usuario.

Por lo tanto las Actions no sólo determinarán que vista mostrar, sino que proporcionarán datos necesitados también por la Vista.

6. Se renderiza la salida. Tras la ejecución del último Action, se determina cuál es la página a devolver y se realiza un forward a la misma.

7. Se devuelve la petición, ejecutándose los correspondientes interceptores, y se devuelve la petición al cliente. Aquí también se podrá añadir lógica adicional

8. Se muestra el resultado al Cliente, que podrá visualizar el resultado en el navegador.

Una vez vista una gráfica genérica, y detallados los pasos concretos que se llevan a cabo en el ciclo de vida de una petición, vamos a mostrarlo de forma más técnica mediante un diagrama de secuencia:

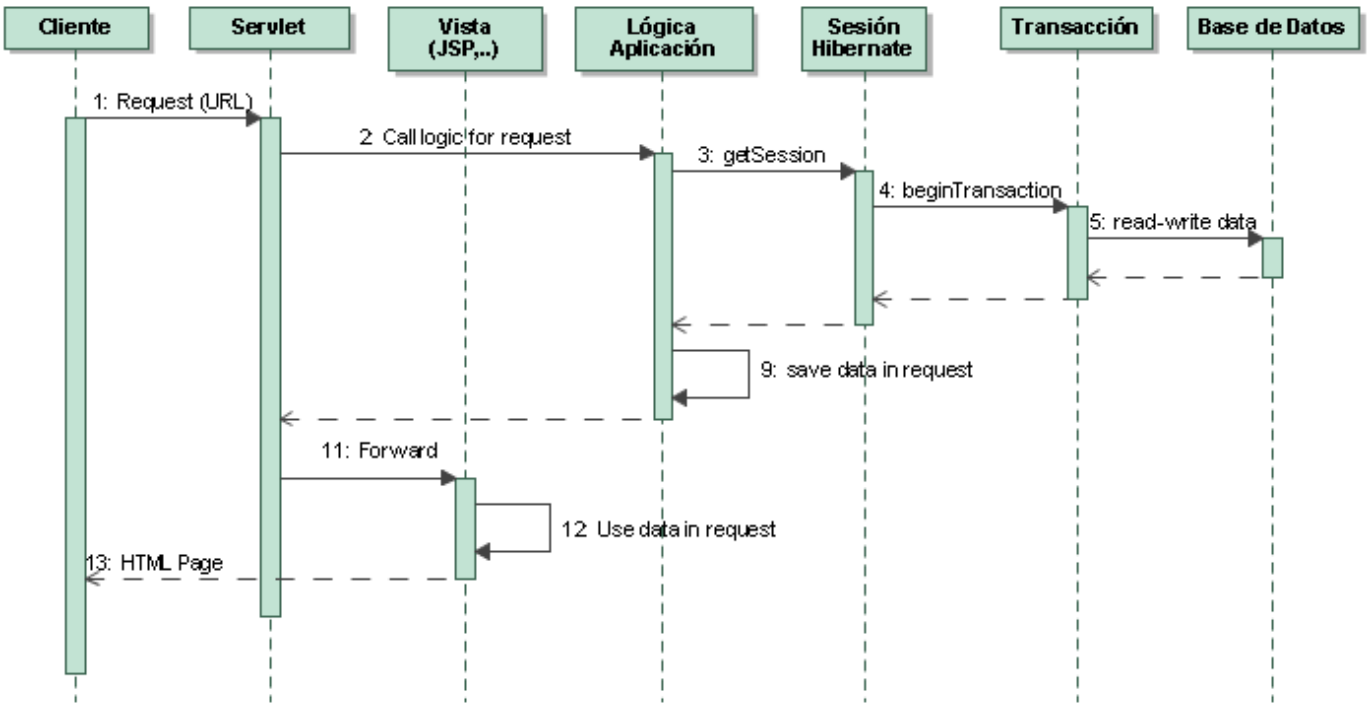


Fig.3.5: *Diagrama de Secuencia de una petición con Struts2*

3.2.5 Principales Componentes

Struts2 está compuesto de los siguientes principales componentes: ***Filter Dispatcher***, ***Interceptores***, ***Action*** y ***Librerías de Acciones***.

a) Filter Dispatcher:

Representa el punto de entrada a la aplicación, dirigiéndose a él todas las peticiones que el cliente. Analiza la petición recibida y determina con el apoyo de otros objetos auxiliares y de la información almacenada en el archivo de configuración struts.xml el tipo de acción a ejecutar.

Filter Dispatcher forma parte del API de Struts 2, concretamente, se incluye dentro del *apache.struts2.dispatcher*, y como bien dice su nombre, es implementado mediante un filtro, por lo que debe ser registrado en el archivo web.xml de la aplicación.

b) Interceptores:

Una vez decidida la acción a ejecutar, *FilterDispatcher* pasa el control de la petición a un objeto intermediario de tipo *Action Proxy*, éste crea un objeto *Action Invocation* en el que guarda la información de la petición y pasa el control de la misma a los interceptores.

Los interceptores son una serie de objetos que realizan tareas de pre-procesamiento antes de ejecutar la acción (validación de datos del usuario, etc.) y después un post-procesamiento de los resultados una vez devueltos por los Action.

Son como filtros Servlet, que se invocan al recibir la petición según un orden indicado en el fichero de configuración struts.xml y en orden inverso cuando se realiza el envío de la respuesta final.

El API de Struts proporciona numerosos interceptores para que seamos nosotros los que decidamos cuáles ejecutar indicándolo en el archivo de configuración struts.xml (también podríamos implementar nuestros propios Interceptores implementando el interfaz *Interceptor*).

c) Action

Los objetos Action forman parte del Modelo, aunque por norma general, la lógica de negocio se suele aislar en clases independientes o EJB, incluyéndose en las clases de acción las llamadas a los métodos expuestos por estos objetos.

Las clases Action no tienen por qué heredar ni implementar ninguna clase o interfaz del API. Son clases estándar (POJOs), cuyo objetivo es proporcionar un método que se llamará *execute(),* donde se llevará a cabo el procesamiento de la acción y será invocado por el último interceptor de la cadena.

Los datos procedentes del formulario cliente también los almacenará el Action, y se deberán implementar los datos miembro para el almacenamiento de los campos con sus correspondientes métodos *set/get,* (de forma similar a como se hacía en los *Action Form* de Struts1).

La pila de valores y el OGNL proporcionan un thread común, enlazando y habilitando la integración entre los otros componentes.

Y además de todos los componentes mencionados, habrá numerosa información relacionada con la configuración, configuración para la aplicación Web además de configuración para los *Actions*, *Interceptors*, *Results*, etc.

d) Librerías de Acciones

Struts2 proporciona un amplio conjunto de acciones o tags que facilitan la creación de las vistas JSP (o en otras tecnologías como Velocity, Freemaker, etc.) para la respuesta al Cliente. Estas acciones se incluyen en la librería de *uri* asociada "/struts-tags.

Además de añadir una gran variedad de componentes gráficos y elementos de control de flujo, esta librería proporciona acciones que nos permitirán acceder directamente desde la vista a la acción que se acaba de ejecutar, facilitándonos de este modo el acceso a los datos generados por el modelo.

3.2.6 Configuración

Para la configuración se contará con el fichero ***struts.xml***. Las aplicaciones Struts2 usarán tal fichero, donde se registrarán y configurarán los distintos componentes de la aplicación.

Aquí se deben registrar los Action, con sus correspondientes reglas de navegación, y los interceptores. Struts2 proporciona además ***herencia de archivos de configuración***, lo que se traduce en poder utilizar una serie de configuraciones por defecto en las aplicaciones sin tener que reescribirlas de nuevo en cada struts.xml particular.

Cada configuración definida dentro de struts.xml se incluye dentro de un ***paquete***, el cual se definirá por el elemento "*package*". Los paquetes nos permitirán agrupar un conjunto de elementos (habitualmente acciones) que compartan una serie de atributos de configuración comunes.

Struts2 proporciona un archivo de configuración por defecto llamado "*struts-default.xml*", en el que se incluye un ***paquete*** de nombre "*struts-default*" con una serie de configuraciones predefinidas disponibles para ser utilizadas en cualquier aplicación. Entre otras cosas, este paquete configura una serie de interceptores predefinidos de Struts 2, por lo que si se quiere disponer de la funcionalidad de los mismos en nuestra aplicación, será conveniente crear paquetes que hereden de éste.

En una aplicación grande con un elevado número de elementos de configuración, nos puede llegar a interesar distribuir estas configuraciones en archivos diferentes. Desde el interior de struts.xml se incluirá una referencia a cada uno de estos archivos mediante el elemento "*include*", el cual indicará archivo a incluir. Con esto se logrará conseguir una ***alta modularidad*** en los ficheros de configuración.

3.3 Tapestry

3.3.1 Introducción

Tapestry es otro de los Frameworks MVC elegidos basados en JAVA, donde se introduce el enfoque de desarrollo basado en componentes y orientados a eventos para el desarrollo de aplicaciones web con Java, frente al clásico desarrollo basado en acciones que emplea por ejemplo Apache Struts.

Se encarga, entre otros, de la validación de las entradas recibidas, del proceso de internacionalización, gestión de estado y persistencia, correspondencia de parámetros de la solicitud y construcción de URLs.

Los controladores que lo forman son clases JAVA. En Frameworks como Struts y Grails se trabaja en términos de URL y parámetros, mientras que en Tapestry en términos de componentes y objetos, métodos y propiedades. Tapestry se encarga de convertir los parámetros de la URL al tipo adecuado y de dejarlo en una propiedad del componente de ese tipo, todo ello de forma transparente al cliente.

Cada Controller se asocia con una vista que adopta la forma de una plantilla XML. Los modelos son también objetos planos JAVA que, como en todo Framework MVC, servirán de conexión entre el controlador y la vista.

A diferencia de los JSP, que permiten la inclusión de código Java, Tapestry organiza las vistas en base a plantillas, las cuales serán altamente flexibles gracias a su estructura XML, que además contienen

una lógica muy escasa. El objetivo de esto es para conseguir aislar al máximo la lógica dejándosela a la función principal y encargándose meramente de mostrar los datos. En parte esta filosofía es interesante, porque obliga al desarrollador a modularizar su aplicación. Consiguiendo reducir la vista, se mostrará muy simplificada y serán mucho más mantenibles.

3.3.2 Principales características

Entre las diferentes características que nos presenta este Framework, se destacarán las siguientes:

- ✓ Es un Framework robusto, basado en estándares.

- ✓ Alta productividad y la reutilización de código que se puede conseguir con los componentes. Son reutilizables abstrayéndonos de sus detalles, sólo conociendo los parámetros que recibe. Dentro de esta abstracción incluimos CSS/Javascript, que los añadirá automáticamente en función de la definición del componente.

- ✓ Modular y contenedor de inversión se control integrado. Cada funcionalidad de Tapestry puede ser modificada o extendida a través de su contenedor de inversión de control, mediante un breve desarrollo en el contenedor. Ese desarrollo no será vía XML sino JAVA, con las ventajas que ello conlleva como la asistencia del compilador y la refactorización del IDE.

- ✓ Libertad en el uso de otros Frameworks. Tapestry no obliga a emplear un determinado Framework de persistencia, de testing, de seguridad u otra cosa por lo que se tendrá libertad de elegir la que más adecuado nos parezca en cada ámbito a resolver.

- ✓ Reporte de Excepciones. Tapestry no sólo no solo nos da la traza de la excepción que se ha producido en la consola del servidor, sino que nos muestra una página con toda la información de la aplicación y de la petición realizada y que ha fallado.

- ✓ Aumento de productividad. El Framework nos permitirá desarrollar cambios en las clases de los componentes, páginas y servicios o a las plantillas de los mismos, sin tener que redesplegar la aplicación en la mayoría de los casos.

- ✓ Alto rendimiento y escalabilidad. Esto se está consiguiendo mediante la eliminación del pool de páginas, mediante sólo una instancia para todas las peticiones de la aplicación, eliminación de espacios en blanco y compresión del contenido devuelto, agregación de Javascripts y CSS para reducir el número de peticiones a realizar al servidor, etc.

- ✓ Se realiza validación de los datos de entrada de un usuario de la aplicación. Además es capaz de presentar los errores al usuario en caso de que alguna haya producido algún error. También soporta la especificación JSR-303.

- ✓ Permite diferentes lenguajes de programación. Se pueden desarrollar componentes, páginas y servicios con otros lenguajes, ya sea Java, Groovy, Scala o cualquier otro que se pueda ejecutar sobre la máquina virtual.

3.3.3 Arquitectura MVC

El Framework Tapestry, al igual que resto de Frameworks mencionados, implementa la arquitectura MVC, siguiendo un modelo basado en componentes y en un patrón de diseño ***FrontController***.

Ya no sólo se centra en separar en capas la parte de la lógica de la presentación, sino incluso a nivel de componente individual, bien sean tablas como formularios. Además se implementan diversos componentes importantes en un contexto de interfaces públicas, con el fin de proporcionar a los desarrolladores una alta flexibilidad en el uso del Framework.

Tapestry nos ofrece una arquitectura extensible y abierta. Se proporcionan implementaciones sencillas de cara a poder añadir extensiones propias o incluso para sobrescribir las ya existentes. Tapestry además proporciona integración con Hibernate, JPA y Spring, contando también con extensiones de otros fabricantes como Lucene y Quartz.

Las aplicaciones basadas en Tapestry estarán básicamente compuestas de páginas, y cada página estará formada de un fichero plantilla y una case JAVA.

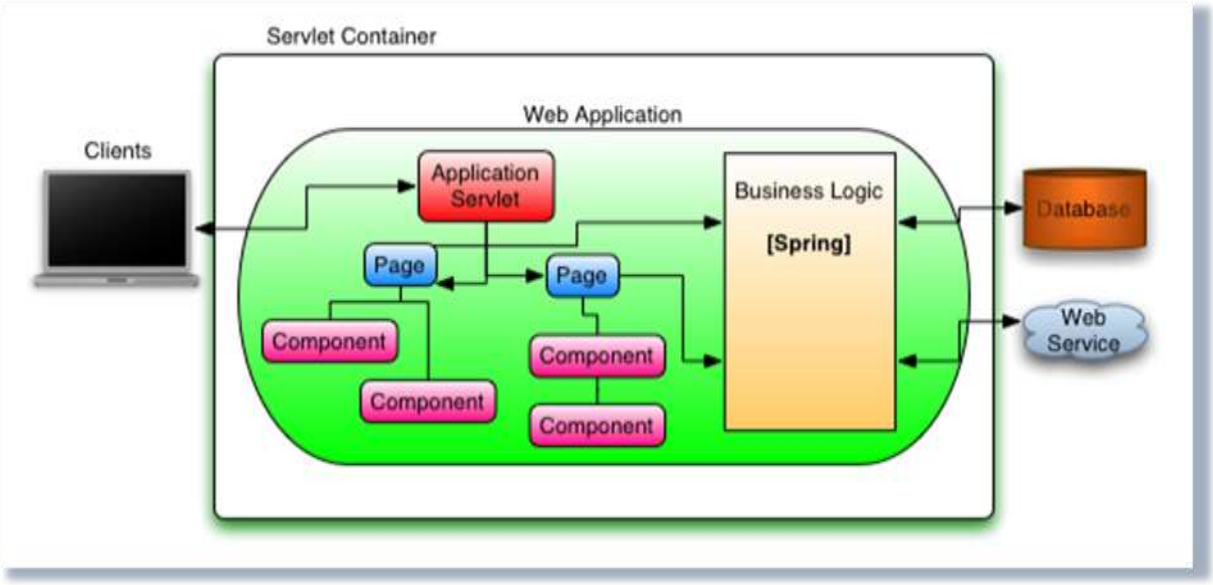


Fig.3.6: *Esquema general de la arquitectura de Tapestry*

3.3.4 Ciclo de Vida y Renderización de un componente

Como ya se ha comentado, la filosofía de Tapestry se basa primordialmente en los componentes, por lo que vamos a hacer un poco más de hincapié.

Tapestry se basa en componentes, como parte de las páginas, y éstos tienen un ***ciclo de vida*** específico:

- Los componentes se instancian y se añaden a las páginas y a los componentes contenedores.
- Una vez que la página está totalmente construida, los componentes reciben una notificación a la vez informando de que la página se ha cargado.
- En la generación y durante la existencia de una request, una página se añadirá a tal request. Se genera una notificación para la página que está siendo agregada y otra notificación posterior cuando la página se desligue (antes de ello, será devuelta al pool de páginas).

En cuanto a la ***renderización*** de un componente, vamos a realizar un breve resumen de los pasos /fases que se van dando a través de tal proceso:

Tapestry parte el proceso de renderización de un componente en diferentes fases. Cada fase corresponde a un método de la clase del componente (posiblemente un método heredado de una clase base). El método se nombre haciéndolo coincidir con el nombre de la fase de renderización.

Normalmente cada fase sigue a su inmediatamente anterior, pero podría haber fases que devolviesen "false", por lo que saltarían hacia atrás.

Los métodos de renderización pueden tener cualquier visibilidad, incluso podrían no tener parámetros. Los componentes por tanto empezarán normalmente en la fase "*BeginRender*" y finalizarán en la "*AfterRender*".

Una fase podría también devolver un componente instancia o un bloque de instancias. Tanto uno como el otro se harán cargo y serán renderizados completamente antes de que el componente actual pase a la siguiente fase.

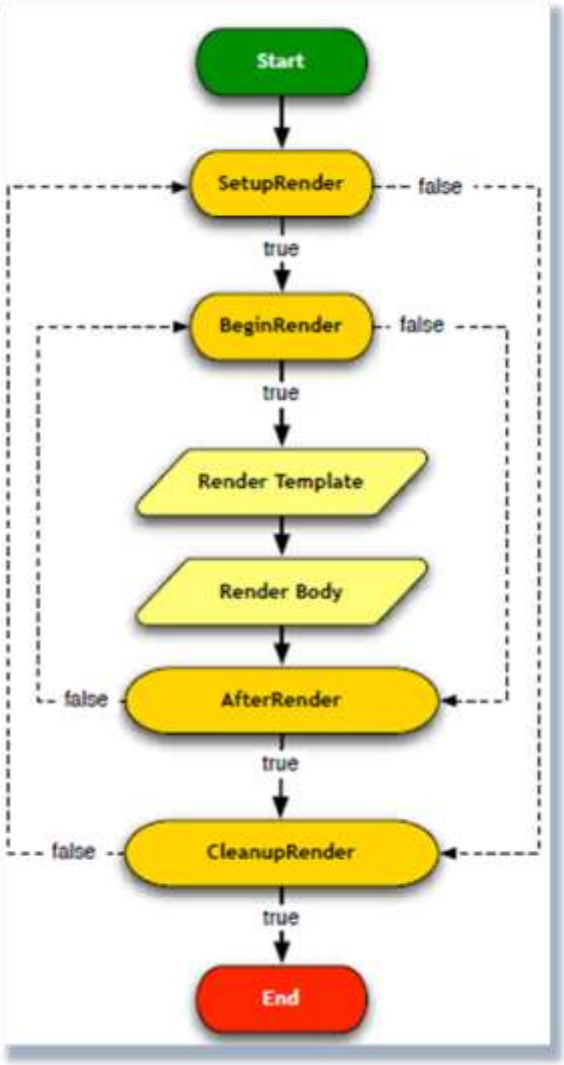


Fig.3.7: *Ciclo de vida de una petición en Tapestry*

3.3.5 Principales Componentes

Del conjunto de engranajes y componentes que forman el Framework de Tapestry, vamos a realizar una enumeración de los más característicos:

a) Application Engine:

Por cada navegador que envía una petición al Framework se crea una instancia del "Application Engine" que se emplea para seguir la actividad del cliente en la aplicación.

b) Application Servlet:

Sirve de puente entre el "Application Engine" y el contenedor de Servlet. Su única función es crear el "Application Engine" tras la primera petición de un cliente a la aplicación y de localizarlo en peticiones futuras.

c) Application Specification:

El fichero de especificación de la aplicación es usado para darle a Tapestry una descripción de la aplicación. En él se especifica el nombre de la aplicación, la clase del "Application Engine", la lista de páginas y nombre de la clase que implementa el Visit Object (si es necesario).

d) Visit Object: Será creado uno por cada una de las conexiones de un cliente a la aplicación. Sirve para almacenar información compartida entre varias páginas de la aplicación.

e) Page Specification:

Cada página posee un fichero de especificación en el que se especifica el nombre de la clase que implementa la página y la lista de componentes.

f) Page Template: Este será el fichero HTML que dará lugar a la representación visual de la página.

g) Page Implementation:

En ella, para cada una los nombres de los parámetros reales, que aparecen en las ligaduras de los parámetros formales a reales en la especificación de los componentes que forman la página, se implementarán métodos *get* y *set*.

3.4 Grails

3.4.1 Introducción

Grails es el tercer Framework de presentación que sometemos a análisis. Éste del mismo modo que los anteriores se dedica al desarrollo de aplicaciones web basado en el lenguaje de programación Groovy, que a su vez se basa todo ello en la Plataforma Java.

Una de las premisas de Grails está basada en los paradigmas convención sobre configuración y DRY (*don't repite yourself*) en las que permite al programador olvidarse en gran parte de los detalles de configuración.

Grails permite al programador olvidarse de gran parte de la configuración típico que incluyen los Frameworks MVC. Además se aprovecha de un lenguaje dinámico como Groovy para acortar los tiempos de desarrollo para invertirlos directamente en la elaboración de código, actualizar, testear y depuración de fallos. Con esto conseguimos que el desarrollo de la aplicación sea mucho más ágil que con otros Frameworks MVC.

Grails además se puede considerar que es una plataforma completa, puesto que incluye también un contenedor web, bases de datos, sistemas de empaquetado de la aplicación y un completo sistema para la realización de tests sobre nuestra aplicación.

De esta forma, no se debe perder el tiempo buscando y descargando un servidor web para nuestra futura aplicación o un gestor de base de datos. Ni tan siquiera será necesario escribir complicados scripts de configuración para el empaquetado de la aplicación. Todo esto se convierte en una tarea tan sencilla como instalar Grails.

3.4.2 Principales Características

De entre las diferentes cualidades que nos ofrece Grails, se destacarán un conjunto, que son las siguientes:

- ***Convención sobre la configuración***:

  En vez de tener que tener que implementar ficheros de configuración en formato XML (como sucede con otros frameworks), Grails se basa en una serie de convenciones para que el desarrollo de la aplicación sea mucho más rápido y productivo sin tener que repetir el trabajo con cada proyecto.

- ***Test***:

  Cada vez que se genera una clase de dominio o controlador en Grails, de forma paralela se genera también un test para comprobar tal clase o controlador. Grails también distingue de controles unitarios (test sin dependencias) y test de integración (tienen acceso al entorno de grails y a la base de datos). También permite la creación de test funcionales.

- ***Scaffolding***:

  Grails permite el uso de scaffolding en las aplicaciones. Esto permite la generación de código automáticamente para las cuatro operaciones básicas de toda aplicación *CRUD* (*create*, *read*, *update* y *delete*). Con Grails se consigue el scaffolding con poco esfuerzo.

- ***Mapeado de objeto relacional:***

  Este Framework proporciona un potente mecanismo para el mapeado objeto-relacional conocido como *GORM* (*Grails Object Relational Mapping*). Como cualquier Framework de persistencia, GORM permite mapear objetos contra bases de datos relacionales y representar relaciones entre dichos objetos del tipo uno-a-uno o uno-a-muchos.

- ***Plugins***:

  Grails dispone de una arquitectura de plugins con una comunidad de usuarios detrás (cada vez más grande) que ofrecen plugins para seguridad, AJAX, testeo, búsqueda, informes y servicios web. Este sistema de plugins hace que añadir complicadas funcionalidades a nuestra aplicación, simplemente configurándola de un modo determinado.

3.4.3 Arquitectura MVC

Como la mayoría de los Frameworks de desarrollo web, Grails está basado en el patrón Modelo Vista Controlador (MVC). En Grails los modelos son tratados como clases de dominio que permiten a la aplicación mostrar los datos en la vista.

A diferencia de en otros Frameworks, las clases de dominio de Grails son automáticamente persistidas y es incluso posible generar el esquema de la base de datos. Los controladores por su parte, permiten gestionar las peticiones a la aplicación y organizar los servicios proporcionados. Por último, la vista por defecto en Grails son las Groovy Server Pages (*GSP*) y habitualmente nos muestra el contenido en formato HTML.

Se ha de tener en cuenta que uno de los principales componentes que emplea internamente Grails es ***Spring.*** Se aprovecha de las ventajas que le da este Framework, en la facilidad de crear componentes reutilizables, además de que se adapta fácilmente con otros Frameworks como Hibernate, iBatis, Struts, etc.

Spring proporciona un nivel de abstracción muy elevado sobre la API de Java EE. Por ejemplo, en vez de tener que tratar con detalles del manejo de transacciones, Spring nos permitirá declarar transacciones a través de POJOs, dejándonos centrar en la lógica del negocio.

Se tienen por tanto las 3 capas de la arquitectura:

- ✓ **Modelo**:
  - El modelo es el responsable de mantener el estado de la aplicación. A veces este estado es transitorio, durando sólo una par de iteraciones con el usuario u otras es permanente almacenándose fuera de la aplicación, normalmente en una base de datos.
  - Hará cumplir todas las reglas de negocio que se vayan a aplicar a los datos. Colocando tales reglas de negocio en el modelo, nos aseguraremos que nada de la aplicación generará datos inválidos.
  - El Modelo actuará tanto como almacén de de datos como de "portero".
  - El *Dispatcher Servlet* se empleará como patrón *Front Controller*, que será en sí un Servlet desarrollado por el Framework apoyado en uno o N patrones *Mapper* para determinar a qué Controller ha de llevarse cada petición del usuario.

✓ **Vista**:

- La vista es la responsable de generar la interfaz de usuario, normalmente basada en los datos del modelo. Mostrará vistas que accedan al modelo y las formateará para el usuario final.
- Aunque la vista muestre diferentes formatos de los datos del modelo, nunca manipulará tales datos.
- Existen muchas vistas que acceden al mismo modelo de datos, normalmente con diferentes propósitos.
- La aplicación en sí podrá implementar varios *Controller* (patrón *Application Controller*) encargados de ejecutar la lógica y de recuperar los datos del modelo para luego entregarlos a la Vista.

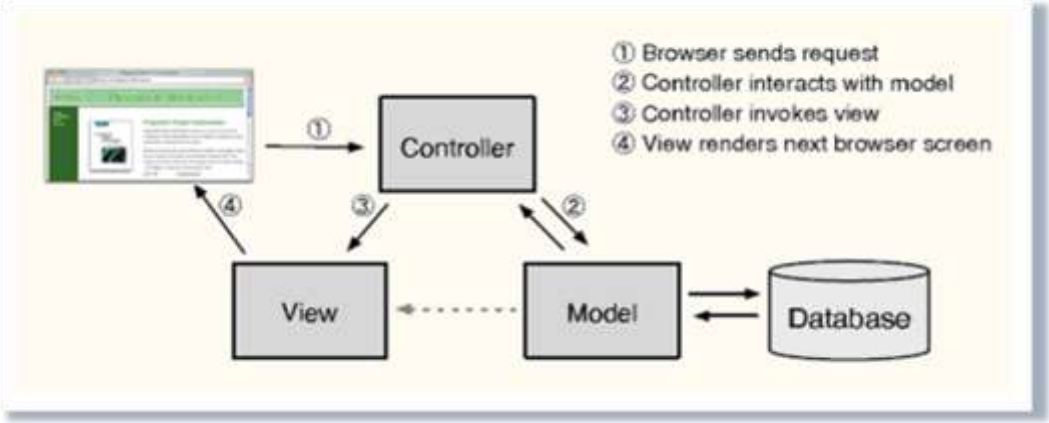


Fig.3.8: *Ciclo de vida de una petición con Grails*

✓ **Controlador**:

- El Controlador orquesta la aplicación. Recibirá la peticiones/eventos desde el exterior, interactuando después con el Modelo y mostrando la vista apropiada al usuario.
- Para decidir a qué vista se deberá entregar el control, el *Dispatcher Servlet* se apoyará en los *View Resolver* que transforman los nombres lógicos de vistas devueltos por el Controller a nombres físicos de recursos.

El funcionamiento es básicamente el triunvirato estándar: Modelo, Vista y los Formularios Controladores de una arquitectura ya bien conocida como MVC:

Y vistas las diferentes capas de la arquitectura y el ciclo de vida de una petición, se mostrará mediante un diagrama de secuencia los pasos por los que transita una request cualquiera al sistema:

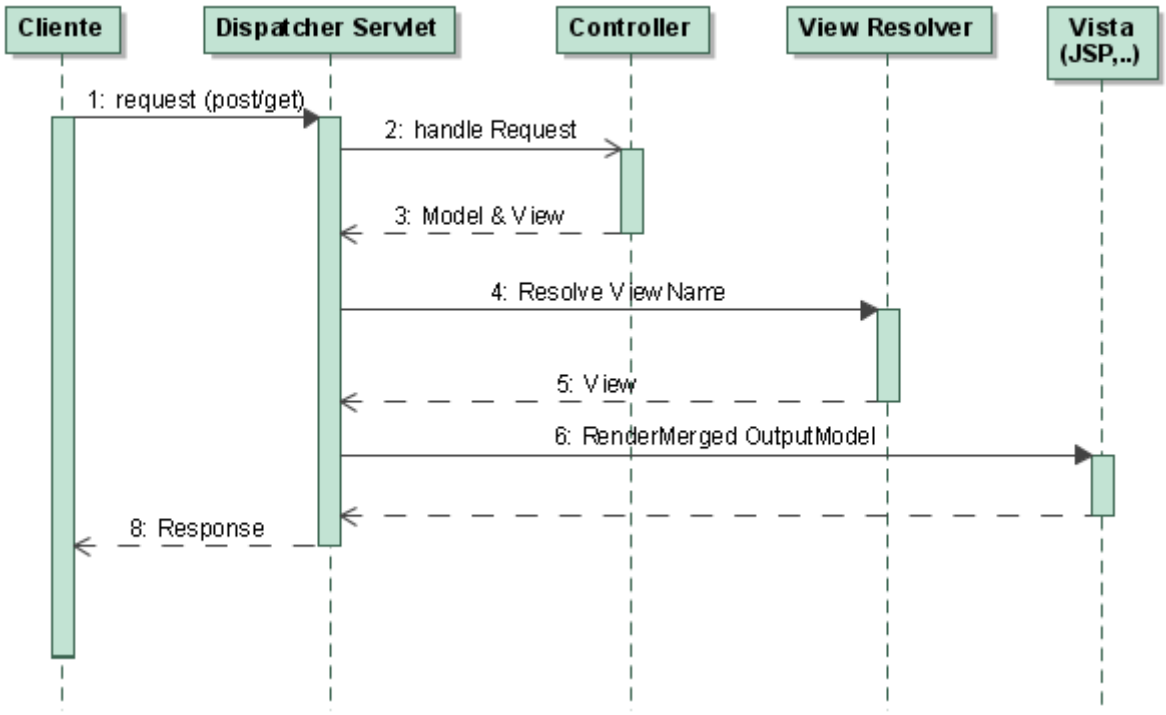


Fig.3.9: *Diagrama de Secuencia de una request con Grails*

3.4.4 Principales patrones empleados

Inversión Of Control (IoC):

Un *patrón* muy importante empleado por Grails será la ***Inversión de Control (IoC)***:

Según la definición de tal patrón, las dependencias de un componente no se gestionarán desde el propio componente, de tal forma que éste sólo contenga la lógica necesaria para hacer lo que debe hacer.

Como bien dice su nombre, con la inversión del control se pretende invertir el flujo de ejecución de un programa con respecto a los métodos de programación tradicionales, en los que la interacción se expresa de forma imperativa haciendo llamadas a los procedimientos o funciones correspondiente.

Normalmente el desarrollador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones. Sin embargo, con este método se especificarán respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

El objetivo de todo esto es por tanto es mantener los componentes lo más sencillos que sea posible, incluyendo sólo código referente a la lógica de negocio, de esta manera se consiguen aplicaciones más fáciles de mantener y reutilizar.

Inyección de Dependencias (DI):

La *Inyección de Dependencias* es un *patrón* de diseño orientado a objetos, derivado del de *Inversión de Control* (*IoC*) que es más genérico, en el que se proporcionan todos los objetos requeridos para una clase en lugar de ser ésta quien tenga que crear tales objetos.

El modo habitual de implementar tal patrón es mediante un contenedor de inyección de dependencias y objetos POJO, donde el contenedor inyecta a cada objeto los objetos necesarios según las relaciones plasmadas en un fichero de configuración.

Estos contenedores serán Spring, Grails, etc. y siempre externos a la aplicación, por lo que, tanto la inyección de dependencias como la inversión de control serán características a emplear por la aplicación, sin formar parte de su lógica.

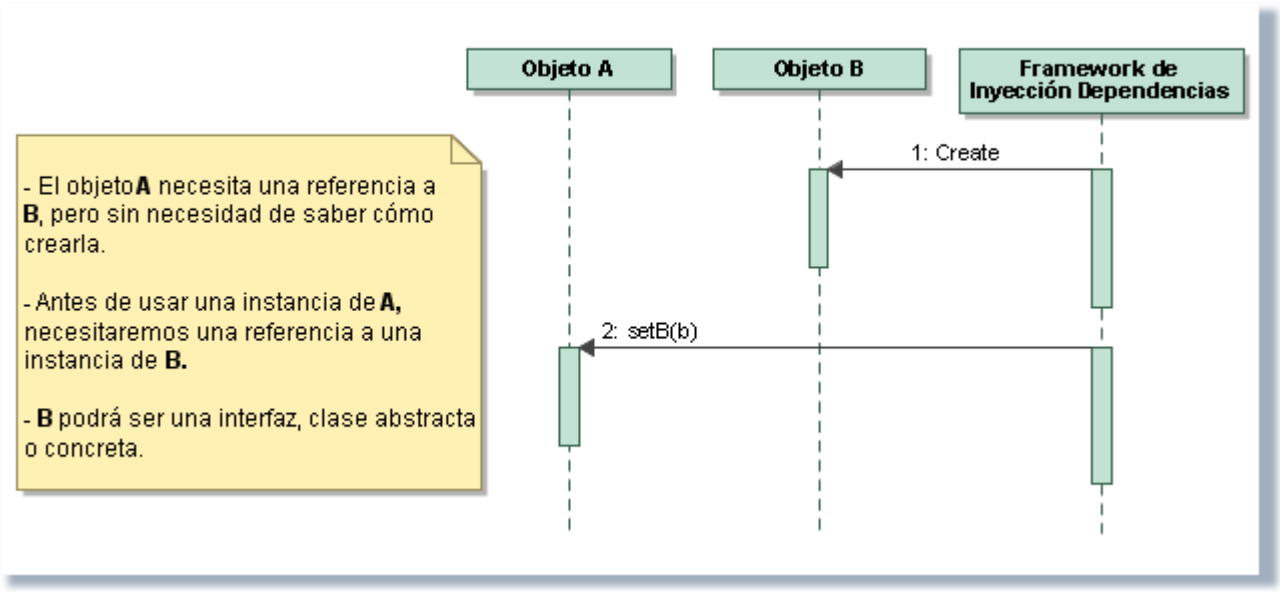


Fig.3.10: *Diagrama se Secuencias de una Inyección de Dependencias*

¿Cuándo podríamos emplear la inyección de dependencias?

No se aplicará siempre la *Inyección de Dependencias* cuando una clase dependa de otra, sino más bien en determinados casos como los siguientes:

- Para inyectar información de configuración en un componente.
- Cuando se requiera alguno de los servicios proporcionados por un contenedor.
- Para inyectar una misma dependencia en varios componentes.
- Para inyectar diferentes implementaciones de una misma dependencia.
- Para inyectar la misma implementación en varias configuraciones.

La *IoC* por ejemplo no es necesaria si uno va a usa siempre la misma implementación de una dependencia o la misma configuración, ya que en estos casos no dará grandes ventajas.

Patrón Service Locator:

El patrón de diseño J2EE *Service Locator* es un componente que contiene referencias a los servicios y encapsula la lógica que los localiza dichos servicios abstrayéndola del cliente. De tal forma que, en nuestras clases, utilizamos tal patrón para recuperar instancias de servicios que se necesiten para nuestra lógica de negocio.

Service Locator no se encarga de instanciar servicios. Proporciona una manera de registrar servicios y mantener una referencia a los mismos. Una vez registrado un determinado servicio, el *Service Locator* podrá localizarlo.

Este patrón además proporciona formas de localizar cualquier servicio sin especificar el tipo. Por ejemplo, puede usar una clave de tipo cadena o directamente el tipo del interfaz. Esto permite un fácil reemplazo de la dependencia sin modificar el código fuente de la clase.

Por tanto se ve la principal funcionalidad del patrón *Service Locator*, por lo que queda en el tejado de la *Inversión del Control* (IoC) el determinar la forma en que se localizará el servicio requerido.

Service Locator frente a Inyección de Dependencias:

Tanto una solución como otra proporcionan un gran desacoplamiento ya que en ambos casos el código de la aplicación es independiente de la implementación concreta del interface del servicio. La

diferencia más importante entre los dos patrones es la forma de proporcionar la implementación a la clase de aplicación.

Con *Service Locator*, la clase de aplicación enviará un mensaje explícito al *Locator*, sin embargo, con inyección no hay petición explícita, el servicio ya aparece en la clase de aplicación gracias a la inversión de control.

Con un Service Locator cada usuario de un servicio tiene una dependencia del *Locator*. El locator puede ocultar dependencias de otras implementaciones, y nosotros no necesitaremos ver el locator.

Usar la *Inyección de Dependencias* puede ayudar a hacer más fácil de ver donde están las dependencias de componentes, mirando al mecanismo de inyección y sus dependencias., mientras que con el *Service Locator* se tendrá que buscar en el código fuente las llamadas a tal locator.

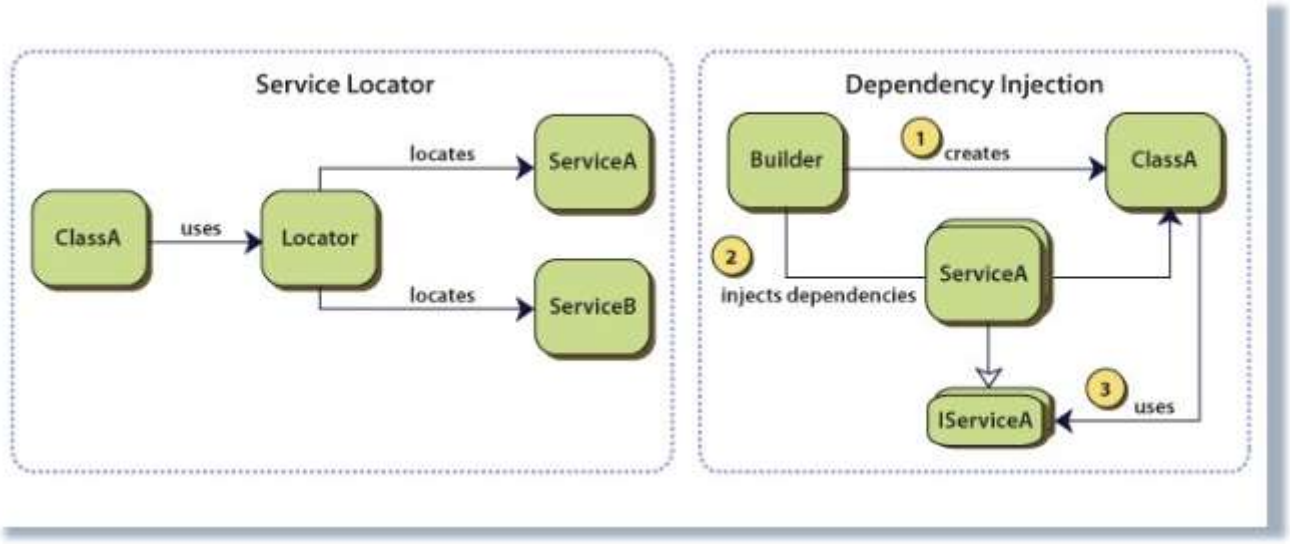


Fig.3.11: *Diferencias entre el Patrón Service Locator y la Inyección de Dependencias*

3.4.5 Principales Componentes

De entre los diferentes elementos de que se compone el Framework de Grails, se destacarán los siguientes:

a) ***Spring:***

Empleará el Framework de Spring para los flujos de trabajo e inyección de dependencias. De ahí que implementará el patrón MVC partiendo de tal tecnología.

b) *Groovy:*

Se empleará tal lenguaje para la creación de propiedades y métodos dinámicos en los objetos de la aplicación. Con Groovy necesitamos menos código para obtener el mismo resultado que si lo hiciéramos con Java tradicional. El hecho de tener menos código supondrá menos fallos y más facilidad de mantenimiento.

c) *Hibernate* para la persistencia de base de datos. Es el estándar por defecto para ORM.

d) *Ant* para la gestión del proceso de desarrollo.

e) *SiteMesh*: Es un Framework robusto y estable para la composición de la vista y renderizar layouts.

Por ejemplo, para el caso de Sitemesh se emplea el **patrón de diseño *Decorator***:

Con el *Patrón Decorator* se responde a la necesidad de añadir dinámicamente funcionalidad a un Objeto.

Por lo tanto se empleará cuando queramos extender una necesidad pero no hay razones para extenderlo a través de la herencia.

Con esto se conseguirá no tener que crear sucesivas clases que hereden de la primera incorporando la nueva funcionalidad, sino otras que la implementan y se asocian a esta.

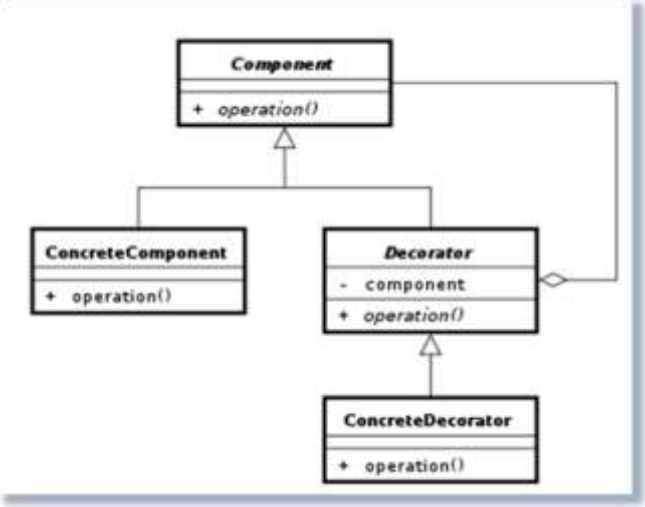


Fig.3.12: *Arquitectura de Patrón Decorator*

A continuación se mostrará un esquema muy general de la arquitectura de Grails y de los diferentes componentes/Frameworks de los que se sirve y otros en los que podría llegar a apoyarse:

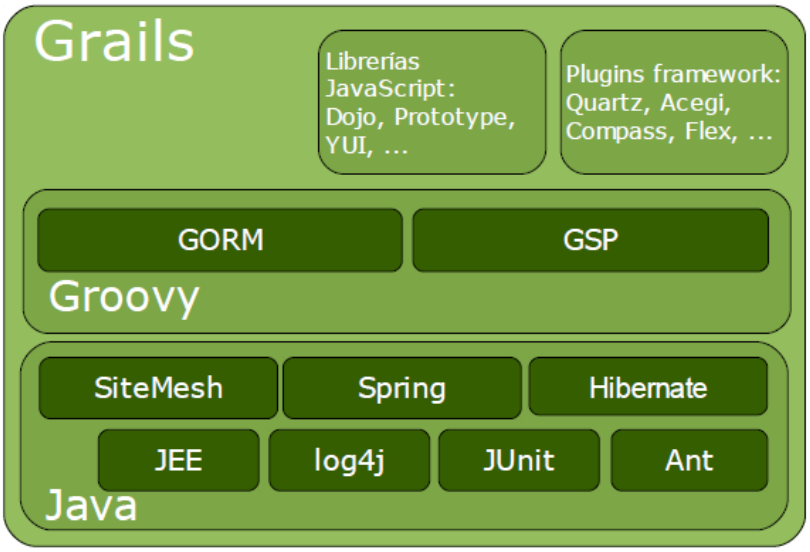


Fig.3.13: *Esquema de componentes Arquitectura Grails*

3.5 Comparativa de los Frameworks estudiados

Una vez estudiadas todas las características de los diferentes Frameworks por separado, se realizará además una comparativa de las posibles diferencias, ventajas e inconvenientes del uso de unos con respecto a los otros.

Dichas diferencias se muestran en la siguiente tabla:

	Struts2	***Tapestry***	***Grails***
Basado en...	Framework J2EE (WebWork)	Framework J2EE	Framework J2EE
Lenguajes de Programación	Java	Java, Groovy, Scala	Java, Groovy
Enfoque orientado a...	Acciones, URLs	Componentes	Acciones y Componentes
Patrón de Diseño	Inyección Dependencias	MVC, Inyección Dependencias	MVC, Inyección Dependencias
Configuración	XMLs y anotaciones Java	Convention over Configuration (anotaciones Java)	Convention over Configuration (ficheros groovy.properties)
Trabaja en términos de...	URLs y parámetros	Componentes y Objetos	URLs y parámetros
¿Permite inyección de Dependencias?	Si	Si	Si
Separación Present./Lógica	Servlets/JSPs pueden contener HTML	Total separación Present./lógica usando plantillas	Total separación Present./lógica usando plantillas

Tipo Controlador	Servlet Filter	Servlet	Servlet
Modelo		Objetos planos Java	
Tipo de Vista	Ficheros JSP, Velocity, FreeMarker, XSLT, etc.	En base a plantillas, flexibles gracias a su estruct. XML	Ficheros GSP
Arquitectura de Plugins, permite Plugins	Medio. Permite, añadiéndolos al ClassPath sin configuración previa	Medio. Al igual que Struts2, puede gestionarlos pero no destacan en este ámbito	Alto. Posee un gestor/wizard y numerosos plugins
¿Libertad de uso Framework Persistencia?	Si	Si	Si
Soporte Validación	Soporte del lado Cliente y Servidor	Soporte del lado Cliente y Servidor	Soporte del lado Cliente y Servidor
Gestión/Compartición entre proyectos de componentes UI	Bajo (basados en request)	Alto (basados en componentes)	Medio
Autenticación	Interceptor	Spring Security	Plugin de Spring Security
Testing "built-in"	Si	Si	Si
Soporte Internacionalización/ Localización (i18n, l10n)	Resource Bundle con Taglibs (soporte internac. de Java estándar)	Ficheros properties (soporte internac. de Java estándar)	Ficheros properties (soporte internac. de Java estándar)

Ventajas	Arquitectura sencilla y fácilmente extensible	Framework muy productivo una vez que se aprende.	Facilidad creación de controladores y componentes. Scaffolding proporciona gran potencia
	La librería de etiquetas se pueden personalizar con FreeMarker o Velocity	Gestión eficiente de recursos. Gran optimización en cuanto a CPU y Memoria.	Plugins para todo tipo de aplicaciones
	Se puede definir la navegación basada en Controladores o basada en páginas.	Plantillas son HTML, productivo trabajando con diseñadores gráficos.	Buena y creciente documentación y comunidad activa
	Integración fácil con Spring, herramientas Debugging y resultados especializados.	Cada nueva versión aporta gran cantidad de innovaciones.	DRY (Don´t Repeat Yourself). Permite alta reutilización de código

Inconvenientes	Documentación mal organizada.	Documentación más conceptual que pragmática.	Tareas de debugueo y trazas de error pobres
	Concentración excesiva en las nuevas funcionalidades.	Curva de aprendizaje alta.	Requiere buen conocimiento de Hibernate, Spring de cara a desarrollo efectivo.
	No proporciona componentes de interfaces usuario ni modelo de gestión de eventos para crear interfaces ricas.	Los ciclos definidos para lanzar nuevas versiones son muy largos.	Escalabilidad media (depende en cierta medida del estado de sesión)

Url Oficial	http://struts.apache.org/2.x/	http://tapestry.apache.org	http://grails.org/

Fig.3.14: *Tabla Compartiva de los Frameworks*

4. Framework Propuesto: "FCUOC"

4.1 Características generales y Requisitos

Para modelar el Framework que de soporte a la capa de presentación, se ha basado en una arquitectura en tres capas, siguiendo el patrón arquitectural: Modelo Vista Controlador (MVC) y en concreto el Model2:

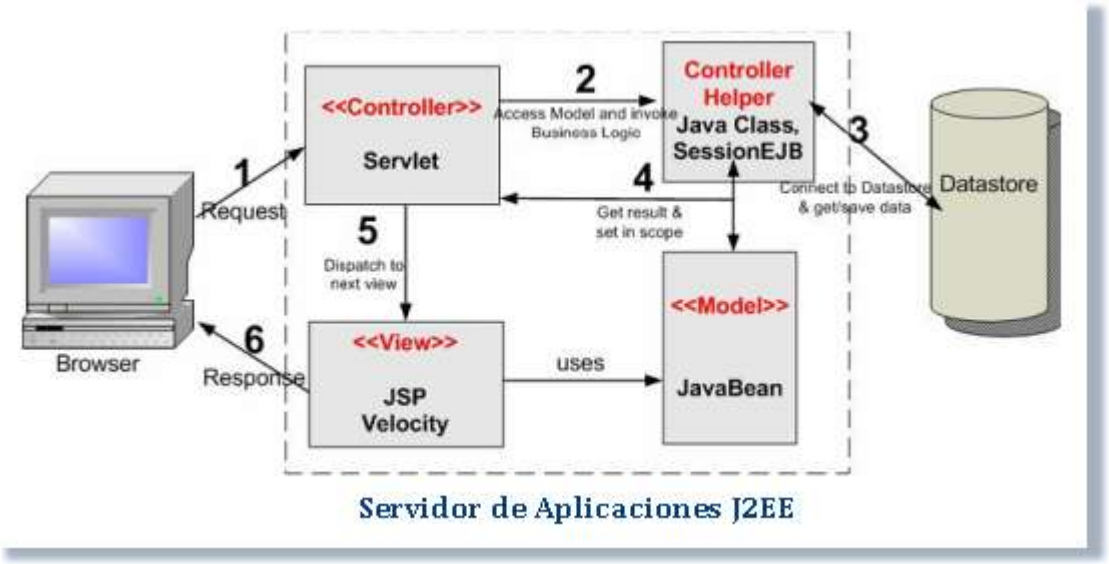


Fig.4.1: *Arquitectura MVC Model-2*

En esta arquitectura propuesta, tanto los Servlets como la Vista (JSP) interaccionan para modelar la capa de presentación de una aplicación web. El Servlet Controller (controlador) se encuentra entre las peticiones de los clientes (peticiones que se efectuarán mediante un navegador) y entre las vistas (JSP) encargadas de mostrar la salida de datos.

El principal objetivo del Servlet es gestionar la lógica de selección de la Vista (view) en función de los parámetros que ha recibido por parte del cliente, también del resultado de la invocación a la lógica de negocio, incluso el estado de la aplicación.
También lleva a cabo la invocación a la lógica de negocio correspondiente en función de la petición que le haya sido efectuada.

Todas estas operaciones se implementarán en el Framework, creando así un producto válido, y dejando puertas abiertas de cara a posibles evoluciones en diferentes aspectos (se detallarán posibles evoluciones más adelante), para que los desarrolladores que la utilicen (para la implementación de su aplicación) puedan evolucionarla.

Esta arquitectura MVC model-2 que se seguirá en el Framework, estará modelada con patrones Core J2EE aplicados para la capa de presentación). Estos patrones aportarán diferentes líneas de soluciones a los problemas que se den en el contexto de la capa de presentación. El motivo de apoyarnos en patrones, es como ya se ha mencionado anteriormente, porque nos ofrecen soluciones probadas, que han sido usadas con anterioridad de manera repetida, garantizando su funcionamiento.

Tales patrones serán reutilizables correspondiendo a problemas que no son específicos de un caso Concreto (se presentarán repetidamente en distintas aplicaciones) y sobre todo son expresivos, permitiendo que cualquier desarrollador con conocimientos, entienda de una manera fluida y precisa la arquitectura ideada para el Framework.

Una vez analizados los tres Frameworks, nos vamos a decantar por los orientados a ***acciones***, como son el caso de *Struts2* y *Grails*, en contraposición de *Tapestry* el cual se implementa orientándose a ***componentes***.

El hecho de orientarse a acciones nos proporcionará una serie de características, de entre las que se pueden destacar fundamentalmente las tres siguientes:

- ✓ Una Acción para cada tipo de interacción con el usuario.
- ✓ Una Vista para cada Acción (normalmente).
- ✓ Mapeo directo de URLs con acciones y vistas (*http://server/controller/action/view*)

La misión fundamental que cumplen los componentes de los que está constituida una aplicación web es gestionar las interacciones con el usuario. Esta gestión se puede descomponer en los siguientes pasos:

1. Procesar la request http.
2. Invocar los componentes de servicio asociados a esa request.

3. Generar el código HTML necesario para la página web que contenga los resultados dinámicos.

Estos pasos se corresponden con el patrón MVC, mencionado en la introducción. El controlador maneja la request, el modelo representa los servicios y, la View dibujará el contenido dinámico en forma de páginas web. Este patrón se encontrará en el núcleo del presente Framework. Además permitirá la separación de la presentación, el control de flujo y la lógica de negocio presente en el gestor.

El aislamiento de estas tres capas se logra aplicando MVC, el cual, como ya se ha explicado, en J2EE se llevará a cabo mediante un controlador central implementado mediante un Servlet.

Para la implementación de tal Sistema ser hará uso de componentes todos ellos compatibles con la especificación J2EE. Se usará una máquina virtual Java versión 1.6.37, junto con un Servidor de Aplicaciones JBoss 6.1.0, atacando contra una Base de Datos MySQL versión 5.5.28 y empleando, entre otras herramientas Ant (versión 1.8.4) para la compilación y generación de los entregables finales.

Con todo ello se obtendrá por un lado la librería JAR con nombre *FUOC-Framework-1.0*, y por otro la aplicación WAR con nombre *ClubCiclistaUOC*, que incluirá tal Framework, entre otras librerías.

4.2 Análisis

4.2.1 Servicios Implementados

Usando como base algunas de las características de Struts, nuestro Framework presentará una serie de servicios y funcionalidades, en donde se destacarán:

- ***Control del flujo de forma declarativa:***

El Framework proporcionará un flujo de navegación de forma declarativa, de tal forma que se especificarán y se definirán las acciones que se van a ejecutar en función de las peticiones

recibidas, se asociarán con los objetos de validación correspondientes mediante reglas específicas y se configurarán las diferentes vistas por las que irá transitando la aplicación.

- ***Servicio de Validación de datos:***

Se proporcionarán mecanismos de validación y conversión de todos aquellos datos que lleguen a la aplicación, provenientes de las diferentes peticiones del cliente, notificando a éste de todas aquellas incidencias o errores que se pudiesen dar.

En primer lugar, se instanciarán diferentes tipos de Formularios en función de los parámetros de entrada que se encuentren, de este modo siempre estarán controlada toda la información de entrada al Sistema. Por otro lado, se implementarán validaciones mediante JAVA que aseguren el correcto estado de la información que se introduce.

En cuanto al ámbito de las validaciones, se pueden realizar de dos tipos:

a) *Validación de los datos en sí:*
Se encarga del tratamiento de la entrada correcta de tipo de datos, en la que no influye el estado de la aplicación. En estas validaciones se tiene en cuenta si un campo es obligatorio, si cumple un determinado patrón (Email, DNI, código postal), etc.

b) *Validación funcional:*
Este tipo de validaciones ya dependen de los datos de negocio y del estado de la aplicación, no depende ni se validará con nuestro Framework.

Paralelamente a las validaciones del lado del Servidor que proporcionará el Framework, se realizarán en la posterior Aplicación del lado del Cliente, empleando librerías Javascript (JQuery).

- ***Servicio de Internacionalización (multiidioma):***

Con el objetivo de poder mostrar al usuario la aplicación en un determinado idioma personalizado, el sistema contará con mecanismos que parametricen los diferentes literales/mensajes/rótulos de la aplicación extrayéndolos de las vistas y almacenándolos en ficheros de configuración. De este modo será más fácil parametrizar un determinado idioma (o

nuevos que se pudiesen añadir) sin ampliar necesariamente la lógica de la aplicación. Los idiomas implementados son: *Castellano*, *Catalán* e *Inglés*.

La aplicación la podríamos internacionalizar de dos modos diferentes:

a) *Duplicando* las diferentes *páginas* por cada idioma existente (aunque muy usada, es poco recomendable por el coste de implementación y mantenimiento de tantas páginas).

b) Mediante la *internacionalización de Java* y extensiones (recomendable y será la que se use en nuestro caso, donde se podrán parametrizar todos los mensajes, textos, etc. añadiendo nuevos idiomas sin que el código sufra cambios).

- ***Gestión de excepciones/errores:***

Se implementarán mecanismos de gestión de excepciones (de forma declarativa) que nos proporcionen un modo de tratar los posibles errores que se produjesen al ejecutarse las diferentes acciones de la capa de negocio. Se generarán por tanto mensajes de error tanto de log que recoja la consola, como para la vista al usuario, informando del error cometido.
Además se dará la posibilidad de inicio de poder indicar la activación del "*modo verbose*", para generar o no logs (info/error) a deseo del cliente.

- ***Sesiones:***

Se prestará también atención a la gestión de las sesiones por parte del Framework. Todas las acciones que se definan para el sistema tendrán asociado un "scope" bien sea de sesión como request, que el Framework almacenará de una forma diferente para cada caso. Por tanto, se encontrarán acciones con un ámbito request (acciones comunes, como insertar, recuperar datos, etc.) y otras donde los datos se almacenarán en la sesión durante toda la conexión del usuario (operaciones de login y logout).

- ***Navegación:***

En la aplicación se podrá observar dos tipos de "navegación" y que además vendrán definidos en el fichero de Properties para cada acción, una "navegación estática" (*Navigation Action*) en donde no se realizará ningún procesado y se redirigirá a la View indicada y por otro lado "navegación

dinámica" donde además de redirigir a la View se realizará un procesado con la información recibida por el cliente o recuperada de la base de datos.

4.3 Arquitectura

Nuestro Framework estará basado en la arquitectura Modelo Vista Controlador como ya se ha ido explicando en los puntos anteriores. En los análisis que se han realizado de los diversos Frameworks del mercado (*Struts2*, *Tapestry* y *Grails*) se han visto que unos siguen un modelo basado en componentes y otro en acciones, pues bien, para el caso que ocupa, se empleará un modelo basado en acciones.

En los próximos apartados se comentarán las estrategias más relevantes que se han empleado en lo que a patrones de diseño se refiere, siendo los más importantes el patrón Front Controller y el Application Controller, entre otros.

4.3.1 Patrones empleados

Para la implementación de nuestro Framework se han analizado los diferentes patrones de diseño software presentes (patrones core J2EE y patrones de diseño GoF –*Gang of Four*-) y se ha tratado de hacer uso en la medida de lo posible éstos.

Los patrones de diseño nos proporcionan la experiencia pasada de otros diseñadores, con el consecuente incremento de velocidad en el proceso de diseño y manteniendo siempre unos niveles aceptables de calidad.

La aplicación de patrones de diseño nos proporciona un vocabulario que agilizará las discusiones acerca de diferentes aspectos del software y, aunque pueden aumentar la complejidad de éste, normalmente se aceptan ya que su empleo en el desarrollo mejora a futuro la flexibilidad y la calidad del producto.
Diseñar con patrones no consiste en aplicar todos los patrones que se conocen para nuestra arquitectura, sino más bien, en resolver cada problema de diseño con el patrón apropiado. El uso

excesivo de patrones nos conducirá inevitablemente a una arquitectura excesivamente complicada.

Cada patrón ofrece una ventaja y una serie de características que afectan positivamente al rendimiento y al proceso de cambio del software. El patrón arquitectónico y mixto conocido como MVC (Model–View-Controller) constituirá el patrón central de este proyecto y se basará fundamentalmente en él.

Para el presente Framework se van a analizar y desarrollar los siguientes tipos de patrones:

- ✓ ***Patrón Context Object:***

Empleado en el *CreadorContextos* (ContextFactory), para compartir los parámetros presentes en la petición Http haciendo que los detalles estructurales de ésta resulten transparentes en otros contextos dentro de la capa de presentación.

Tras realizar una petición por parte del cliente, nuestro controlador pide la instanciación de un objeto *ContextObject* y le asigna una sesión. También determina de forma declarativa, qué instancia concreta (acción) deberá solicitar basado en el fichero de configuración y mapeos que tenga establecidos.

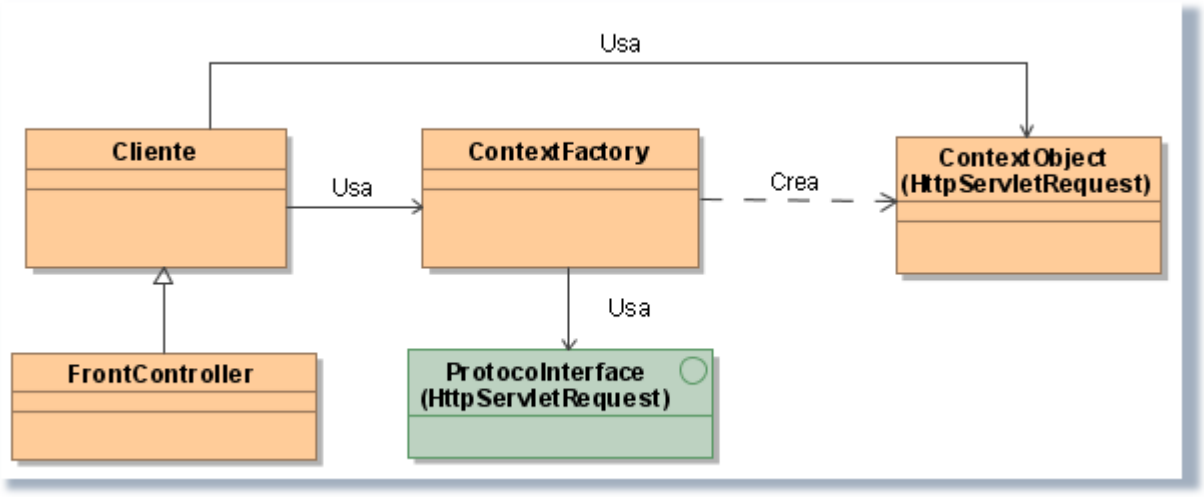


Fig.4.2: *Arquitectura del patrón Context Factory*

Conocidos los elementos que forman parte de este patrón, se mostrará un diagrama de las diferentes peticiones que se llevarán a cabo:

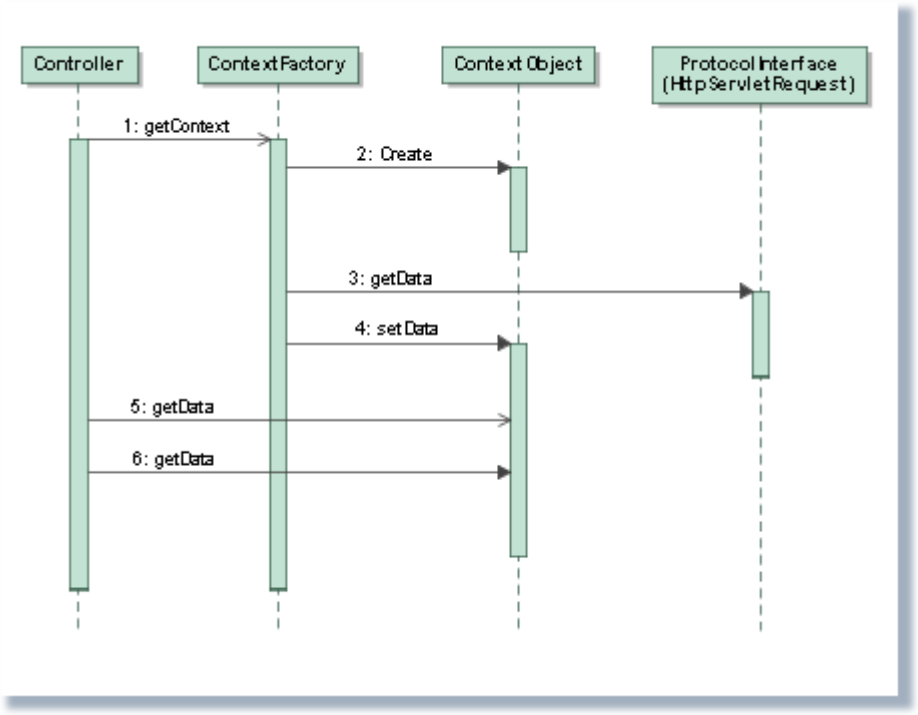


Fig.4.3: *Diagrama Secuencias Patrón Context Object*

- ✓ ***Patrón Singleton:***

Se contará con un componente que será un *Configurador* y se encarga de procesar y guardar todas las configuraciones de nuestro Framework. Esta carga se realizará una única vez al arrancar la aplicación, por lo que podríamos hacer uso de tal patrón (patrón "*Creacional*" GoF). Tal patrón nos asegura una clase única con una única instancia, proporcionando un punto de acceso global a ella:

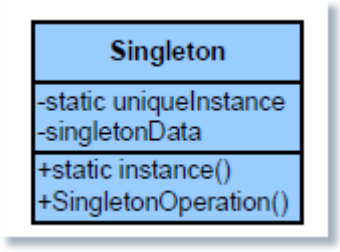


Fig.4.4: *Arquitectura Patrón Singleton*

Entre otros casos, por ejemplo se aplicará tal patrón al *Application Controller*, de tal forma que se instanciará una única vez y toda la aplicación hará uso de ella de forma unívoca. Otro caso son

también los diferentes servicios declarados (*UsuarioService*, etc.) que se obtendrán instancias Singleton para las diferentes acciones.

✓ *Patrón Command:*

Empleado dentro del core del Framework para implementar las '*Actions*', en concreto la clase abstracta *AbstractAction*, de la cual parten todas las acciones que se realizan en la aplicación.

Define métodos abstractos (Ejecutar, Validar) que ejecutará el *Application Controller*, instanciando la subclase apropiada de ésta en función de la request recibida (a su vez el método 'Ejecutar' será el encargado de comunicarse con la capa de servicios para obtener posteriormente los datos para la Vista).

El esquema que se seguirá de cara a la implementación seguirá las premisas de tal patrón tal que así:

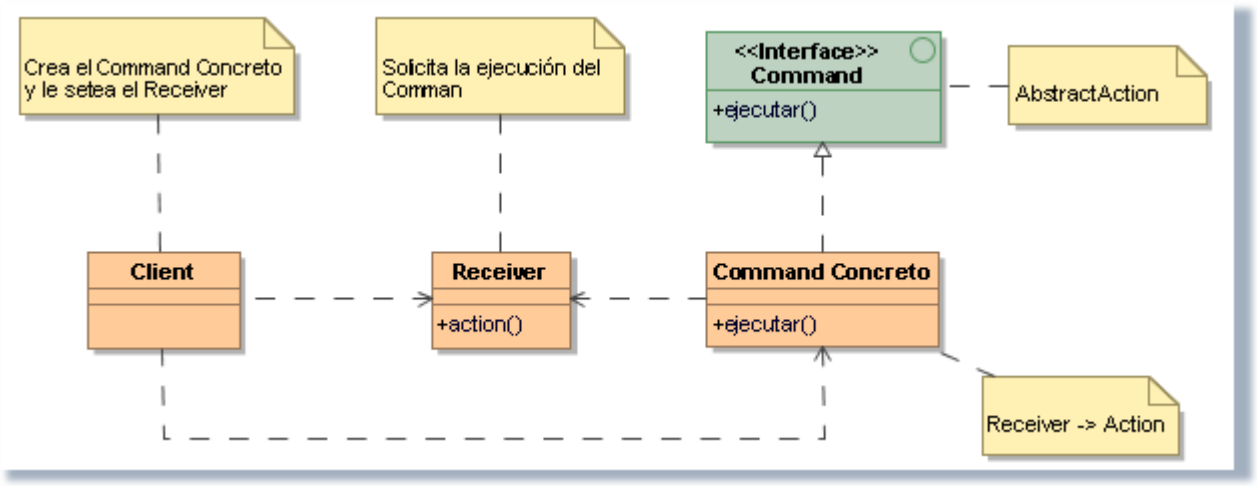


Fig.4.5: *Arquitectura Patrón Command*

✓ *Patrón Front Controller:*

La misión fundamental del patrón *Front Controller* es ser el encargado de manejar las peticiones, delegando al patrón *Application Controller* el manejo de las acciones y las vistas.

Recibe las peticiones del cliente y se encarga de dirigir el curso de tales peticiones (*request*). Es el punto inicial de contacto para el manejo de todas las request, donde el control está centralizado, no duplicado. Centralizando la lógica de control, el *Front Controller* ayuda a reducir la cantidad de lógica incluida directamente en las *Views*.

A la hora de implementar el controlador, se usará la estrategia llamada "*Servlet Front Estrategy*", donde el controlador será implementado como un Servlet y mediante una determinada configuración del fichero descriptor (*web.xml*) se podrá redirigir las diferentes peticiones hacia el Servlet controlador.

Es conveniente y muy importante el eliminar la gestión de comandos y de *Views* de la parte del *Front Controller*, reduciéndolo a las operaciones de manejo de protocolo y transformación de contexto, y aquellos que tengan que ver con comandos y vistas se las pasará al *Application Controller* (que se estudiará a continuación).

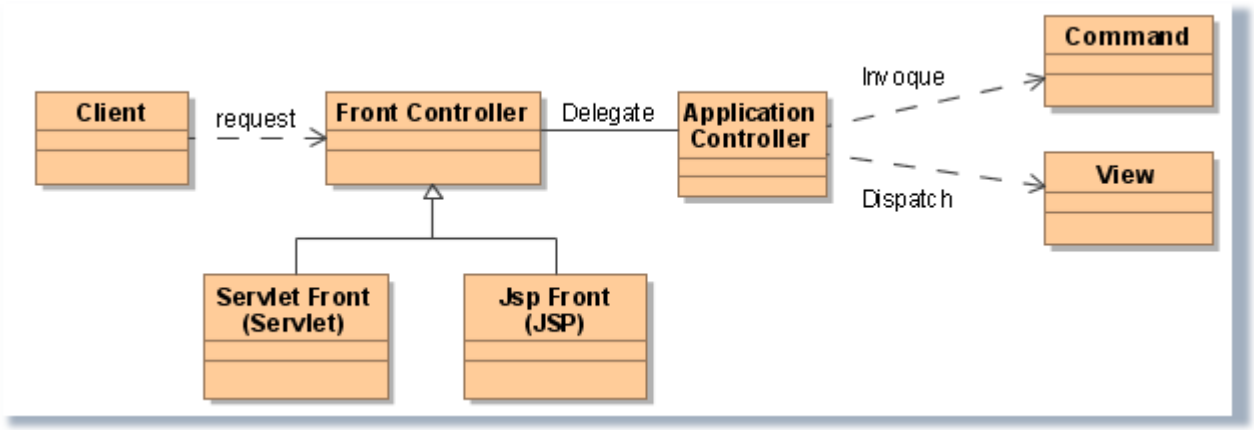


Fig.4.6: *Arquitectura Patrón Front Controller con estrategia "ServletFront"*

- ✓ ***Patrón Application Controller:***

Es en sí el encargado del manejo de las acciones y vistas. Su misión es elegir la acción apropiada y su vista correspondiente para satisfacer una determinada petición. Para los casos más simples, este comportamiento podría incluso estar incluido en el propio *Front Controller.*

El *Application Controller*, por tanto, hará el papel de *Dispatcher* y se usará con una estrategia de "*Command Handler*" (Dispatcher), donde, las tareas que realizará, serán la resolución de las peticiones entrantes al comando apropiado, la posterior invocación del comando y finalmente la entrega del control a la vista apropiada.

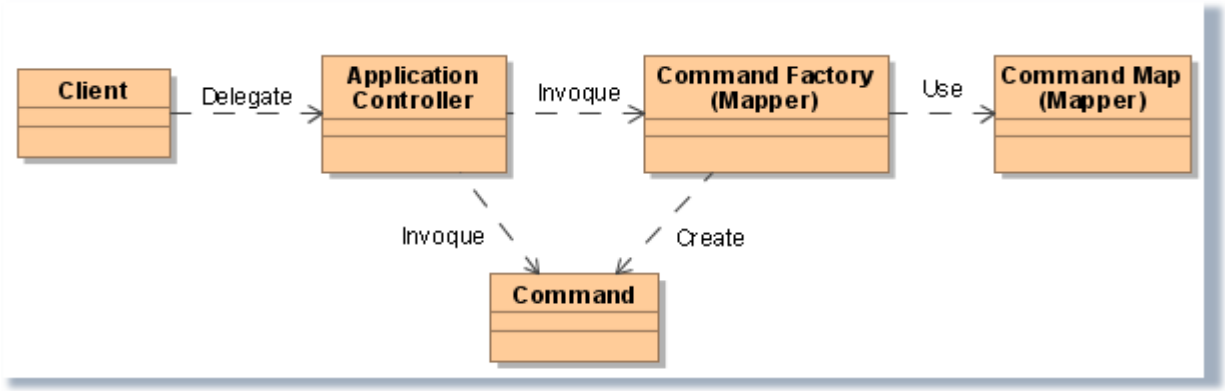


Fig.4.7: *Arquitectura patrón Application Controller con estrategia "Command Handler"*

Nuestro binomio *Servlet Controller* (compuesto por el *Front Controller* + *Application Controller*) se encargará en resumen de las tareas siguientes:

Cargar configuración de inicio, crear los datos de contexto, mapear el modelo, funciones de autorización, validación de datos de entrada, ejecución del modelo, mapeo de la vista y por último, de la ejecución de la misma.

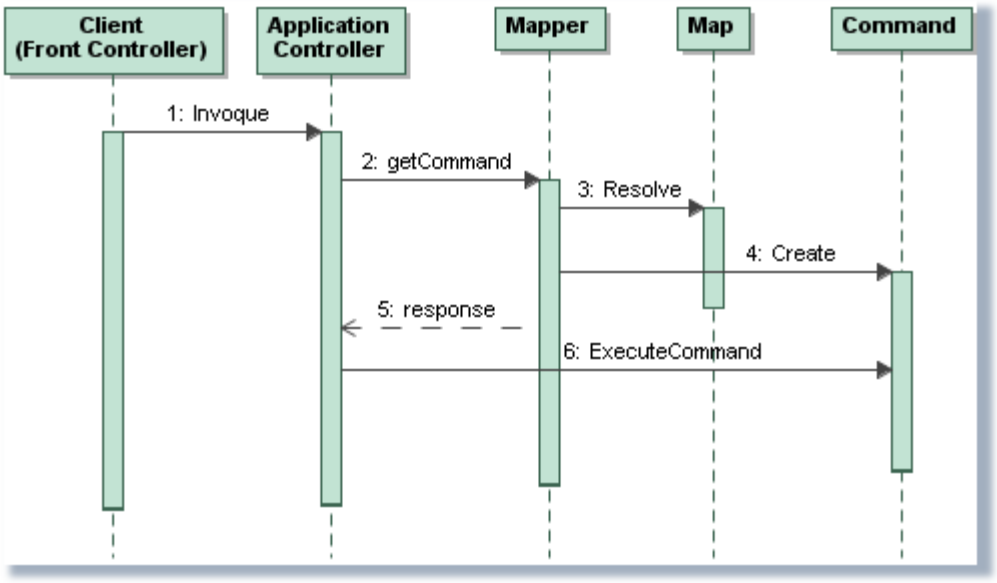


Fig.4.8: *Diagr. Sec. Request Patrón Application Controller estrategia "Command Handler"*

- ✓ ***Patrón View Helper:***

Un *Helper* se encarga de colaborar con una vista o con un controlador para alcanzar un determinado objetivo. El *ViewHelper*, por tanto, recupera y adapta los datos para ayudar a que sean representados por la vista.

En nuestro caso, el Framework debe hacer visibles a la vista los datos cargados en el modelo y por otro lado, introducir los diferentes componentes de la interfaz de usuario. Para ello, para operaciones complejas podría usarse normalmente Tag Files o clases planas Java, que faciliten esta operación, empleando tal patrón y evitar en lo posible programar y añadir lógica en la vista que vaya más allá de las etiquetas de las librerías.

- ✓ ***Patrón Composite View:***

Nos permitirá construir vistas mediante composición de otras, permitiendo reutilizar componentes comunes entre varias de ellas, como podría ser el caso de menús, pies, cabeceras comunes, etc.
La vista será por lo tanto en la mayoría de los casos un *Composite View*.

- ✓ ***Patrón Service to Worker:***

Se trata de un patrón mixto, está compuesto de otros patrones que ya han sido explicados. El control centralizado de las request y la creación de las view (vistas) se llevarán a cabo con los patrones *Front Controller*, *Application Controller* y *View Helper*.

El *Front Controller* se encarga del manejo de los elementos que forman parte de la petición. El papel del control se delega al patrón *Application Controller*, que recibirá un objeto de tipo *Context Object* que encapsulará la petición del cliente.

El *Application Controller* actúa en este momento como un *Command Handler*, de tal forma que se encargará del mapeo de los nombres lógicos a sus correspondientes comandos y acciones, para posteriormente invocar al comando correspondiente.

Posteriormente, el *Application Controller* invoca a la lógica de negocio necesaria recuperando la información devuelta (y almacenándola en el modelo) y acto seguido dirigiéndose a la vista creando una respuesta dinámica según los datos guardados en el modelo (haciendo uso del patrón *View Helper*).

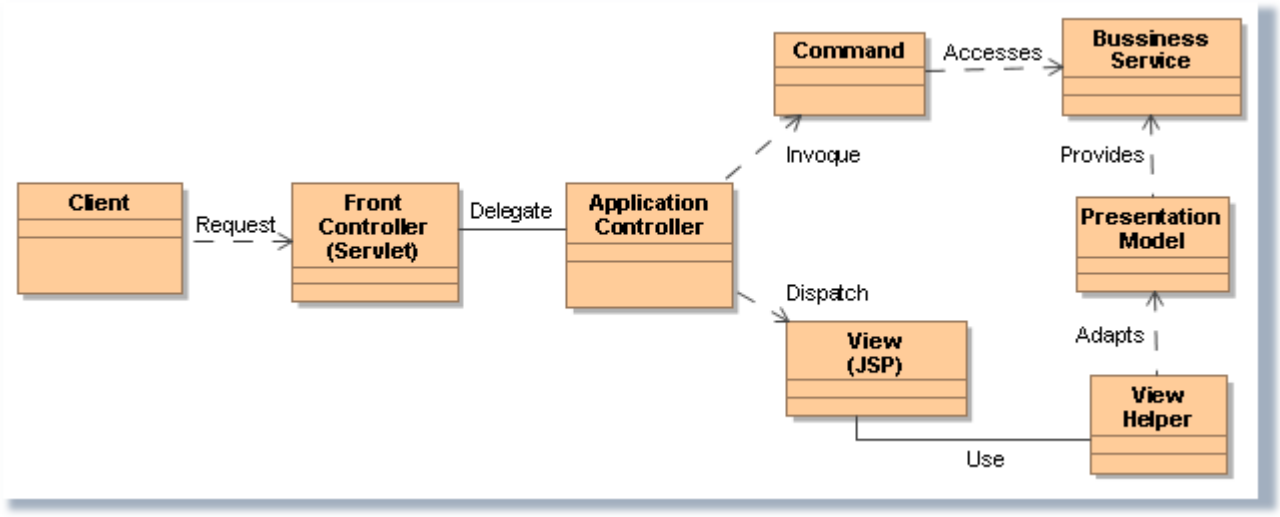


Fig.4.9: *Arquitectura del patrón Service to Worker con estrategia "Command"*

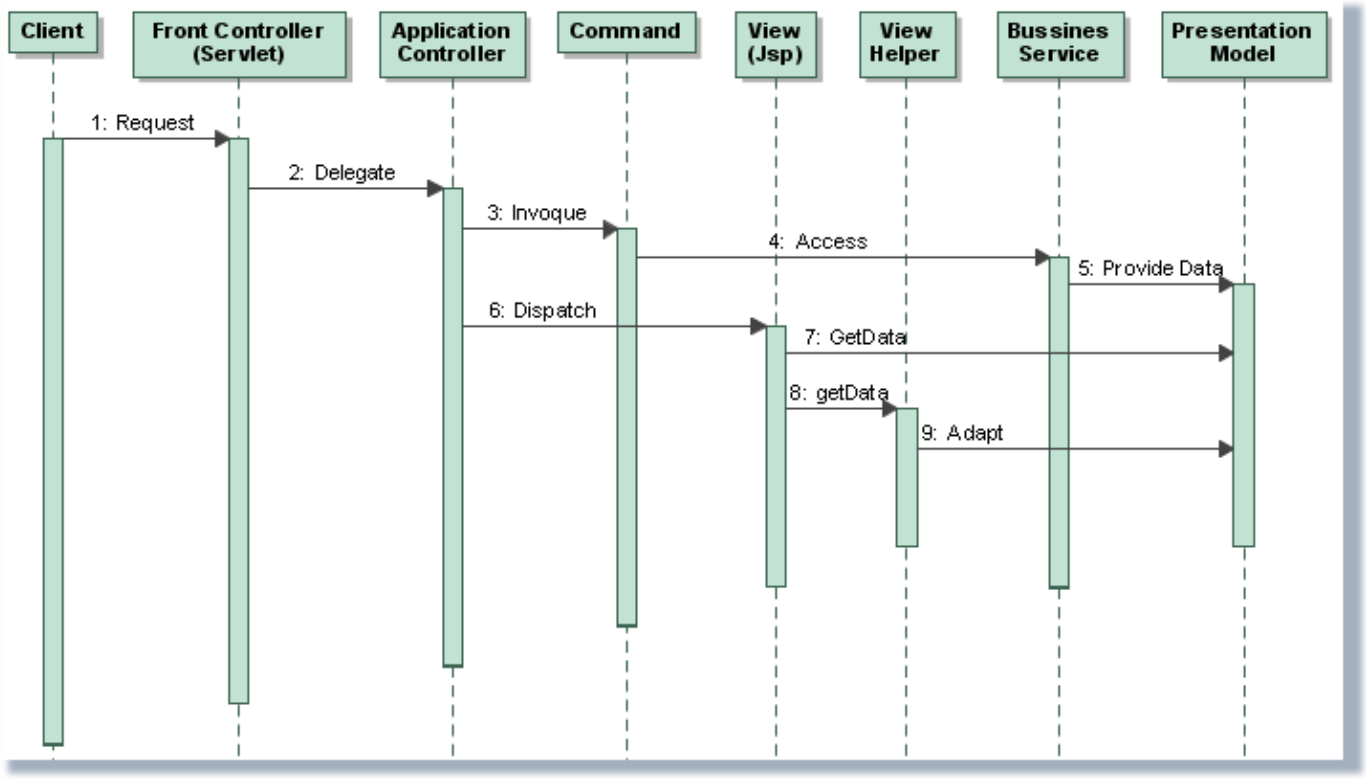


Fig.4.10: *Diagrama de Sec. petición aplicando Patrón Service to Worker*

4.4 Diseño

4.4.1 Diagrama de Clases del Framework

A continuación se mostrará un diagrama de clases de los diferentes componentes que forman nuestro Framework y posteriormente se hará un repaso general del cometido de todos y cada uno de los componentes:

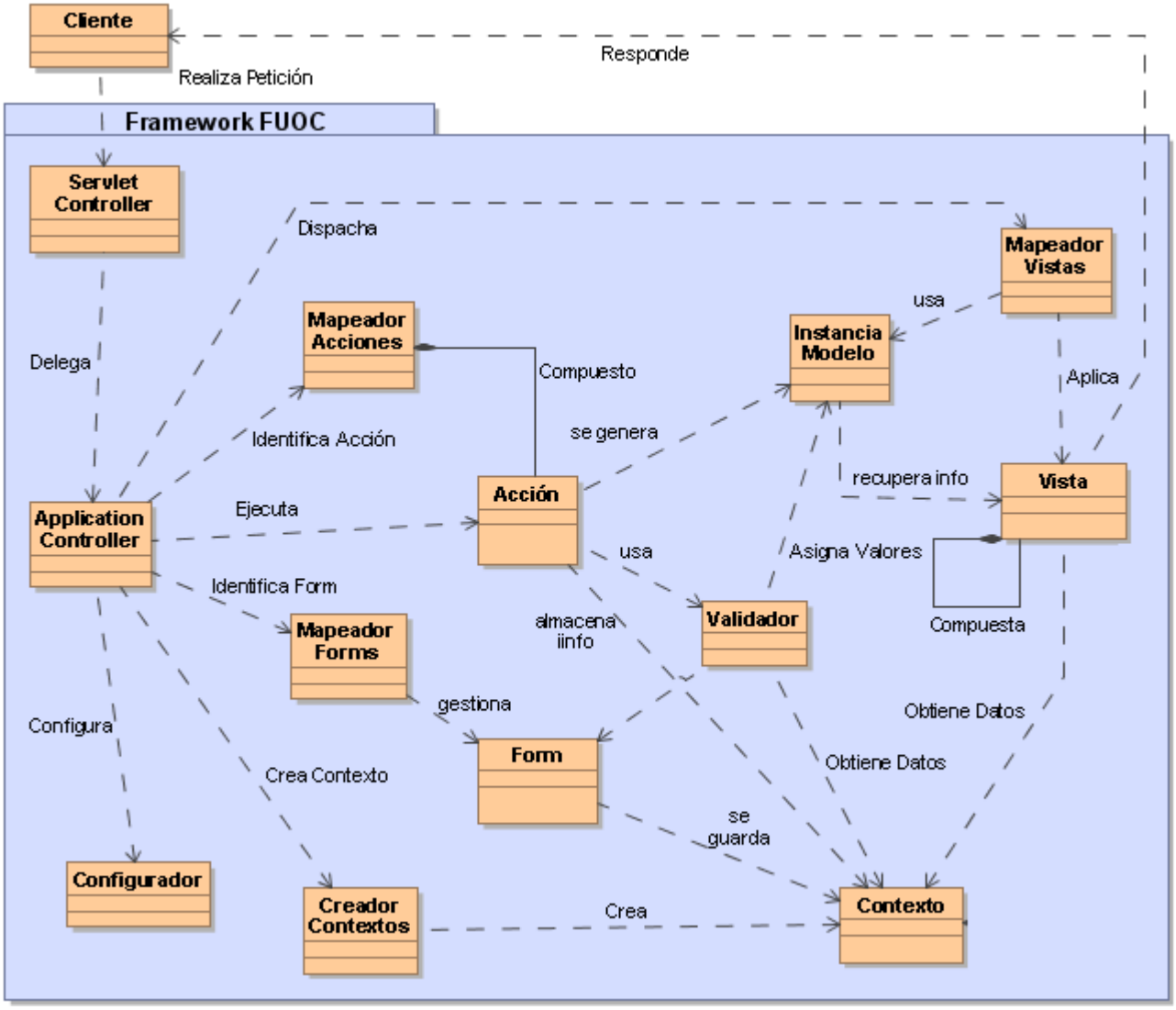


Fig.4.11: *Diagrama de clases de los componentes del Framework FUOC*

Como se puede observar en la figura anterior, se dispone de los siguientes componentes en nuestro sistema:

- ***Front Controller:***

  Empleado para centralizar las peticiones sobre la aplicación (y nuestro Framework). Delegará sus funciones sobre el controlador de aplicación (*Application Controller*) que se encargará de la inicialización de toda la aplicación y de ejecutar las acciones provocadas por las peticiones, entre otras, descargando de este modo la carga del Servlet.

- ***Application Controller:***

  Sobre él se delegan las diferentes peticiones del cliente (enviadas por el Servlet) además de ejecutar otras operaciones, como la configuración inicial del sistema (vía fichero XML), mapeo de las acciones, funciones de validación, mapeo de la vista, etc.

  Como ya se ha mencionado, se empleará una estrategia "*Command handler* ", donde este patrón determinará sobre un CommandFactory cuál es la acción a llevar a cabo (Command) en base al mapeado de las acciones sobre las diferentes request posibles.

- ***Validador:***

  La validación se realiza por cada Acción que se implementa, de tal forma que cada "*Action*" sabrá exactamente qué tipo de validaciones ha de realizar según el caso (y estas se llevarán a cabo siempre como paso previo a la ejecución final de la acción).
  Se contará también con una clase auxiliar Validador que facilitará las labores de validación (validación de email, etc.) a la acción concreta.

  La validación se realiza sobre la entrada de datos del formulario, siempre en el servidor empleando los diferentes mecanismos para ello, no desde el cliente (como mejoras al Framework se propone la posible validación en el cliente vía Javascript, aportando mayor agilidad al sistema y menor carga de peticiones al servidor).

  Si durante el proceso de validación algún dato fuese erróneo, se generará la excepción correspondiente (*FUOCException*) y si todo fuese bien, los datos de la Instancia Modelo se enviarán para la View correspondiente.

- ***Configurador:***

Elemento que en el momento de arrancarse la aplicación, leerá y gestionará los diferentes parámetros de inicio tanto de un archivo de configuración XML (*web.xml*), como los parámetros y configuraciones necesarias a tener en cuenta por la aplicación (ficheros Properties). Se creará una vez por arranque la aplicación, por lo que la instancia será única.

- ***MapeadorAcciones:***

  Encargado de gestionar el mapa de acciones general, de tal forma que, para cada acción concreta extraerá la información necesaria (request, form asociado, vista de error, ámbito, etc.) y la almacenará, de cara a facilitar la labor al Application Controller a la hora de distribuir el trabajo.

- ***Acción:***

  El controlador en base al mapa de acciones y la petición recibida, decide qué acción se ajusta a la necesidad y por tanto ha de ejecutar. Todas las acciones (*Action*) heredarán de *AbstractAction*, conteniendo los métodos necesarios para validar y llevar a cabo una acción concreta. El Mapeador de Acciones estará compuesto por N acciones, y éstas serán las que se invoquen en función de la petición recibida.

- ***MapeadorForms:***

  Elemento que gestionará los diferentes Forms a crear y "rellenar" de información, conocida la acción a ejecutar, obteniéndose en base al nombre su clase "*Form*" asociada. La clase "*Form*" elegida será una subclase de la clase abstracta "*Abstract Form*" y se instanciará de forma dinámica en tiempo de ejecución en función de los datos recibidos.

- ***Form:***

  Componente que se encargará de almacenar el conjunto de parámetros de entrada que son procesados y cargados en un formulario. Se instanciará un Form (heredando de *Abstract Form*) determinado en función del formulario de entrada, y será al que acceda la Action en el momento de ejecutar la acción y el que se almacene en el contexto por si luego es necesario su uso en las vistas.

- ***Modelo:***

Es el resultado devuelto por la ejecución de una determinada acción, que contiene el Modelo y todos aquellos datos que fuesen posteriormente necesarios para mostrar o gestionar en la vista. El objeto generado resultado de la acción se le entregará a la vista (almacenándolo en el contexto), donde posteriormente el Framework gestionará qué vista mostrar al cliente, que será quien use esos datos.

- ***MapeadorVistas:***

  Elemento distribuidor que se encargará de identificar la vista a aplicar en función del resultado obtenido tras la ejecución de la acción (y creación del Modelo). Analizará si se ha producido un error o no y acto seguido decidirá si con ese error (o ejecución correcta) ha de reenviar el flujo a otra vista o acción determinada. La información que necesita para realizar estas redirecciones viene dada en primer lugar por el resultado de la acción y en segundo lugar por los datos extraídos del fichero "*actionProperties*".

- ***Vista:***

  Componente que se va a crear y enviar al cliente final, decidido por su mapeador y en función de la acción ejecutada. Para la composición de la vista se podrán hacer uso de las librerías de Tags definidas para poder acceder al modelo creado y a los diferentes elementos de la petición (request/session).

- ***CreadorContextos:***

  Como ya se ha comentado en el apartado de patrones (en este caso implementa un patrón Context), el cometido de este elemento es crear un contexto al que asignarle una sesión. El Creador de Contextos creará por tanto una instancia que recogerá los datos del Servlet HTTP, de la request y del response, y se encargará, entre otros, de almacenar en request o en sesión las diferentes acciones, según la necesidad.

4.4.2 Inicialización de la Aplicación

La inicialización de la aplicación también seguirá un proceso ordenado de pasos, en los que como ya se ha comentado, el *Front Controller* delegará por completo en el *Application Controller* de cara a la inicialización y a la posterior gestión de las peticiones.

Para la inicialización se destacarán los siguientes ficheros, fundamentales para el correcto arranque:

- WebContent/WEB-INF/**web.xml:**

```
<display-name>ClubCiclistaUOC</display-name>
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

<servlet>
  <description>Controlador del framework FUOC</description>
  <display-name>FrontController</display-name>
  <servlet-name>FrontController</servlet-name>
  <servlet-class>edu.uoc.pfc.drsimo.fuoc.core.FrontController</servlet-class>

  <init-param>
    <description>Fichero de Configuraciones Generales</description>
    <param-name>configFile</param-name>
    <param-value>configFile.properties</param-value>
  </init-param>
  <init-param>
    <description>Fichero de Acciones</description>
    <param-name>actionsFile</param-name>
    <param-value>actionsFile.properties</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>FrontController</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

Fichero de entrada a la Aplicación (redirigirá a la primera acción index.html)

Ubicación de la clase FrontController (controlador de la App)

Fichero de Configuraciones Globales

Fichero con el mapeo de todas las Acciones implementadas

Patrón de URL de las peticiones a las que responderá el Controlador

Fig.4.12: *Fichero Web.xml*

- **configFile.properties**: Contendrá todos aquellos parámetros globales necesarios para el arranque de la aplicación.

- **actionsFile.properties:** Es uno de los ficheros más importantes a tener en cuenta para el desarrollo de la aplicación. En él se describen todas las acciones, asociándolas a su correspondiente formulario, action, vistas/acciones a ejecutar tras la ejecución de la acción y por el último el ámbito (*scope*) de cada acción.

El patrón de cada una de las acciones es como el que sigue:

```
http://<server_name>:<puerto>/<Context_root>/<accion>.htm
```

Un ejemplo que muestre tal patrón es, por ejemplo, la acción de Login de un Usuario:

http://localhost:8080/clubciclistauoc/***loginusuario.htm***

Para una petición de este tipo, se recuperará la siguiente información:

```
<accion>
```

Nombre de la *Clase Action* (hereda de *Abstract Action*) que procesará la petición.

```
<accion.form>
```

Nombre de la *Clase Form* (hereda de *FormBean*) que empleará un Formulario concreto para la recepción de los diferentes parámetros asociados a tal petición.

```
<accion.next>
```

Siguiente acción a realizar. Esta puede ser de 2 tipos:

- ✓ **view-<nombre_jsp>** → Redirige a la vista (JSP) indicada <nombre_jsp>.jsp
- ✓ **redirect-<otra_accion>** → Redirecciona a la acción indicada en <otra_accion>.
- ✓ **Vista general de Error** → En caso de no ser ninguna de las anteriores.

```
<accion.scope>
```

Qué tipo se ámbito se le asociará a la acción concreta (*Session*, *Request*). Por ejemplo, para el caso del "*login*", se encontrará la siguiente información:

```
# ------------------------------------------------------------------- #
# Acción/Página de 'Login de Usuario':
loginusuario       = edu.uoc.pfc.drsimo.ccuoc.acciones.LoginAction
loginusuario.form  = edu.uoc.pfc.drsimo.ccuoc.forms.LoginForm
loginusuario.next  = view-homeUsuario
loginusuario.error = view-login
loginusuario.scope = session
# ------------------------------------------------------------------- #
```

Fig.4.13: *Definición de Acción "Login de Usuario" (actionConfig.properties)*

Como se puede observar, la petición de Login, se asociará a una acción *Login Action*, que llevará consigo un formulario de entrada *Login Form* (con los campos que se esperan para esa acción), la petición será de tipo *Session* (indicado en el scope) y por último de indica que, en el caso de ejecutarse correctamente la acción, se redirigirá a la vista de "*homeUsuario*" y en caso contrario de nuevo a la de "*login*" (mostrando además al usuario el correspondiente mensaje de error).

En líneas generales, el proceso de inicialización de la aplicación pasará por los siguientes pasos, representados en el siguiente diagrama de secuencia:

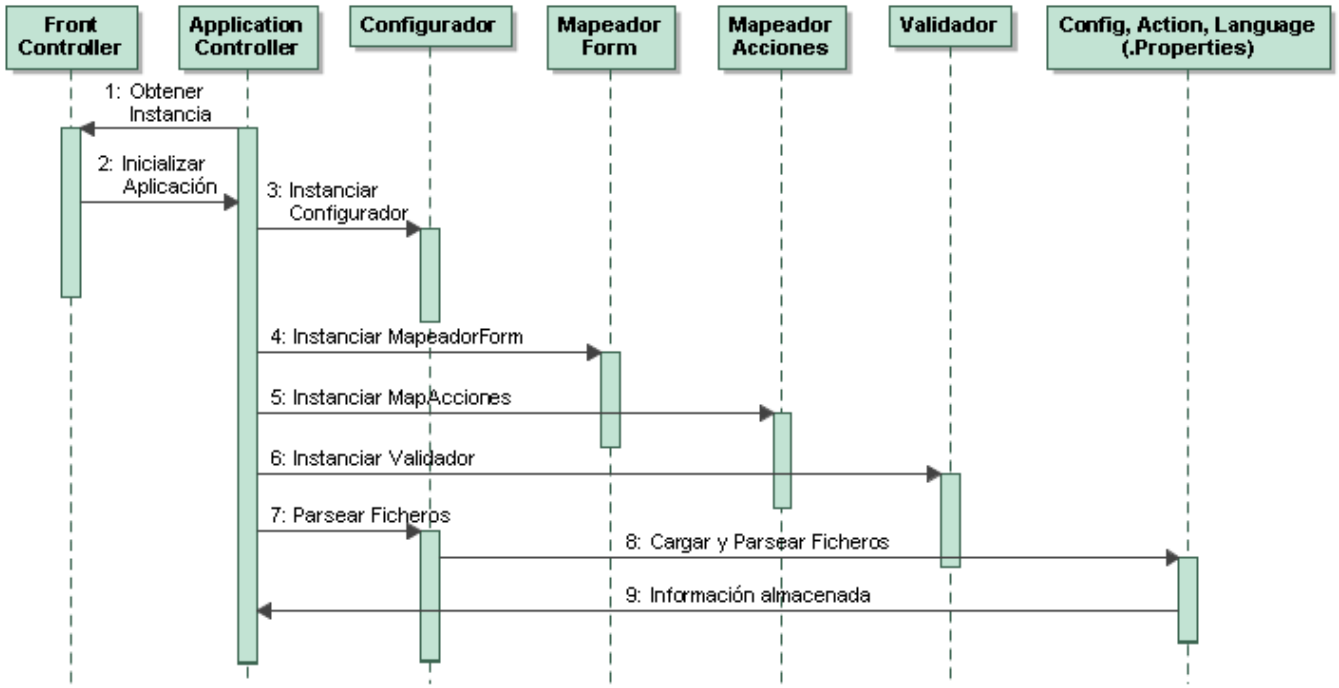


Fig.4.14: *Diagrama de Secuencia de la Inicialización del Framework FUOC*

Como se puede apreciar y ya se ha comentado anteriormente, el *Front Controller* delega por completo al *Application Controller* la inicialización de la aplicación (y la posterior gestión de peticiones, que se verá después), y éste se encargará de realizar las diferentes gestiones con el resto de componentes.

4.4.3 Diagrama de Secuencia de una petición

Una vez conocidos a nivel general los diferentes componentes que forman nuestro sistema, vamos a realizar una vista por los diferentes pasos por los que transcurriría el ciclo de vida de una petición al llegar a nuestro Framework.

Al arrancar la aplicación y configurarse nuestro Framework como está previsto, nuestro sistema estará listo para recibir peticiones, que serán asociadas a las diferentes acciones disponibles y devueltas al usuario en una vista determinada:

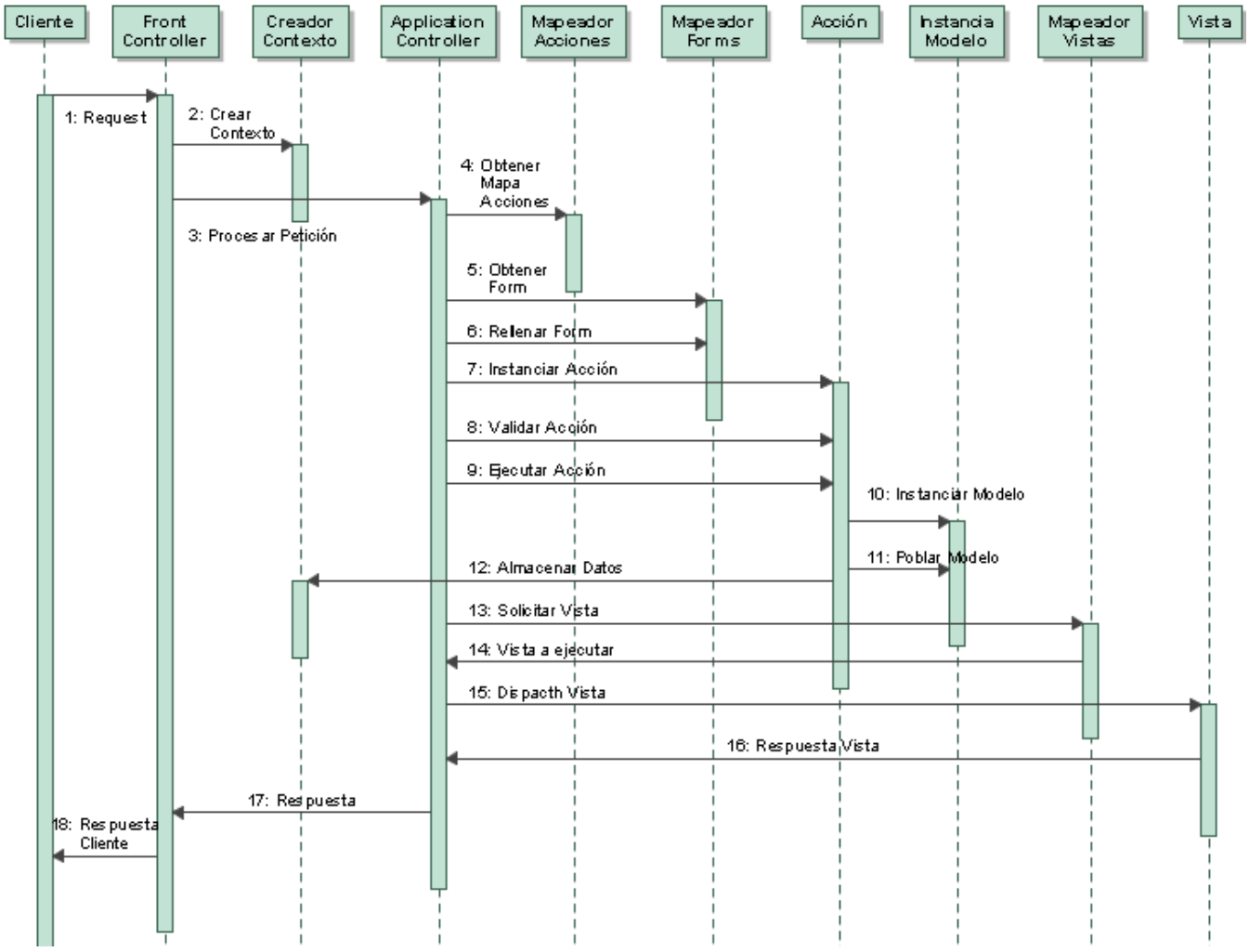


Fig.4.15: *Diagrama de Secuencia de una petición al Framework FUOC*

En términos generales, estos son los pasos que se llevan a cabo:

- Llega la petición del Cliente que capturará el *Front Controller*. Le petición se hará corresponder con un nombre de acción. Éste crea una nueva instancia haciendo uso del

CreadorContextos, donde se guarda los componentes *Request*, *Response* y el *Contexto*. Acto seguido lanza la petición al *Application Controller,* delegando en él los siguientes pasos a dar para la gestión de la petición.

- El ApplicationController orquesta a partir de aquí los siguientes pasos:
    - Realiza una llamada al *MapeadorAcciones*, a partir del cual obtiene la siguiente información:
        - Petición realizada.
        - Acción y Formulario que se asociarán a tal petición.
        - Vista o Acción a donde redirigirá el flujo tras la ejecución de la acción.
        - Ámbito de la petición.
    - Con toda la información anterior, instancia la "*claseAction*" (que implementa de "*AbstractAction*") asociada a la Acción correspondiente, que será quien se encargue de realizar la diferentes operaciones.
    - Instancia la clase MapeadorForms, encargada de instanciar y rellenar la clase "*Form*" concreta en función de los datos recibidos de el Formulario HTML.
    - Una vez creada la clase Form correspondiente, lo "rellenará" con los datos introducidos por el usuario en el formulario de entrada.
    - Con el Form y todos sus datos, se somete a un proceso de validación de la información recibida. Esta acción se lleva a cabo por la "*Action*" que lleva a cabo esta petición, y en algunos casos podrá hacer uso de la clase Validador para realizar comprobaciones complementarias.
    - Realizadas todas las validaciones necesarias (y suponiendo que no se ha generado error alguno, de lo contrario se lanza la *FUOCException* correspondiente), se invoca a la operación "ejecutar" de la acción, almacenándose los resultados tanto en el contexto como en el modelo que se instancia para tal efecto.
    - Ejecutada la acción, se dispacha a la vista/acción correspondiente (delegando en el *MapeadorVistas*). Si por el contrario se produjese un error, también se

redireccionaría a una vista concreta o a la de error general (esta información se extrae del *actionConfig.properties*).

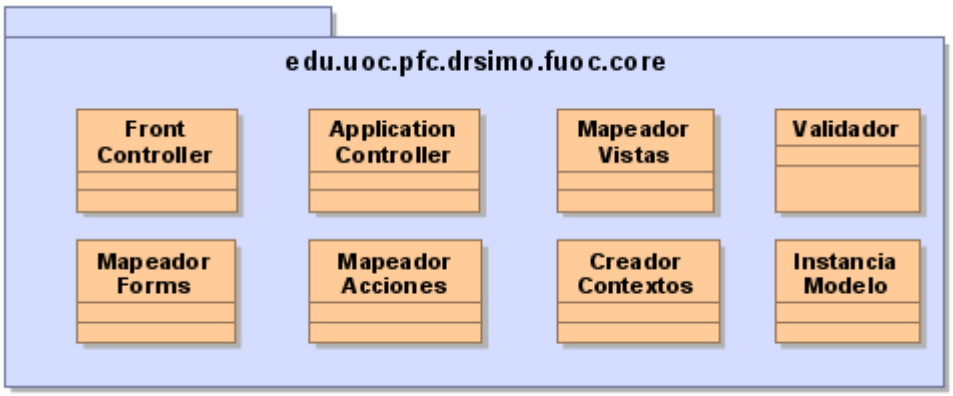
4.5 Estructura de paquetes

La implementación de las diferentes clases se ha organizado en diferentes paquetes Java organizándolos con una estructura lógica. En los siguientes gráficos se muestra la organización de los diversos paquetes y la funcionalidad que posee cada uno de ellos. Estos son los siguientes:

Paquete	Descripción
edu.uoc.pfc.drsimo.fuoc.core	Conjunto de Clases principales "core" del Framework que soportarán las principales funcionalidades del mismo.
edu.uoc.pfc.drsimo.fuoc.actions	Contiene AbstractAction, clase abstracta a partir de la cual heredarán todas las Actions.
edu.uoc.pfc.drsimo.fuoc.config	Contiene las clases encargadas de la configuración inicial del Framework y funciones varias.
edu.uoc.pfc.drsimo.fuoc.exceptions	Contiene FUOCExcepcion, clase que gestiona el lanzamiento de excepciones generadas.
edu.uoc.pfc.drsimo.fuoc.forms	Contiene AbstractForm, clase abstracta a partir de la cual heredarán todos los Forms.

Fig.4.16: *Estructura de Paquetes de FUOC*

A continuación se verá el contenido de estos mismos paquetes en las diferentes clases que participan en el ciclo de vida del Framework:



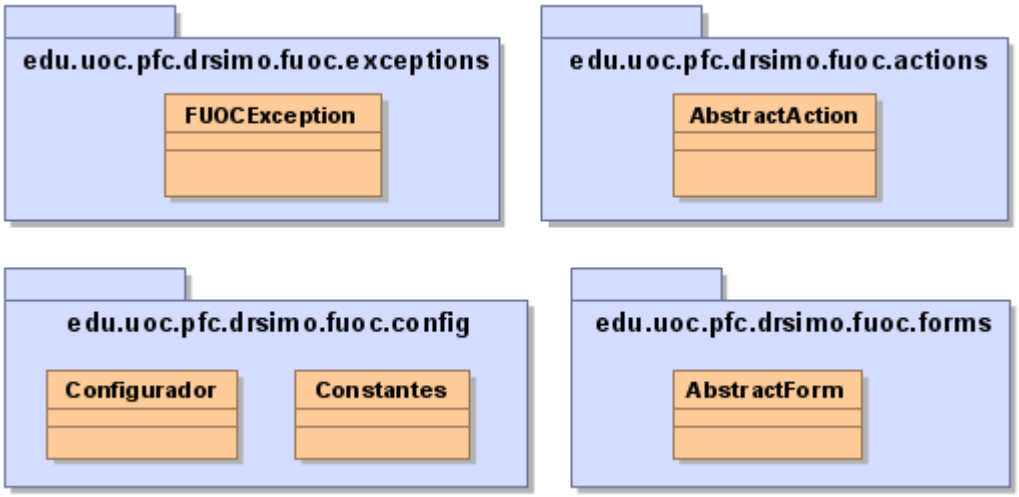


Fig.4.17: *Diagrama de Paquetes de FUOC*

4.6 Diccionario de Clases y métodos

Tanto para la implementación del Framework como para la Aplicación, se ha generado el diccionario de las diferentes Clases y Métodos mediante la herramienta *Javadoc*, la cual se generará (mediante *Ant*) toda la documentación resultante bajo el directorio /Javadoc de ambos proyectos.

Todas las Clases y Métodos tendrán en el código un encabezado con la siguiente estructura:

```
/**
 * - Fichero: ActualizarUsuarioAction.java <p>
 * - Paquete: edu.uoc.pfc.drsimo.ccuoc.acciones <p>
 * - Función: Clase encargada de actualizar (y guardar en BBDD)....
 * @author  Daniel Rguez. Simó
 */
public class ActualizarUsuarioAction extends AbstractAction {
```

Fig.4.18: *Ejemplo cabecera (Javadoc) para una clase*

Se informará del nombre del fichero que lo contiene, el paquete que lo engloba y la funcionalidad de cada uno de ellos.

5. Aplicación Web: "Club Ciclista UOC"

5.1 Introducción

El Club Ciclista UOC es un caso real de aplicación del Framework implementado (*FUOC-Framework*). La aplicación trata fundamentalmente sobre una Web de socios de un club ciclista, donde se da la posibilidad además de poder logarse como usuario registrado, listar los socios, editar, dar de alta nuevos o eliminar de la base de datos los ya existentes. A continuación se entrará más en detalle de las características a satisfacer.

5.2 Análisis

5.2.1 Requisitos

5.2.1.1 Requisitos funcionales

Las principales funciones que proporciona la aplicación se pueden resumir en los siguientes puntos:

1. ***Grupo funcional GF-01 Funcionalidad general:***

    a. ***RF-01: Logado.***

    El sistema implemente una función de *login*, donde el socio registrado podrá acceder al sistema introduciendo sus credenciales (*login/password*) dándole acceso a todas las opciones de la aplicación. Sus datos se mantendrán en sesión durante toda la interacción con el usuario (*Session*).

2. ***Grupo funcional GF-02 Gestión de usuarios:***

    a. ***RF-02:Listado de Socios***:

    Se proporciona un listado completo de todos los socios (incluyendo el Administrador) que forman actualmente parte del Club, mostrando todos sus datos personales más el login/password correspondientes.

b. ***RF-03: Consultar*** Datos Personales:

   A partir de la lista de Socios, se podrá acceder a los datos de un socio concreto, y en la misma página se dará la posibilidad de modificar sus datos o directamente eliminarlo.

c. ***RF-04: Creación*** de un nuevo Socio:

   Se presenta un formulario de entrada donde el Administrador podrá añadir un nuevo usuario en el sistema.

d. ***RF-05: Modificar*** Datos Personales:

   Pantalla de gestión de los datos personales de un Socio, donde se podrá modificar cualquiera de sus campos (login, password, nombre, apellidos, email y número de trofeos) para posteriormente actualizarse en la base de datos. Tanto la página de edición de datos personales como la de creación de nuevo usuario validarán la corrección de los datos como paso previo a almacenar los datos en la base de datos.

e. ***RF-06: Borrar*** Socio:

   Opción de borrado/eliminación de un socio del sistema. Previamente se comprobará que tal socio existía en la base de datos.

5.2.1.2 Requisitos no funcionales

En este ámbito deberíamos mencionar todas aquellas características que debería cumplir la aplicación fuera del contexto de su funcionalidad, como son rendimiento, accesibilidad, usabilidad, seguridad, etc.

5.3 Diseño de la Aplicación

5.3.1 Configuración de la Aplicación:

Como ya se ha explicado anteriormente (en el apartado 4.4.2 "*Inicialización de la Aplicación"*), durante el proceso de inicialización de la aplicación se llevan a cabo una serie de acciones relacionadas directamente con la configuración de la aplicación.

La configuración gira principalmente en torno a dos ficheros:

- **configFile.properties**: Contendrá todos aquellos parámetros globales necesarios para el arranque de la aplicación. Para la aplicación actual contendrá el parámetro correspondiente al idioma a emplear y por otro lado, la activación o no del "modo verbose".
- **actionsFile.properties:** Es uno de los ficheros más importantes a tener en cuenta para el desarrollo de la aplicación. En él se describen todas las acciones, asociándolas a su correspondiente formulario, action, vistas/acciones a ejecutar tras la ejecución de la acción y por el último el ámbito (scope) de cada acción.

La aplicación por tanto, durante el proceso de inicialización tendrá que gestionar y parsear ambos ficheros, generando las clases necesarias, para poder acceder posteriormente a cada uno de sus atributos. Por un lado se mapea el fichero del conjunto de acciones que se ejecutarán según las peticiones que reciba la aplicación, y por otro lado se guardarán en sesión el "paquete" de mensajes correspondientes al idioma seleccionado en el *configFile*.

Para estos ficheros de configuración también podrían haberse empleado ficheros XML, pero se ha optado por ficheros .*properties*, ya que su parseo es directo, sin tener que realizar procesos previos de validación del XML ni generación de reglas, además de la claridad que aporta tal y como están estructuradas sus diferentes acciones.

A continuación se muestra un pequeño fragmento donde se puede observar la claridad y organización del conjunto de acciones definidas (***actionsFile.properties***):

```
# ------------------------------------------------------------------- #

# Acción 'Eliminar un Socio':
borrarusuario       = edu.uoc.pfc.drsimo.ccuoc.acciones.BorrarUsuarioActio
borrarusuario.form  = edu.uoc.pfc.drsimo.ccuoc.forms.IdUsuarioForm
borrarusuario.next  = redirect-listusuarios
borrarusuario.error = redirect-listusuarios
borrarusuario.scope = request

# ------------------------------------------------------------------- #

# Navegación a la Página Home de Usuario:
home         = edu.uoc.pfc.drsimo.ccuoc.acciones.Navegation
home.form    = edu.uoc.pfc.drsimo.ccuoc.forms.NavegationForm
home.next    = view-homeUsuario
home.scope   = request

# ------------------------------------------------------------------- #
```

Fig.5.1: *Fragmento de actionsFile.properties*

5.3.2 Clase Action

En nuestro Framework se implementarán diferentes acciones que se corresponderán con el abanico de peticiones que puede recibir la aplicación. Todas estas acciones se ejecutan en el servidor asociadas al procesamiento o submit de un formulario HTML. La acción se encargará en primer lugar de realizar los procesos de validación pertinentes de los datos entrantes y posteriormente de realizar el procesamiento de los datos necesario en el servidor, ya sea invocando a otros componentes de la capa de negocio o él mismo.

Las Acciones son identificadas por el Framework por un identificador unívoco que forma parte de la URL de la petición. La aplicación Web esta parametrizada en el fichero descriptor web.xml para que todas las URL que se dirigen a una acción (*.htm*) sean capturadas por el Servlet que hace el papel de *Front Controller*.

Una vez tomado el control por el *Front Controller*, se analizara la URL de la petición para obtener el identificador de la acción. Con esta información, se accederá al fichero *actionConfig.properties* de donde se extraerán todos los datos necesarios, como son la *Action* concreta a ejecutar, el *Form* asociado, las acciones a realizar a posteriori y el ámbito de aplicación de la petición.

En cuanto a la Action asociada, será quien ejecute la lógica y será instanciada dinámicamente en tiempo de ejecución e invocada por el *Application Controller*.

Todas las Action implementadas se encontrarán bajo el siguiente paquete:

edu.uoc.pfc.drsimo.ccuoc.acciones

Y todas ellas heredarán de *AbstractAction*, redefiniendo los métodos "*ejecutar*" y "*validar*" e implementándolos según las necesidades de la situación.

Todas las Action recibirán los siguientes parámetros:

- ✓ ***Contexto*** (*CreadorContextos*), que será el contexto del Servlet con toda la información de la request.
- ✓ ***Form*** (*Abstract Form*), que identificará al formulario generado para la ejecución de la acción y que contendrá los datos necesarios para su ejecución.

Todas estas *Actions* como ya se ha comentado dispondrán de un método propio "*validar*" donde se comprobará antes de lanzar la ejecución, si todos los parámetros que le han llegado del formulario son correctos.

No todas las acciones han de realizar una operación en sí, pueden darse casos en los que sólo se pretenda realizar una navegación de un sitio a otro de la aplicación sin necesidad de realizar validaciones, ni operativa alguna. Para esos casos de simple "navegación" se dispondrá de la clase *NavigationAction*, definida también en el mismo paquete que el resto de acciones de la aplicación.

A continuación de muestra un diagrama de clases de las relaciones de herencia entre la clase abstracta Abstract*Action* del Framework y las clases *Action* de la aplicación:

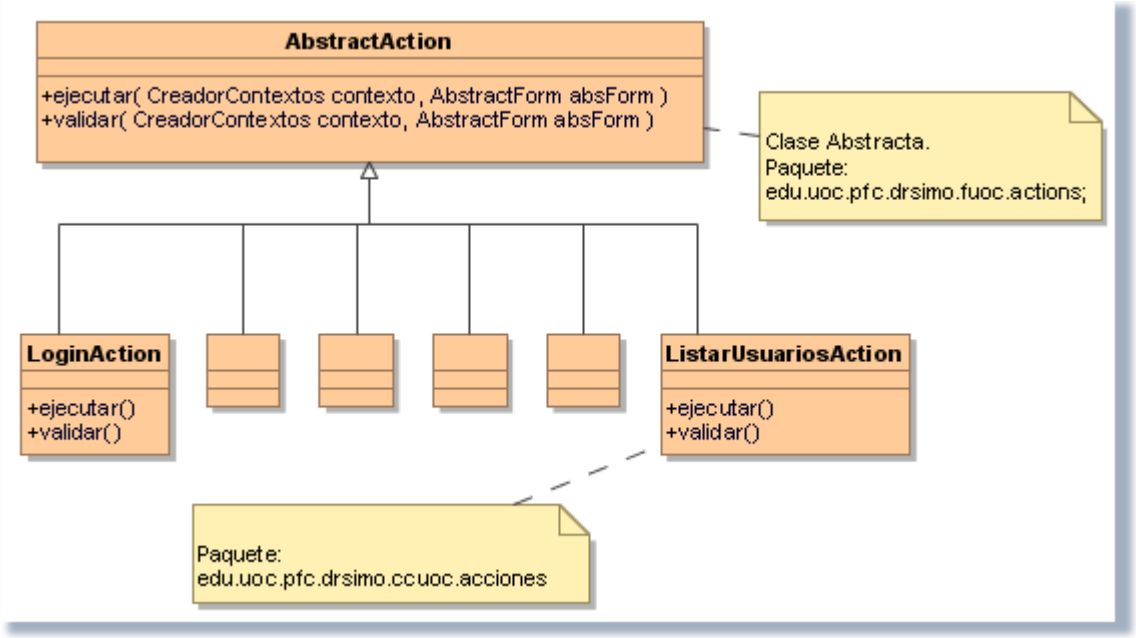


Fig.5.2: *Diagrama de clases de AbstractAction y Actions*

5.3.3 Clase Form:

Las clases *Form* se emplean para albergar los diferentes parámetros que pueden llegar desde los formularios de entrada a la aplicación. Se emplean para facilitar la labor a la hora de acceder a tales datos y para prepararlos de cara a la posterior validación y ejecución por parte de la acción.

Tales clases se instanciarán con el mismo número y tipo de parámetros que se esperan de entrada y además irán asociados a una acción concreta, de tal forma que han de existir todos los parámetros necesarios, de lo contrario al instanciar el *Form* se generarían excepciones. Cabe decir que tales clases además de contar con los atributos que correspondan, irán acompañados de sus métodos get/set necesarios para acceder a sus valores.

El procedimiento a seguir por lo tanto es el siguiente:

Cuando un usuario rellena un formulario y pulsa la acción de enviar, guardar, etc. el *Application Controller* instancia y rellena/configura las clases *Form* y la *Action* adecuadas para esa acción de forma dinámica, realizándose un mapeo entre los nombres de los campos del formulario y los atributos de la clase de forma automática. Se ha prestar especial atención al número, el tipo y el nombre de los parámetros, pues ambos han de coincidir, de lo contrario se generaría una excepción en tiempo de ejecución.

La misma lógica aplicada para el caso de los *Action*, se ha seguido para el caso de los *Form*, de tal forma que se dispondrá de una clase abstracta *AbstractForm* definida dentro del Framework, y en la aplicación, el conjunto de formularios que heredarán de esta, cada uno con sus respectivas características y atributos.

Todas los Form implementadas se encontrarán bajo el siguiente paquete:

edu.uoc.pfc.drsimo.ccuoc.forms

Y un diagrama de clases de las relaciones de herencia entre la clase abstracta de *AbstractForm* del Framework y las clases *Form* de la aplicación lo mostramos a continuación:

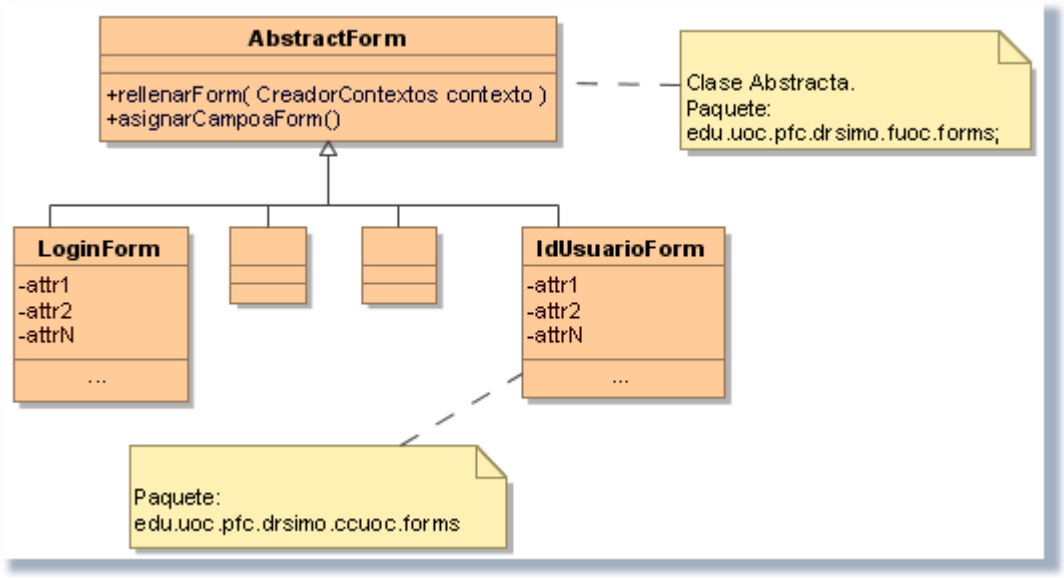


Fig.5.3: *Diagrama de clases de AbstractForm y Forms*

5.3.4 Ámbito de las Peticiones

En la presente aplicación de han tenido en cuenta dos tipos de ámbitos/contextos que se asociarán a cada una de las acciones (debe recordarse que esta asociación viene seteada en el fichero *actionsFile*), son el ámbito a nivel de petición (*Request*) y a nivel de sesión (*Session*):

5.3.4.1 A nivel de petición (Request)

Este contexto se genera por cada petición HTTP que llega del cliente, donde se instancia un objeto que implementa de ***javax.servlet.http.HttpServletRequest*** y que contendrá una colección de

pares de atributos clave-valor empleados para almacenar objetos, cuya vida coincidirá con la vida de la petición. La clave será en sí una cadena, y el valor puede ser cualquier tipo de objeto.

Una vez que el servidor atiende una solicitud y la respuesta es devuelta al cliente, la solicitud y sus atributos ya no estarán disponibles y se eliminan.

Para nuestro caso real se hará uso de este ámbito en la mayoría de los casos (peticiones), pero en otros será necesario almacenar la información durante más tiempo (se explicará a continuación).

En el momento de realizarse la petición, tendrán acceso a la información asociada todos aquellos objetos que tengan una referencia al objeto "*Contexto*", el cual se ha definido como "eje central" para cada una de las peticiones realizadas. Una vez que la respuesta ha sido devuelta al cliente, la visibilidad desaparece y también sus objetos asociados. Cabe también decir que aquellos objetos que se almacenan en el ámbito request no son visibles para otras peticiones del cliente.

5.3.4.2 A nivel de sesión (Session)

Esta ámbito también se reflejará en la implementación de la aplicación, y, aunque no es el predominante (la mayoría de las acciones tendrán ámbito request), sí será necesario para otras peticiones que se deseen almacenar durante más tiempo.

En este caso, el contenedor web instancia un objeto que implementa la interfaz ***javax.servlet.http.HttpSession***, que se empleará para identificar a un mismo usuario a través de múltiples solicitudes por las diferentes páginas.

La sesión también nos permitirá definir una colección de objetos que se almacenen en base a un esquema de pares clave-valor del mismo modo que se definen en el ámbito request. La diferencia entre este y en request es la duración de los objetos. Los objetos de sesión existen a través de múltiples peticiones del cliente. Y del mismo modo que en el caso de la request, los objetos almacenados en una sesión de usuario nunca serán visibles para los usuarios con una sesión diferente.

5.3.5 Arquitectura:

Club Ciclista UOC se pueden diferenciar claramente los tres niveles de Diseño:

5.3.5.1 Capa de Presentación:

La aplicación Web está configurada (*web.xml*) para que toda petición entrante por parte del usuario la trate nuestro Framework (*FUOC-Framework*), concretamente el *Front Controller*, bajo el paquete: **"*edu.uoc.pfc.drsimo.fuoc.core.FrontController*"**

```
<servlet>
  <description>Controlador del framework FUOC</description>
  <display-name>FrontController</display-name>
  <servlet-name>FrontController</servlet-name>
  <servlet-class>edu.uoc.pfc.drsimo.fuoc.core.FrontController</servlet-class>
</servlet>
```

Fig.5.4: *Definición del Servlet Controller (web.xml)*

Cada una de las acciones (que heredan de la clase abstracta *AbstractAction* definida en el Framework) se instanciarán para llevar a cabo las diferentes peticiones del usuario, y se podrán encontrar bajo el paquete: **"*edu.uoc.pfc.drsimo.ccuoc.acciones*".**

El conjunto de formularios (que heredan de la clase abstracta *AbstractForm*) se instanciarán acordes con las diferentes acciones solicitadas y con el abanico de posibles parámetros de entrada de los diversos formularios. Tales formularios se encontrarán en el paquete: **"*edu.uoc.pfc.drsimo.ccuoc.forms*".**

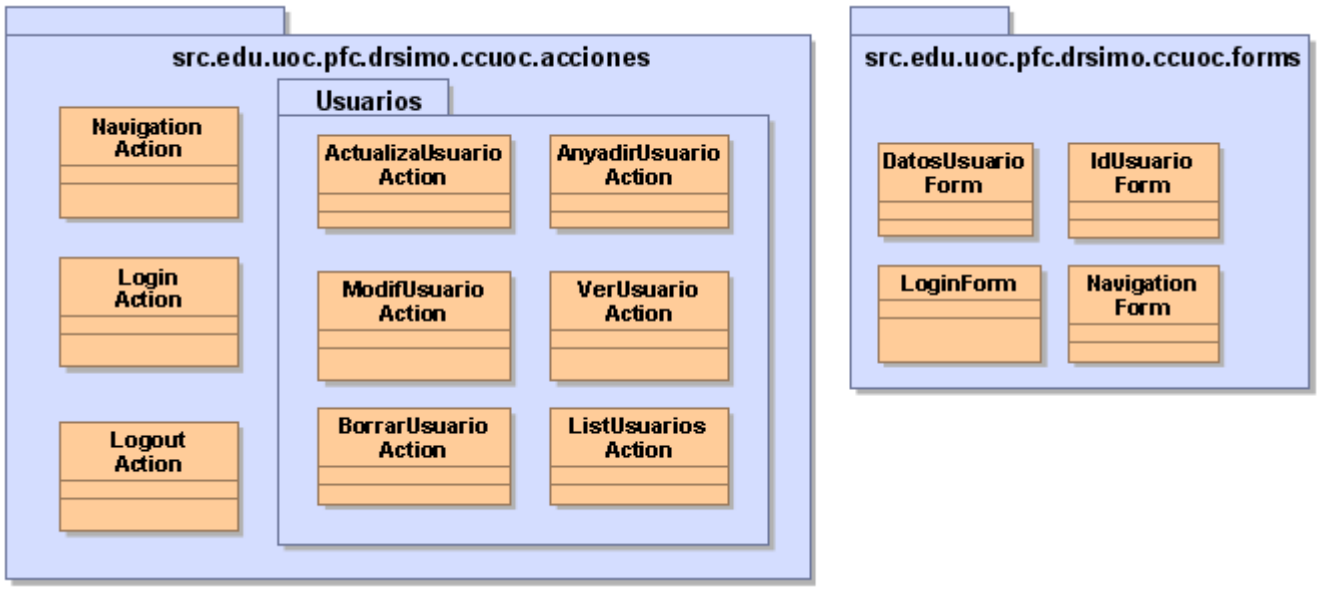


Fig.5.5: *Diagrama de Paquetes de "Acciones" y "Forms"*

Toda la parte de la vista (JSPs) se tendrá bajo el directorio WEB-INF/jsp, el cual, como ya se ha mencionado, se va a implementar la técnica de *Composite View*, agrupando en otro directorio (/includes) todas aquellos módulos que pueden ser reutilizados en la aplicación, consiguiendo de este modo que el código sea más reutilizable y mantenible.

Ejemplos de componentes que son reutilizables, serán los siguientes:

- ✓ Cabecera y Pie.
- ✓ Lista de Errores y Lista de Mensajes.

También aunque se ha implementado una página de error genérica (para los casos en que no se redirija a ninguna acción ni haya una vista configurada), tanto los mensajes de confirmación como los de errores que se produjesen durante el transcurso de la interacción se mostrarán "*inline*" en la misma pantalla o en la home de usuario con un icono y mensaje explicativo del error. Con esto conseguimos que el usuario no pierda el foco de donde estaba realizando su última operación y pueda corregir el posible error que hubiese cometido, en vez de llevarle a una página de error aislada:

Ejemplos de *Mensajes de Error*:

Fig.5.6: *Ejemplos de Mensajes de Error*

Ejemplos de *Mensajes de Confirmación*:

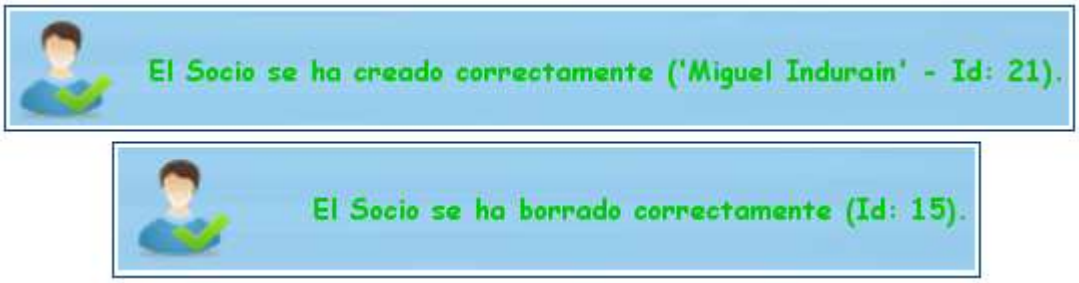


Fig.5.7: *Ejemplos de Mensajes de Confirmación*

(A nivel de la capa de presentación, en el apartado 5.4 "*Interfaz de Usuario*" se detallan más acciones realizadas en el lado del Cliente.)

5.3.5.2 Capa de Negocio

La capa de negocio está compuesta por un conjunto de servicios que accederán a la capa de persistencia para dar el modelo en la capa anterior. Estos servicios estarán implementados en el paquete **"*edu.uoc.pfc.drsimo.ccuoc.modelo*".**

En cuanto al servicio, se implementa "*UsuarioServicio*", que se encargará de todas y cada una de las acciones que se puedan llevar a cabo con los diferentes socios de la aplicación. *UsuarioServicio* por tanto, engloba todos aquellos métodos necesarios para dar todas las operaciones sobre la entidad Usuario y necesarios para la capa web. Estas operaciones son de creación, modificación y alta, consulta y listado de socios, junto con la validación por *Login* y *Password* de usuario.

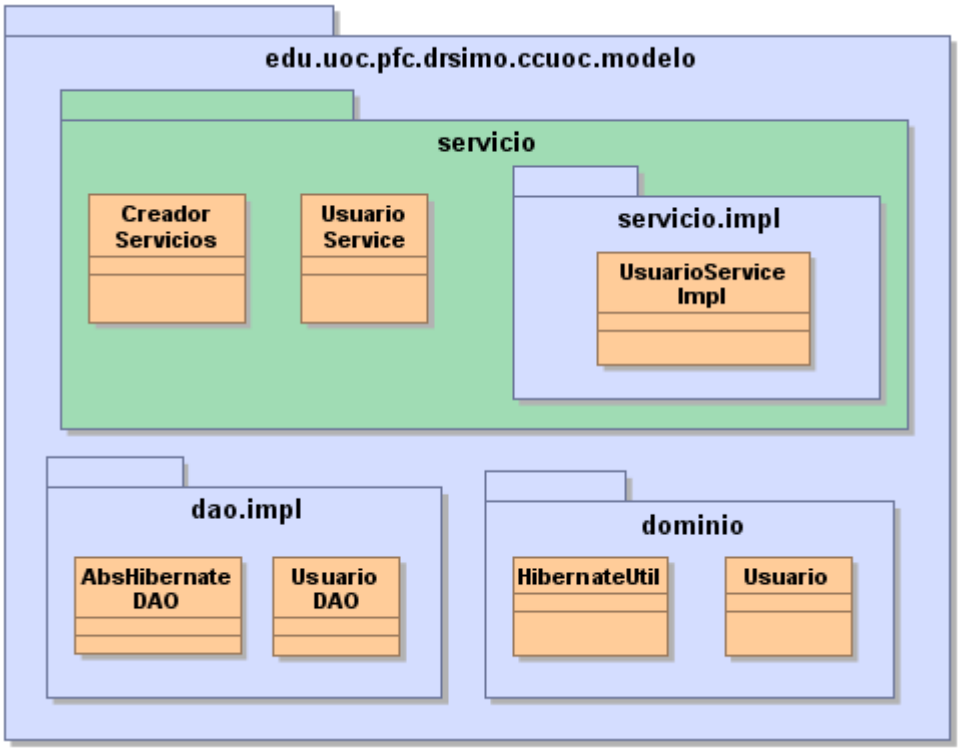


Fig.5.8: *Diagrama de Paquetes del Modelo Club Ciclista*

5.3.5.3 Capa de Integración:

La capa de persistencia estará gestionada por una base de datos MYSQL, donde se implementará un mapeado de sus diferentes entidades y posibles relaciones mediante clases java que se podrá encontrar en el paquete: ***"edu.uoc.pfc.drsimo.ccuoc.modelo.dominio".***

Este mapeado se hará mediante el Framework de persistencia *Hibernate* versión 3.2.5 siguiendo un patrón *DAO* (*Data Access Object*) por cada entidad implementada también en el paquete ***"edu.uoc.pfc.drsimo.ccuoc.modelo.dao".*** A continuación, se muestra un esquema de la arquitectura que mantiene nuestra aplicación diferenciando las diversas capas y cómo fluye información de unos componentes a otros, partiendo en primera instancia por la petición que formula el cliente llegando hasta la base de datos, y el flujo de vuelta que llega al Framework, que haciendo uso de las vistas, muestra finalmente al usuario los resultados obtenidos:

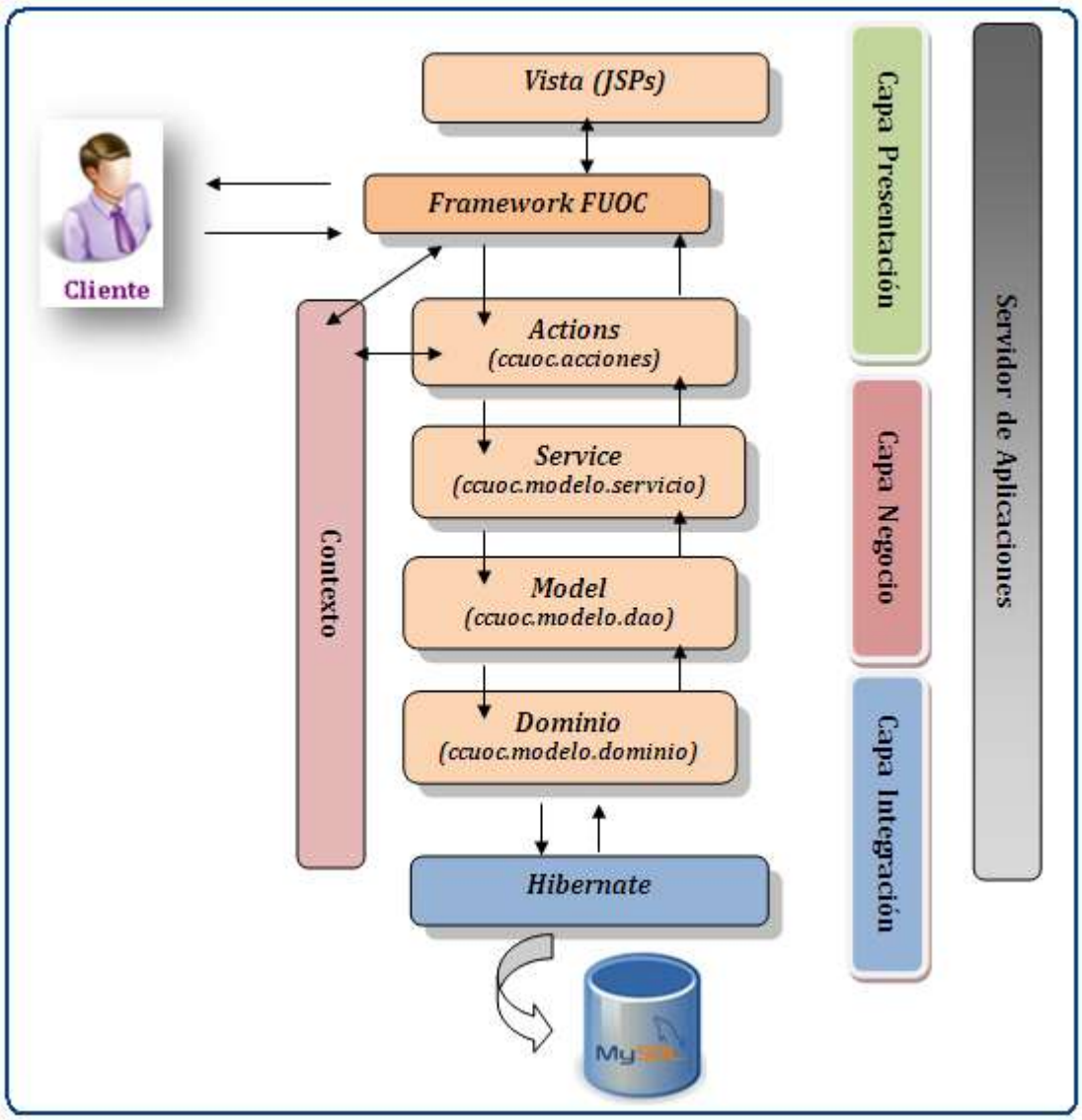


Fig.5.9: *Arquitectura de capas del Club Ciclista*

5.4 Modelo de Datos

Para la implementación de la aplicación se han diseñado las siguientes tablas:

- ***Tabla Usuario:***

Reflejará los datos personales de cada uno de los socios del Club, incluyendo su *login* y *password* para el acceso al sistema.

Para el presente Proyecto no se han implementado más clases relacionadas, pero una posible iteración podría pasar por generar tablas como "Competición", "Marcha", "Palmarés", que relacionándolas aportasen más información a cada una de las entidades.

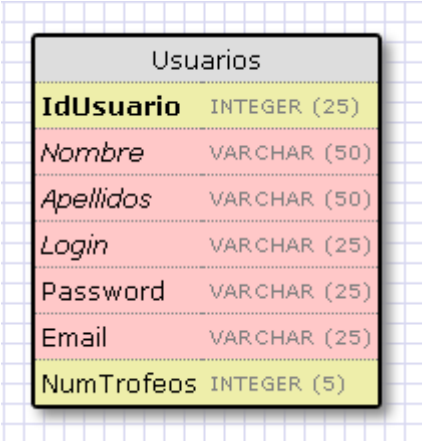


Fig.5.10: *Tabla Usuario*

5.5 Diagrama de Estados/Navegación

La página inicial de la aplicación permite solicitar el alta como socio o identificarse como usuario. Una vez identificado como usuario aparece el menú de opciones, donde se podrá gestionar los usuarios existentes, viendo sus datos, modificándolos o borrándolos.

También se podrán añadir nuevos usuarios al sistema. Para ello, se dispondrá de un formulario en el cual habrá que rellenar una serie de campos, los cuales unos serán campos obligatorios y otros habrá que informarlos correctamente (como por ejemplo, el campo email, añadiendo la "@" y el punto).

Además desde todos los puntos de la aplicación se podrá acceder a la página principal (*Home*), ir hacia atrás (a la pantalla anterior) y cerrar la sesión (*Logout.htm*). A continuación se muestra un diagrama que muestra claramente todas las transiciones que se pueden llevar a cabo:

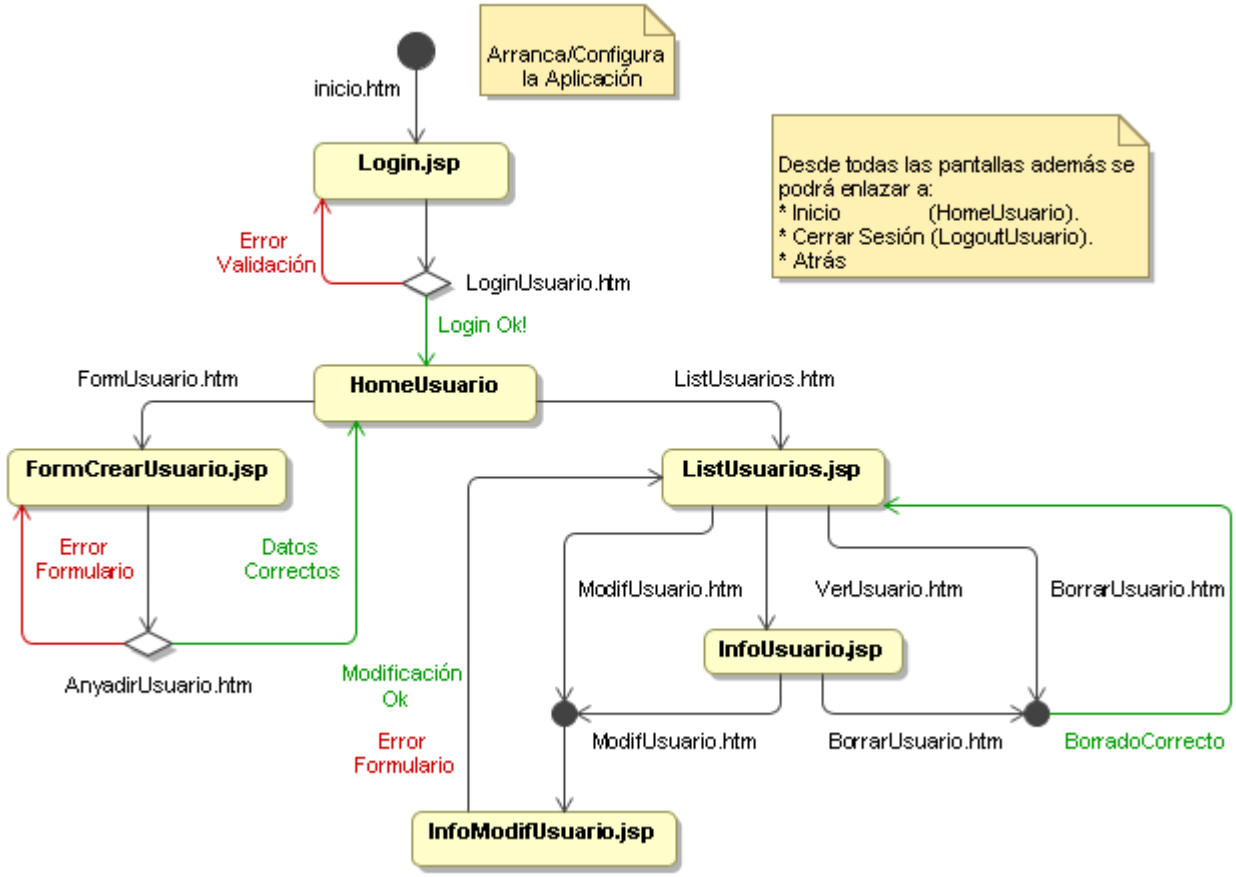


Fig.5.11: *Diagrama de Estados de la Aplicación*

5.6 Capa Vista/Interfaz de Usuario

Además de lo ya expuesto en el apartado anterior de la "Capa de Presentación", de cara a la Interfaz de Usuario se van a destacar los siguientes puntos:

5.6.1 Composición de las Vistas:

La aplicación se encuentra totalmente modularizada, en cuanto a vistas se refiere tratando de conseguir la mayor reutilización del código, favoreciendo así su mantenibilidad. Todas las páginas se encuentran compuestas de diversos componentes (JSPs), siguiendo por ejemplo el *Patrón Composite*, cuya estructura se puede apreciar claramente en el siguiente esquema:

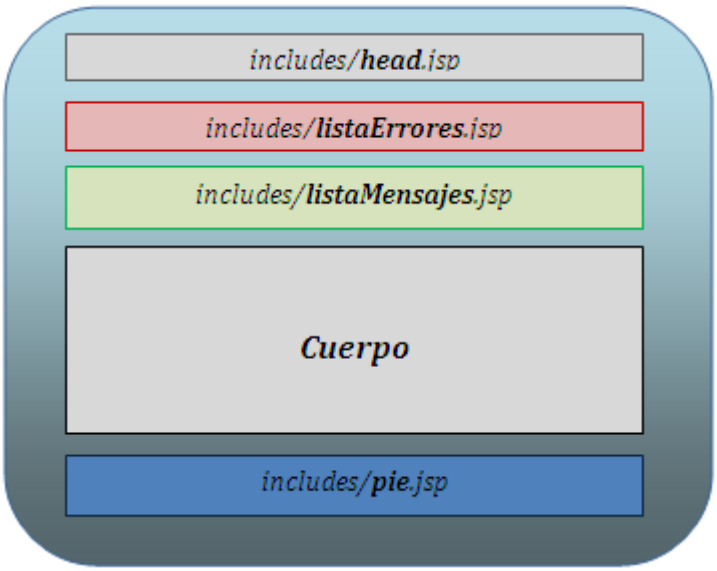


Fig.5.12: *Esquema de composición de una página*

5.6.1.1 Recursos

Además de las diferentes JSPs, la parte de la vista se ha organizado de la siguiente manera:

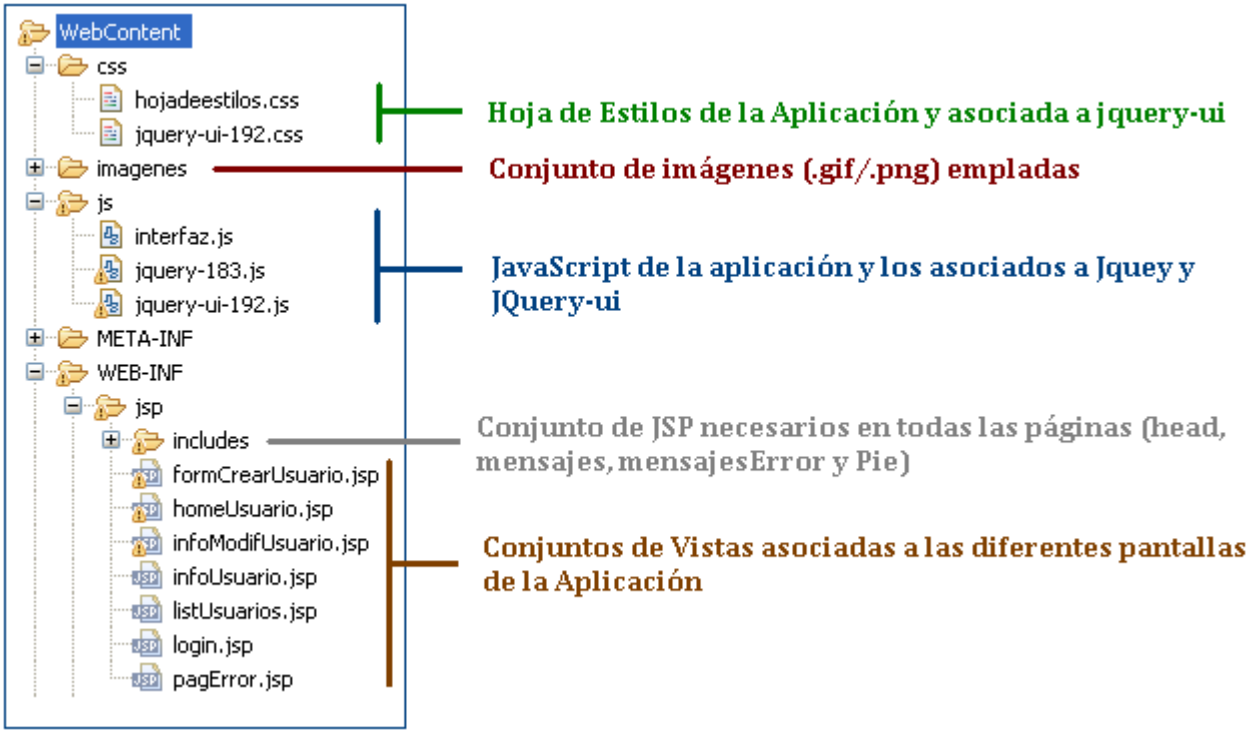


Fig.5.13: *Esquema general del Proyecto (Web Content)*

5.6.2 Jquery/JQueryUI

Como punto de partida para futuros desarrollos, se ha introducido Javascript en la parte cliente, en concreto Jquery/JqueryUI. Principalmente se ha añadido para los siguientes puntos:

5.6.2.1 Validación en el lado del cliente

Toda la aplicación se ha desarrollado realizando las diferentes validaciones en el lado del servidor, mapeando los formularios correspondientes con sus clases apropiadas, y las acciones eran las encargadas de "validar" los datos previamente a "ejecutar" la acción. Pues bien, mediante Jquery se puede agilizar esas validaciones, ahorrando peticiones al servidor si éstas contienen datos erróneos de partida.

Como ejemplo, se han realizado validaciones con Jquery en los formularios, tanto de "Modificar Usuario" como de "Añadir Usuario", en el campo "*NumTrofeos*", donde se comprueba que el valor introducido es un campo numérico, de lo contrario, se genera un mensaje de alerta indicándolo:

Fig.5.14: *Validación en cliente de campos mediante Jquery*

5.6.2.2 Usabilidad/Accesibilidad/Dinamicidad

- ✓ Con Jquery-ui también se ha impulsado sobremanera la ***Usabilidad*** de toda la aplicación, pues se han añadido textos alternativos en todas y cada una de las imágenes, campos de formularios y botones de la implementación, indicando en todo momento al usuario qué acciones puede realizar y como texto alternativo a todas las imágenes:
- ✓ Por el hecho de ser *Javascript* se podría encontrar el problema de la ***Accesibilidad*** al desactivar *Javacript* en nuestro navegador. Esta posibilidad también queda controlada pues

todos aquellos elementos que fuesen susceptibles de tener textos alternativos los tendrán asignados:

Fig.5.15: *Ejemplos de uso Jquery (Usabilidad)*

- ✓ Además de cara a darle otra estética "***Dinámica***" a la Aplicación, se ha hecho uso de Jquery para dotar a las tablas de total flexibilidad, de tal forma que el usuario podrá modificar con el ratón sus dimensiones cuando lo desee:

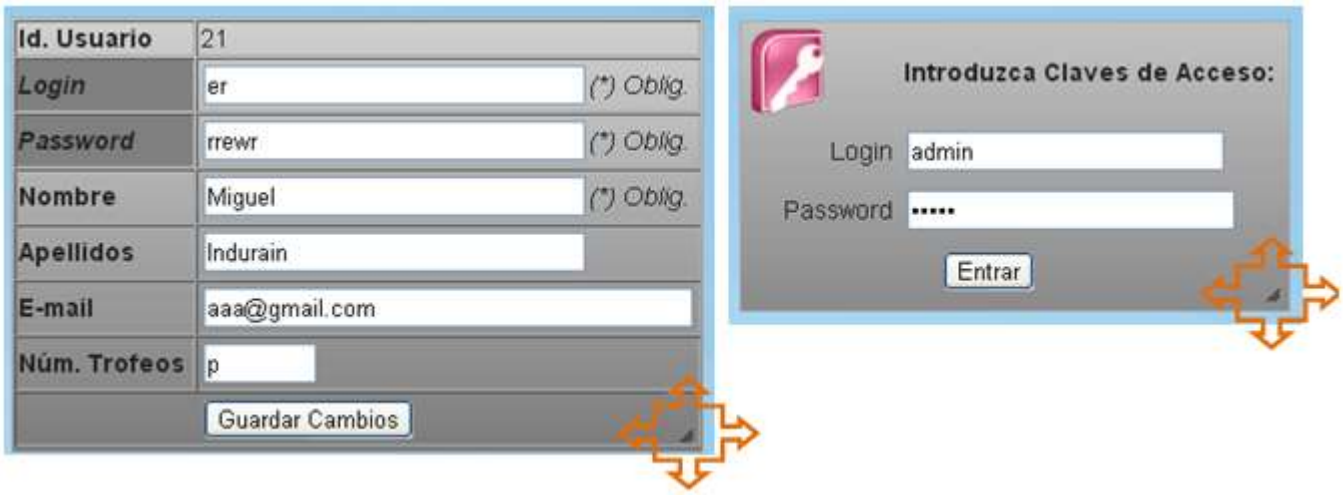


Fig.5.16: *Ejemplos de "dinamicidad "que nos proporciona JQuery"*

5.7 Dependencias

Para la implementación de la Aplicación se ha requerido de la existencia de un conjunto de librerías externas además de la del Framework en sí, con las que se han permitido entre otras,

operaciones con Hibernate, MySQL, etc. A continuación con la siguiente ilustración se muestran claramente el conjunto de librerías externas empleadas para tal desarrollo:

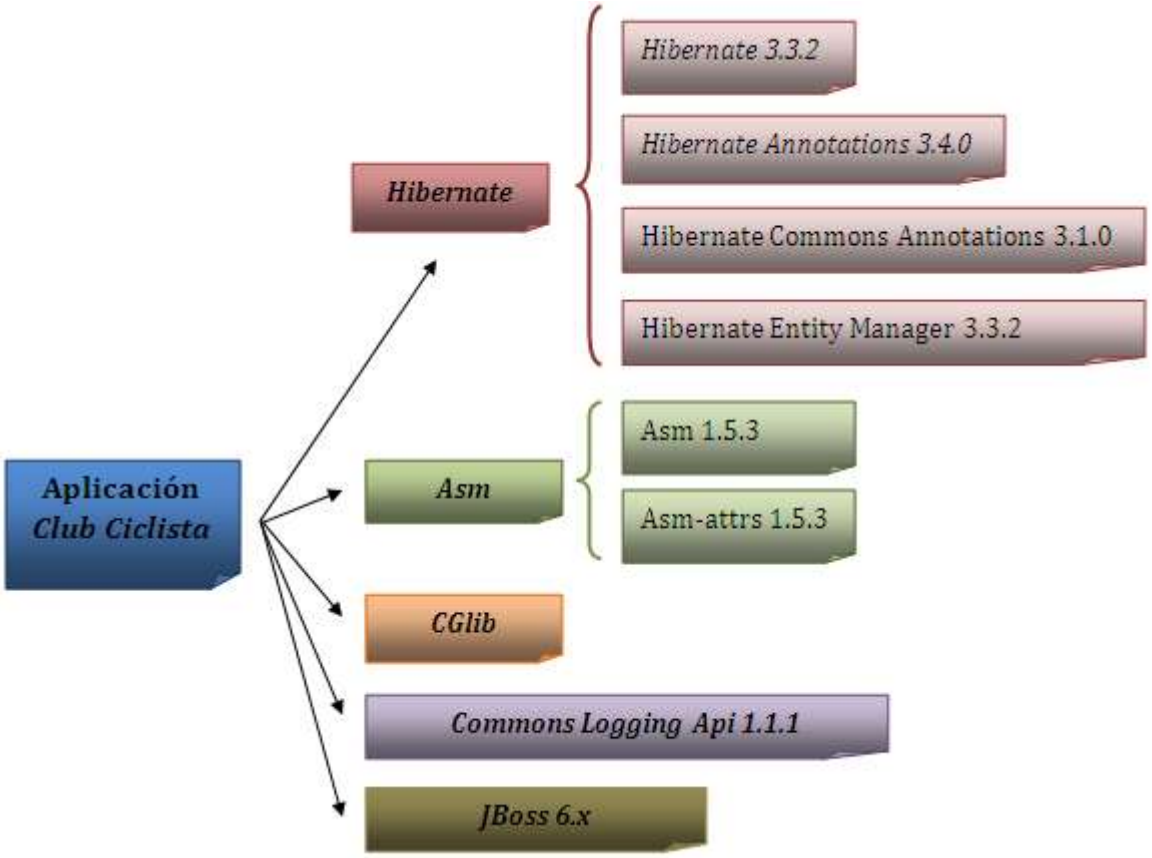


Fig.5.17: *Diagrama de dependencias con librerías externas"*

Las librerías aquí mencionadas serán aquellas que se deberán descargar al Proyecto y añadirlas al directorio */lib* (se encontrarán sus URLs de descarga mencionadas en el apartado 9.2.2.1 "*Librerías a Importar*") y las correspondientes a JBoss nos las incluirá el Servidor de Aplicaciones.

Por otro lado se harán uso de otro conjunto de librerías, pero que en vez de añadirlas al proyecto en sí (en el directorio */lib* como las anteriores), se harán referencia desde el *CLASSPATH.* Estas serán ***log4j.jar***, ***jbossall-client.jar***, ***tools.jar*** (JBoss-6.1.0.Final) y por otro lado el conector a la Base de Datos MySQL (***mysql-connector-java-5.1.22-bin.jar***).
Todos estos detalles y más se explican más adelante en el apartado 9.1 ("*Software y Herramientas empleadas*").

5.8 Posibles evoluciones y ampliaciones

Se abre la puerta posibles mejoras y evoluciones de la Aplicación/Framework, de las cuales podríamos destacar las siguientes:

- ✓ Implementación de ***roles***/control de acceso:

  Una posible evolución podría ser la implementación de roles y diferentes mecanismos, que en función de un perfil u otro, pudiese realizar una determinada serie de acciones o acceder a un conjunto de vistas concreto.

- ✓ Servicio de ***Logs/Trazas***:

  Basándonos en el estándar log4j, es una herramienta muy útil para la depuración de errores en tiempo de desarrollo y de diagnóstico una vez desplegada la aplicación.
  El objetivo sería poder alcanzar un estado en el que se pudiesen configurar en la aplicación dos cosas:

  1. El nivel de log a visualizar (a nivel de información, debugueo, warnings y de error).
  2. El tipo de salida a obtener (por consola, a fichero, por correo, etc.)

- ✓ Crear más mecanismos de ***validación*** en la parte cliente:

  Ya se ha implementado algún mecanismo de validación en el Cliente vía Jquery como prueba de concepto, pero podrían realizarse muchas más validaciones de cara a hacer más robusta la aplicación, y por otro lado, para agilizar todo el proceso de validación de posibles errores en los formularios, etc., sin tener que llegar la petición hasta el servidor.

- ✓ ***Forms dinámicos***:

  Con el fin de poder añadir más flexibilidad en la creación de los formularios, buscar algún método de evitar tener que generar un Form para cada tipo de petición (Action).

- ✓ Implementar ***cacheo*** para la aplicación Web:

Nuestro sistema será "de juguete", pero en la vida real si tuviese mucha relevancia podría estar sometido a miles de peticiones en espacios cortos de tiempo, por lo que sería considerable tener el cuenta sistemas de cacheo, ya no para la parte BackOffice (a nivel de gestión no habrá cargas muy elevadas de peticiones además de que no interesa cachearlas), sino para el FrontEnd, la parte a la que accederían los navegantes.

- ***Jquery/Ajax***:

  Para hacer más "vistosos" y dinámicos los formularios (y también de la mano de la validación de los mismos, comentado en el punto anterior) y para poder realizar peticiones asíncronas al servidor, también se podrían añadir más funcionalidades y mecanismos sirviéndonos de Jquery y Ajax de los que se han añadido a modo concepto.

- ***Web Responsive***:

  Se podría hacer la aplicación *Web Responsive*, adaptándola mediante implementación a otros dispositivos (móvil, tabletas, etc.) modificando la capa cliente/presentación (mediante JS, CSS), de tal forma que la aplicación sea capaz automáticamente de ajustarse a la pantalla independientemente del dispositivo que la contenga.

- ***Sistema RESTful:***

  Cada día se impone más esta tecnología (estilo arquitectónico REST) como medida para hacer más robustas y escalables las aplicaciones. Para nuestro caso, se implementaría una API en la parte Servidor proporcionándonos los servicios ***CRUD*** (*Create*, *Read*, *Update* y *Delete*), de tal forma que desde el Cliente se pudiesen realizar las mismas operaciones (mediante *URIs*) de ahora, pero abstrayéndonos de cómo estuviese implementada la parte Servidor. Esta estrategia sería también muy interesante si quisiéramos exportar nuestra aplicación en dispositivos móviles, consiguiendo una interfaz muy ligera (y mediante el uso de JSON).

- Añadir más ***idiomas*** al servicio de internacionalización (vasco, francés, etc.)

- ✓ Empleo de ***librerías de Tags*** JSP:

  Aunque no ha sido preciso actualmente para la implementación realizada, se abre la puerta para futuros desarrollos el uso de librerías de etiquetas para acceder a los diferentes objetos sin necesidad de hacer uso de código JAVA.

6. Conclusiones

La investigación y desarrollo del presente proyecto nos ha permitido ampliar la visión de la situación actual del mercado en cuando a ***Frameworks de Presentación*** se refiere, y por otro lado, se nos ha brindado la posibilidad de conocer el amplio abanico de herramientas de diseño (denominadas "***patrones***") que se tienen a nuestro alcance y la gran importancia de hacer uso de éstas, de cara a diseñar y modelar una aplicación con unos estándares de calidad.

A lo largo de las diferentes fases del trabajo se ha observado la necesidad real del uso de patrones de diseño que se apoyen en soluciones ya aprobadas, como base para una buena definición arquitectónica de una aplicación y como referencia de buenas prácticas a la hora de desarrollar implementaciones de estas dimensiones.

El estudio de las principales características de los Frameworks analizados del mercado, nos ha proporcionado referencias e ideas que han sido de utilidad de cara al enfoque del nuevo *Framework FUOC*, bien sea a nivel de desarrollo como funcional.

Además, el desarrollo del presente trabajo no sólo se ha centrado en el aprendizaje de las características en cuanto al ámbito de Frameworks se refiere, sino que también nos ha proporcionado conocimientos en otro tipo de responsabilidades, como son la planificación y gestión de proyectos, gestión de recursos y herramientas utilizadas (incluyendo instalación y configuración de las mismas), administración y gestión de bases de datos, gestión de la configuración y despliegues, metodología de pruebas, y por supuesto todo ello acompañado de la implementación bajo el marco de desarrollo Java.

Se ha realizado un especial hincapié en estructurar de forma muy clara cada uno de los diferentes "*engranajes*" de que están compuestos los dos productos, realizando un esfuerzo en proporcionar

la mayor sencillez en los aspectos de configuración y estructuración de los diversos componentes (sin dejar de lado la documentación generada), lo cual redundará en una mayor facilidad de comprensión, evolución y mantenimiento del desarrollo de cara a terceras personas.

Como conclusión se puede afirmar que tanto el Framework como la Aplicación que lo contiene realizan un "*overview*" de todas y cada una de las características más relevantes que han de contener dos productos de esta naturaleza, satisfaciendo positivamente un número importante de funcionalidades y objetivos marcados, si bien cabe decir que, siempre existirán aspectos que podrán ser mejorables bajo la experiencia de un profesional dedicado en este ámbito, y por otro lado también se abren muchas puertas a posibles evoluciones del producto (tanto a nivel de Servidor como de Cliente).

7. Glosario de Términos

- ✓ ***Ajax***: Acrónimo de *Asynchronous JavaScript And XML*, es una técnica de desarrollo web de aplicaciones interactivas que se ejecutan en el navegador manteniendo una comunicación asíncrona con el servidor.

- ✓ ***AOP*** (*Aspect-Oriented Programming*): Es un paradigma de programación orientada a aspectos que pretende conseguir una modularización, encapsulando los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos.

- ✓ ***API*** (*Application Programming Interface*): Conjunto de especificaciones de comunicación entre componentes de software que proporciona una abstracción entre los diferentes niveles de una aplicación.

- ✓ ***Ant:*** Herramienta usada en las fases de implementación, para la realización de diferentes tareas, como son la compilación, despliegues, paquetizados de una aplicación, etc.
  Es un software para procesos de automatización de compilación, similar a Maven pero desarrollado en lenguaje Java. se basa en archivos de configuración XML y clases Java para la realización de las distintas tareas.

- ✓ ***Bean:*** Componente de software reutilizable. Disponen de una interfaz pública para instanciarlos y unos métodos para acceder a los atributos e insertar valores en los atributos.

- ✓ ***Capa de datos***: En el modelo definido J2EE es la encargada de almacenar de forma persistente los datos de la aplicación al SGBD.

- ✓ ***Capa de negocio***: En el modelo definido J2EE es la capa que encapsula toda la lógica del negocio de la aplicación. Una de las funciones más importantes es la de separar la presentación del acceso a los datos.

- ✓ ***Capa de persistencia***: Tercera capa del modelo J2EE, donde se guardan los datos persistentes, enviados/ordenados desde la capa de negocio.

- ✓ ***Capa de presentación***: En el modelo J2EE, es también conocida como la capa Web de una aplicación. Su principal función es separar el interfaz con el cliente de la lógica de negocio y servir de puente entra ambas.

- ✓ ***Controller***: Elemento del patrón Modelo Vista controlador que responde a los eventos (normalmente acciones del usuario) y que hace las correspondientes peticiones al modelo y a vistas determinadas.

- ✓ ***Core J2EE Patterns***: Conjunto de patrones de diseño y arquitectónicos para los objetivos de desarrollo comunes en J2EE. Esta organizado en patrones de la capa de presentación, patrones en la capa de negocio y patrones de la capa de integración.

- ✓ ***CSS*** (*Cascading Style Sheets*): Lenguaje de programación utilizado para definir el estilo de un documento html.

- ✓ ***DAO*** *(Data Access Object)**:*** Patrón de diseño utilizado para la capa de persistencia realizando la abstracción y encapsulación de los métodos de acceso a los datos.

- ✓ ***Dispatcher***: Componente que se encarga de dar/gestionar la vista cuando hace una petición al controlador en un patrón modelo vista controlador.

- ✓ ***EJB*** (*Enterprise Java Bean*): Modelo de componentes del lado del servidor para desarrollar la lógica de negocio en aplicaciones distribuidas solventando la problemática de gestión de la concurrencia, transacciones, seguridad.

- ✓ ***Framework***: Software que aporta una arquitectura estandarizada para ser aprovechado por otras aplicaciones, resolviendo una serie de problemáticas, como puede ser la implementación del patrón MVC. Consiste en un diseño reutilizable o un subsistema que proporciona ayudas o implementaciones a ciertos casos de uso comunes en los programas.

- ✓ ***Grails***: Framework para aplicaciones web libre desarrollado sobre el lenguaje de

programación *Groovy* (basado en Java). Grails pretende ser un marco de trabajo altamente productivo siguiendo paradigmas tales como convención sobre configuración o no te repitas (DRY) proporcionando un entorno de desarrollo estandarizado y ocultando gran parte de los detalles de configuración al programador.

- ✓ ***Hibernate***: Herramienta de mapeo objeto relacional (*ORM*) para la JAVA, que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (*XML*) o anotaciones en los *beans* de las entidades que permiten establecer estas relaciones.

- ✓ ***Internacionalización*** (*i18n*): Proceso para desarrollar software que pueda adaptarse a diferentes idiomas y regiones sin necesidad de hacer cambios de ingeniería o de código. La Internacionalización también es conocida como "i18n", que es la abreviatura de Internacionalización.

- ✓ ***Inyección de Dependencias***: Patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase quien se encargue de crear tal objeto.

- ✓ ***IOC*** (*Inversion of Control*): Principio de la programación orientada a objetos por el que se reduce el acoplamiento inherente en los desarrollos. El control se pasa de la aplicación al Framework.

- ✓ ***JAR*** (*Java Archive)*. Es un contenedor de recursos de Java para distribuir varios recursos, como clases Java u otros ficheros. Para el caso actual, el Framework FUOC se distribuirá en el archivo con este formato.

- ✓ ***JBoss***: Servidor de Aplicaciones J2EE de código abierto implementado en Java. JBoss puede ser utilizado en cualquier sistema operativo para el que esté disponible Java.

- ✓ ***JDK*** (*Java Development Kit*). Hace referencia a la especificación Java, que incluye el cliente (*JRE – Java Runtime Environment*) y otras utilidades como el compilador.

- ✓ ***JEE***: También denominado Java EE es una plataforma de programación para desarrollar y ejecutar software escrito con el lenguaje Java con una arquitectura distribuida con niveles, basada en componentes de software, todo ello ejecutándose en un servidor de aplicaciones.

- ✓ ***JNDI*** (*Java Naming and Directory Interface*): Mecanismo que permite al cliente de este servicio buscar objetos y datos mediante su nombre y poder invocarlo posteriormente.

- ✓ ***JSP*** (*Java Server Pages*): Implementación que se ejecuta como servlets pero que permite una aproximación más natural a la creación de contenido estático. También llamadas vistas pues las JSP implementan las vistas del modelo MVC en la capa de presentación.

- ✓ ***Lógica de negocio***: Conjunto de rutinas que realizan entradas de datos, consultas a los mismos, generación de informes y más específicamente todo el procesamiento que se realiza detrás de la aplicación transparente para el usuario.

- ✓ ***MVC***: Abreviatura de Modelo-Vista-Controlador, patrón de arquitectura en ingeniería del software. Se aplica por norma general en todos los desarrollos, separando los datos (Modelo) y los problemas de la interfaz de usuario (Vista), de tal manera que los cambios en la interfaz no afecten el manejo de los datos, y que los datos puedan ser organizados sin cambiar la interfaz de usuario. MVC soluciona este problema desacoplando el acceso a los datos y la lógica de negocio de la presentación.

- ✓ ***MVC2 (MVC Model 2)***: Término desarrollado por *Sun* para describir una arquitectura web basada en peticiones HTTP en el que se pasa la petición de un cliente a un *Servlet* controlador que actualiza el modelo e invoca posteriormente a la correspondiente vista.

- ✓ ***Patrón***: Solución probada y reconocida para una problemática concreta de la ingeniería del software u otras áreas de ingeniería. Un patrón de diseño es una descripción de cómo resolver un problema que se pueda utilizar en muchas y diferentes situaciones.

- ✓ ***Request***: Es cada una de las solicitudes que se envían desde el navegador del usuario al servidor de aplicaciones para que realice una determinada acción. Petición que implemente de *HttpServletRequest*.

- ✓ ***Response***: Encapsula la información de la respuesta HTTP de una operación. Implementa *HttpServletResponse*.

- ✓ ***Servlet***: Clase Java ejecutada por un servidor de aplicaciones y que responde a invocaciones HTTP, sirviendo páginas dinámicas. Recibe invocaciones y genera respuestas en función de los datos de la invocación, del estado del propio sistema y los datos a que pueda acceder. Se ajusta a la implementación de API Java Servlet (Java EE).

- ✓ ***Sesión***: En Java la sesión es un objeto que persiste durante toda la secuencias de interacciones de un navegador con la aplicación, y sirve para mantener el estado entre todas las peticiones (request).

- ✓ ***Struts***: Es una herramienta de soporte para el desarrollo de aplicaciones Web bajo el patrón MVC bajo la plataforma Java EE (*Java Enterprise Edition*). Struts se desarrollaba como parte del proyecto Jakarta de la Apache Software Foundation, pero actualmente es un proyecto independiente conocido como Apache Struts.

- ✓ ***Tapestry***: Framework de código abierto para la creación de aplicaciones web de forma

dinámica, robusta y altamente escalable en Java. Tapestry divide una aplicación web en un conjunto de páginas, cada una compuesta de componentes. Lanzado bajo la licencia de Apache Software License 2.0.

- ✓ ***WAR***: Siglas de "*Web Application Archive*". Contenedor de una aplicación web de J2EE, con todos los recursos, como clases Java, librerías, ficheros de configuración, un descriptor de despliegue… necesarios para su despliegue en un servidor de aplicaciones. Para nuestro caso, la aplicación Club Ciclista se distribuirá en un fichero con este formato.

8. Bibliografía

Bibliografía:

- ✓ Deepak Alur, John Crupi and Dan Malks. (2003). "*Core J2EE Patterns: Best Practices and Design Strategies".* (http://corej2eepatterns.com/index.htm).
- ✓ Martín Sierra, Antonio (2007). "*Struts. 2ºEdición*". (http://es.scribd.com/doc/70705402/94/COMPONENTES-DE-STRUTS-2).
- ✓ Scribd Documents (2009). "*Struts2*". (http://es.scribd.com/doc/74281388/TEMA-8-2-Struts2).
- ✓ Casanovas, Josep (2004). "*Usabilidad y Arquitectura Software*". (http://www.alzado.org/articulo.php?id_art=355).
- ✓ Canarias, Iker (2012). "*Frameworks J2EE*". (http://www.slideshare.net/ikercanarias/Frameworks-j2ee).
- ✓ Gamma, Erich & Johnson Ralph (2010). "*Design Patterns CD*". (https://www.box.com/shared/kgqyooaboh).
- ✓ Cheriton School of Computer Science (2011). "*Gang of Four. OO Design Patterns*". (https://cs.uwaterloo.ca/~a78khan/cs446/lectures/2011_05-may_11_DesignPatterns_01.pdf).
- ✓ Moreno, A. y Sánchez, M. "*Patrones de Usabilidad*". (http://www.willydev.net/descargas/prev/PatronesUsa.pdf).
- ✓ Raible, Matt. (2010). "*Web Frameworks Comparison*". (http://es.scribd.com/doc/50467194/3/Pros-and-Cons).
- ✓ Wikipedia. "*Comparison of Web Application Frameworks*". (http://en.wikipedia.org/wiki/Comparison_of_web_application_Frameworks).
- ✓ Martir Fowler (2004). "*Inversion of Control Containers & the Dependency Inyection Pattern*". (http://martinfowler.com/articles/injection.html).
- ✓ Barrios, Juan Manuel. "*Estudio Componentes J2EE*". (*http://users.dcc.uchile.cl/~jbarrios/J2EE/node22.html*).
- ✓ Dpto. Tecnologías de la Información (Facultad Informática A Coruña). "*Aplicaciones J2EE y Patrones Arquitectónicos*". (http://www.tic.udc.es/~fbellas/teaching/is-2001-2002/Tema1.pdf)
- ✓ Escuela Politécnica Superior (UAM). "*Patrones de Diseño*". (http://astreo.ii.uam.es/~jlara/TACCII/8_Patrones.pdf).
- ✓ Source Making. Teaching IT Professionals. "*Design Patterns Simply*". (http://sourcemaking.com/design_patterns).
- ✓ Mestras Pavón, Juan (Universidad Complutense Madrid). "*Patrón Modelo Vista Controlador –MVC-".* (http://www.fdi.ucm.es/profesor/jpavon/poo/2.14.MVC.pdf).

- ✓ M. Keith Mortensen. "*ASP.NET & Struts: Web Application Architectures*". (http://msdn.microsoft.com/en-us/library/aa478961.aspx).
- ✓ Torrijos, Ricard Lou. "*Patrones de Diseño J2EE. Capas de Negocio e Integración*". (http://www.programacion.com/articulo/catalogo_de_patrones_de_diseno_j2ee_y_ii:_capas_de_negocio_y_de_integracion_243/xinha_editor.html/).
- ✓ Spring Source Community. *Spring.* (http://www.springsource.org/).
- ✓ Apache Software Foundation. *Struts2*. (http://www.springsource.org/).
- ✓ Apache Tapestry. *Tapestry 5*. (http://tapestry.apache.org/).

9. Anexos

9.1 Software y herramientas empleadas

De cara a la elaboración de la Memoria y la Implementación del Proyecto se han empleado las siguientes herramientas, añadiendo (donde proceda) las URLs de donde se puede descargar las versiones necesarias para su instalación:

9.1.1 Herramientas para la Documentación

- Elaboración de la Planificación Inicial: *Smartsheet* (https://www.smartsheet.com/)
- Documentación de la Memoria Final: *Microsoft Word/Excel (2007).*
- Elaboración de la Presentación: *Microsoft Power Point (2007).*
- Diagramas UML: *Magic Draw versión. (versión 17.0)*

9.1.2 Herramientas para la Fase de Implementación

9.1.2.1 Descarga de las diferentes herramientas:

- ***Java JDK***, (Java Platform, S.E.: Java SE 6 Update 37)
  - *URL* de Descarga: Descarga Java6 *(Oracle)*
- ***Ant***, versión 1.8.4:
  - *URL* del site de Ant: http://ant.apache.org/
  - *URL* de Descarga Directa: Ant 1.8.4
- Base de Datos ***MySQL***:
  - *URL* MySQL: http://dev.mysql.com/downloads/mysql/5.5.html
    - URL de Descarga Directa: MySQL 5.5.28 win32
  - ***MySQL Workbench*** versión 5.2 CE:
    - URL Descargas Workbench: http://www.mysql.com/downloads/workbench

- URL de Descarga Directa: MySQL Workbench5.2_win32

- ***Conector Java MySQL*** versión 5.1.22:
  - URLDescargasConnectors: http://dev.mysql.com/downloads/connector/j/5.1.html
  - URL de Descarga Directa: MySqlConnnector-5.1.22.zip

- Servidor de Aplicaciones ***JBoss***, versión 6.1.0:
  - URL de Descargas JBoss: http://www.jboss.org/jbossas/downloads
  - URL de Descarga directa: JBoss 6-1-0.Final-win32.zip

- Entorno de Desarrollo: ***Eclipse*** *IDE for Java EE Developers Indigo SR2*
  - URL de Descargas de Eclipse: http://www.eclipse.org/downloads
  - URL de Descarga directa: Eclipse-jee-indigo-SR2_win32.zip

✓ ***JBoss AS Tools*** versión 3.3.0 (Integración de Eclipse con el Servidor de Aplicaciones JBoss):

- Descarga e Instalación:

  Dentro de Eclipse → Eclipse Marketplace → JBoss Tools (Indigo) → "*JBoss Tools 3.3.x*".

- URLs de Documentación e Información:
  - http://www.jboss.org/tools/download.html
  - http://docs.jboss.org/tools/3.3.0.Final/

✓ ***Hibernate***, versión 3.6.10.Final:

- URL de Descargas Hibernate: http://sourceforge.net/projects/hibernate/files/hibernate3/
- URL de Descarga Directa: 3.6.10.Final

9.1.2.2 Configuración de las Herramientas instaladas

Para la fase de implementación, las descargas de las diferentes herramientas y sus correspondientes versiones se detallan en el apartado anterior, y en este apartado se destacarán las configuraciones básicas que se han de realizar para poder implementar y ejecutar la aplicación:

Configuración de las Variables de Entorno y librerías:

Para la ejecución será necesario configurar las siguientes variables:

JAVA_HOME: Apuntará al directorio en el que está instalado Java (por ejemplo, C:\Archivos de programa\Java\jdk1.6.0_37). Esta variable es imprescindible para el correcto funcionamiento de Eclipse.

ANT_HOME: Apuntará al directorio donde está instalado Ant (por ejemplo, C:\apache-ant-1.8.4).

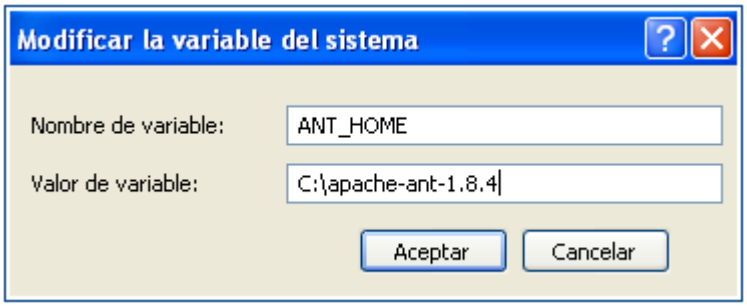


Fig.9.1: *Variable de Entorno ANT_HOME*

JBOSS_HOME: Apuntará al directorio de JBoss (por ejemplo, C:\Jboss-6.1.0.Final):

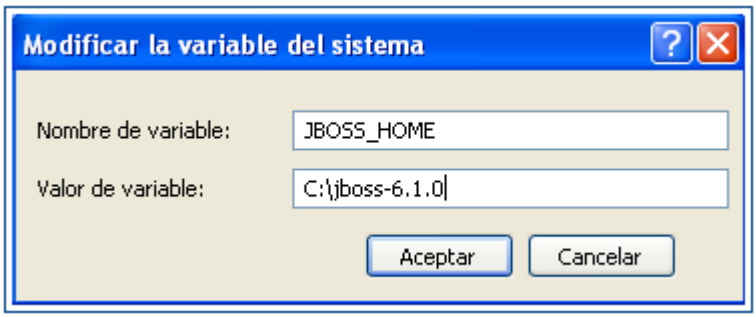


Fig.9.2: *Variable de Entorno JBOSS_HOME*

La variable ***PATH*** deberá apuntar a la dirección correspondiente a las carpetas bin de cada software, por ejemplo:

%JAVA_HOME%\bin; %ANT_HOME%\bin; %JBOSS_HOME%\bin

La variable ***CLASSPATH***, apuntará a los JAR, por ejemplo:

.;%JBOSS_HOME%\client\log4j.jar;%JBOSS_HOME%\client\jbossall-client.jar;C:\mysql-connector-java-5.1.22\mysql-connector-java-5.1.22-bin.jar;%JAVA_HOME%\lib\tools.jar

Como se puede comprobar en la variable de entorno del ***CLASSPATH***, se tendrán en cuenta las librerías ***log4j.jar***, ***jbossall-client.jar, tools.jar*** y ***mysql-connector-java-5.1.22-bin.jar,*** de tal forma que *JBoss* al arrancar las cargará y no se necesitará tener que estar importándolas de forma adicional. Estos tres JAR nos proporcionarán diferentes servicios a nuestras aplicaciones, que explicamos a continuación:

- ✓ ***log4j.jar:***
  - *Utilidad*: Esta librería proporciona servicios de logging. Estos logs son controlados por el fichero conf\log4j.xml, que define un conjunto de entradas, especifica los ficheros log, qué categorías de mensajes acepta, el formato de los mensajes y el nivel de filtro. Hay cuatro niveles de login básicos: DEBUG, INFO, WARN y ERROR.
  - *Ubicación*: Se encontrará la librería en el directorio***: Jboss-6.1.0.Final\client\.***
- ✓ ***jbossall-client.jar:***
  - *Utilidad:* Guarda configuraciones y ficheros JAR que son necesarios para una aplicación cliente Java o para un contenedor web externo.
  - *Ubicación:* Se encontrará la librería en el directorio: ***Jboss-6.1.0.Final\client\.***
- ✓ ***tools.jar:***
  - *Utilidad:* Librería que proporciona acceso del contenedor de servlets al JDK completo, principalmente el compilador de Java.
- ✓ ***mysql-connector-java-5.1.22-bin.jar:***
  - *Utilidad:* Este es el conector para MySQL que se ha descargado previamente como se ha señalado en al apartado anterior.
  - *Ubicación:* Éste se ha descargado en el apartado anterior y se descomprimirá en el directorio, por ejemplo: ***C:\mysql-connector-java-5.1.22*** (acorde con la ruta anteriormente puesta en la definición del CLASSPATH).

    De cara a que los componentes que se ejecutan dentro del servidor de aplicaciones también puedan acceder al conector JDBC para MySQL, es necesario que copiar también el JAR en el directorio JBOSS_HOME***\server\default\lib*** y en el directorio JBOSS_HOME\lib.

Nótese que el ***CLASSPATH*** también incluye el directorio actual ('.'). Con algunas configuraciones del sistema operativo y JDK esto no sería necesario; en otras, sí. Por lo tanto, se incluirá para evitar posibles problemas.

9.1.2.3 Creación de la Base de Datos y Configuración del recurso JNDI

Creación de la Base de Datos:

En primer lugar se creará la Base de Datos, sus estructuras y datos de muestra con los que partir en la aplicación. Para ello se contará un fichero SQL que nos creará tales elementos:

- ✓ Ubicación del Fichero: ***\ClubCiclistaUOC\BaseDatos\import_inicial.sql***
- ✓ Descripción:
  - Se crea el SCHEMA "*clubciclista*".
  - Se crea un usuario Administrador con todos los permisos (***Usuario:*** *uoc* ***/ Pass:*** *uoc*).
  - Se crean las diferentes tablas con sus restricciones y se rellenan con datos de muestra.

Configuración de acceso al recurso JNDI:

En el Fichero de configuración de ***Hibernate*** (en ***\ClubCiclistaUOC\src\hibernate.cfg.xml***)

Se describen las características necesarias para localizar el recurso JNDI y conectarnos a la BBDD. A destacar las siguientes:

```
<!-- Configuraciones de conexion a la BBDD -->
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost/clubciclista</property>
<property name="connection.username">uoc</property>
<property name="connection.password">uoc</property>
<!-- Dialecto SQL -->
<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
<!-- Recurso JNDI -->
<property name="connection.datasource">java:/clubciclista</property>
.............
```

Fig.9.3: *Fragmento de Hibernate.cfg.xml*

Por otro lado, para que la aplicación encuentre tal recurso, es necesario que lo definamos el ***Datasource*** dentro de JBoss.

Para ello, se debe copiar el fichero **mysql-ds.xml** (antes de arrancar JBoss) que se proporciona ya modificado en: ***/ClubCiclistaUOC/Configuraciones/mysql-ds.xml*** y que se deberá copiar tal cual en el directorio de JBoss ***/server/default/deploy.***

9.2 Configuración y Generación de Entregables

9.2.1 Framework: FUOC Framework

9.2.1.1 Generación del Entregable:

Para la generación del entregable del Framework (*FUOC-Framework-0.1.jar*) se hará uso de la herramienta ya configurada ***Ant,*** y se ha implementado un fichero ***build.xml*** que se encargará de las diferentes tareas de inicialización, copias de ficheros, compilación y generación del JavaDoc y del entregable (en nuestro caso de la librería *JAR*).

Dentro del proceso de compilación y generación del Javadoc, se tendrán en cuenta por tanto aquellas librerías que sean necesarias, recurriendo en algunos casos a las variables de entorno definidas y JBoss:

```
<!-- Variables de Entorno -->
<property environment="env" />
<!-- Librería a generar -->
<property name="libreria" value="FUOC-Framework-0.1" />
. . . . .
<!-- Directorios del Servidor -->
<property name="jboss.home" value="${env.JBOSS_HOME}" />
<property name="jboss.lib"  value="${jboss.home}/lib" />
. . . . . .
<target name="all" depends="clean, init, compile, javadoc, dist"/>
. . . . . .
```

Fig.9.4: *Fragmento de build.xml (Framework)*

Para lanzar la ejecución de Ant, nos dirigiremos al Proyecto de Eclipse y con el botón derecho se hará sobre el fichero build.xml: *Run → Ant build.*

El entregable final (JAR) se encontrará generado en la siguiente ubicación:

\FUOC-Framework\dist\lib\FUOC-Framework-0.1.jar

9.2.2 Aplicación: ClubCiclista UOC

De cara a la configuración, ya se ha mencionado en diferentes sitios del documento, y es que la configuración para el arranque de la Aplicación girará principalmente en torno a los siguientes ficheros:

- ✓ Fichero ***descriptor de la Implementación*** (/ClubCiclistaUOC/WebContent/WEB-INF/***web.xml***):

  Contendrá los parámetros básicos de arranque de la aplicación, referencia al ServletControlador y a los ficheros *Properties* configFile y actionFile.

- ✓ Fichero de ***definición de Acciones*** (/ClubCiclistaUOC/src/***actionsFile.properties***):

  Contiene toda la información sobre las acciones a ejecutar de la Aplicación y los Formularios, Vistas, Acciones y Ámbitos asociados a cada una de ellas.

- ✓ Fichero de ***definición de Configuraciones*** (/ClubCiclistaUOC/src/***configFile.properties***):

  Mantiene los parámetros iniciales a tener en cuenta en la aplicación, como puede ser el idioma (español, catalán o inglés), la activación del "modo debug", etc.

- ✓ Fichero de ***configuración de Hibernate*** (/ClubCiclistaUOC/src/***hibernate.cfg.xml***).

- ✓ Fichero de ***propiedades Log4j*** (/ClubCiclistaUOC/src/***log4j.properties***).

- ✓ Fichero con definición del ***DataSource*** de conexión a la Base de Datos (en el directorio de ***JBoss*** /server/default/deploy/***mysql-ds.xml***)

9.2.2.1 Librerías a importar:

Para la implementación de la aplicación *Club Ciclista UOC* será precisa la importación de la siguiente serie de librerías, que son las siguientes:

- ✓ ***FUOC-Framework-0.1.jar*** → Librería generada por el Framework.

- ✓ ***log4j-1.2.16.jar***:

  - *Utilidad*: Librería necesaria para la generación de Logs. *(versión 1.2.16).*

  - URL Descarga: http://www.java2s.com/Code/Jar/l/Downloadlog4j1216jar.htm

- ✓ ***commons-logging-api-1.1.1.jar:*** *(versión 1.1.1)*
  - *Utilidad*: Librería empleada para la generación de trazas de Log (*Log*, *LogFactory*).
  - *URL*Descarga: http://www.java2s.com/Code/Jar/c/Downloadcommonsloggingapi111jar.htm
- ✓ ***cglib-2.1.3.jar***:
  - *Utilidad*: Biblioteca de generación de código que se utiliza para extender las clases e interfaces Java en tiempo de ejecución. *(versión 2.1.3).*
  - *URL* Documentación: http://cglib.sourceforge.net/
  - *URL* Descarga: http://www.java2s.com/Code/Jar/c/Downloadcglib213jar.htm
- ✓ ***asm-attrs.jar*** *(versión 1.5.3)* y ***asm.jar*** *(versión 1.5.3)*:
  - *Utilidad*: ASM es un marco de manipulación de bytecode Java y análisis. Se puede utilizar para modificar las clases existentes o generar dinámicamente clases, directamente en forma binaria. Permiten montar fácilmente transformaciones complejas y herramientas de análisis de código.
  - *URLs* de Descarga:
    - http://www.java2s.com/Code/Jar/a/Downloadasmattrs153jar.htm *(versión 1.5.3).*
    - http://www.java2s.com/Code/Jar/a/Downloadasm153jar.htm *(versión 1.5.3).*
- ✓ ***hibernate3.jar***: *(versión 3.3.2.GA)*
  - *Utilidad*: Librería de utilidades necesarias para interacción con *Hibernate*.
  - *Descarga*: *http://sourceforge.net/projects/hibernate/files/hibernate3/3.3.2.GA/*

    (Se descomprimirá y extraerá únicamente el fichero *hibernate3.jar*).
- ✓ ***hibernate-annotations.jar:*** *(versión 3.4.0.GA)*
  - *Descarga*: *http://sourceforge.net/projects/hibernate/files/hibernate-annotations/3.4.0.GA/*

    (Se descomprimirá y extraerá únicamente el fichero *hibernate-annotations.jar*).

- ✓ ***hibernate-commons-annotations.jar:*** *(versión 3.1.0.GA):*
  - *Descarga*:*http://sourceforge.net/projects/hibernate/files/hibernate-commons-annotations/3.1.0.GA/*

    (Se descomprimirá y extraerá sólo el fichero *hibernate-commons-annotations.jar*).
- ✓ ***hibernate-entitymanager.jar:*** *(versión 3.3.2.GA):*
  - *Descarga*: http://sourceforge.net/projects/hibernate/files/hibernate-entitymanager/3.3.2.GA/

    (Se descomprimirá y extraerá sólo el fichero *hibernate-entitymanager.jar*).
- ✓ El resto de posibles librerías básicas y componentes necesarios para ejecutar la aplicación las proporcionará de forma implícita el mismo ***JBoss*** en su instalación.

9.2.2.2 Generación del Entregable:

Para la generación del entregable de la aplicación (*clubciclista.war*) se hará uso del mismo modo de la herramienta ***Ant*** cuyo fichero ***build.xml*** que se encargará de las diferentes tareas de inicialización, copias de ficheros, compilación, generación del JavaDoc y del entregable (en nuestro caso el WAR) y además realizará una copia automática al directorio de despliegue del JBoss (tarea ***deploy***, en el directorio de *JBoss/server/default/deploy*):

Dentro del proceso de compilación y generación del Javadoc, se tendrán en cuenta por tanto aquellas librerías que sean necesarias, recurriendo en algunos casos a las variables de entorno definidas y JBoss:

```
<!-- Variables de Entorno -->
<property environment="env" />
<!-- Proyecto a generar -->
<property name="proyecto"     value="clubciclistauoc" />
<property name="package.name" value="${proyecto}.war" />
. . . .
<!-- Directorios del Servidor -->
<property name="jboss.home"   value="${env.JBOSS_HOME}" />
<property name="jboss.lib"    value="${jboss.home}/lib" />
<property name="jboss.deploy" value="${jboss.home}/server/default/deploy"/>
. . . . .
<target name="all" depends="clean, init, compile, package, javadoc, deploy'/>
. . . . .
```

Fig.9.5: *Fragmento de build.xml (Aplicación)*

Para lanzar la ejecución de Ant, nos dirigiremos al Proyecto de Eclipse y con el botón derecho se hará sobre el fichero build.xml: *Run* → *Ant Build.*

El entregable final (WAR) se encontrará generado en la siguiente ubicación:

\ClubCiclistaUOC\dist\clubciclistauoc.war

9.2.3 Estructura general de ambos Proyectos:

Aplicación Web "Club Ciclista"

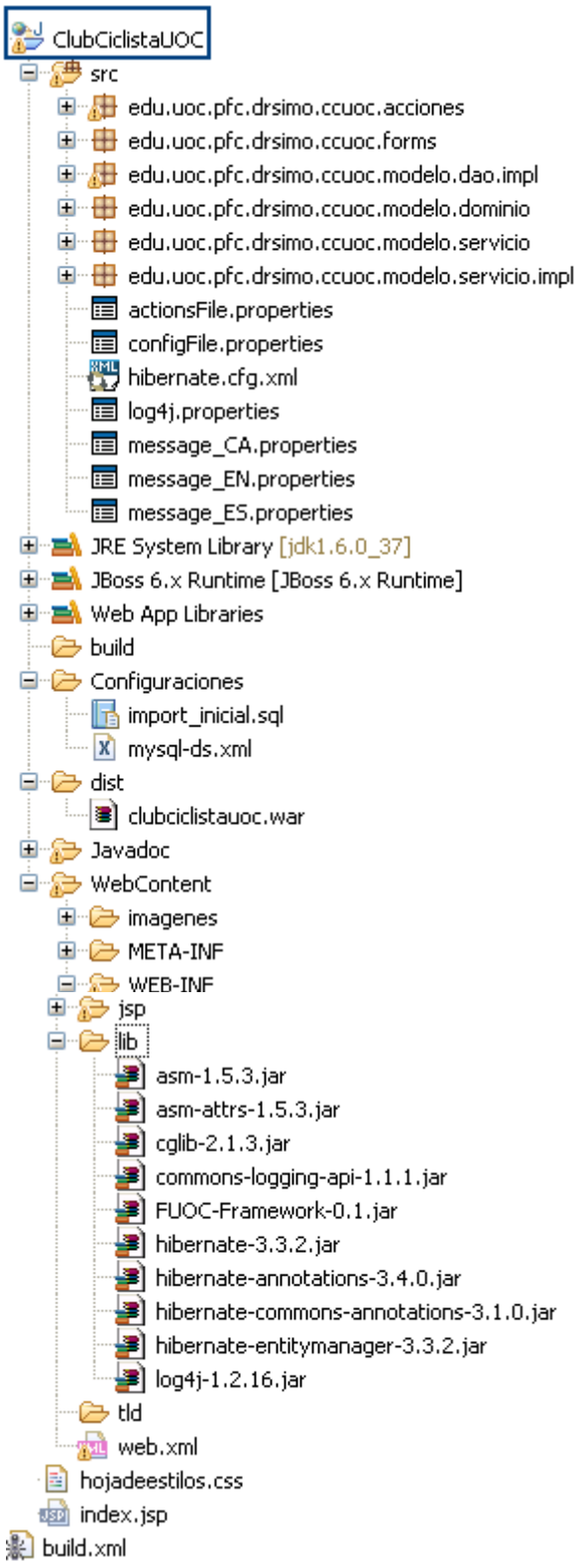


Fig.9.6: *Estructura del Proyecto Club Ciclista*

Framework "FUOC"

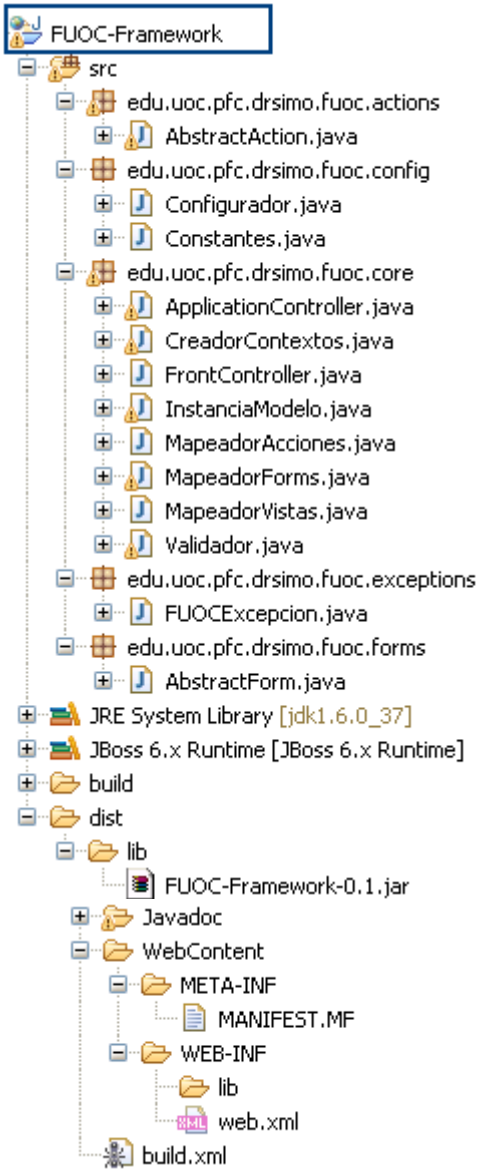


Fig.9.7: *Estructura del Framework FUOC*

9.3 Metodología de Test/Pruebas Software:

Para garantizar la correcta implementación y funcionamiento del sistema (tanto del *Framework FUOC* como de la *Aplicación ClubCiclista*), se ha sometido a un estudio de pruebas, que se dividirá en ***Metodología de Pruebas Funcionales***, ***Plan de Pruebas Funcionales*** y ***Cross Browser Testing***. A continuación se detallan con más profundidad:

9.3.1 Metodología de pruebas funcionales

Para garantizar la correcta implementación y funcionamiento del sistema (tanto del Framework FUOC como de la Aplicación ClubCiclista), él mismo se ha sometido a unas baterías de pruebas funcionales que validan y verifican la correcta funcionalidad del sistema.

La metodología de pruebas utilizadas es propia y se basa en las siguientes fases:

- ✓ *Análisis de pruebas*: Esta actividad consiste en realizar un análisis de la fase de pruebas funcionales de una aplicación con el fin de conocer el nivel de profundidad de la fase de pruebas para las distintas funcionalidades del aplicativo.
- ✓ *Requisitos:* Los requisitos funcionales son especificaciones relacionadas con el funcionamiento del sistema, como interactúa con su entorno y las cuales van a ser su estado y funcionamiento, así como indicar lo que el sistema no debe hacer.
- ✓ *Especificación del plan de pruebas*: El plan de pruebas funcional se define como un conjunto de casos de prueba que permiten verificar que el sistema cumple las necesidades establecidas por el usuario en los requisitos. El plan de pruebas diseñado contendrá tantos casos de prueba como sean necesarios para que toda la funcionalidad del requisito quede cubierta teniendo presente que se deberán documentar caminos principales, caminos secundarios o alternativos y caminos de error funcional.
- ✓ *Ejecución del plan de pruebas*: En esta fase se diseñan los ciclos de pruebas funcionales, que son un conjunto de casos de prueba que se ejecutan a la vez y que tienen un objetivo común.

La nomenclatura que se ha utilizado para definir los casos de pruebas es ***CP-XXX-Breve_descripción***, en donde *XXX* es un código único secuencial que se corresponde con un identificador del caso de prueba.

9.3.2 Plan de pruebas funcionales

9.3.2.1 CP-001_Logado_Exito → RF-01:

Se accede al sistema introduciendo los datos de usuario (login/password) y se verifica que se entra en el sistema correctamente. Debe mostrarse la pantalla de gestión de usuarios.

9.3.2.2 CP-002_Logado_Error → RF-01:

Se verifica que no se entra en el sistema si se introduce unos datos (login/password) erróneos, se debe poder probar todas las combinaciones posibles (usuario erróneo, password erróneo o datos vacíos).

Debe aparecer el mensaje "*FUOC ERROR: El Login o Password introducidos no son correctos.*"

9.3.2.3 CP-003_Listar_socios_Exito → RF-02 (caso de prueba reutilizable):

Pulsando en el enlace "*Listado y gestión de usuarios*" se debe acceder a una página que muestra los datos (identificados, nombre y apellidos) de los usuarios existentes y tres iconos (ver, modificar y borrar).

9.3.2.4 CP-004_Consultar Datos Personales_Exito →RF-03:

Tras ejecutar las acciones indicadas en el caso de prueba CP-002_Listar_socios_Exito, seleccionar un usuario y pulsar en el botón "*Ver*". Se debe comprobar que aparece la información (identificado de usuario, nombre, apellidos, email, etc.) correspondiente al usuario que se ha consultado.

9.3.2.5 CP-005: Creación_Socio_Exito: →RF-04:

Pulsando en el enlace "*Crear un nuevo usuario*" se presenta un formulario de entrada donde el Administrador deberá rellenar los campos obligatorios (nombre, password y login) en el formato indicado y se pulsará en el botón "*Dar de alta*".

Debe aparecer el texto "*El socio se ha creado correctamente ("nombre y apellido" id: XX)".*

9.3.2.6 CP-006: Creación_Socio_Error: → RF-04:

Pulsando en el enlace "Crear un nuevo usuario" se presenta un formulario de entrada donde el Administrador deberá realizar pruebas comprobando que si falta alguno de los datos de prueba obligatorios (login, password, nombre o número de trofeos) o no están en el formato indicado muestra los siguientes mensajes. Se deberán probar todas las posibles combinaciones.

Si el Login/Password/Nombre están vacíos → Mensaje de Error: "*El Login"/"Password"/ "Nombre" no puede estar vacío*".

Si el Email no tiene un formato válido →Error: "*El formato del "Email" ha de ser válido*".

Si el campo Número de Trofeos NO es numérico el error que muestra en la misma pantalla (Validado con JQuery): → "*El campo Número de Trofeos ha de ser un valor numérico*".

9.3.2.7 CP-007: Modificar_Datos_Personales_Exito: →RF-05:

Tras ejecutar las acciones indicadas en el caso de prueba *CP-002_Listar_socios_Exito*, seleccionar un usuario y pulsar en el botón "*Modificar*". Pantalla de gestión de los datos personales de un Socio, donde se podrá modificar cualquiera de sus campos para posteriormente actualizarse en la base de datos.

Debe aparecer el texto "*El socio se ha modificado correctamente ("nombre y apellido" id: XX)".*

9.3.2.8 CP-008: Modificar_Datos_Personales_Error: →RF-05:

Tras ejecutar las acciones indicadas en el caso de prueba CP-002_Listar_socios_Exito, seleccionar un usuario y pulsar en el botón "Modificar". Pantalla de gestión de los datos personales de un Socio, donde se podrá modificar cualquiera de sus campos para posteriormente actualizarse en la base de datos. Se deberán probar todas las posibles combinaciones del juego de datos de entrada para obtener los siguientes mensajes.

Si el Login/Password/Nombre están vacíos → Mensaje de Error: "*El Login"/"Password"/ "Nombre" no puede estar vacío*".

Si el Email no tiene un formato válido →*Error: "El formato del "Email" ha de ser válido".*

Si el campo Número de Trofeos NO es numérico el error que muestra en la misma pantalla (Validado con JQuery): → "*El campo Número de Trofeos ha de ser un valor numérico*".

9.3.2.9 CP-009: Borrar Socio: → RF-06

Tras ejecutar las acciones indicadas en el caso de prueba *CP-002_Listar_socios_Exito*, seleccionar un usuario y pulsar en el botón "*Eliminar*".

Debe aparecer el texto "*El socio se ha borrado correctamente ("id: XX")".*

9.3.3 Cross Browser Testing

La compatibilidad de navegadores es un aspecto de creciente importancia en el diseño web, principalmente por la diversidad cada vez mayor de plataformas desde las cuales se puede navegar por la red. El hecho de que un sitio web se vea de una manera en Firefox y de otra completamente distinta en Safari puede generar problemas.

El objetivo macado ha sido también lograr que la aplicación tenga el aspecto y la funcionalidad deseada sin importar el navegador donde se visualice. Por ello, la aplicación (con el Framework) se han sometido a diferentes pruebas con los principales navegadores en la actualidad, comprobando que la Web es compatible entre todos ellos, algo que se conoce como "*Cross-browser Testing*".

Las Pruebas realizadas se han desarrollado satisfactoriamente con los siguientes navegadores:

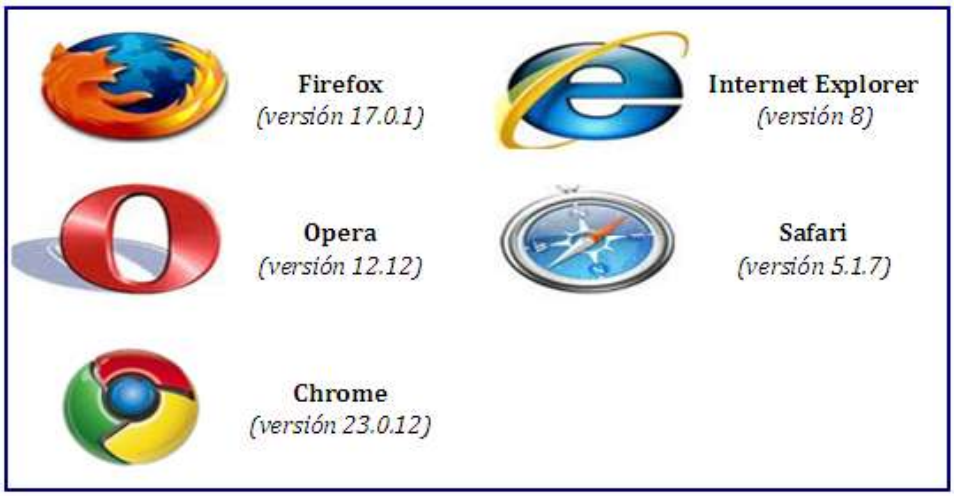


Fig.9.8: *Navegadores empleados para Fase de Pruebas*

9.4 Ejecución de la Aplicación:

Para la ejecución de la aplicación se seguirán los siguientes pasos:

1. Generación del entregable (y Javadoc) del Framework (***FUOC-Framework-0.1-jar***).

   Para ello, dentro del Proyecto del Framework, se ejecutará Ant (Buld.xml: *Run As > Ant Build*).

2. Añadir la librería generada al resto de librerías (en WEB-INF/lib) de la Aplicación Club Ciclista y generar el entregable de la Aplicación (***clubciclistauoc.war***).

   Para ello, dentro del Proyecto del Club Ciclista, se ejecutará Ant (Buld.xml: *Run As > Ant Build*).

   La misma tarea de generación del WAR, lo desplegará automáticamente en el Servidor de Aplicaciones JBoss (en el directorio de JBosss ***/server/default/deploy***).

3. Con JBoss arrancado, el servidor responderá a la siguiente URL de inicio:

   http://localhost:8080/clubciclistauoc/

   Que nos cargará la página de inicio (*index.htm*) que coincidirá con la pantalla de Login al Sistema.

   El login/password que vienen por defecto serán los del Administrador, pero se podrá entrar con cualquiera de los Socios ya dados de alta, por ejemplo ***Login****: usu1*, ***Password****: pass1* (y así hasta 5).

   A partir de ahora y una vez logados, se podrá observar como nuestro nombre de usuario se mantiene en la cabecera (y se "despide" al salir de la aplicación), y la aplicación se cargará en el idioma que hayamos configurado (*configProperties*), que lo indicará la bandera inferior de la pantalla (con tu texto alternativo).

✓ ***Ejemplos:***

Pantalla de con el ***Listado de Usuarios*** (configurado el idioma ***catalán***):

Fig.9.9: *Pantalla Listado de Usuarios (catalán)*

Pantalla de ***Modificación de un Usuario*** existente (configurado el idioma ***inglés***):

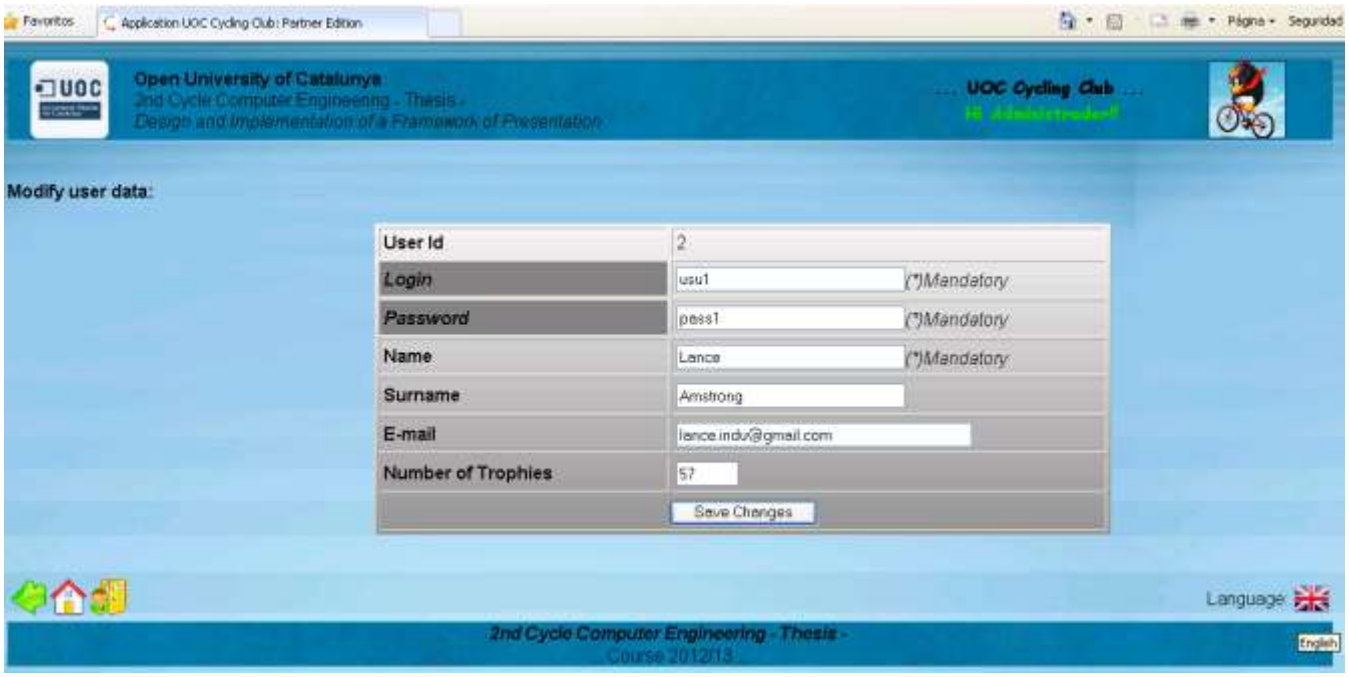


Fig.9.10: *Pantalla de Modificación de Usuario (Inglés)*