Diseño e implementación de un framework de presentación

Alejandro Marmelo Insua

Ingeniería Informática 2º ciclo

Director: Óscar Escudero Sánchez

Universitat Oberta de Catalunya

Barcelona, Enero de 2015

Agradecimientos

Quisiera agradecer el apoyo incondicional de mi familia y amigos desde el momento en el que decidí retomar mis estudios e iniciar el segundo ciclo de Ingeniería Informática.

Agradezco la compresión de toda aquella gente de mí alrededor a la que he descuidado durante estos años y a la que espero poder compensar ahora que este proyecto acaba.

Por último, quisiera dedicar este proyecto a la memoria de mi abuelo, quién siempre supo regalarme palabras de ánimo y apoyo.

Resumen ejecutivo

El presente proyecto consiste en el estudio y desarrollo de un framework de presentación para aplicaciones Web 'Thin Client' desarrolladas en la plataforma Java EE.

El proyecto comprende tanto el estudio de las características generales de los principales frameworks de presentación (prestando especial atención a su arquitectura) como de los patrones de diseño recomendados en la implementación de la capa de presentación en aplicaciones Web.

El resultado obtenido es un framework, bautizado con el nombre de JavaMVC, basado en el patrón arquitectónico Modelo-Vista-Controlador y en el patrón de diseño Service To Worker que proporciona funcionalidades comunes a la mayoría de frameworks de la actualidad. Y, finalmente, el desarrollo de una sencilla agenda telefónica on-line multiusuario, en la que se muestra el comportamiento del framework en una aplicación real.

Índice

Índice de Figuras

Capítulo I – Introducción

1. Descripción del PFC

El proyecto de fin de carrera documentado a lo largo de esta memoria recoge tanto el estudio como el diseño y la implementación de un framework que simplifique y agilice el desarrollo de la capa de presentación en aplicaciones Web Thin Client con tecnología Java EE.

Toda aplicación puede sufrir actualizaciones a lo largo de su vida (nuevas funcionalidades, cambios en su apariencia, etc.) sin embargo, en el caso particular de las aplicaciones Web, una parte significativa de estas actualizaciones se concentra en la capa de presentación. Es por este motivo que, un diseño erróneo o poco afortunado de una aplicación de este tipo, puede traducirse en una mayor dificultad a la hora de mantenerla o evolucionarla en el futuro. Para evitar estas situaciones, las arquitecturas web a lo largo de su historia han evolucionado a modelos cada vez más eficientes (Sin MVC → Arquitectura JSP Modelo-1 → Arquitectura JSP Modelo-2/MVC2/Web MVC → Framework basado en Modelo-2).

A lo largo de esta memoria, se desarrollará un framework basado en la arquitectura JSP Model-2 para implementar el patrón arquitectónico MVC (en su versión adaptada para aplicaciones web), con el objetivo de establecer un marco de trabajo para toda aplicación Web, que simplifique y agilice el proceso de desarrollo de la capa de presentación.

En la actualidad son muchos los frameworks que implementan el patrón arquitectónico MVC adoptando la arquitectura JSP Model-2, por ello, de entre todos los existentes, se hará especial hincapié sobre **Spring MVC**, **Java Server Faces**, **Struts** y **Struts2**.

Finalmente se desarrollará una sencilla agenda on-line que ponga a prueba el framework y demuestre cómo gracias a éste, se consiguen construir aplicaciones web en las que:

- La capa de presentación y negocio estén desacopladas.
- Se simplifique la gestión del flujo de navegación de la aplicación.
- Aumente el nivel de reutilización.
- Se simplifiquen tareas repetitivas.
- Etc.

2. Objetivos generales y específicos

El objetivo general de este proyecto consiste en profundizar en el conocimiento de la plataforma Java EE, así como en las tecnologías y buenas prácticas necesarias para llevar a cabo la implementación de un framework que asegure unos mínimos que garanticen la calidad del producto final.

Objetivos específicos:

- Estudio de las distintas arquitecturas de implementación del patrón arquitectónico MVC en aplicaciones Web.
- Investigación y análisis de los principales frameworks de presentación para aplicaciones Web Java EE.
- Estudio del catálogo de patrones de diseño propuestos por Oracle como buenas prácticas (BluePrints) en el desarrollo de la capa de presentación en aplicaciones web.
- Desarrollo de un framework para la capa de presentación de aplicaciones Web Thin client desarrolladas con tecnología Java EE.
- Desarrollo de una aplicación que muestre claramente el uso del framework.
- Redacción de un documento (memoria) en el que se recoja la información relevante así como los estudios y conclusiones obtenidas a lo largo del proyecto.

Objetivos funcionales:

- Un framework que ofrezca las funcionalidades básicas comunes de los frameworks analizados y facilite el desarrollo de la capa de presentación en aplicaciones Web Java EE.
- Una aplicación web de prueba que sirva para como ejemplo para mostrar el funcionamiento del framework.

3. Planificación del proyecto

El proyecto estará planificado en cuatro entregas tal y como establece el calendario de la asignatura.

Plan de trabajo – PEC1- (17 Septiembre 2014 – 1 Octubre 2014)

- Descripción del PFC.
- Objetivos generales y específicos.
- Planificación de hitos principales.

Análisis y diseño – PEC2 - (2 Octubre 2014 – 5 Noviembre 2014)

- Análisis Tecnología Web Java EE.
- Análisis y documentación de los frameworks **Struts**, **Struts2**, **Spring MVC** y **JSF**.
- Análisis y diseño del framework de presentación.

Implementación – PEC3 – (6 Noviembre 2014 – 19 Diciembre 2014)

- Implementación framework de presentación.
- Desarrollo aplicación de prueba.

Entrega Final (20 Diciembre 2014 – 12 Enero 2015)

- Memoria y Presentación.

4. Diagrama de Gantt

Nombre de tarea	Duración	Comienzo	Fin
Proyecto de fin de carrera	**84 días**	**mié 17/09/14**	**lun 12/01/15**
Plan de trabajo - PEC1	11 días	**mié 17/09/14**	**mié 01/10/14**
Análisis y diseño - PEC2	**25 días**	**jue 02/10/14**	**mié 05/11/14**
Tecnología Web JEE	**2 días**	**jue 02/10/14**	**vie 03/10/14**
Investigación	1 día	jue 02/10/14	jue 02/10/14
Documentación	1 día	vie 03/10/14	vie 03/10/14
Framework Struts	**3 días**	**lun 06/10/14**	mié 08/10/14
Investigación	2 días	lun 06/10/14	mar 07/10/14
Documentación	1 día	mié 08/10/14	mié 08/10/14
Framework Struts 2	**4 días**	**jue 09/10/14**	**mar 14/10/14**
Investigación	3 días	**jue 09/10/14**	lun 13/10/14
Documentacion	1 día	mar 14/10/14	mar 14/10/14
Framework Spring Web MVC	**4 días**	**mié 15/10/14**	lun 20/10/14
Investigación	3 días	mié 15/10/14	vie 17/10/14
Documentación	1 día	lun 20/10/14	lun 20/10/14
Framework JSF	**4 días**	**mar 21/10/14**	vie 24/10/14
Investigación	3 días	mar 21/10/14	jue 23/10/14
Documentación	1 día	vie 24/10/14	vie 24/10/14
A/D Framework presentación	**8 días**	**lun 27/10/14**	**mié 05/11/14**
Análisis	2 días	lun 27/10/14	mar 28/10/14
Diseño	6 días	mié 29/10/14	mié 05/11/14
Entrega análisis y diseño	0 días	**mié 05/11/14**	**mié 05/11/14**
Implementación - PEC3	**32 días**	**jue 06/11/14**	**vie 19/12/14**
Implementación framework	27 días	jue 06/11/14	vie 12/12/14
Desarrollo aplicación de prueba	5 días	lun 15/12/14	vie 19/12/14
Entrega implementación	0 días	**vie 19/12/14**	**vie 19/12/14**
Entrega final	**17 días**	**sáb 20/12/14**	**lun 12/01/15**
Memoria	13 días	**sáb 20/12/14**	mar 06/01/15
Presentación	4 días	mié 07/01/15	lun 12/01/15
Proyecto de fin de carrera entregado	**0 días**	lun 12/01/15	lun 12/01/15

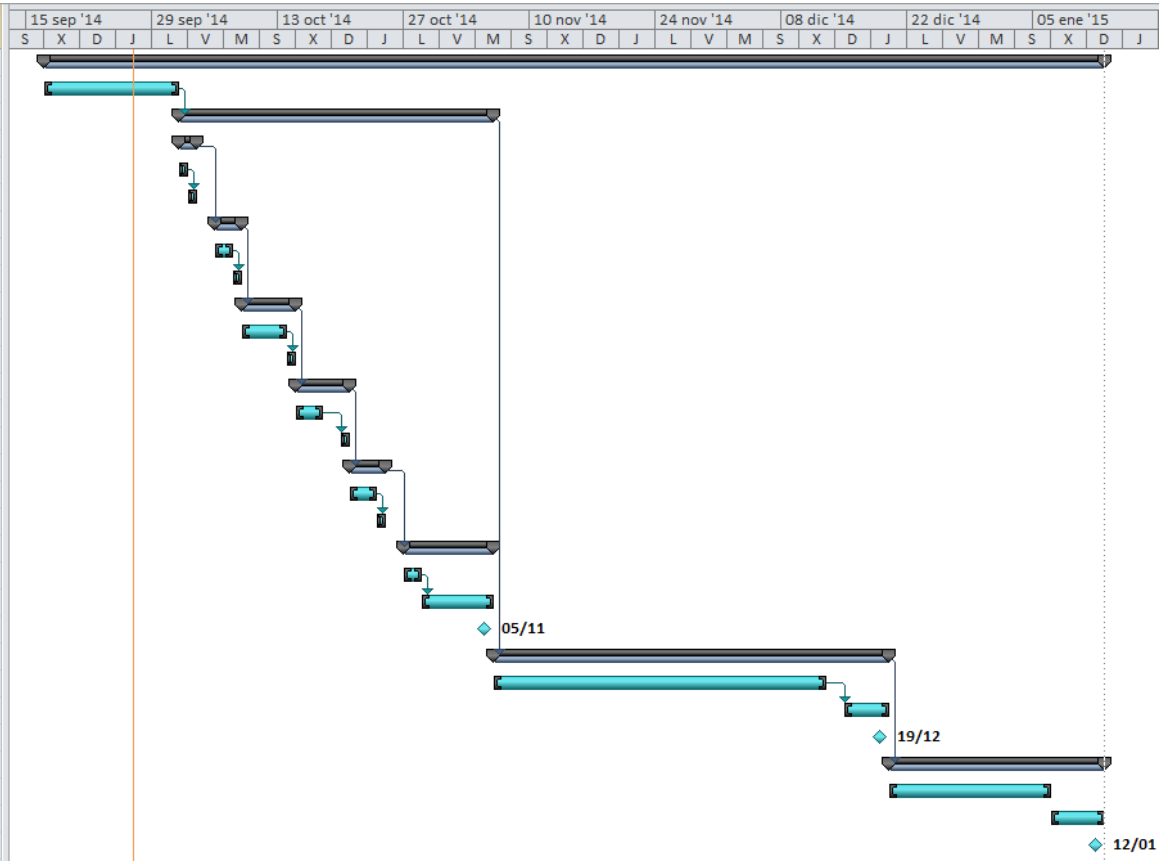

Figura 1 - Diagrama de Gantt

Capitulo II – Estudio previo

Para plantear el desarrollo de un framework de presentación para aplicaciones web en tecnología Java EE es importante conocer aquellos patrones (de diseño, arquitectura, etc.) que pueden resultar útiles a la hora de abordar este tipo de proyectos. Actualmente, el patrón arquitectónico MVC se ha convertido en un estándar de *facto* que multitud de frameworks implementan, con el objetivo de facilitar tanto el desarrollo como el mantenimiento de la capa de presentación.

1. Patrón arquitectónico MVC – Modelo-Vista-Controlador

El patrón MVC es un patrón arquitectónico destinado a separar los datos y la lógica de negocio de una aplicación de su interfaz de usuario. Está indicado especialmente para aquellos aplicativos que disponen de una interfaz gráfica que cambia con frecuencia.

Este patrón arquitectónico persigue la separación de conceptos y la reutilización de código con el objetivo de facilitar tanto las tareas de desarrollo como los posteriores mantenimientos.

El patrón MVC divide el sistema en tres componentes:

- **Modelo**. Contiene los datos, las reglas y los procesos de negocio. Es el encargado de enviar a la **Vista** los datos que ésta le solicite y jamás deberá tener ninguna información sobre la interfaz de usuario. La **Vista** y el **Controlador** acceden al **Modelo**.
- **Vista**. Es la encargada de presentar los datos del **Modelo** en el formato adecuado.
- **Controlador**. La función del **Controlador** es la de recibir las peticiones del usuario e invocar los procesos del **Modelo** necesarios para responder a dichas peticiones. Posteriormente será también el encargado de determinar qué **Vista** presentará los resultados de cada petición. Por ello podría decirse que el **Controlador** actúa como intermediario entre la **Vista** y el **Modelo**.

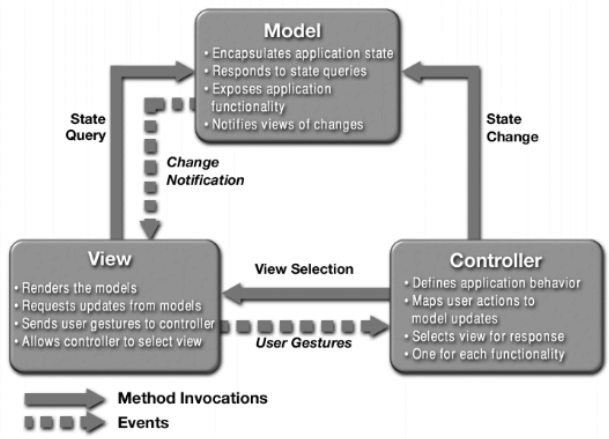

Figura 2 - Estructura del patrón Modelo-Vista-Controlador [Modelo tradicional]

Este patrón, en general, encaja bastante bien en aplicaciones Web, sin embargo, la naturaleza sin estado de las peticiones Web, las limitaciones de las interfaces Web y la distribución de los clientes hacen necesario adaptar este patrón para las aplicaciones Web.

En el patrón MVC tradicional, la **Vista** observa el **Modelo** por lo que pueden actualizar automáticamente los datos que se muestran por pantalla cuando éste cambia. En una aplicación Web, una **Vista** sólo actualizará sus datos si el usuario realiza una nueva petición.

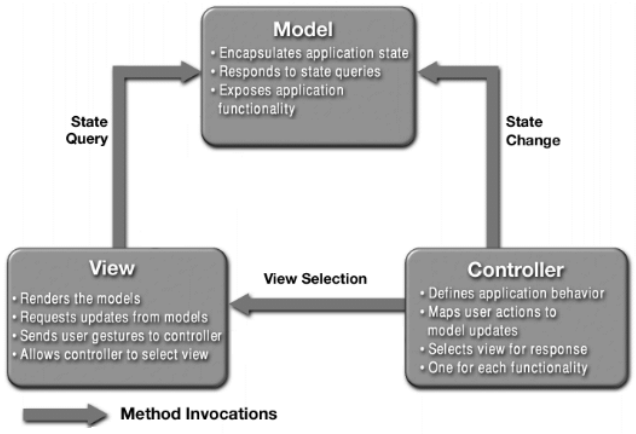

Figura 3 – Estructura del patrón Modelo-Vista-Controlador [adaptado a aplicaciones Web]

Tal y como se aprecia en la figura 3, el patrón MVC debe sufrir ciertos cambios con respecto a la figura 2 para así adaptarse al contexto de las aplicaciones Web. A esta adaptación del patrón MVC tradicional se la conoce como **Web MVC, MVC2 o JSP Modelo 2**.

Dado que esta memoria se focaliza en el desarrollo de un framework de presentación para aplicaciones Web desarrolladas en la plataforma Java EE, es conveniente conocer qué tipos de arquitecturas se utilizan habitualmente a la hora de afrontar este tipo de aplicaciones en esta plataforma.

2. Arquitecturas de implementación en aplicaciones Web en Java EE

Previo al estudio de las arquitecturas de implementación utilizadas en el desarrollo de aplicaciones Web Java EE, es conveniente presentar, a grandes rasgos, tanto las bases de la plataforma sobre la que se implementarán, como las tecnologías en las que se basan para llevarlo a cabo.

2.1. Plataforma Java Enterprise Edition

Java Enterprise Edition es una plataforma de programación (parte de la plataforma Java) utilizada para desarrollar y ejecutar aplicaciones Java. Su objetivo es proporcionar un conjunto de APIs que ayuden a reducir los tiempos de desarrollo (reduciendo la complejidad y mejorando el rendimiento de la aplicación).

Las aplicaciones Java EE se construyen a base de componentes y la especificación Java EE define los siguientes tipos:

- Los clientes de aplicación y los Applets que se ejecutan en la máquina del cliente.
- Los componentes **Java Servlet**, **JavaServer Faces** (JSF), y **JavaServer Pages** (JSP) son ***componentes Web*** que se ejecutan en el servidor.
- Los componentes EJB son ***componentes de negocio*** que se ejecutan en el servidor.

Para que estos componentes puedan ejecutarse, en primer lugar, deben ensamblarse en un módulo Java EE adecuado y posteriormente desplegarse en un contenedor apropiado.

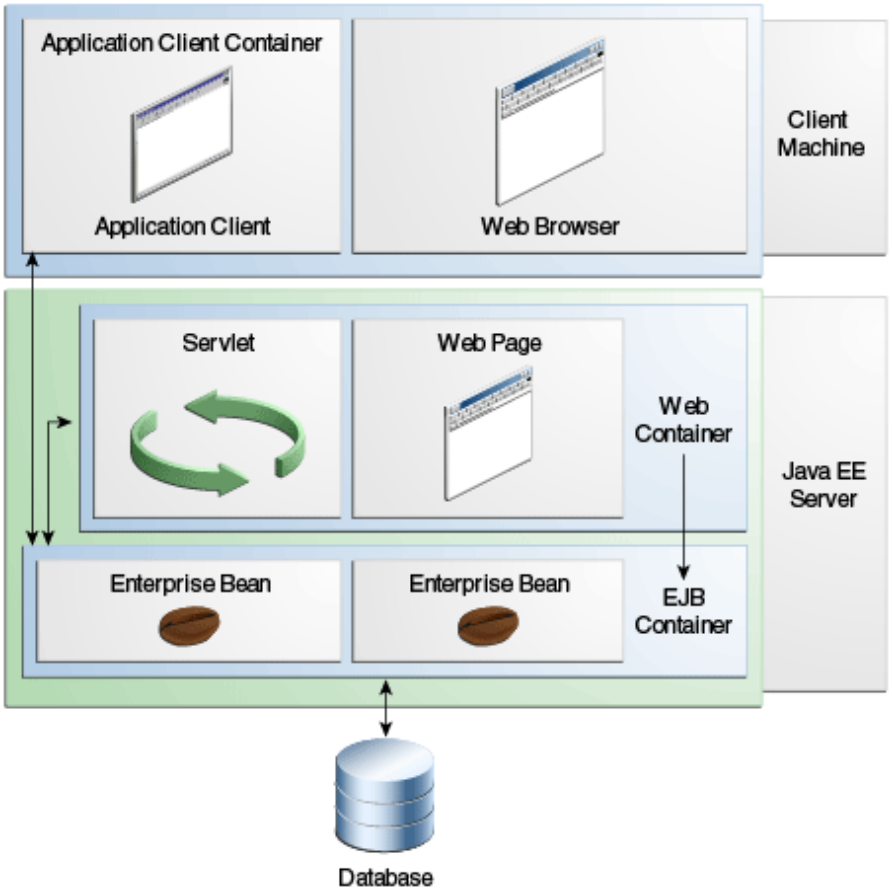

Figura 4 – Servidor Java EE y Contenedores

Un **contenedor** puede ubicarse en la máquina cliente como en el caso de los *clientes de aplicación* y los *Applets* o bien, en un servidor Java EE como es el caso de los *componentes Web* y los *componentes de negocio*.

Un servidor Java EE proporcionará *contenedores Web* y *contenedores EJB* para gestionar la ejecución de los *componentes Web* (páginas Web y Servlets) y de los *componentes EJB* respectivamente.

2.1.1. Componentes Web

Los componentes Web Java EE son **Servlets** o páginas Web creadas utilizando o bien **JavaServer Faces** y/o **JavaServer Pages** (JSP).

- Un **Servlet** es una clase escrita en lenguaje Java que procesa peticiones y construye respuestas.
- Una **JSP** (JavaServer Pages) es un documento de texto que puede estar formado por:
    - Plantilla de datos estática expresada en algún lenguaje de marcas como (X)HTML o XML.
    - Elementos JSP que determinan cómo construirá la página el contenido dinámico.
- La tecnología **JavaServer Faces** se construye sobre **Servlets** y **JSP**s y proporciona un framework de componentes de interfaz de usuario, para aplicaciones Web, que será tratado en mayor profundidad más adelante.

2.1.2. Aplicación Web Java EE

Una aplicación Web es una extensión dinámica de un servidor Web o aplicación. En la plataforma Java EE, es mediante los *componentes Web* que se llevan a cabo estas extensiones dinámicas. Estos componentes, tal y como se ha comentado anteriormente, pueden ser **Servlets**, páginas web implementadas con tecnología **JavaServer Faces**, endpoints de servicios web o páginas **JavaServer Pages** (JSP).

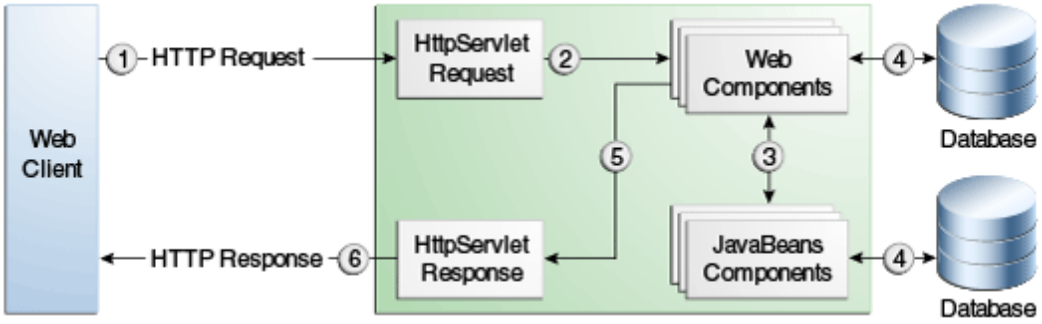

Figura 5 – Ciclo de vida de una petición en una aplicación Web

Tal y como muestra la Figura 5, el ciclo de vida de una petición Web sigue los siguientes pasos:

1. El cliente envía una petición HTTP al servidor Web. El servidor Web que implementa Java Servlet y/o JavaServer Pages convierte la petición en un objeto **HttpServletRequest**.
2. El objeto se entrega al componente web que puede o bien interactuar con componentes JavaBeans (3) o bien con base de datos (4) para generar contenido dinámico.
5. El componente web puede entonces pasar la petición a otro componente web o generar finalmente un objeto **HttpServletResponse**.
6. El servidor web convertirá el objeto **HttpServletResponse** en una respuesta HTTP que devolverá al cliente.

Arquitecturas de implementación en aplicaciones Web Java EE (cont.)

En la introducción a la plataforma Java EE se ha visto que las aplicaciones Web forman parte de lo que se conoce como *componentes Web* y que las principales tecnologías para llevar a cabo estos componentes son los **Servlets** y las **JavaServer Pages** (JSPs).

En el desarrollo de aplicaciones Web Java existen dos arquitecturas de implementación ampliamente utilizadas conocidas como: Arquitectura JSP Modelo-1 y Arquitectura JSP Modelo- 2.

Ambas arquitecturas separan la generación del contenido (lógica de negocio) de la presentación de éste, sin embargo, la diferencia entre ambas radica en el lugar en el que se lleva a cabo el procesamiento de la petición.

2.2. Arquitectura JSP Modelo-1

Esta arquitectura se caracteriza por dar a la JSP la responsabilidad tanto de procesar la petición entrante, como de generar la respuesta al cliente. Puede observarse, por tanto, que hablando en un contexto MVC esta arquitectura separaría **Vista** y **Controlador** llevados a cabo por la JSP, del **Modelo**.

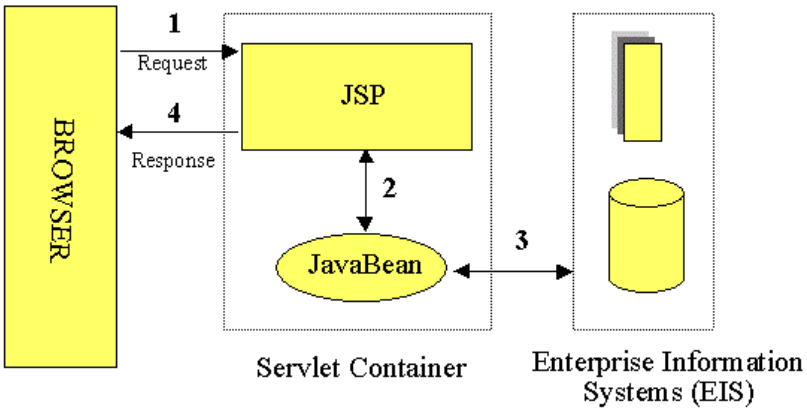

Figura 6 – Arquitectura JSP Modelo 1

Ciclo de vida de una petición

1. El explorador envía una petición a una página JSP.
2. La página JSP se comunica con un JavaBean.
3. El JavaBean se comunica con los sistemas de información necesarios.
4. La respuesta generada por la JSP se envía como respuesta al explorador.

Inconvenientes

Esta arquitectura conduce al uso indiscriminado de scriptlets embebidos en las páginas JSP, complicando con ello la creación y el mantenimiento de las mismas y dificultando la participación de diseñadores en el desarrollo, cosa por otra parte muy habitual en proyectos de cierta envergadura.

Ventajas

Resulta más rápida de implementar que arquitecturas más elaboradas pero, sus posibilidades son tan limitadas que sólo resulta aplicable en aplicaciones muy sencillas.

2.3. Arquitectura JSP Modelo-2 – MVC2 – Web MVC

La arquitectura JSP Modelo 2 consiste en un enfoque híbrido en el que se combina el uso de Servlets y JSPs aprovechando los puntos fuertes que aporta cada una de estas tecnologías. Las JSPs se utilizan para generar la presentación y los Servlets para las tareas de procesado.

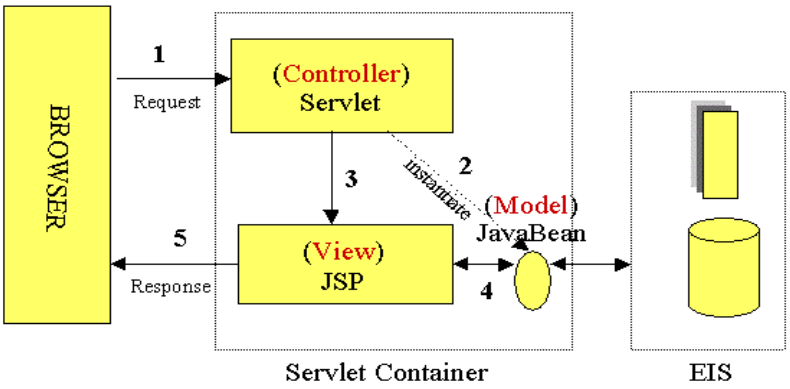

Figura 7 - Arquitectura JSP Modelo 2 - MVC2 – Web MVC

Ciclo de vida de una petición

1. El explorador envía una petición que recibe un Servlet.
2. El Servlet instancia un JavaBean que se comunica con un origen de datos.
3. El Servlet determina qué página JSP debe mostrar el resultado.
4. La página JSP se comunica con el JavaBean para obtener los datos que necesita.
5. La página JSP genera la respuesta que se envía al explorador.

En esta arquitectura ya no hay lógica de procesamiento en las JSPs. Éstas únicamente son responsables de recuperar los datos de los objetos o Beans que hayan sido creados previamente por el Servlet y extraer el contenido dinámico para insertarlo en plantillas estáticas de código (X)HTML. En esta arquitectura el Servlet actúa como **Controlador**, la página JSP como **Vista** y los JavaBeans como el **Modelo**, es por este motivo que a esta arquitectura también se la conoce como MVC2 o Web MVC.

Ventajas

- Resulta sencillo modificar el modelo, la vista y el controlador por separado.
- Facilita la separación de tareas en los equipos.

Inconvenientes

- La división del sistema supone un sobrecosto en la comunicación entre modelo y la vista.

3. Patrones de diseño de la capa de presentación

Una vez introducidas las distintas arquitecturas de implementación en aplicaciones Web Java EE, Oracle proporciona un conjunto de patrones de Diseño que considera buenas prácticas (blueprints) a la hora de afrontar el desarrollo de éste tipo de aplicaciones.

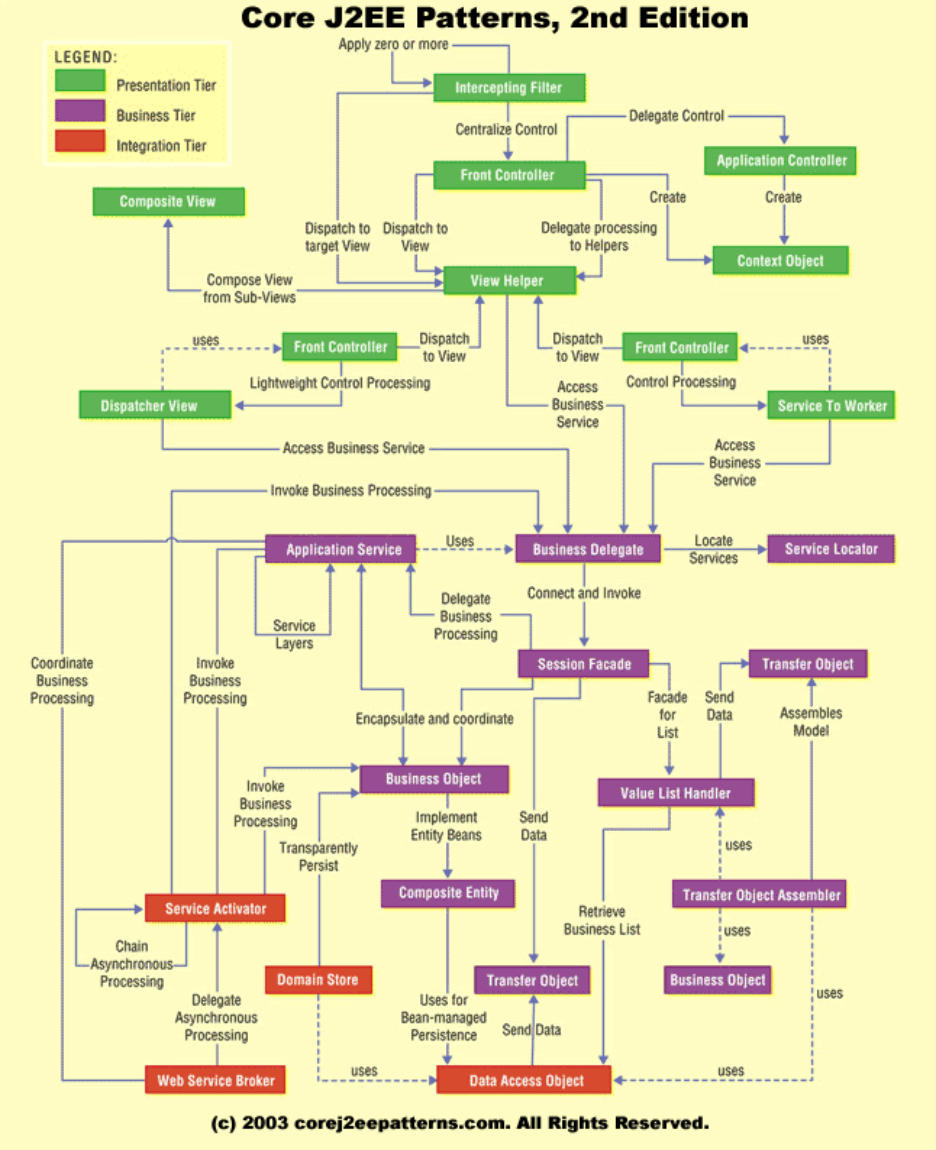

Figura 8 – Catálogo de patrones Java EE

Dado el tipo de framework a desarrollar, el estudio de patrones de diseño se concentrará principalmente en aquellos ubicados en la capa de presentación (Presentation Tier). A continuación se expondrán las características más destacables de aquellos que, tras un estudio preliminar, se ha considerado de mayor relevancia y que con mayor probabilidad formarán parte del diseño final del sistema.

3.1. Patrón Intercepting Filter

El patrón *filtro interceptor* proporciona un mecanismo mediante el cual decorar el tratamiento de las peticiones recibidas con un pre-procesamiento y un post-procesamiento adicional.

El problema

El responsable de manejar las peticiones y las respuestas en la capa de presentación recibe muchos tipos de peticiones y, en algunos casos, estas peticiones requieren un pre-procesado y un post-procesado común:

- Autenticación de usuarios.
- Validación de sesión.
- Compresión / Descompresión de la petición.
- Cifrado / Descifrado de la petición.
- Etc.

Hacer que un único componente lleve a cabo todas estas tareas plantea inconvenientes:

- Escasa reutilización del componente (baja cohesión).
- Código en el que probablemente abundará lógica condicional anidada que dificultará, a posteriori, tanto la adición como la eliminación de tareas.

La solución

La solución pasa por utilizar un mecanismo que permita separar cada tarea en distintos componentes independientes que puedan añadirse o eliminarse fácilmente.

Mediante esta solución, cada petición entrante pasará a través de la cadena de filtros declarados antes de llegar al componente responsable de tratar la petición. Una vez obtenida la respuesta, éste la enviará de vuelta al usuario pero antes pasará nuevamente a través de cada filtro dónde se llevarán a cabo las tareas de post-procesado.

Diagrama de Clases

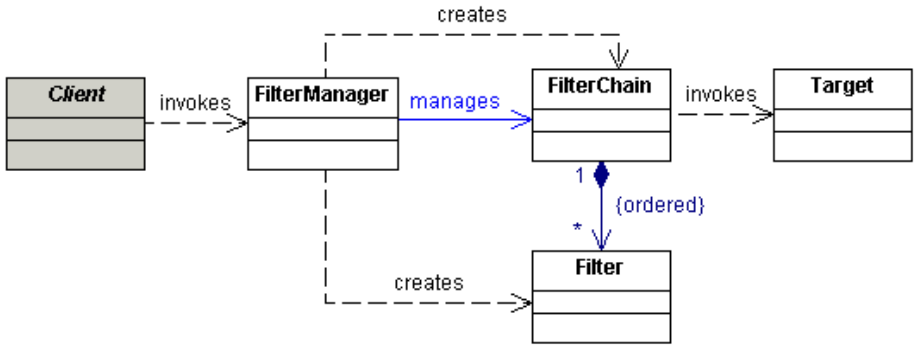

Figura 9 – Diagrama de clases patrón filtro interceptor

Diagrama de secuencia

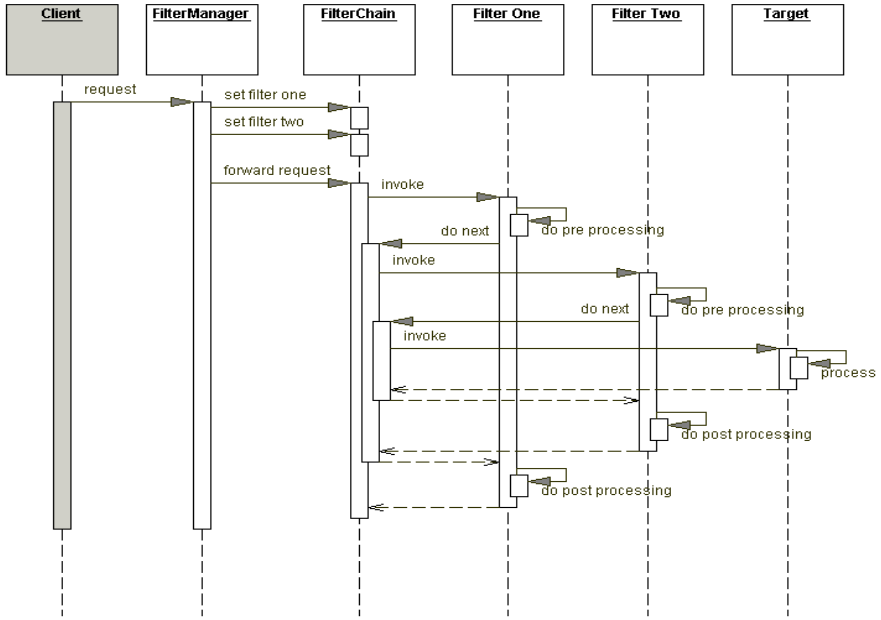

Figura 10 – Diagrama de secuencia del patrón filtro interceptor

Estrategias de implementación

- **Custom Filter**. Necesario en versiones previas a Java Servlets 2.3. Implica la implementación total de la estrategia.
- **Filtro estándar**. Disponible desde Java Servlets 2.3. Implica la implementación de una clase que extiende de la interfaz Filter y su declaración en el fichero de descripción de despliegue de la aplicación (web.xml).
- **Base Filter**. Clase base escrita como una superclase de la que heredan el resto de clases filtro de forma que se aproveche la funcionalidad común implementada.

- **Template Filter**. Mediante el uso de la estrategia Base Filter junto con el patrón ***Template Method*** que permite que la superclase defina la estructura del algoritmo, en base a métodos abstractos que se implementarán en las subclases.

Ventajas

- Los filtros pueden aplicarse a todas las peticiones entrantes sin tener código duplicado.
- Los filtros tienen un bajo acoplamiento por lo que son fácilmente combinables y gracias a su alta cohesión son fácilmente reutilizables entre aplicaciones.
- Los filtros pueden añadirse o eliminarse de un recurso de forma declarativa.

Inconvenientes

- Los filtros son independientes lo que los convierte en ineficientes si es necesario compartir muchos datos entre filtros.

3.2. Patrón Front Controller

El patrón Front Controller proporciona un único punto de entrada a la capa de presentación. Puede manejar la seguridad, validación y control de flujo y ayuda a simplificar los componentes de la vista.

El problema

Los usuarios de un sistema Web envían múltiples peticiones y éstas peticiones requieren cierta lógica de control que determine cómo deben procesarse. Esta lógica de control puede estar dispersa en múltiples componentes o centralizada en un conjunto cerrado. Si las peticiones se envían directamente a una vista sin pasar antes por un controlador centralizado pueden suceder varias cosas:

- Cada vista tendrá que proporcionar sus propios servicios de sistema lo que conllevará a una duplicación de código.
- La navegación se deja en manos de la vista y ello implica mezclar el contenido de la vista con la navegación.
- El control distribuido es más difícil de mantener dado que cualquier cambio implica replicarlo en varios sitios.

La solución

Utilizar un componente **FrontController** que actúe como puerta de entrada a la aplicación Web y gestione el tratamiento inicial de la petición. Las responsabilidades de este componente pueden incluir desde servicios de seguridad hasta la auditoría de peticiones.

Habitualmente el componente **FrontController** delegará la petición al componente **ApplicationController** que determinará qué lógica de negocio llevar a cabo así como la vista que mostrará su resultado. La introducción de un **FrontController** en la arquitectura proporciona un contenedor centralizado que controla y gestiona el manejo de cada petición.

Diagrama de clases

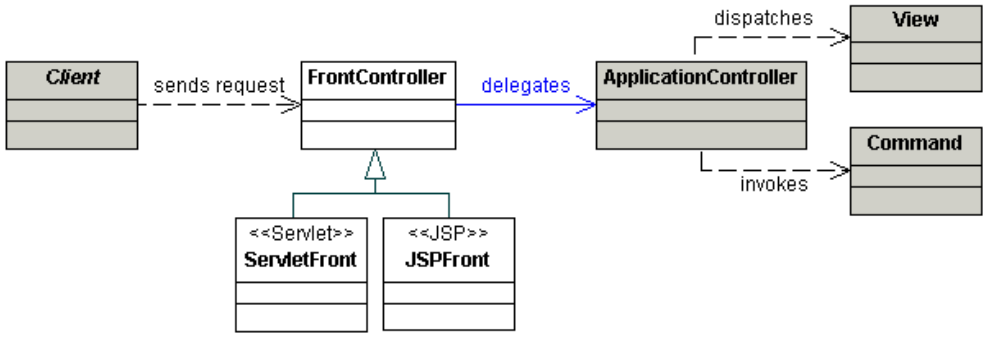

Figura 11 – Diagrama de clases del patrón Front Controller

Diagrama de secuencia

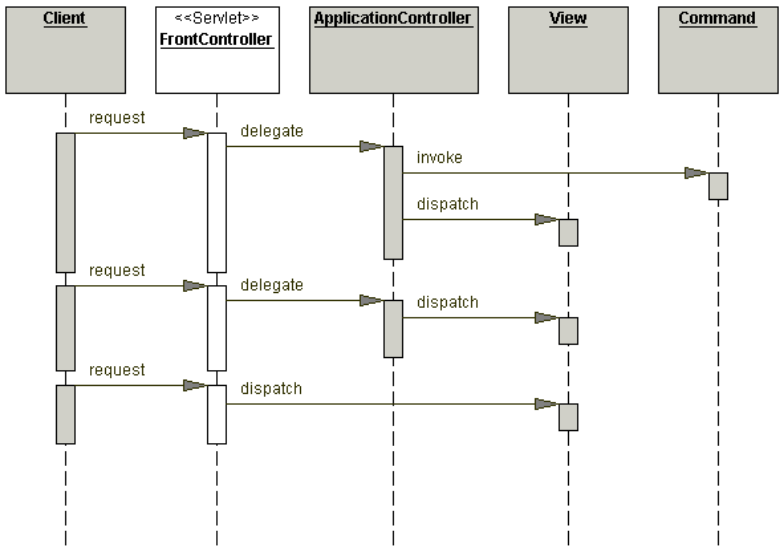

Figura 12 – Diagrama de secuencia del patrón Front Controller

Estrategias de implementación

- **JSP Page Front** – Estrategia que implementa el controlador en una JSP.
- **Servlet Front** - Estrategia que implementa el controlador en un Servlet. Generalmente preferible sobre la anterior.
- **Command & Controller** – Estrategia que sugiere una interfaz genérica para los componentes de ayuda (helpers) mediante la cual el controlador pueda delegar la responsabilidad.
- **Physical Resource Mapping** – Las peticiones se hacen a nombres de recursos específicos en lugar de a nombres lógicos.

- **Logical Resource Mapping** – Las peticiones se hacen a nombres lógicos de recursos en lugar de nombre físicos específicos. Esta estrategia se prefiere sobre la anterior debido a la flexibilidad de proporciona.
- **Multiplexed Resource Mapping** – Se trata de una sub-estrategia de la anterior que permite mapear un conjunto de nombres lógicos con un único recurso físico.
- **Dispatcher en el Controller** – Si el **FrontController** apenas debe despachar peticiones a otros componentes puede plantearse la posibilidad de incluir el **Dispatcher** en el propio componente **FrontController**.
- **Base Front** – Estrategia que sugiere implementar una clase base que otros controladores deberán extender. La clase base puede contener implementaciones comunes por defecto que cada subclase podrá sobrescribir si lo cree conveniente.
- **Filter Controller** – Algunas de las funcionalidades del **FrontController** puede ubicarse en filtros.

Ventajas

- Evita la duplicación de código común de procesamiento de peticiones.
- Proporciona un punto central para el control de flujo en aplicaciones Web.
- Centraliza la gestión de las validaciones y el tratamiento de errores.

Inconvenientes

- Baja cohesión del controlador en función de la implementación del mismo.

3.3. Patrón Application Controller

Patrón que proporciona la forma de separar del componente FrontController la gestión de invocación a las acciones y la gestión de envío a las vistas.

El problema

Los componentes del controlador de la capa de presentación tienen dos responsabilidades principales:

- **Gestión de las acciones**: En función de la petición entrante, el controlador decide qué acción debe ser invocada y la invoca.
- **Gestión de vistas**: En función de la petición entrante y del resultado obtenido de la invocación de la acción, el controlador determina qué vista debe ser generada y pasa la petición a esa vista.

La gestión de las vistas y de las acciones puede ubicarse en el componente **FrontController** sin embargo, ello conlleva a una pérdida de cohesión por parte de éste, además de acoplar el código con la tecnología Servlet (**FrontController** normalmente se implementa mediante un Servlet) lo que dificulta la reutilización y las pruebas.

La solución

- La gestión de las acciones vinculadas con una petición, así como la gestión de las vistas debe ser reutilizable.
- El componente **FrontController** debe mantener una cohesión razonablemente alta.
- Las pruebas sobre la gestión de las vistas y las acciones debe ser posible sin la necesidad de requerir un cliente Web.

Diagrama de clases

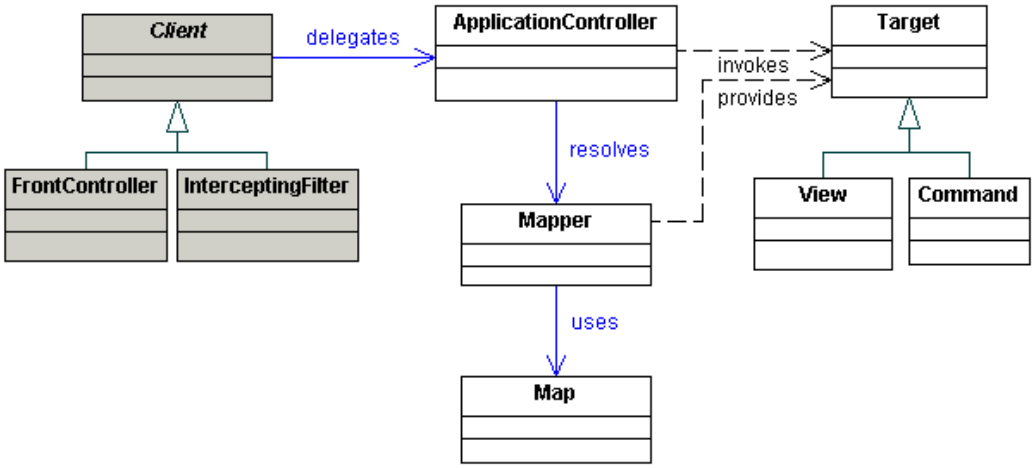

Figura 13 – Diagrama clases del patrón Application Controller

Diagrama de secuencia

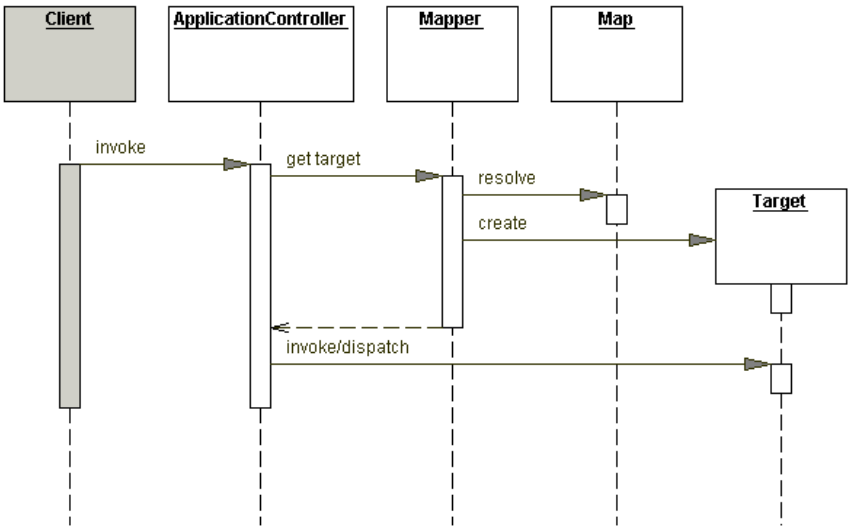

Figura 14 – Diagrama de secuencia del patrón Application Controller

Estrategias de implementación

- **Command Handler** – Un controlador de aplicación que lleva a cabo la gestión de las acciones. Para ello, habitualmente, se utiliza el patrón **Command** y mediante la utilización de un fichero de configuración XML se determina qué acción invocar en función de la URL solicitada.

- **View Handler** – Un controlador de aplicación que lleva a cabo la gestión de las vistas. Normalmente el controlador de la aplicación es tanto **Command Handler** como **View Handler**. Utilizando un fichero de configuración XML, el resultado de una acción puede ser utilizado para determinar a qué vista debe redirigirse la petición.
- **Transform Handler** – En lugar de utilizar JSP como tecnología para la generación de vistas, algunas aplicaciones pueden utilizar XSLT para transformar un XML del modelo en código HTML u otro contenido para mostrarlo.
- **Navigation and Flow Control** – De una forma u otra, el controlador de la aplicación controla el flujo de la aplicación por ello puede contener validaciones que aseguren que ciertas acciones se completan antes de que puedan llevarse a cabo otras.
- **Estrategias basadas en mensajes SOAP y JAX-RPC.**

Ventajas

- Mejora la cohesión separando la gestión de las acciones y la gestión de las vistas del **FrontController**.
- Mejora la reusabilidad manteniendo al margen el protocolo de control.

Inconvenientes

- Introduce un nuevo componente que puede ser innecesario en aplicaciones pequeñas.

3.4. Patrón Context Object

Patrón que permite pasar datos de objetos de un contexto específico a otro, sin pasar esos objetos fuera de su contexto.

El problema

Objetos de un contexto específico como **HttpServletRequest**, a menudo contienen datos que se usan durante el procesamiento de la petición incluso en otros contextos como el de la capa de negocio. Si los objetos específicos de un contexto pasan dentro de clases a otros contextos, la flexibilidad y reutilización de estos se reduce.

La solución

Crear un objeto que encapsule los datos del objeto específico de un contexto. El objeto contexto (**ContextObject**) expone una interfaz genérica, pudiendo mantener una referencia a una clase especifica del protocolo y delegar las peticiones a esta o alternativamente mantener una copia de los datos del objeto específico del protocolo.

El **ContextObject** puede también contener lógica de validación para verificar que los datos sean válidos. Un **ContextObject** puede verse como un **TransferObject**, dado que ambos objetos son similares en cuanto a que son simples estructuras de datos que proporcionan una sencilla interfaz con la que poder acceder a sus datos aunque sean distintos en su motivación.

Diagrama de clases

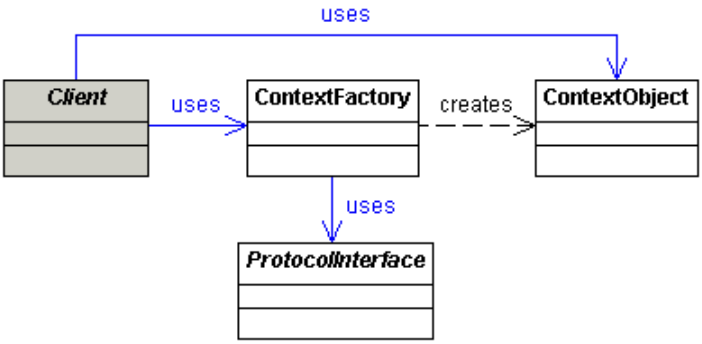

Figura 15 – Diagrama de clases del patrón Context Object

Diagrama de secuencia

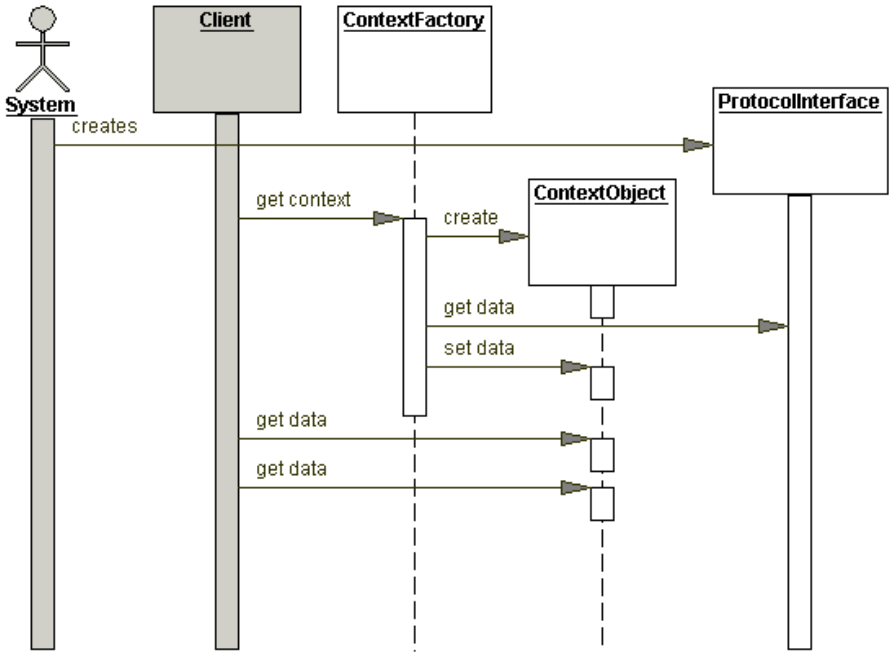

Figura 16 – Diagrama de secuencia del patrón Context Object

Estrategias de implementación

- **Request Context Map** – Los datos del objeto **ServletRequest** se ubican en un objeto Map dentro del **ContextObject**.
- **Request Context POJO** - Se crea un POJO para el objeto **ContextObject** y éste contiene métodos *get* para cada uno de los datos que guarda. El constructor toma como parámetro un objeto **ServletRequest** y puebla las variables de instancia con los datos del objeto **ServletRequest**. Dado que en esta alternativa el **ContextObject** tiene métodos set para datos específicos, será necesario tener diferentes **ContextObject** para cada formulario.
- **Request Context Validation** – Independientemente de la estrategia que se utilice, es muy probable que se requiera lógica de validación a nivel de formulario. Validaciones sencillas que comprueben si todos los datos necesarios han sido informados o si éstos tienen el tipo y el tamaño correcto.
- **Configuration Context** – Un **ContextObject** puede encapsular el estado de la configuración. En algunos casos los **ContextObject** pueden incorporar datos de múltiples protocolos o clases específicas de una capa.

- **Security Context** – El estado de seguridad puede encapsularse en un **ContextObject** llamado en un contexto de seguridad.
- **Context Object Factory** – Usar una factoría para crear objetos **ContextObject** puede permitir controlar la creación y población de objetos en una localización centralizada.
- **Context Object Auto-Population** – En lugar de tener que llamar a métodos para pasar los datos de un objeto a un **ContextObject** de uno en uno, puede utilizarse código que copie todos los datos de una vez.

Ventajas

- Mejora la reutilización de componentes desacoplándola de las clases especificas del protocolo.
- Mejora la capacidad de hacer pruebas dado que los objetos contexto pueden rellenarse con datos de testeo en lugar de con objetos específicos del protocolo.
- Reduce las dependencias con clases específicas de un protocolo que pueden cambiar entre versiones.

Inconvenientes

- Incrementa la cantidad de código que debe ser desarrollado y mantenido.
- Reduce significativamente el rendimiento haciendo copias de datos y añadiendo una capa de indirección.

3.5. Patrón View Helper

El patrón View Helper encapsula "lógica de ayuda" separándola de los componentes de la Vista. Si la vista es una página JSP este patrón minimiza la cantidad de código Java que habrá en la JSP, simplificando el desarrollo y mejorando la mantenibilidad.

El problema

La lógica de negocio y de procesamiento de datos es necesaria para crear contenido dinámico. Los cambios en la capa de presentación ocurren con frecuencia y su desarrollo y mantenimiento se complica cuando la lógica de formateo y de acceso a los datos de negocio se mezclan. Esto implica una pérdida de flexibilidad, reutilización y adaptación al cambio que unido a la falta de modularidad que provoca, implica una pobre separación de roles entre programadores y desarrolladores de contenido web.

La solución

La vista debe focalizarse en presentar el contenido y delegar las responsabilidades de procesamiento a clases de ayuda (**helpers**). Estas clases deberán adaptar el modelo de negocio para los componentes de la vista y llevar a cabo la preparación de los datos y la lógica de formateo. Las clases **Helper** (clase de soporte) pueden implementarse mediante **POJOs**, **JavaBeans** o **Custom Tags** (etiquetas personalizadas) que ayudan a eliminar de la vista toda aquella lógica que no esté directamente relacionada con la presentación.

Diagrama de clases

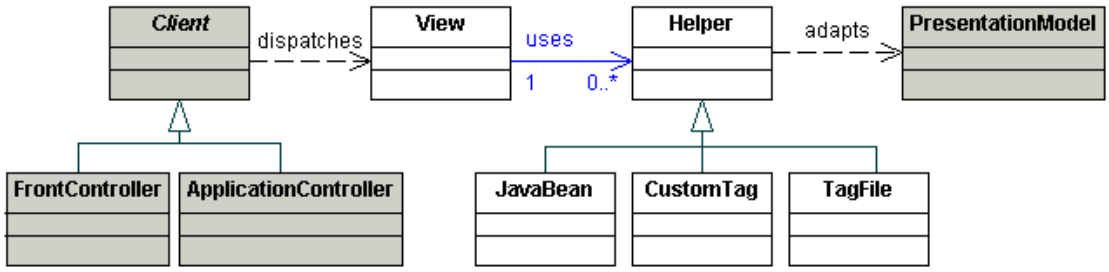

Figura 17 – Diagrama de clases del patrón View Helper

Diagrama de secuencia

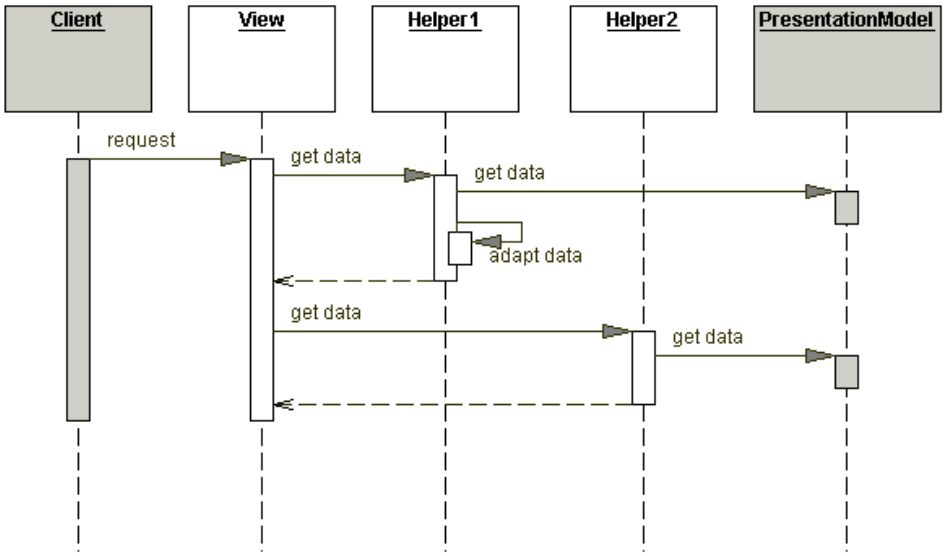

Figura 18 – Diagrama de secuencia del patrón View Helper

Estrategias de implementación

- **Servlet View**- Estrategia que utiliza un **Servlet** como vista. Es una estrategia más engorrosa para los equipos de desarrollo de software y de producción Web ya que embebe código HTML dentro de código Java.
- **JSP Page View** – Estrategia que sugiere la utilización de una JSP como componente vista. Todo y que semánticamente es equivalente a la estrategia de **Servlet View**, es una alternativa más elegante y ampliamente utilizada.
- **JavaBeans Helper** – El componente **ViewHelper** se implementa mediante un componente JavaBean que ayuda en la recuperación de contenido, adapta y almacena el modelo para usarlo en la vista.
- **Custom Tag Helper** – El componente **ViewHelper** se implementa como un Tag personalizado (sólo JSP 1.1+), donde la lógica de negocio se encapsula en un componente de etiqueta personalizada, que ayuda en la recuperación del contenido y en la adaptación del modelo para su utilización en la vista.
- **Tag File Helper** – JSP 2.0 proporciona una alternativa sencilla a los **Custom Tag**, moviendo la preparación de los datos y la lógica de formateo a archivos Tag mejorando la reutilización y abstrayendo los detalles de implementación.

- **Business Delegate** – Si el componente **ViewHelper** necesita acceder a servicios de negocio, es recomendable utilizar **Business Delegate** (delegados de negocio) para así conseguir ocultar los detalles de la implementación subyacente.
- **Transformer Helper** – El componente **ViewHelper** se implementa mediante XSLT (Extensible Stylesheet Language Transformer). El **ViewHelper** obtiene el modelo, que normalmente son datos en formato XML, y los transforma a un lenguaje de presentación como por ejemplo HTML.

Ventajas

- Utilizar componentes ViewHelper separa a la vista de la lógica de formateo y la preparación de los datos.
- La lógica de preparación y formateo de los datos fuera de las páginas JSP, en componentes **JavaBean** o **Custom Tag** permitirá reutilizar estos componentes en distintas JSPs evitando la replicación de código y facilitando de esta forma el mantenimiento.

3.6. Patrón Composite View

El patrón Composite View proporciona una forma sencilla de crear vistas combinando fragmentos de otras vistas.

El problema

La mayor parte de las Vistas que aparecen en una aplicación Web comparten contenido común entre ellas como la cabecera, el pie, el área de navegación, el contenido, etc.

Si el contenido común que debe aparecer en diversas páginas se ha de copiar y pegar en cada una de ellas aparecen una serie de inconvenientes:

- Cambiar el contenido común implica la modificación de todas las páginas que lo tienen.
- Mantener una apariencia (look & feel) consistente entre distintas páginas se vuelve más complicado.
- Crear tantas vistas como sean necesarias para disponer de todas las combinaciones de contenido requeridas.

La solución

La solución pasa por la creación de vistas compuestas basadas en la inclusión y sustitución de fragmentos de plantilla. De esta forma se promueve la reutilización de porciones atómicas y se asegura un diseño modular. Este patrón es extremadamente útil cuando es necesario generar páginas que muestran componentes que podrían combinarse en gran variedad de formas.

Diagrama de clases

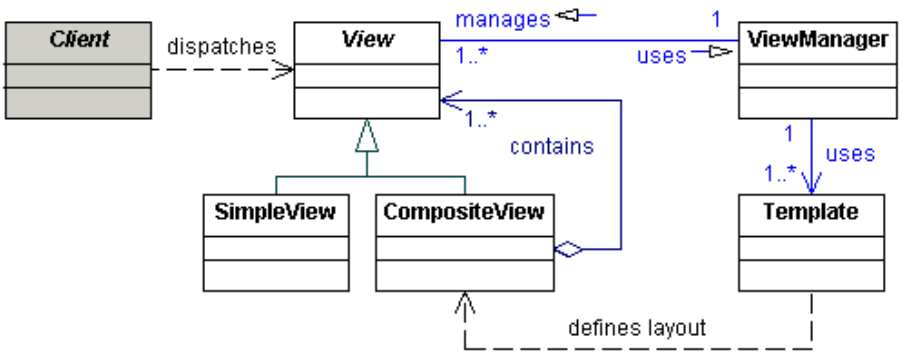

Figura 19 - Diagrama de clases del patrón Composite View

Diagrama de secuencia

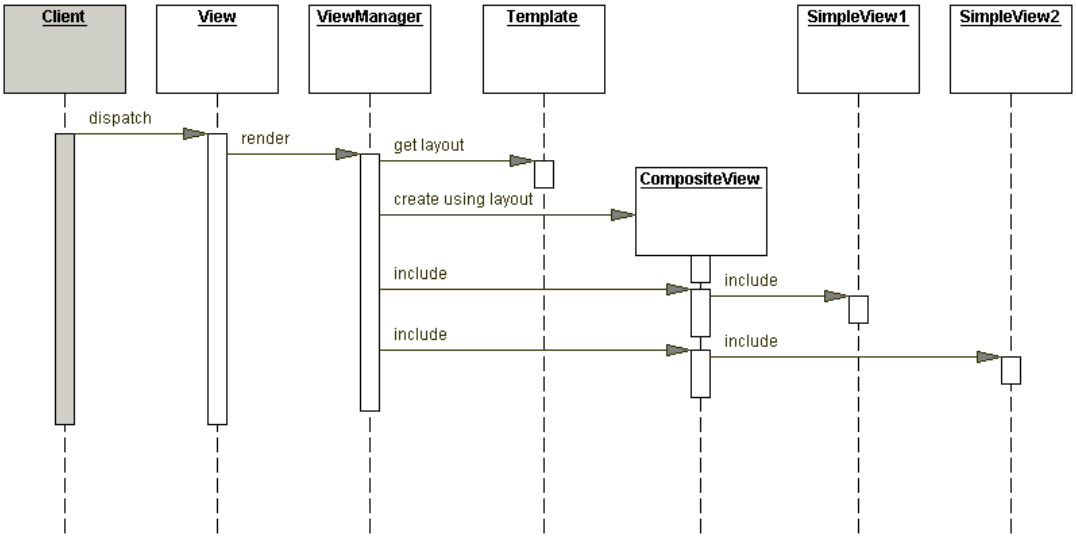

Figura 20 - Diagrama de secuencia del patrón Composite View

Estrategias de implementación

- **Early-Binding Resource** – Consiste en incluir partes de la plantilla en el momento de traducir la página JSP mediante la inclusión de la directiva **<%@ include>**. Esta estrategia proporciona un buen rendimiento y suficiente flexibilidad para páginas que no cambian continuamente.
- **Late-Binding Resource** – Consiste en la inclusión de partes en la plantilla en tiempo de ejecución, utilizando la acción **<jsp:include>**. Esta estrategia ofrece un rendimiento peor que la anterior pero es muy sencilla si los fragmentos se actualizan a menudo.
- **Custom Tag View Management** – Las vistas se incluyen utilizando etiquetas personalizadas.
- **JavaBeans Component View Management** – Las vistas se incluyen utilizando componentes JavaBeans.
- **Transformer View Management** – Las vistas se gestionan utilizando un Transformador XSL.

Ventajas

- **Mejora la modularidad y la reutilización**. Los fragmentos creados pueden reutilizarse en distintas vistas incluso con distintos contenidos.
- **Mejora el mantenimiento y la manejabilidad**. Las plantillas o los fragmentos de vistas pueden cambiarse en un único lugar provocando cambios en todas aquellas vistas que las utilicen.

Desventajas

- **Impacto en el rendimiento.** La inclusión de fragmentos en tiempo de ejecución puede penalizar en el rendimiento.
- **Contenido inválido.** Puede provocar la creación de contenido inválido si las plantillas no están configuradas correctamente. Por ejemplo la inclusión de tags HTML en zonas en las que no estén permitidos.

3.7. Patrón Service To Worker

El patrón Service To Worker es un patrón que combina los patrones **View Helper**, **Front Controller** y **Application Controller**. El sistema controla el flujo de ejecución y accede a los datos de negocio desde donde crea el contenido de presentación.

El problema

Si un sistema no utiliza un mecanismo de gestión de peticiones centralizado como el que proporcionan **Front Controller** y **Application Controller**, las actividades del controlador pueden acabar dispersas en una variedad de componentes Vista que darán como resultado un sistema en el que habrá código duplicado y, por tanto, resultará más difícil de adaptar y extender.

Además, si el sistema no elimina la lógica de negocio y la lógica de presentación de los componentes de la vista tal y como hace el patrón **View Helper**, los componentes de la vista acabarán teniendo embebido código scriptlet que dificultará, posteriormente el mantenimiento así como la depuración del código.

Todos estos problemas se solucionarían utilizando estos tres patrones en conjunto, pero aun así quedaría por determinar si el **Front Controller** o los componentes de la vista solicitarían los servicios del negocio. Asignar esta funcionalidad a los componentes de la vista (patrón **Dispatcher View**) podría llevar a una baja cohesión de los componentes de la vista y a la duplicación de código.

La solución

Utilizar el patrón Service To Worker para centralizar el control y el manejo de las peticiones para recuperar un modelo de presentación antes de pasarle el control a la vista. La vista generará una respuesta dinámica basada en el modelo de presentación.

Diagrama de clases

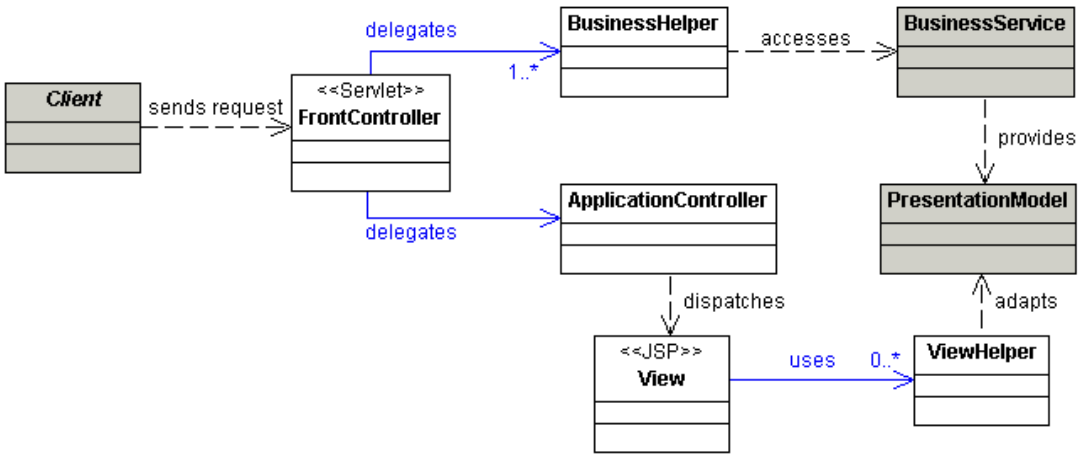

Figura 21 – Diagrama de clases del patrón Service To Worker

Diagrama de secuencia

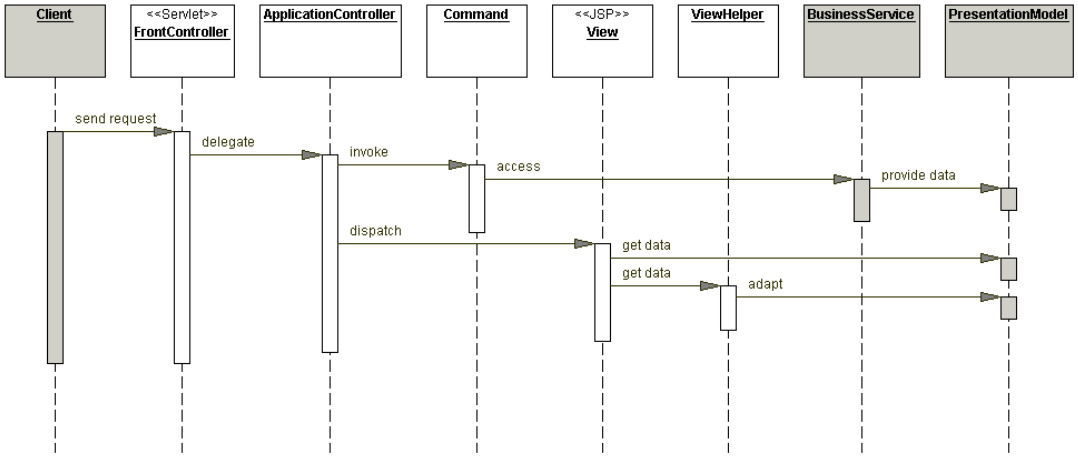

Figura 22 – Diagrama de secuencia del patrón Service To Worker

Ventajas

- Proporciona todas las ventajas de los patrones **Front Controller**, **Application Controller** y **View Helper**.
- Comparado con el patrón **Dispatcher View**, los componentes de la vista son probablemente más cohesivos, tienen menos código scriptlet y son más fáciles de adaptar y mantener.

Desventajas

- Poca cohesión del componente controlador en función de la implementación.

4. Patrones creacionales

Los patrones creacionales especificados por GoF (Gang of Four) están destinados a ocultar y abstraer los detalles de la creación de objetos.

Algunas de las ventajas que proporcionan son:

- Exponen métodos en lugar de constructores.
- Ocultan la diferencia entre crear un nuevo objeto y reutilizar uno existente.

4.1. Patrón Singleton

Patrón creacional cuyo objetivo consiste en restringir la creación de objetos pertenecientes a una clase y proporcionar un único punto de acceso global a ésta.

El problema

Se admite exactamente una instancia de una clase. Los objetos necesitan un único punto de acceso global.

La solución

Definir un método estático de la clase que devuelva el singleton

Diagrama de clases

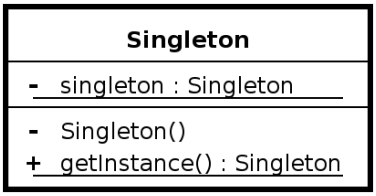

Figura 23 - Diagrama de clases del patrón Singleton

Ventajas

- El cliente puede obtener fácilmente una referencia al objeto.
- La clase Singleton podrá construir su instancia durante la carga de la clase o bajo demanda.

Desventajas

- La clase Singleton debe proporcionar instancias bajo demanda sin problemas de concurrencia (Thread-safe).

4.2. Patrón Factoría – Factoría (Concreta)

Patrón creacional que define un objeto **Fabricación Pura** "factoría" para la creación de objetos.

El problema

Se desea separar las responsabilidades de creación de objetos para mejorar la cohesión.

La solución

Crear un objeto **Fabricación Pura** denominado factoría que maneje la creación.

Diagrama de clases

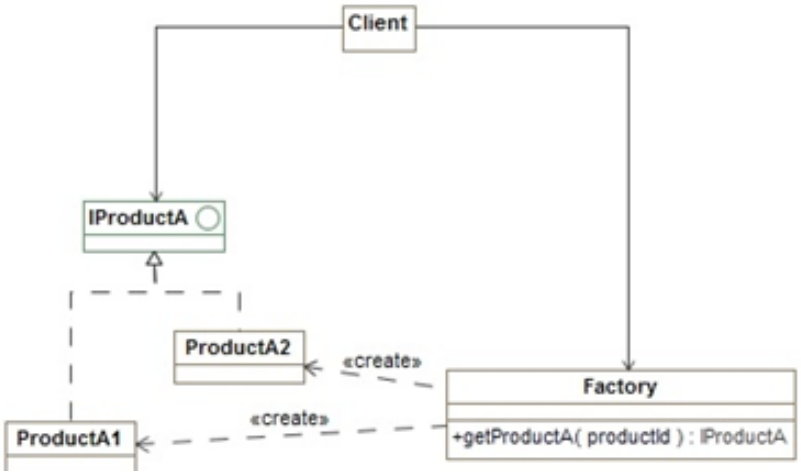

Figura 24 - Diagrama de clases patrón factoría concreta

Ventajas

- Separa la responsabilidad de la creación compleja en objetos de apoyo cohesivos.
- Oculta la lógica de creación potencialmente compleja.
- Permite la introducción de estrategias para mejorar el rendimiento de la gestión de la memoria.

Desventajas

- Su utilización puede complicar el código innecesariamente.

5. Framework

Desde el punto de vista del desarrollo del software, un framework o marco de trabajo, es una estructura de soporte definida para el desarrollo de una aplicación. Típicamente puede incluir soporte de programas, bibliotecas así como lenguajes de script que ayuden con el desarrollo y la unión de distintos componentes de un proyecto. Un framework podría considerarse como una aplicación genérica incompleta y configurable a la que pueden añadírsele las últimas piezas para así construir una aplicación completa.

Los objetivos principales que persigue un framework son:

- Acelerar el proceso de desarrollo.
- Reutilizar código ya existente.
- Promover las buenas prácticas de desarrollo mediante el uso de patrones.

Inconvenientes

- Su utilización supone un cierto coste inicial de aprendizaje aunque a largo plazo es probable que facilite tanto el desarrollo como el mantenimiento.

En el marco de las aplicaciones Web, un framework Web podría definirse como un conjunto de componentes (clases java, descriptores, archivos de configuración en XML) que componen un diseño reutilizable que facilita y agiliza el desarrollo de sistemas web.

Entre los numerosos frameworks Web que existen actualmente en el mercado, concentrándonos sólo en aquellos compatibles con la plataforma Java EE, se estudiarán algunos de los que mayor aceptación han tenido en el desarrollo Web. La figura adjunta nos da una idea de las tendencias actuales en este marco de desarrollo.

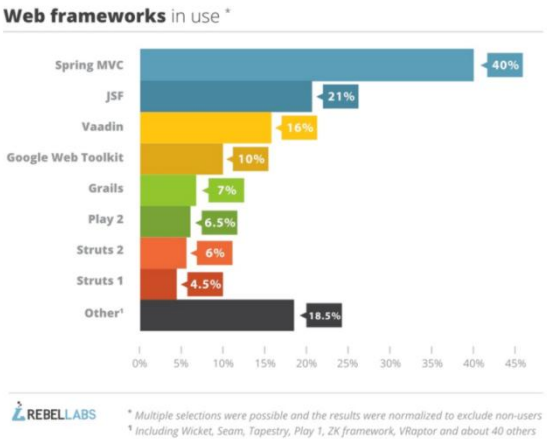

Figura 25 - Frameworks Web más utilizados

Tomando la figura anterior como referencia unida además a la experiencia personal en el desarrollo de aplicaciones Web en el entorno laboral, se ha seleccionado para su estudio los siguientes frameworks Web: Struts, Struts^2, Spring MVC y JavaServer Faces.

Dada la multitud de funcionalidades y posibilidades que estos frameworks ofrecen (a veces pudiendo combinarse con otros frameworks) resulta inviable con el tiempo disponible, llevar a cabo un estudio en profundidad de cada uno de ellos. Por ello, considerando que tampoco es el objetivo de este proyecto, se ha divido el estudio de cada uno en cuatro puntos comunes y vitales para la correcta comprensión sobre su funcionamiento. Estos puntos son:

- **Introducción sobre el framework**. Breve pincelada sobre sus orígenes, situación actual, versiones, etc.
- **Arquitectura del framework**. Se presentan y describen los componentes básicos del núcleo del framework, facilitando de esta forma la compresión de las ilustraciones sobre las que se apoya este estudio.
- **Ciclo de vida de una petición**. Donde se describe paso a paso el flujo de trabajo y la relación existente entre los componentes que participan.
- **Implementación del patrón MVC**. Por último, el estudio de cada framework concluye con la identificación de aquellos componentes que, habiendo sido presentados en puntos anteriores, forman parte del Modelo, la Vista y el Controlador según el patrón arquitectónico MVC.

6. Apache Struts

Apache Struts es un framework open-source destinado al desarrollo de aplicaciones Web en Java EE. Éste se caracteriza por la utilización tanto de tecnologías estándar, como de patrones de diseño mediante los que crea un entorno de desarrollo extensible para el desarrollo de aplicaciones Web.

Su objetivo principal es el de separar el **Modelo** (lógica/procesos de negocio) de la **Vista** (presentación de los datos al usuario) y el **Controlador** (unidad de control central que actúa como puente entre el Modelo y la Vista). Para ello proporciona su propio componente **Controlador** y se integra con otras tecnologías para soportar el **Modelo** y la **Vista**.

Para el **Modelo** el framework puede interactuar con tecnologías de acceso a datos estándar como JDBC, EJB así como paquetes de terceros como Hibernate, iBATIS, etc. Para la **Vista** utiliza normalmente JavaServer Pages, incluyendo JSTL y JSF, así como Velocity Templates, XSLT y otros sistemas de presentación.

Originalmente Struts fue creado por Craig McClanahan y donado a **Apache Foundation** en Mayo de 2000. Actualmente Apache Struts es un proyecto al que se dejó de dar soporte en Marzo de 2012 en su versión 1.3.10. Desde entonces, éste fue sustituido por Struts[2] con el que convivió apenas dos años antes de su total desaparición.

6.1. Arquitectura de Struts

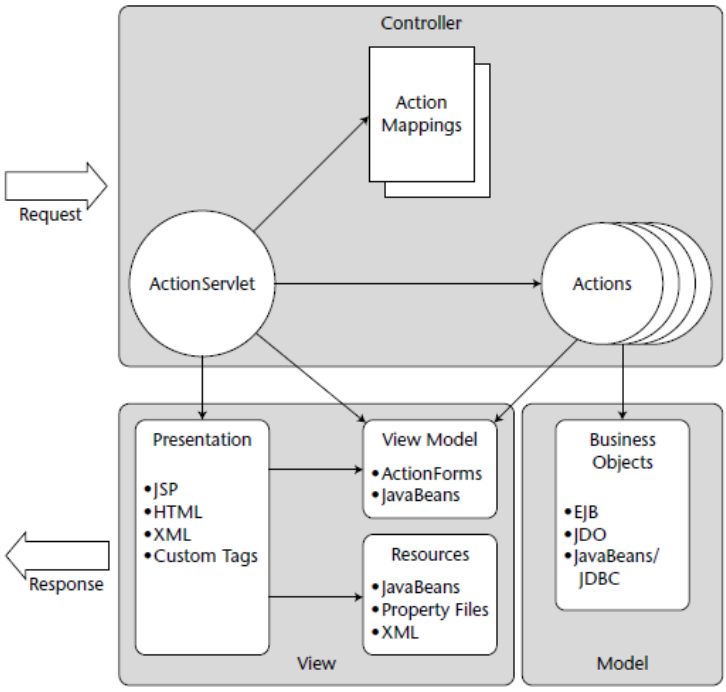

Figura 26 – Arquitectura de Struts

La figura anterior muestra los distintos componentes que participan en el marco de trabajo proporcionado por Struts. Una breve descripción de los más importantes servirá para comprender mejor la forma de trabajar de este framework.

- **ActionServlet:** Servlet que Struts utiliza como componente principal del **Controlador**, éste implementa el patrón de diseño *Front Controller* que lo convierte en el punto de entrada de todas las peticiones dirigidas a la aplicación web.
- **Action:** Adaptador entre el contenido de una petición http y la correspondiente lógica a ejecutar para procesar la petición.
- **ActionMapping:** Representa la información que el controlador conoce sobre la relación existente entre una petición y el **Action** que debe servirla.
- **ActionForm:** JavaBean opcionalmente asociado con uno o más **ActionMappings** que tendrá sus propiedades inicializadas con los valores enviados por el usuario. Esta clase dispone del método validate() que permite al **Action** validar si los datos recibidos son correctos antes de pasar a ejecutar la lógica de negocio asociada. Implementación del patrón Context Object.
- **ActionForward:** Representa un destino al cual el controlador deberá ser redirigido para llevar a cabo el resultado de la ejecución de un **Action**.
- **struts-config.xml:** Fichero de configuración que utiliza Struts para inicializar sus recursos. Estos recursos incluyen **ActionForm**s para recoger las entradas de los usuarios, **ActionMappings** para dirigir las entradas a los **Action** de la parte servidor y **ActionForwards** para seleccionar las páginas de salida.
- **JavaServer Pages – JSP:** Principal componente de la Vista. Las páginas JSP están formadas por código HTML estático y librerías de etiquetas que permiten generar código HTML dinámico.
- **Struts Taglib:** Componente que proporciona un conjunto de librerías de etiquetas que ayudan a los desarrolladores a crear aplicaciones interactivas basadas en formularios. Está compuesto por cuatro librerías de etiquetas distintas:
    - Bean – Definición de nuevos Beans.
    - HTML – Creación de formularios de entrada y otras interfaces de usuario basadas en HTML.
    - Logic – Generación de código HTML bajo ciertas condiciones, recorrido sobre colecciones, etc.
    - Nested

  Struts utiliza el patrón View Helper para implementar estas librerías de etiquetas.

6.2. Ciclo de vida de una petición en Struts

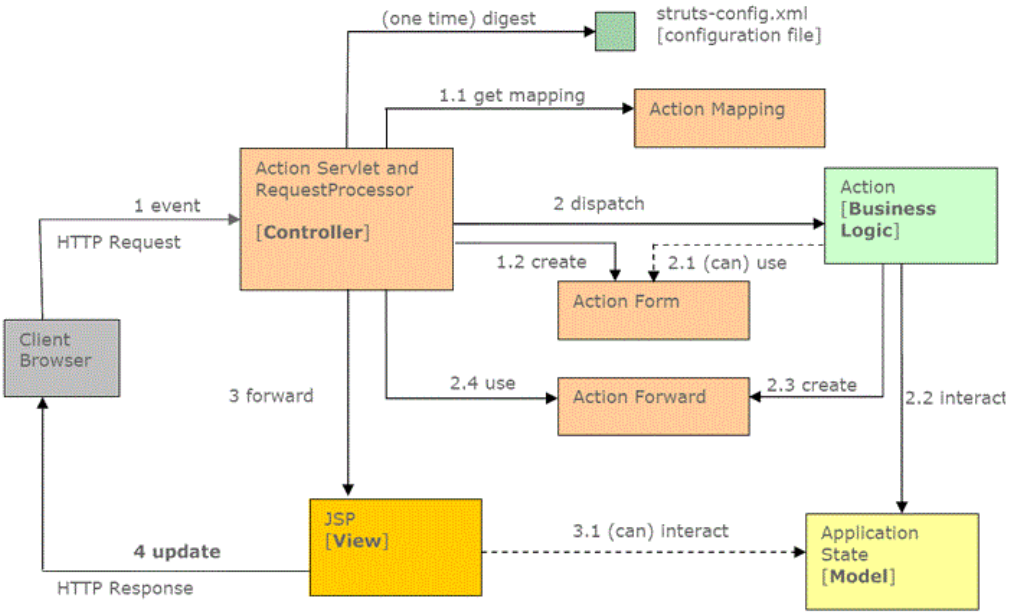

Figura 27 – Ciclo de vida de una petición en Struts

1. Inicialmente el cliente enviará una petición HTTP (normalmente mediante un cliente web). La petición es recibida por ***ActionServlet*** que, durante el arranque de la aplicación ha cargado en memoria una colección de objetos a partir de la lectura del fichero de configuración ***struts-config.xml***.
   1.1. Recupera el ***Action*** al que habrá que dirigir la petición en función de la URL solicitada por el usuario.
   1.2. Se crea el objeto ***ActionForm*** asociado a la petición (si procede) y se cargan todos los valores de los parámetros enviados en la petición del cliente.
2. ***ActionServlet*** determina a qué ***Action*** hay que llamar y le pasa el control.
   2.1. El ***Action*** (si procede) ejecuta las validaciones de los datos enviados por el usuario.
   2.2. La clase ***Action*** procesa la petición con ayuda del modelo.
   2.3. La clase ***Action*** construye y devuelve el ***ActionForward*** al ServletDispatcher.
3. El ***ServletDispatcher*** analiza el ***ActionForward*** enviado por el ***Action*** y, consultando los mapeos previamente cargados (***struts-config.xml***) determina a qué vista debe invocar para que muestre los resultados.
   3.1. La vista prepara los resultados accediendo a los resultados recuperados por el modelo (si procede).
4. La vista, una vez generada la respuesta, envía el resultado al cliente.

6.3. Struts y el patrón MVC

Modelo

Struts no impone ninguna restricción sobre cómo implementar el **Modelo** en una aplicación Web. El modelo de dominio representado por objetos de negocio, normalmente se representa mediante **Enterprise JavaBeans** (EJBs), **Java Data Object**s (JDO), o **JavaBeans** que pueden acceder a bases de datos vía **JDBC**. Pese a que la arquitectura del framework es flexible al respecto, es recomendable separar la lógica de negocio del rol que lleva a cabo un **Action**.

Vista

La **Vista** consiste en JSPs y un conjunto de etiquetas a medida proporcionadas por el framework, que ayudan a separar la **Vista** del **Controlador.** (Véase patrón *View Helper*).

Controlador

El componente principal del **Controlador** en el framework es un Servlet llamado ***ActionServlet*** que implementa el patrón *Front Controller*. Este Servlet es el punto de acceso a la aplicación Web y se configura mediante un conjunto de ***ActionMappings*** definidos en el fichero **struts-config.xml**. Recordemos que un ***ActionMapping*** definía la relación existente entre la URL de la petición entrante y la clase **Action** que debe ejecutarse para servirla, mientras que los **Action** encapsulan las llamadas a las clases que contienen la lógica de negocio, interpretan los resultados y finalmente despachan el control al componente vista apropiado para crear la respuesta.

7. Struts^2

Apache Struts^2 es un framework open-source para el desarrollo de aplicaciones Web Java EE. Originariamente era conocido como WebWork2 y su aparición fue el resultado de la convergencia entre las comunidades de desarrolladores de WebWork y Struts tras años de desarrollos independientes. Su primera versión fue liberada el 22 de febrero de 2007 (versión 2.0.6) y a día de hoy aún continúa evolucionando.

7.1. Arquitectura de Struts2

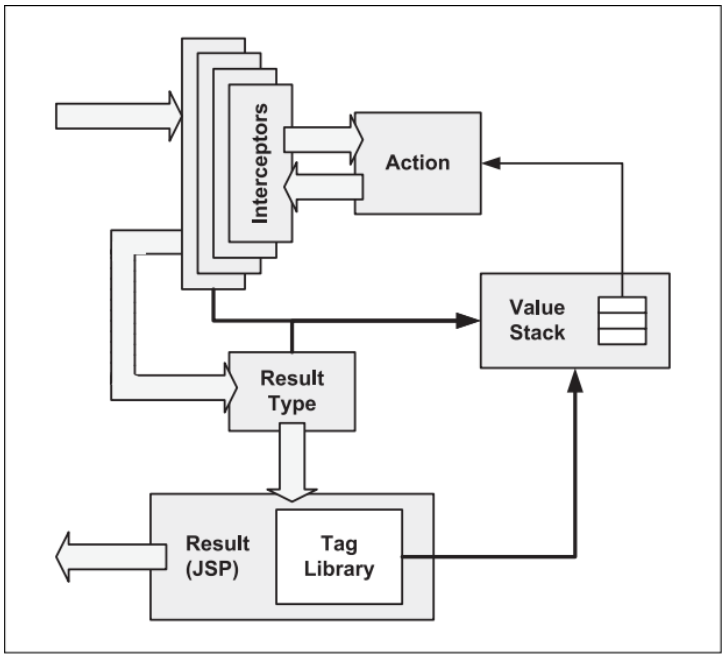

Figura 28 – Arquitectura de Struts^2

La figura 27 muestra la arquitectura del marco de trabajo Struts^2 en la que aparecen una serie de componentes que pasarán a explicarse brevemente a continuación.

- **Action.** Los **Action**s son el núcleo del framework de Struts^2. Cada URL se mapea con un **Action**, el cual proporciona la lógica necesaria para servir la petición de un usuario. El único requisito para un **Action** en Struts^2 es que deben tener un método sin argumentos que devuelva un objeto de tipo String o **Result**. Cuando el **Action** responde a una petición con un String, se obtiene el **Result** correspondiente y se instancia. Un **Action** no tiene porqué implementar ninguna interfaz, no obstante, Struts^2 proporciona la interfaz *Action* y la clase *ActionSupport* para que puedan ser utilizadas como esqueleto.
- **Interceptor**. Un interceptor no es más que una clase Java que implementa la interfaz *Interceptor*. Mediante estas clases se permite definir código que se procesará antes

y/o después de la ejecución de un **Action** y aunque el uso que se les puede dar es muy diverso, normalmente se utilizan para validaciones, seguridad, registros de actividad, etc. Dos o más interceptores pueden encadenarse creando una pila que el framework ejecutará en el orden establecido. Muchas de las funcionalidades que proporciona Struts^2 las llevan a cabo interceptores. (Patrón de diseño filtro interceptor).

- **Value Stack**. La pila de valores es un concepto central del framework. Todos los componentes del núcleo interaccionan con ella de una forma u otra para proporcionar acceso a información del contexto así como a elementos del entorno de ejecución. Pese a tratarse de una estructura de datos tipo Pila, su implementación difiere de la de una pila tradicional. La pila de valores no funciona apilando o desapilando objetos según la necesidad, sino buscando o evaluando expresiones utilizando sintaxis OGNL (Object Graph Navigation Language). Esta pila además consta de cuatro niveles:
    - **Objetos temporales**: Objetos que se almacenan temporalmente durante el procesamiento de una petición.
    - **Modelo de objeto**: Cuando un **Action** implementa la interfaz *ModelDriven*, el modelo de objeto se ubica en la pila frente a la acción que está siendo ejecutada.
    - **Objeto Action**: La acción que está siendo ejecutada.
    - **Objetos con nombre:** A cualquier objeto se le puede asignar un identificador convirtiéndolo así en un objeto con nombre. Estos objetos pueden ser creados por el desarrollador.
- **Result.** Una vez que el **Action** se ha procesado, es necesario enviar la información resultante al usuario que la solicitó. En Struts^2 esta tarea se divide en dos partes, el tipo de resultado (result type) y el resultado en sí.
    - El tipo de resultado proporciona los detalles de implementación para el tipo de información que se devuelve al usuario. Los tipos de resultados suelen estar o bien pre-configurados por Struts^2 o definidos mediante plugins. Por defecto, el tipo de resultado que devuelve un **Action** es *dispatcher*, el cual utiliza una JSP para renderizar la respuesta al usuario.
    - El resultado en sí define qué sucede una vez que el **Action** se ha ejecutado.
- **Tag library.** Las librerías de etiquetas proporcionan la intersección entre acciones y vistas. La diferencia entre la librería de etiquetas de Struts^2 y otras librerías de etiquetas JSTL es que las primeras están fuertemente integradas con el framework, aprovechando la **Pila de Valores** para acceder a los métodos de un **Action** y además, aprovechan *OGNL* para evaluar expresiones y llevar a cabo la creación de objetos al

vuelo y generación de código. La librería de etiquetas de Struts[2] se divide en cuatro categorías distintas que varían según su función: Etiquetas de control, de datos y de formularios y de no formularios (aquellas utilizadas en formularios pero con funcionalidad diferente a las etiquetas de formularios).

7.2. Ciclo de vida de una petición en Struts[2]

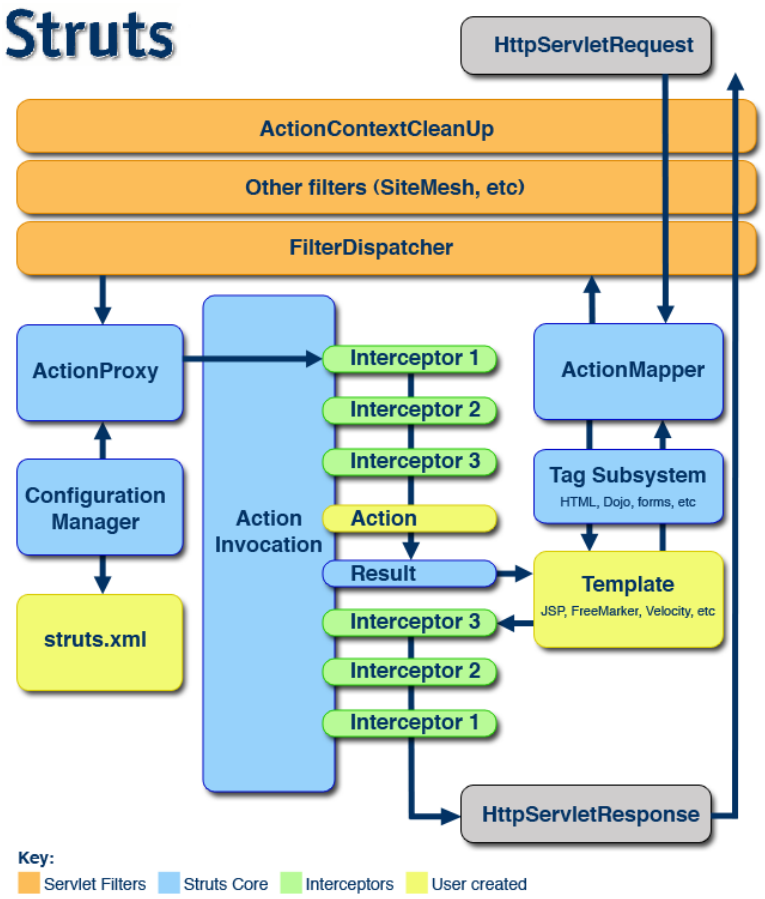

Figura 29 – Ciclo de vida de una petición en Struts2

- Inicialmente el contenedor de Servlets en el que está desplegada la aplicación web recibe una petición HTTP solicitando algún recurso.
- Esta petición pasa a través de una cadena de filtros estándar que finaliza con el **FilterDispatcher** que consulta el **ActionMapper** (interfaz que proporciona un mapeo bidireccional entre peticiones HTTP e invocación de **Actions**) para determinar si la petición debe invocar a un **Action**.
- Si **ActionMapper** determina que un **Action** debe ser invocado, **FilterDispatcher** delega el control al **ActionProxy**.

- El **ActionProxy** consulta los ficheros de configuración del framework (struts.xml) y crea un **ActionInvocation.** Unida a esta creación, se invoca a un conjunto de interceptores antes de invocar al **Action**.
- Una vez ejecutado el **Action**, el **ActionInvocation** se responsabiliza de buscar el resultado asociado al código resultante de la ejecución del **Action** (definido en el fichero struts.xml).
- A continuación se construye del resultado y ello implica, en ocasiones, la renderización de alguna plantilla escrita en JSP o FreeMarker. Mientras se lleva a cabo el renderizado de las plantillas, si éstas incluyen Tags (proporcionados por el framework), puede ser necesario llegar a trabajar con el **ActionMapper** de forma que si un Tag implica la construcción de una URL, ésta se construya correctamente.
- Finalmente, antes de entregar el resultado de la petición inicial recibida por el contenedor, todos los interceptores se ejecutarán nuevamente, esta vez en orden inverso al inicial y a continuación se enviará la respuesta.

7.3. Struts2 y el patrón MVC

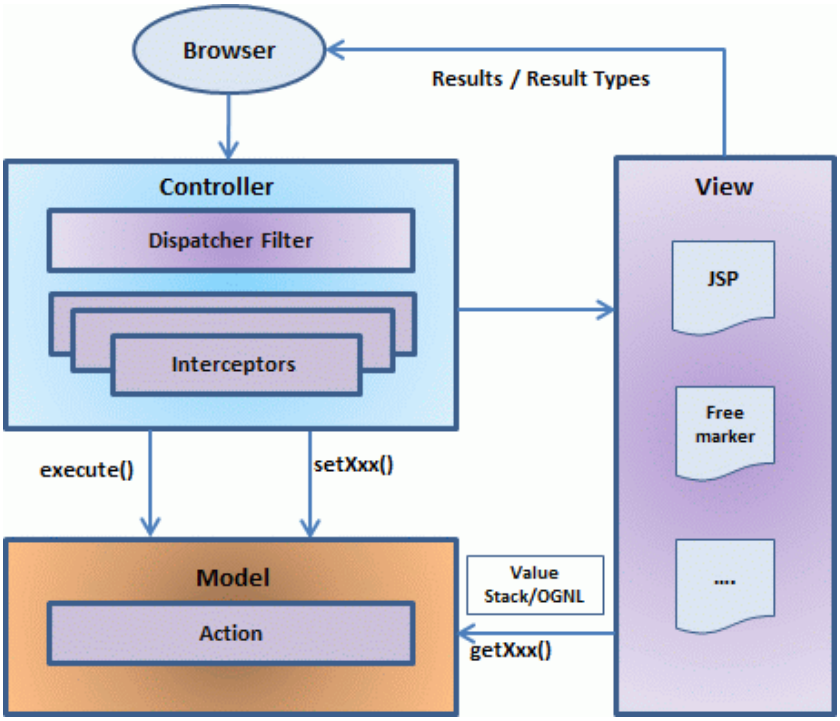

Figura 30 - MVC en Struts2

Tal y como se aprecia en la captura, Struts[2] implementa el patrón arquitectónico MVC de una manera significativamente distinta a la de un framework tradicional.

En su implementación las clases **Action** toman el rol de **Modelo** (en lugar de Controlador) y el **Controlador** se implementa mediante **Interceptores** y el **FilterDispatcher** (implementación del front-controller mediante un filtro). Por último, la **Vista** en Struts^2 permite su implementación mediante la utilización de diversas tecnologías como: JSP, Velocity Template y Freemaker entre otras.

La implementación que lleva a cabo Struts^2 del patrón MVC es conocida como Pull-MVC. En esta implementación (que difiere de su alternativa Push-MVC) es la **Vista** quién solicita los datos que necesita (a través de la pila de valores - **Value Stack**) en lugar de ser el **Modelo** quién envía estos datos a la **Vista** mediante el uso de objetos Request o Session tal y como se hacía en Struts.

8. Diferencias entre Struts y Struts2

Clases Action

- Struts requiere que las clases **Action** extiendan de una clase base abstracta.
- Struts^2 permite que cualquier POJO con un método execute() pueda utilizarse como objeto **Action** aunque también proporciona la posibilidad de opcionalmente implementar la interfaz **Action** y extender la clase **ActionSupport** si se desea aprovechar la implementación de ciertas funcionalidades útiles.

Modelo de Threads

- Los **Actions** de Struts son singletons por lo que todas las solicitudes enviadas hacia un **Action** concreto son gestionadas por una única instancia. Ello implica que los recursos utilizados por un **Action** deben o bien estar sincronizados o ser Thread-safe para evitar que distintas peticiones a un mismo **Action** obtengan resultados condicionados por otra petición.
- Los **Action** de Struts^2 se instancian en cada petición por lo que no pueden aparecer problemas entre distintas peticiones a un mismo **Action**.

Dependencia de Servlets

- Struts muestra una fuerte dependencia de la API de Servlets dado que los objetos HttpServletRequest y HttpServletResponse se pasan al método execute() cuando el **Action** se invoca.
- Los **Action** de Struts^2 no están acoplados con el contenedor, ello permite su testeo de forma aislada. Los **Action** de Struts^2 pueden mantener el uso de la request y la response pero esta necesidad se reduce debido a ciertos elementos de su arquitectura.

Manejo de pruebas

- En Struts dada la dependencia de los **Action** de la API de Servlets, se hace necesario utilizar extensiones de terceros como (Struts TestCase) para poder llevar a cabo pruebas.
- En Struts^2 los **Action** pueden ser probados instanciando el propio **Action**, estableciendo propiedades e invocando métodos

Recolección de datos de entrada

- En Struts es necesario un objeto **ActionForm** para capturar las entradas. Al igual que las clases **Action**, todos los **ActionForm** deben extender de una clase base.
  DynaBeans pueden utilizarse como alternativa a la creación de clases **ActionForm**.
- Struts^2 usa las propiedades de los **Action** para manejar las entradas, eliminando de esta forma la necesidad de un segundo objeto de entrada.

Expression Language (EL) – Lenguaje de expresiones

- Struts está integrado con JSTL de forma que puede utilizar JSTL-EL (lenguaje de expresiones de JSTL)
- Struts^2 además de JSTL soporta un lenguaje de expresiones más potente y flexible llamado OGNL.

Vinculación a objetos de la vista

- Struts utiliza el mecanismo JSP estándar que vincular objetos al contexto de la página para hacerlos accesibles.
- Struts^2 utiliza una tecnología basada en una pila de valores (**Value Stack**) de esta forma las librerías de etiquetas pueden acceder a valores sin estar acoplados a la vista en la que se renderizará el objeto con independencia del contexto (sesión, aplicación o petición) siendo de esta forma más fáciles de reutilizar.

Conversiones de tipo

- En Struts las propiedades de los **ActionForm** son normalmente Strings.
- Struts^2 utiliza OGNL para la conversión de tipos. El framework ya incluye conversores para objetos básicos y tipos primitivos.

Validación

- Struts permite validación manual por medio de métodos en el **ActionForm** o a través de una extensión de los **Commons Validator**. Las clases pueden tener diferentes contextos de validación para la misma clase pero no pueden encadenar validaciones en sub-objetos.
- Struts^2 soporta validación manual por medio del método *validate* y del framework **XWork Validation**. El framework Xwork soporta encadenamiento de validaciones en

las sub-propiedades usando validaciones definidas para las propiedades de la clase y el contexto de validación.

Control de la ejecución del Action

- Struts soporta ciclos de vida diferentes para cada módulo pero, todas las acciones de un módulo deben compartir el mismo ciclo de vida.
- Struts[2] soporta la creación de distintos ciclos de vida, uno por **Action** a través de la pila de interceptores, pilas que pueden ser creadas y usadas con diferentes **Action** si es necesario.

9. Spring MVC

Spring es un framework de código abierto para el desarrollo de aplicaciones y contenedor de inversión de control para la plataforma Java. Su primera versión fue escrita por Rod Johnson y lanzada en Junio de 2003. Spring en la actualidad (Versión 4.1.1) es uno de los frameworks de mayor éxito. Éste consiste en un conjunto de funcionalidades organizadas en torno a 20 módulos. Estos módulos se agrupan en: **Core Container**, **Data Access/Integration**, **Web**, **AOP**, **Instrumentation**, **Messaging** y **Test**.

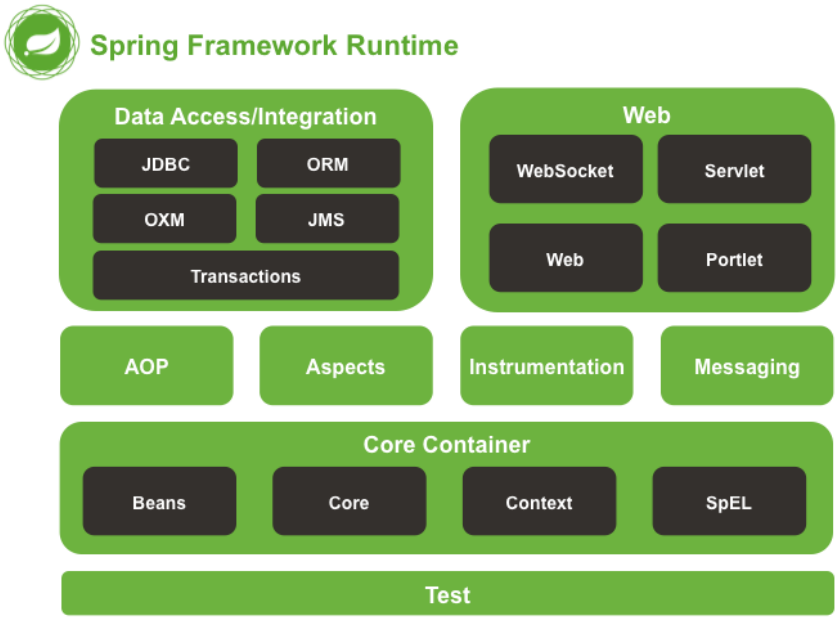

Figura 31 - Framework Spring

De entre todas las funcionalidades que proporciona Spring, **Web** es la que está relacionada con el desarrollo de aplicaciones Web y, de entre todos los módulos que la componen (spring-web, spring-webmvc, spring-websocket y spring-webmvc-portlet) **spring-webmvc** proporciona una arquitectura MVC y componentes (temas, internacionalización, validación, conversión de tipos, formateo, etc.) para el desarrollo de aplicaciones web flexibles y con bajo acoplamiento. Este módulo, no obstante, se apoya sobre spring-core (módulo de la funcionalidad **Core Container)** que proporciona las funcionalidades fundamentales del Framework Spring como IoC (Inversion Of Control) y la DI (Dependency Injection), funcionalidades sobre las que se apoyan el resto de módulos de Spring.

Actualmente Spring MVC es el framework Web más utilizado por encima incluso de JavaServer Faces que forma parte de la plataforma Java EE desde la versión 1.4.

9.1. Arquitectura de Spring MVC

La siguiente figura muestra la arquitectura de Spring MVC a través del flujo seguido por una petición.

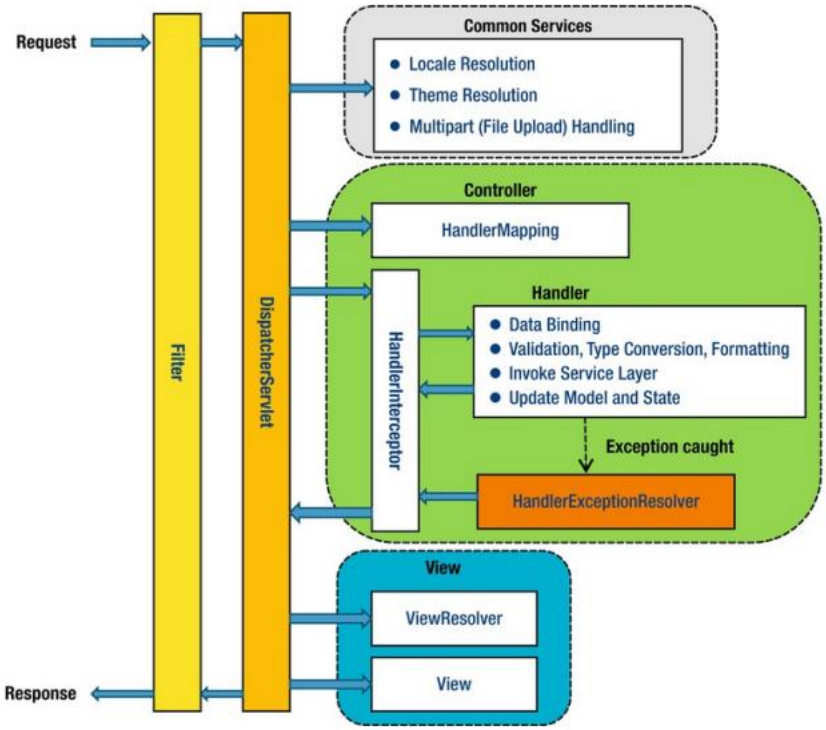

Figura 32 – Arquitectura de Spring MVC

- **Filter.** Componente que permite la ejecución de cierta lógica antes y/o después de procesar una petición.
- **DispatcherServlet**. Servlet que implementa el patrón de diseño Front Controller, siendo por tanto el componente central de Spring MVC encargado de recibir las peticiones enviadas a la aplicación y despacharlas convenientemente.
- **Commons Services.** Servicios comunes que se aplicarán a cada petición para dar soporte a internacionalización, temas, subida de ficheros, etc.
- **HandlerMapping.** Bean configurado en el contexto de la aplicación web que implementa la interfaz *HandlerMapping* y es responsable de devolver el *handler* apropiado para cada petición. Los **HandlerMappings** normalmente mapean una petición con un *handler* en función de la URL solicitada.
- **HandlerInterceptor.** En Spring MVC pueden registrarse interceptores para los *handlers* para llevar a cabo lógica compartida o ciertas comprobaciones.
- **Handler y Controller.** Un handler es un objeto Java arbitrario capaz de atender peticiones web y el tipo más habitual utilizado en Spring MVC es el *Controller*. Un

Controller es un tipo específico de handler que una vez que acaba de procesar una petición, devuelve al **DispatcherServlet** un **Modelo** y una **Vista** (o bien su identificador o bien un objeto vista).

- **HandlerExceptionResolver.** En Spring MVC la interfaz HandlerExceptionResolver está diseñada para tratar con excepciones propagadas durante el procesamiento de la petición.
- **ViewResolver.** Bean configurado en el contexto de la aplicación web que implementa la interfaz *ViewResolver*. Su responsabilidad es devolver un objeto vista a partir de su identificador.
- **View.** Una vez que el **DispatcherServlet** dispone del objeto **Vista**, lo renderizará y le pasará el objeto modelo devuelto por el **Controller**. Será responsabilidad de la vista mostrar al usuario el contenido del modelo.
- **Model.** Representa los datos que se pasan desde una operación (definida en un controlador web) hacia la vista. Permite la completa abstracción de la tecnología utilizada en la Vista.

9.2. Ciclo de vida de una petición en Spring MVC

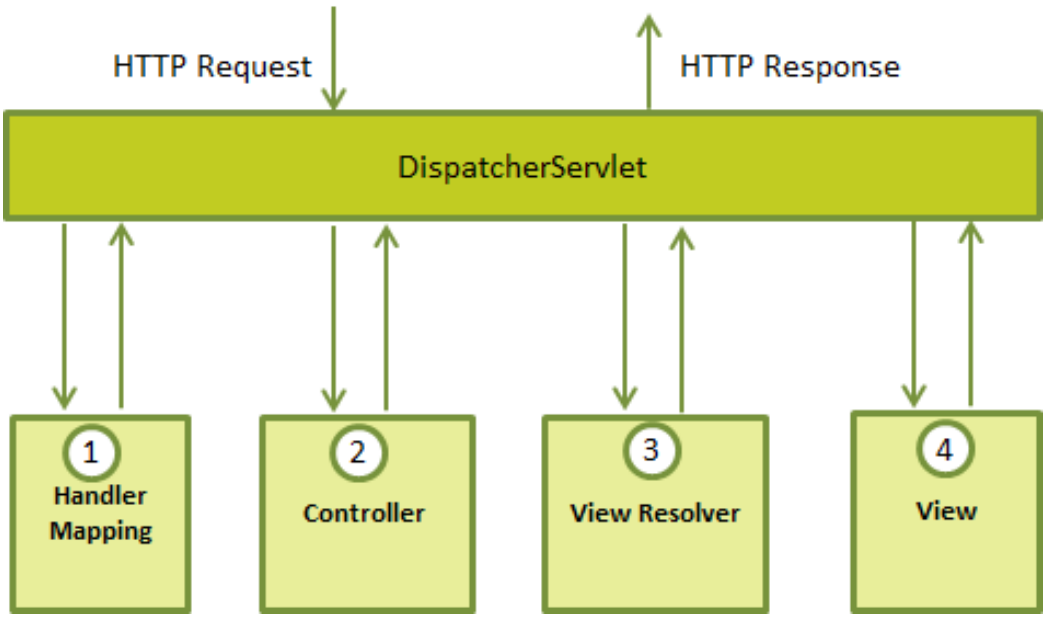

Figura 33 – Ciclo de vida básico de una petición en Spring MVC

- Inicialmente el contenedor de Servlets recibe una petición y ésta se redirige al **DispatcherServlet** (Servlet que implementa el patrón Front Controller).
- El **DispatcherServlet** averigua normalmente a partir de la URL, qué ***Controller*** hay que utilizar para servir la petición y ello lo determina a partir de un **HandlerMapping**.
- Se llama al **Controller** que ejecuta la lógica de negocio, obtiene los resultados y los devuelve al **DispatcherServlet** encapsulados en un objeto de tipo **Model** junto con el nombre lógico de la vista a mostrar.
- Un **ViewResolver** se encarga de averiguar el nombre físico de la vista que se corresponde con el nombre lógico de la vista obtenida en el paso anterior.
- Finalmente **DispatcherServlet** redirige la petición hacia la vista, que muestra los resultados de la operación realizada.

9.3. Spring MVC y el patrón MVC

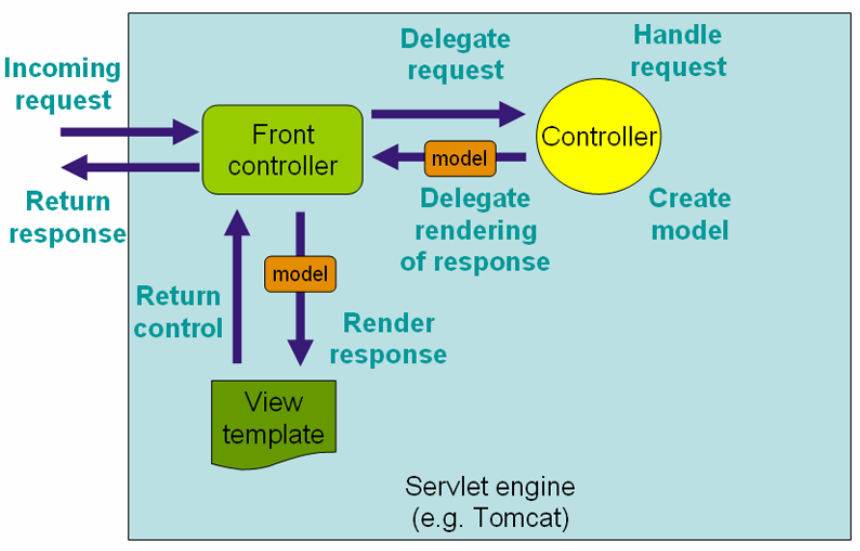

Figura 34 – Modelo Vista Controlador en Spring MVC

Spring MVC como otros muchos frameworks MVC está orientado a peticiones y diseñado alrededor de un Servlet central que despacha peticiones a controladores y ofrece otras muchas funcionalidades con el objetivo de facilitar el desarrollo de aplicaciones Web.

El **Controlador** en Spring MVC es el encargado de proporcionar acceso al comportamiento de la aplicación. Estos interpretan las entradas de usuario y las transforman en el **Modelo** que se presentará al usuario mediante la **Vista**. La noción de Controlador en Spring MVC es muy abstracta dado que permite crear distintos tipos de controladores en función de las necesidades (controladores específicos para formularios, para asistentes (wizards), basados en comandos, etc).

El **Modelo** tal y como se aprecia en la figura anterior es un objeto que devuelve el **Controlador** y se pasa a la **Vista**. El **Modelo** es el objeto encargado de contener los datos que serán mostrados por la **Vista**. Este objeto puede ser de un simple POJO hasta cualquier tipo de objeto de tipo Collection o incluso una combinación de ambos.

La **Vista** es la encargada de mostrar los datos al usuario. Spring MVC, al igual que otros frameworks, utiliza una combinación de JavaServer Pages y librerías de etiquetas para implementarla. Además permite utilizar otros tipos de tecnologías como Tiles, Velocity o Jasper Reports que pueden conectarse con el Framework.

10. JavaServer Faces

JavaServer Faces es un framework, a diferencia de los tres vistos hasta el momento, **basado en componentes**, destinado a la construcción de interfaces de usuario en aplicaciones Web. Éste se centra en simplificar el desarrollo de la interfaz de usuario dado que ésta suele ser de las partes más complejas y tediosas durante el desarrollo de una aplicación Web.

Fue creado a través del **Java Community Process** (JCP) por un grupo de líderes tecnológicos como: Sun Microsystems, Oracle, Borland, BEA e IBM junto con expertos de reconocida reputación en el desarrollo de aplicaciones Web en Java.

Su especificación se inició a mediados de 2001 y finalizó con la inclusión de la especificación de JSF 1.0 en la plataforma Java EE 1.4 en Marzo de 2004.

JSF combina un diseño MVC con un potente framework de desarrollo de interfaces de usuario, basadas en componentes, que simplifican notablemente el desarrollo de aplicaciones Web Java EE.

10.1. Arquitectura de JavaServer Faces

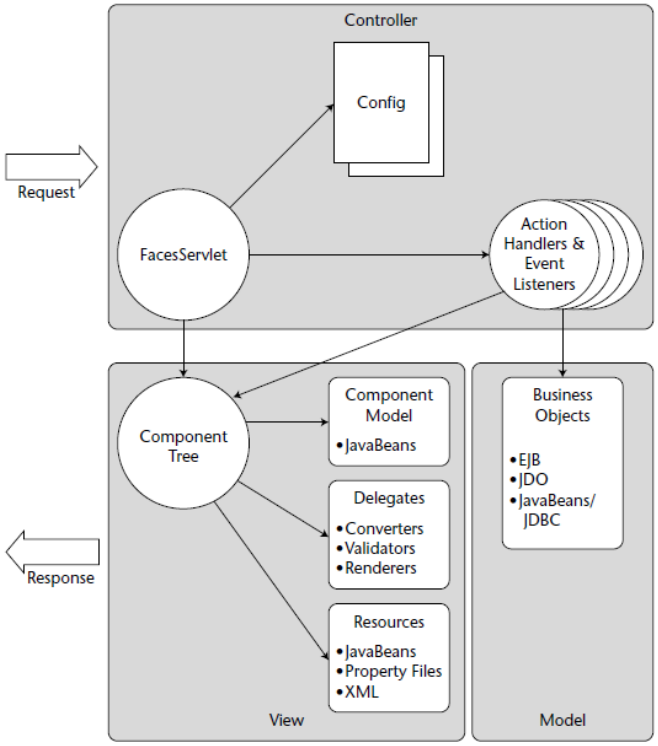

Figura 35 – Arquitectura JavaServer Faces

- **FacesServlet**. Servlet que implementa el patrón *Front Controller* que, junto con el fichero de configuración faces-config.xml y un conjunto de Action Handlers de la aplicación, forman el controlador de JavaServer Faces.
- **Componente UI**. Objeto con estado mantenido por el servidor que proporciona una funcionalidad específica para interactuar con un usuario final. Los componentes UI son JavaBeans con propiedades, métodos y eventos, organizados en una Vista (View), que es un árbol de componentes normalmente mostrado como una página.
- **Renderer**. Responsable de mostrar un componente UI y traducir la entrada del usuario en valores de componentes.
- **Validator**. Responsable de asegurar que el valor introducido por un usuario es correcto. Puede asociarse más de un validador a un componente.
- **Backing Beans.** JavaBeans especializados que recogen valores de los componentes UI e implementan métodos listener de eventos.
- **Converter**. Convierte un valor de un componente a y desde una cadena para mostrarlo.
- **Events y Listeners**. JSF utiliza el modelo event/listener de JavaBeans (usado también en Swing). Los componentes UI generan eventos y los listeners pueden ser registrados para manejar dichos eventos.
- **Mensajes de información que se muestra al usuario.** Backing Beans, Validators, Converters, etc. pueden generar información o mensajes de error que pueden mostrarse al usuario.

10.2. Ciclo de vida de una petición en JavaServer Faces

El ciclo de vida de una aplicación JavaServer Faces empieza cuando un cliente solicita una página (mediante una petición HTTP) y finaliza cuando el servidor responde a éste con la página solicitada traducida a lenguaje HTML.

El ciclo de vida de una petición puede dividirse en dos fases principales: **Ejecución** y **Renderizado**.

La fase de **ejecución** a su vez se fragmenta en fases más pequeñas que permiten soportar el sofisticado árbol de componentes (llamado Vista) que representa una página y que se construye llevando a cabo una serie de pasos ordenados.

Esta estructura requiere que:

- Los datos se validen y se lleven a cabo las conversiones necesarias.
- Los eventos del componente sean tratados.
- Los datos de éste se propaguen de manera ordenada.

El ciclo de vida de una petición-respuesta debe ser capaz de manejar dos tipos de peticiones:

- **Peticiones iniciales (Initial Request)**. Suceden cuando el usuario realiza la petición de una página por primera vez.
- **Devolución (PostBack)**. Ocurren cuando el usuario envía un formulario contenido en una página previamente cargada como resultado de una petición inicial.

Cuando el ciclo de vida trata una petición inicial, únicamente se ejecutan las fases: **Restore View** y **Render Response** debido a que no hay ni datos de usuario ni acciones a procesar. Si por el contrario, el ciclo de vida debe tratar una operación de **PostBack** se ejecutarán todas las fases que se muestran en la figura adjunta.

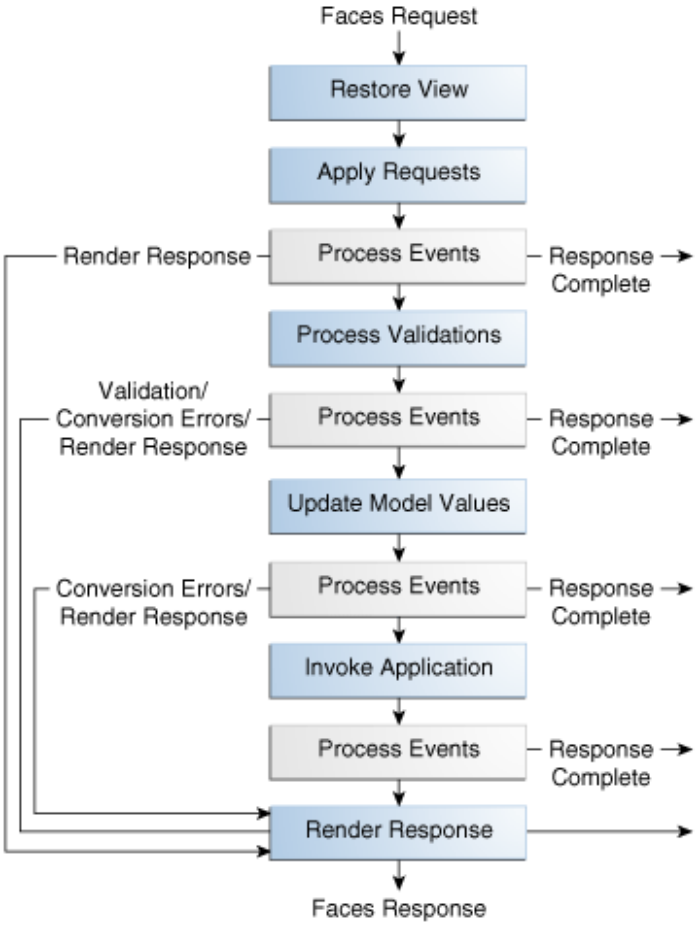

Figura 36 – Ciclo de vida de una petición en JavaServer Faces

Fases del ciclo de vida de una petición:

- **Restore View.** Esta fase se inicia al solicitarse una página JSF como resultado de hacer clic en un botón o enlace. Durante esta fase se construye la vista de la página conectando los controladores de eventos y validadores a los componentes de la vista y, salvando ésta en la instancia de *FacesContext* que contiene toda la información necesaria para procesar una petición.
  - Si la petición es una ***Initial Request***, la implementación de JSF creará una vista vacía durante esta fase y saltará a la fase **Render Response**.
  - Si la petición es de ***PostBack***, en *FacesContext* ya existirá una vista que se corresponda con la página.
- **Apply Request Value.** Después de haber restaurado el árbol de componentes como resultado de una petición de ***PostBack***, cada componente del árbol extraerá su nuevo valor de los parámetros de la petición mediante su método *decode* y lo almacenará localmente en cada componente.
  - Si alguno de los métodos decode o algún escuchador de eventos (event listener) ha llamado al método renderResponse de la instancia actual de FacesContext, se saltará a la fase **Render Response.**
  - Si algún evento se ha incluido en la cola durante esta fase, JSF difundirá los eventos a los *listeners* interesados.
  - Aquellos componentes de la página que tengan atributos inmediatos establecidos, implicarán la ejecución de validaciones, conversiones y eventos durante esta fase.

  Si algún proceso de conversión falla, un mensaje de error asociado con el componente se generará y se añadirá a la cola de *FacesContext* y este mensaje se mostrará en la fase de **Render Response**, así como cualquier error de validación que se produzca durante la fase **Process Validation**.
- **Process Validation.** JSF procesa todos los validadores registrados en el árbol de componentes, utilizando su método de validación, y examina las reglas de los atributos de cada componente y comprueba el valor local almacenado para cada uno. Si el valor local es inválido, se añade un mensaje de error a la instancia de *FacesContext* y el ciclo de vida avanza hasta la fase de **Render Response** mostrando nuevamente la misma página con el mensaje de error. De existir errores de conversión generados durante la fase **Apply Request Values** estos también se mostrarían.

- **Update Model Values.** Una vez determinado que los datos son válidos, JSF recorre el árbol de componentes y configura las propiedades del objeto de la parte servidor con los valores locales del componente. Si los datos locales no pueden ser convertidos a los tipos especificados por las propiedades del bean, el ciclo de vida avanza directamente a la fase de **Render Response** donde se vuelve a mostrar la página con los errores acontecidos.
- **Invoke Application.** Durante esta fase JSF maneja los eventos a nivel de aplicación como el envío de un formulario o el enlace con otra página.
- **Render response.** En esta fase, JSF construye la vista y delega la responsabilidad de renderizar la página al recurso apropiado para dicha tarea.
    - Para una petición inicial, los componentes que forman la página tendrán que añadirse al árbol de componentes.
    - Para una petición de postback, el árbol de componentes ya existirá por lo que no será necesario volver a añadirlos.

  Después de que el contenido de la vista haya sido renderizado, el estado de la respuesta se guarda de forma que futuras peticiones puedan acceder a este.

10.3. JavaServer Faces y el patrón MVC

El framework para aplicaciones Web JavaServer Faces está basado en la arquitectura JSP Modelo 2, aunque proporciona un entorno MVC más rico y cercano al patrón MVC tradicional.

El **Controlador** en la arquitectura JSF consiste en un Servlet **Front Controller** llamado *FacesServlet,* un fichero de configuración centralizado llamado faces-config.xml y un conjunto de *Action Handlers* de la aplicación Web. *FacesServlet* es el responsable de recibir las peticiones de los clientes Web y llevar a cabo los pasos necesarios para despachar y preparar una respuesta.

El **Modelo** en JSF es a menudo similar a los ActionForm de Struts que almacenan y dirigen los datos entre las capas de la Vista y el Modelo en una aplicación Web.

La **Vista** en JSF consiste principalmente en un árbol de componentes. Los componentes pueden mostrarse de distintas formas en función del tipo de cliente y la vista delega este trabajo en distintos motores de renderizado que se ocuparán de un tipo específico de salida. Conversores y Validadores pueden utilizarse para convertir y validar los datos de entrada de un usuario. La Vista también cuenta con recursos que puedan resultar útiles por ejemplo para temas de localización (locale) de la aplicación Web.

11. Comparativa de frameworks

	Struts	Struts^2	Spring MVC	JavaServer Faces
Orientación	Acciones y URLs	Acciones y URLs	Acciones y URLs	Componentes
Acciones y formularios	Objetos que extienden de clases base del framework.	POJOs	POJOs	POJOs
Dependencia API Servlet	Alta (Objetos request y response pasados como parámetros).	Muy Baja. Acceso a través de inyección de dependencias.	Muy Baja. Acceso a través de inyección de dependencias.	Muy Baja
Configuración	XML	XML y anotaciones	XML y anotaciones	XML y anotaciones
Tecnologías para la Vista	JSP, JSTL, JSF, Tiles, Velocity, XSLT, Struts Tags, etc.	JSP, JSTL, Velocity, Freemaker, etc.	JSP, JSTL, Velocity, Freemaker, XSLT, etc.	JSP.
Filtros o Interceptores	No	Sí	Sí	Sí
Recolección de datos	A través de ActionForm o DynaBeans.	Consultando las propiedades del Action.	Java Beans	Java Beans
Validación de datos	ActionForms o extensiones como Commons Validator.	XWork Validation.	Compatible con JSR-303, especificación de validación de los JavaBeans.	Permite validación individual de componentes, creación de validadores estándar, métodos validadores en los backing beans, etc.
Facilidad Testing	Baja, requiere extensiones como Struts TestCase.	Alta	Alta	Alta
Soporte nativo de Ajax	No	Sí. A través de etiquetas (DOJO).	No	Sí
Aceptación	Ya en desuso aunque fue muy alta	Alta	Muy Alta	Muy Alta

Figura 37 - Comparativa frameworks

Capitulo III – Análisis y diseño

1. Requisitos del sistema

Requisitos directos del Proyecto.

- Desarrollo de un framework que simplifique y agilice el desarrollo de la capa de presentación en aplicaciones Web Thin Client desarrolladas en Java EE.
- El framework debe proporcionar la implementación del patrón **Model**-**View**-**Controller** (MVC) y facilitar el desarrollo de la capa de presentación.

Implícitamente relacionados con los requisitos directos del proyecto se identifican los siguientes:

- Debe existir un componente central (**Controlador)** encargado de recibir y procesar las peticiones de los usuarios del sistema, delegando en otros componentes las siguientes acciones:
    - Validación de los datos de entrada de la petición.
    - Procesamiento de la lógica de negocio necesaria para servir la petición.
    - Presentación de los resultados vinculados a una petición.
- El framework permitirá la configuración declarativa de los componentes relacionados con cada petición y el controlador utilizará esta configuración para ejecutar el procedimiento necesario para servir cada una.
- El sistema permitirá la creación de componentes **Vista** que generen la respuesta que convenga en función de las peticiones recibidas.
- El sistema utilizará los componentes **Modelo** para llevar a cabo la lógica de negocio necesaria para servir cada petición y así obtener los resultados que la vista utilizará para generar la respuesta.

Adicionalmente, extraídos tras el estudio de los distintos frameworks del mercado, se añaden como requisitos adicionales las siguientes características:

- **Colecciones de etiquetas**. Soporte para la clara separación entre la lógica de preparación y el formateo de los datos fuera de las páginas JSP.
- **Internacionalización**. Proporcionar mecanismos que permitan internacionalizar los mensajes de una aplicación.
- **Gestión de Excepciones**. Soporte para el tratamiento personalizado de excepciones.

- **Interceptores a nivel de acción**. Soporte para la creación de interceptores que permitan la ejecución de código a nivel de acción antes y después de su ejecución.

2. Modelo de Casos de Uso

A nivel de análisis, los casos de uso permiten identificar las funciones de un sistema software desde el punto de vista de sus interacciones con el exterior.

Dado que el sistema en cuestión es un framework para la capa de presentación de aplicaciones Web, se identifica claramente al desarrollador de una aplicación de este tipo como entidad externa que quiere interaccionar con el sistema. Estas interacciones entre el sistema y el desarrollador (actor) quedan reflejadas en los distintos casos de uso identificados.

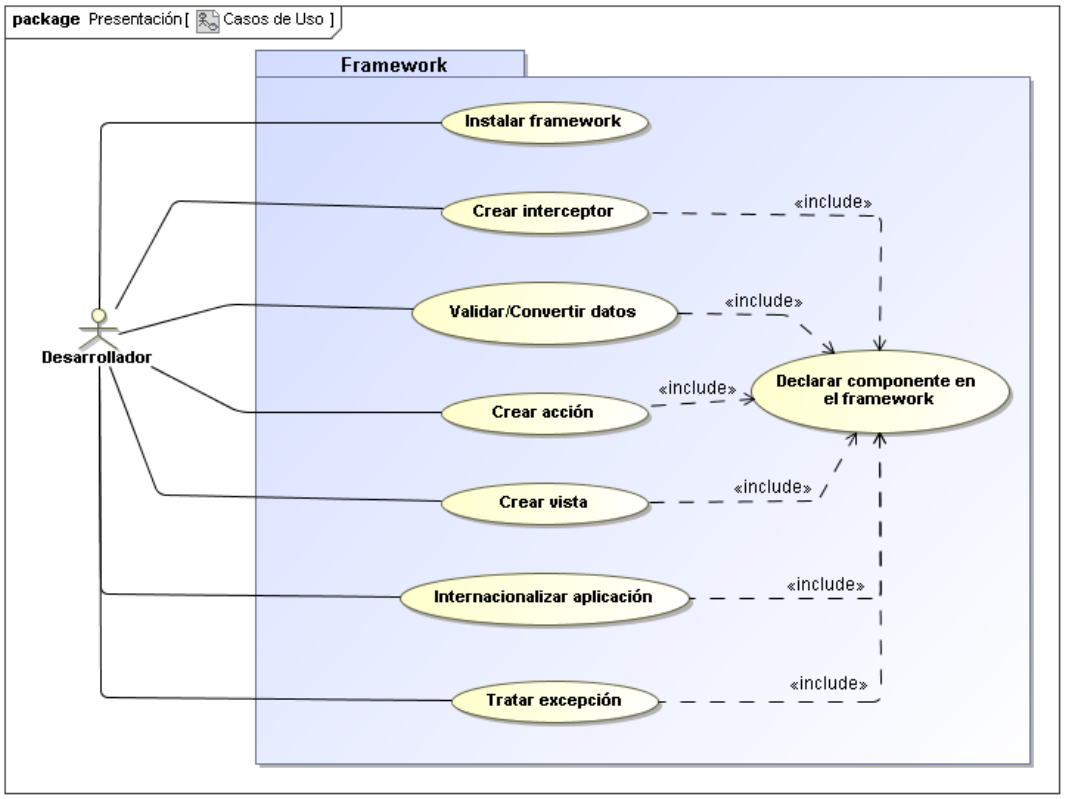

Figura 38 – Modelo de casos de uso

Caso de uso <<Instalar framework>>

- **Resumen de la funcionalidad**: Se configura la aplicación web para que utilice el framework y se proporcionan las librerías y demás utilidades necesarias para su funcionamiento.
- **Actor(es)**. Desarrollador.
- **Casos de uso relacionados**: Ninguno.
- **Pre-condición**: El framework no ha sido instalado previamente en la aplicación web.
- **Post-condición**: La aplicación web dispone del framework instalado y configurado para empezar a desarrollar con él.
- **Propósito.** Configurar un proyecto web para que pueda hacer uso del framework.
- **Flujo de acontecimientos.**
    - El desarrollador quiere utilizar el framework en el desarrollo de la capa de presentación en su aplicación Web.
    - El desarrollador instala y configura en su proyecto las librerías requeridas por el framework para su funcionamiento.

Caso de uso <<Crear interceptor>>

- **Resumen de la funcionalidad:** El desarrollador crea un interceptor con el fin de ejecutar un código antes y/o después de ejecutarse la acción vinculada a una petición.
- **Actor(es)**: Desarrollador.
- **Casos de uso relacionados**: *Declarar componente en el framework.*
- **Pre-condición**: Ninguna.
- **Post-condición**: Se ha definido una lógica vinculada con el procesamiento previo y/o posterior a la ejecución de una acción.
- **Propósito**: Crear una funcionalidad que pueda ejecutarse antes y/o después del procesamiento de una acción.
- **Flujo de acontecimientos.**
    - El desarrollador quiere ejecutar cierta lógica antes y/o después de la ejecución de una acción.
    - El desarrollador implementa el interceptor definiendo la lógica vinculada con el procesamiento previo y posterior a la ejecución de la acción vinculada a una petición.
    - El desarrollador declara el interceptor en el framework (caso de uso “Declarar componente en el framework”).

Caso de uso <<Validar/Convertir datos de entrada>>

- **Resumen de la funcionalidad:** El desarrollador implementa una lógica que permite validar los datos recibidos en una petición así como convertirlos al tipo más conveniente.
- **Actor(es)**: Desarrollador.
- **Casos de uso relacionados**: *Declarar componente en el framework.*
- **Pre-condición**: Ninguna.
- **Post-condición**: Se ha creado una clase que valida/convierte datos que viajan en una petición HTTP y ésta clase está declarada como nuevo componente en el framework.
- **Propósito**: Dado que los datos enviados a través de un formulario viajan en la petición HTTP sin un tipo más allá del de una cadena de texto. El propósito de este caso de uso es el de proporcionar un mecanismo que permita por una parte comprobar que los datos recibidos son válidos, o no se han informado (validaciones simples) y por otra convertir estos datos a tipos más adecuados según los valores que transporten.
- **Flujo de acontecimientos.**
    1. El desarrollador quiere recuperar/convertir los datos recibidos a través de una petición HTTP.
    2. El desarrollador implementa una clase que lleva a cabo la validación/conversión de los datos recibidos en la petición.
    3. El desarrollador declara el Validador/Conversor en el framework (caso de uso “Declarar componente en el framework”).

Caso de uso <<Crear acción>>

- **Resumen de la funcionalidad:** El desarrollador quiere dotar al sistema de una nueva funcionalidad.
- **Actor(es)**: Desarrollador.
- **Casos de uso relacionados**: *Declarar componente en el framework*.
- **Pre-condición**: Ninguno.
- **Post-condición**: Se ha creado una acción encargada de procesar cierta petición.
- **Propósito**: Dotar de una nueva funcionalidad a la aplicación web.
- **Flujo de acontecimientos.**
    1. El desarrollador implementa una nueva funcionalidad que responderá a una determinada petición.

2. El desarrollador declara la acción en el framework (caso de uso "Declarar componente en el framework").
3. El desarrollador podrá vincular una acción a otros componentes como (Vistas, Validación/conversión de datos o interceptores).

Caso de uso <<Tratar excepción>>

- **Resumen de la funcionalidad:** El desarrollador quiere dotar al sistema mediante el que se traten las excepciones sucedidas.
- **Actor(es)**: Desarrollador.
- **Casos de uso relacionados**: *Declarar componente en el framework*.
- **Pre-condición**: Ninguno.
- **Post-condición**: Se ha creado un componente encargado de tratar excepciones.
- **Propósito**: Dotar de una nueva funcionalidad a la aplicación web mediante la cual se capturen y traten convenientemente ciertas situaciones erróneas.
- **Flujo de acontecimientos.**
    1. El desarrollador implementa una nueva funcionalidad que responderá ante un determinado tipo de excepción.
    2. El desarrollador declara el componente encargado de la gestión de la excepción en el framework (caso de uso "Declarar componente en el framework").

Caso de uso <<Crear vista>>

- **Resumen de la funcionalidad:** El desarrollador implementa la presentación de datos.
- **Actor(es)**: Desarrollador.
- **Casos de uso relacionados**: *Declarar componente en el framework*.
- **Pre-condición**: Ninguna
- **Post-condición**: El desarrollador ha definido una nueva plantilla de presentación de datos.
- **Propósito**: Dotar de una plantilla que muestre cierta información en un formato concreto.
- **Flujo de acontecimientos.**
    1. El desarrollador implementa una nueva plantilla de datos.
    2. El desarrollador declara la nueva vista en el framework (caso de uso "Declarar componente en el framework").

Caso de uso <<Internacionalizar aplicación>>

- **Resumen de la funcionalidad:** El desarrollador debe dotar a la aplicación web de la posibilidad de traducir sus textos a varios idiomas.
- **Actor(es)**: Desarrollador.
- **Casos de uso relacionados**: *Declarar componente en el framework*.
- **Pre-condición**: Ninguna
- **Post-condición**: Se dispone de los ficheros de recursos con las traducciones de todos los mensajes a tantos idiomas como se requieran.
- **Propósito**: Dar la posibilidad a la aplicación web de cambiar de un idioma a otro fácilmente.
- **Flujo de acontecimientos.**
    1. El desarrollador crea tanto ficheros de traducciones como idiomas quiera poder utilizar en su aplicación.
    2. El desarrollador habilita la internacionalización de la aplicación en el framework (caso de uso “Declarar componente en el framework”).

Caso de uso <<Declarar componente en el framework>>

- **Resumen de la funcionalidad:** El desarrollador define de forma declarativa un nuevo componente que estará disponible en el framework.
- **Actor(es)**: Desarrollador.
- **Casos de uso relacionados**: Ninguno.
- **Pre-condición**: Es necesario que el componente exista previamente antes de declararlo en el framework.
- **Post-condición**: El componente ha sido declarado y por tanto el framework será capaz de interactuar con él.
- **Propósito**: Declarar el componente implementado con el fin de que el framework tenga constancia de él y pueda utilizarlo.
- **Flujo de acontecimientos.**
    1. El desarrollador tras haber implementado un componente (Acción, Validación, Vista, Interceptor o Excepción), quiere declararlo en la configuración del framework para que pueda utilizarse.

3. Diagrama de clases del modelo estático

El modelo estático de un sistema es aquel en el que se describen las clases y los objetos. En este caso, el diagrama de clases muestra la estructura estática de las clases en el dominio que nos ocupa, presentando tanto las clases como las relaciones entre ellas a lo largo del tiempo.

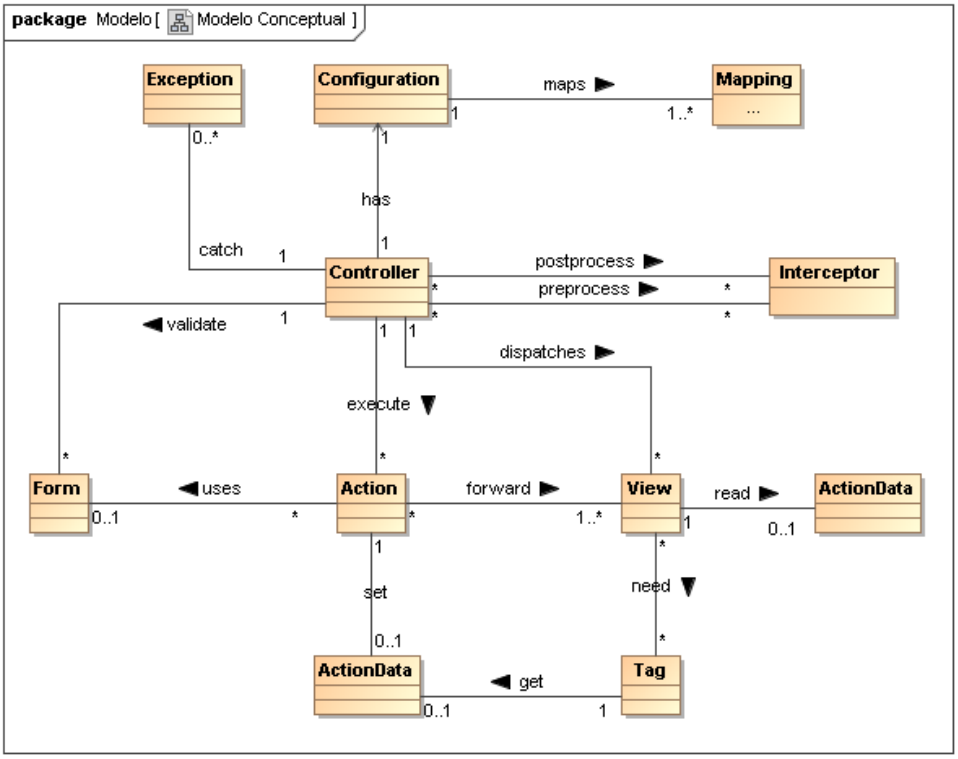

Figura 39 - Diagrama de clases del modelo estático

4. Diseño del modelo conceptual

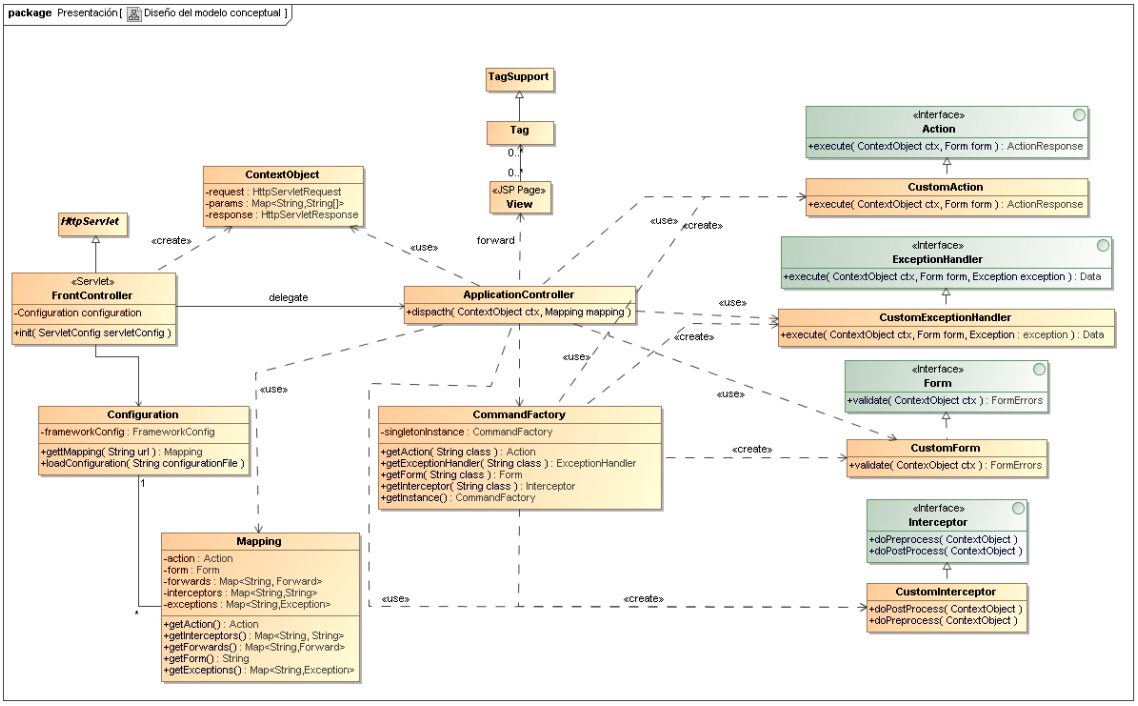

Figura 40 - Diagrama de diseño del modelo conceptual

La figura anterior muestra el diagrama de diseño del framework JavaMVC. En éste, se ha obviado la definición de algunas clases dado que su inclusión en el mismo, habría complicado innecesariamente el diagrama sin aportar información relevante para su comprensión.

Descripción de las clases del diagrama

- **FrontController** – Clase encargada de proporcionar un único punto de acceso a la capa de presentación. Durante el periodo de despliegue será la encargada de iniciar el proceso responsable de llevar a cabo la configuración del framework, posteriormente, será el punto de entrada de todas las peticiones que reciba la aplicación. Esta clase implementa el patrón Front Controller.
- **ContextObject** – Clase encargada de encapsular los detalles de cada petición recibida de forma que puedan ser compartidos en toda la aplicación ocultando los detalles del protocolo utilizado (http). Como su nombre indica se trata de una implementación del patrón Context Object.
- **Configuration** – Clase generada durante la fase de despliegue de la aplicación. Su misión es la de procesar el fichero de configuración del framework (XML en el que el desarrollador ha definido las acciones, formularios, excepciones, etc.) para poder, a posteriori, saber cómo tratar cada una de las peticiones recibidas por la aplicación.

- **Mapping** – Clase que encapsula la configuración asociada a una petición definida en el fichero de configuración del framework. Esta información será necesaria para que la clase ApplicationController sepa de dónde obtener los componentes necesarios para procesar la petición en curso.
- **ApplicationController** – Clase responsable de la gestión las peticiones que llegan al sistema (liberando de esta responsabilidad al componente FrontController), siguiendo el flujo definido en el fichero de configuración del framework. Este flujo puede implicar la ejecución de filtros, validación de los datos de entrada, ejecución de la acción asociada, tratamiento de excepciones, etc. para finalmente delegar en la vista la responsabilidad de presentar los resultados. Implementa el patrón ApplicationController.
- **CommandFactory**. Clase encargada de crear instancias de cada uno de los componentes implicados en el flujo de tratamiento de una petición. Implementa el patrón Factoría Concreta y el patrón Singleton.
- **Form**. Interfaz a implementar cuando es necesario realizar conversiones y/o validaciones sobre los datos enviados en una petición.
- **Action**. Interfaz a implementar para llevar a cabo la acción vinculada a una petición. Esta acción representa el conjunto de flujos y operaciones de negocio necesarias para dar respuesta a una petición.
- **Interceptor**. Interfaz a implementar cuando se requiera dotar a un Action de un tratamiento previo y/o posterior a su ejecución.
- **ExceptionHandler**. Interfaz a implementar cuando se requiera un comportamiento específico por parte de la aplicación ante una situación de excepción concreta.
- **View.** Página JavaServer Pages utilizada para mostrar la respuesta a una petición.
- **Tag**. Clase encargada de asistir a los componentes View durante la generación de contenido dinámico minimizando la cantidad de código scriptlet embebido. Implementa el patrón View Helper.

5. Diagramas de secuencia

Para facilitar la comprensión del funcionamiento del framework, se muestran a continuación los diagramas de secuencia de las dos operaciones realizables. La operación de carga del framework en el contenedor se realizará de forma única, mientras que la gestión de una petición se realizará tantas veces como peticiones reciba el FrontController.

6. Configuración inicial

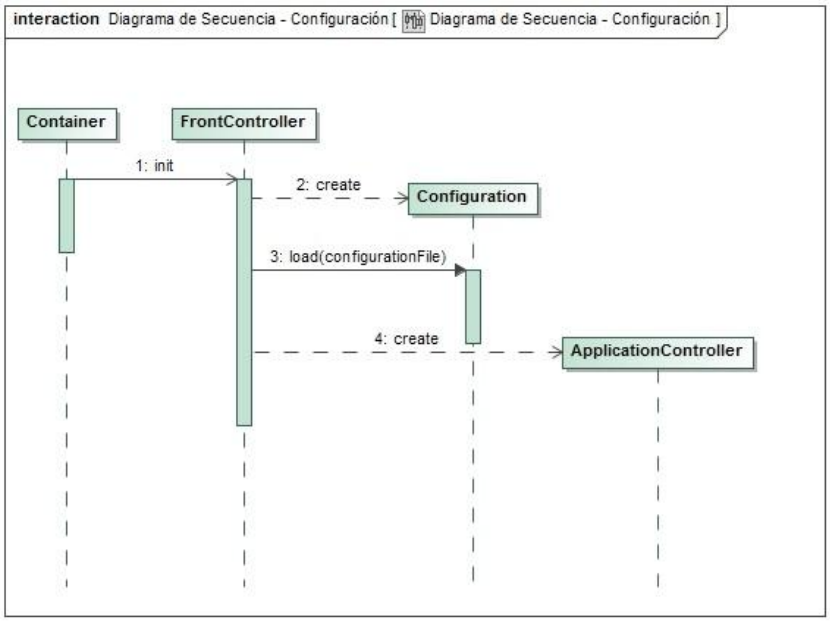

Figura 41 - Diagrama de secuencia - Configuración

El proceso de configuración se inicia en algún momento entre el despliegue de la aplicación en el contenedor de Servlets y la primera petición que deriva en el FronController. En el proceso de configuración, el contenedor lee el fichero de despliegue de la aplicación Web (web.xml) y en ella encuentra definido el Servlet que iniciará el proceso de configuración del framework. Este Servlet actuará como **FrontController** siendo el foco de ahora en adelante de todas las peticiones que reciba la aplicación, bajo cierto patrón de Urls, una vez desplegada.

1. El contenedor carga en memoria el Servlet **FrontController** y ejecuta su método init().
2. La invocación del método init() inicia el proceso de configuración del framework que comienza creando una instancia de la clase **Configuration**.
3. A continuación se llama al método load de la clase **Configuration** que se encargará de leer y procesar el fichero XML que configurará la aplicación en el framework.
4. Finalmente se crea una instancia de la clase **ApplicationController** que será utilizada de ahora en adelante para gestionar todas las peticiones.

7. Petición correcta completa

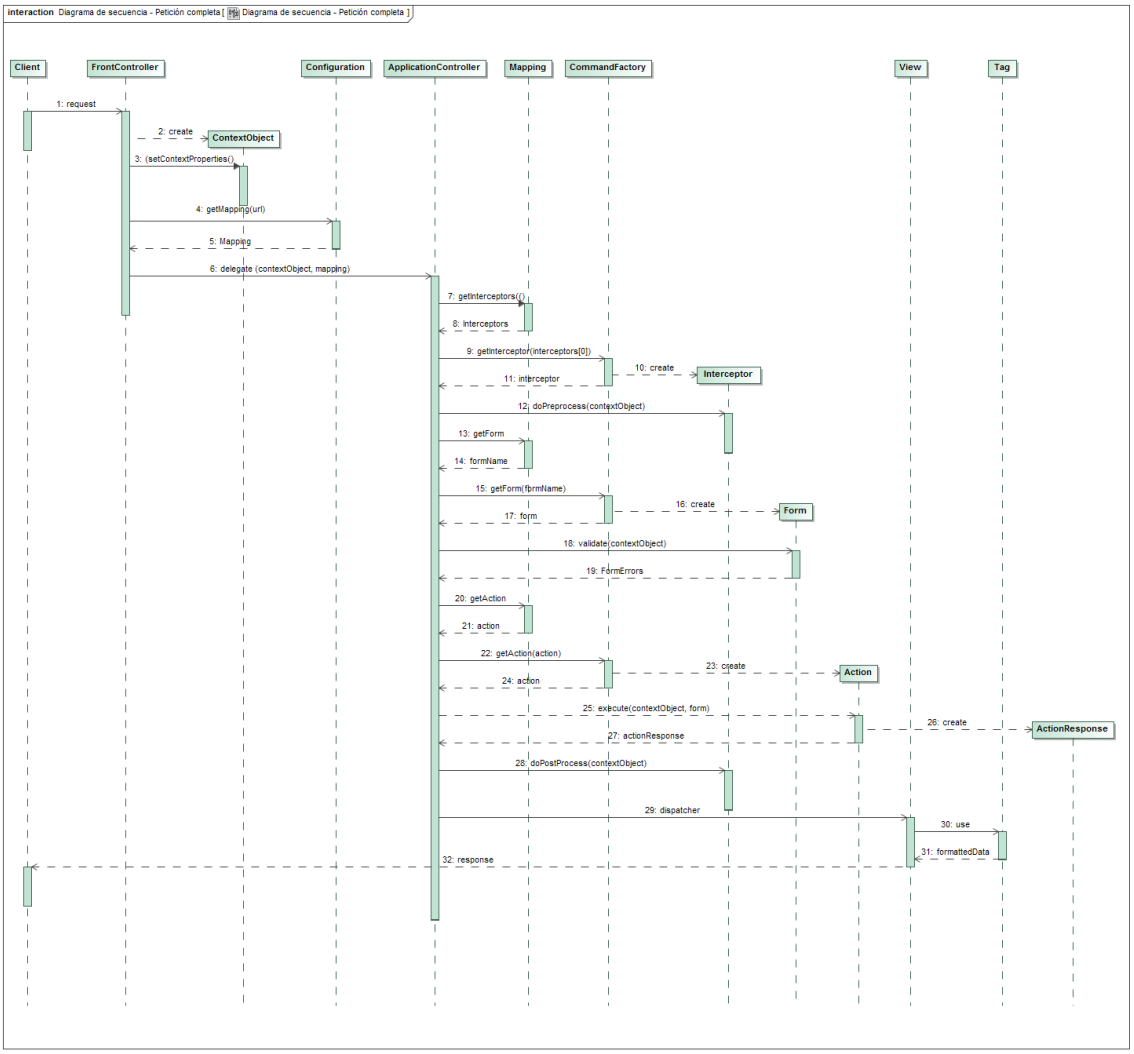

Figura 42 - Diagrama de secuencia petición completa

1. El cliente envía una petición y ésta es capturada por el **FrontController**.
2. El **FrontController** crea un objeto **ContextObject**.
3. Se configura el **ContextObject** con las propiedades relevantes de la petición.
4. **FrontController** solicita a la clase **Configuration** el mapeo vinculado con la petición recibida.
5. La clase **Configuration** devuelve un objeto **Mapping** que contiene la acción, formulario, excepciones, interceptores etc. que puedan ser de utilidad para servir la petición.
6. La clase **FrontController** delega en **ApplicationController** el tratamiento de la petición pasándole el objeto contexto y el objeto **Mapping** vinculado a la petición a tratar.
7. La clase **ApplicationController** pregunta al objeto **Mapping** por los interceptores vinculados con la petición en curso.

8. La clase **Mapping** devuelve los interceptores vinculados a la petición.
9. Se solicita a la clase **CommandFactory** la fabricación del interceptor. Si existiese más de uno este proceso se repetiría para cada uno de ellos, preservando el orden definido en el fichero de configuración.
10. **CommandFactory** crea una nueva instancia del interceptor.
11. **CommandFactory** devuelve la instancia del interceptor.
12. **ApplicationController** ejecuta el método doPreprocess para llevar a cabo las tareas previas a la ejecución de la acción solicitada en la petición.
13. **ApplicationController** solicita el formulario vinculado con la petición en curso.
14. **Mapping** devuelve el formulario vinculado a la petición si lo hay.
15. **ApplicationController** solicita a **CommandFactory** la creación del objeto Formulario.
16. **CommandFactory** localiza la implementación del objeto formulario y la instancia.
17. **CommandFactory** devuelve un objeto de tipo Form.
18. **ApplicationController** invoca al método validate del objeto formulario, proporcionándole el ContextObject vinculado con la petición en curso.
19. El objeto formulario devuelve un objeto de tipo FormErrors. Este objeto estará vacío si la validación de los datos del formulario ha sido correcta y tendrá elementos si por el contrario se han encontrado errores.
20. **ApplicationController** solicita a la clase **Mapping** los datos de la acción vinculada con la petición en curso.
21. **Mapping** devuelve la acción vinculada con la petición.
22. **ApplicationController** solicita a **CommandFactory** la creación del objeto Action.
23. **CommandFactory** localiza la implementación del objeto Action y la instancia.
24. **CommandFactory** devuelve el objeto Action.
25. **ApplicationController** ejecuta el método execute con el que llevar a cabo el procesado de la petición.
26. El **Action** como resultado de la ejecución de la lógica de negocio, crea un objeto **ActionResponse** que encapsula tanto los datos como el recurso que tratará con ellos para dar respuesta a la petición del usuario.
27. El **Action** devuelve al **ApplicationController** el objeto ActionResponse creado.
28. **ApplicationController**, antes de delegar el procesamiento de la petición en la vista para que ésta genere el contenido que se presentará al usuario, ejecutará el método doPostProcessing() en el interceptor vinculado a la petición. En caso de que hubiese más de un interceptor vinculado con el **Action**, estos se ejecutarían en orden inverso al que se ejecutaron al inicio.

29. **ApplicationController** delega en la vista la presentación de los datos.
30. La **View** utiliza a la clase **Tag** para que ésta le proporcione ciertos datos formateados.
31. La clase **Tag** genera el resultado y se lo entrega a la vista.
32. Finalmente el contenido generado por la clase **View** se envía de vuelta al usuario en respuesta a la petición realizada.

Capítulo IV – Implementación

1. Estructura de paquetes

El código fuente del framework JavaMVC está organizado en un conjunto de paquetes según su funcionalidad. De esta forma se facilita la clasificación de la funcionalidad implementada sirviendo como medio para evitar la duplicidad, facilitar la reutilización y agilizar la localización.

Nombre del paquete	Descripción
edu.uoc.framework.action	Recoge los elementos necesarios para la implementación de una acción que pueda ser servida por el framework.
edu.uoc.framework.core	Recoge las clases que forman el núcleo de procesamiento del framework.
edu.uoc.framework.core.configuration	Recoge las clases utilizadas por JAXB durante el proceso de lectura del fichero XML de configuración del framework.
edu.uoc.framework.core.configuration.adapters	Recoge las clases adaptadoras que redefinen el comportamiento de JAXB durante el proceso de unmarshal del fichero de configuración del framework.
edu.uoc.framework.exception	Recoge las excepciones que pueden suceder en el framework durante su ejecución.
edu.uoc.framework.exceptionHandler	Recoge la interfaz necesaria para llevar a cabo el tratamiento personalizado de excepciones en una aplicación.
edu.uoc.framework.form	Recoge los elementos necesarios para trabajar con formularios de datos cuando éstos se envían adjuntos a una petición.
edu.uoc.framework.helpers	Recoge las estructuras de ayuda utilizadas por el framework para la gestión de los datos resultantes tras la ejecución de una operación.
edu.uoc.framework.interceptor	Recoge la interfaz que debe implementar una clase para que ésta sea tratada como interceptor de una acción.
edu.uoc.framework.utilities	Recoge una colección de clases que sirven de apoyo al framework para la internacionalización, reflexión, etc.
edu.uoc.framework.utilities.I18N	Recoge las clases utilizadas por el framework para dar soporte a la internacionalización de aplicaciones.
edu.uoc.framework.view	Recoge aquellas clases relacionadas o de soporte para la construcción de la vista.
edu.uoc.framework.view.tags	Recoge las clases que implementan las etiquetas proporcionadas por el framework y que sirven de apoyo a la Vista.

2. Dependencias

JavaMVC se apoya en una serie de librerías externas que le son de suma importancia para implementar su funcionalidad.

En primer lugar, dado que la estrategia de implementación utilizada para el patrón **Front Controller** es la conocida como **Servlet Front**, será necesario poder utilizar Servlets para llevarla a cabo y por ello, se ha optado por incluir la implementación de la especificación Servlet 3.0.

Por otra parte, la necesidad de mostrar contenido HTML como resultado a las peticiones recibidas por el framework, separando la presentación de la lógica de la aplicación, hace necesario el uso de una tecnología que conviva en armonía con la tecnología Servlet y permita llevar a cabo esta tarea. Para ello se ha utilizado la implementación de la especificación JSP 2.1.

Finalmente, dada la necesidad de contar con un sistema de logs que permita conocer las trazas del framework durante su ejecución de forma que facilite la localización e identificación de los errores que puedan aparecer durante el desarrollo, ha motivado a incluir la librería Log4j 1.2.17.

El siguiente diagrama muestra, por tanto, las dependencias que cualquier proyecto web en Java EE debe satisfacer si quiere utilizar el framework JavaMVC.

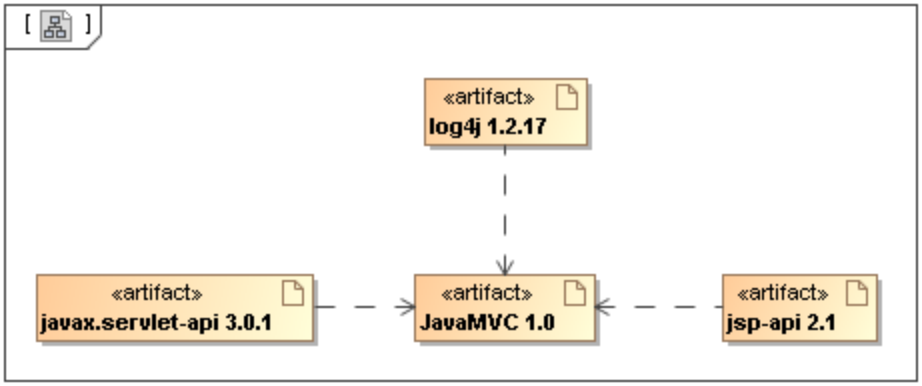

Figura 43 - Diagrama de Dependencias

3. Manual de usuario de JavaMVC

JavaMVC es un framework de presentación implementado con tecnología Java EE. Éste se distribuye en un fichero Jar que debe incluirse como librería en cualquier aplicación Web Java EE que quiera implementar el patrón arquitectónico MVC bajo el soporte de este framework.

Configuración de la aplicación Web

JavaMVC, al igual que otros frameworks de presentación, utiliza el patrón Front Controller para proporcionar un único punto de entrada en la capa de presentación de una aplicación Web. La implementación de este patrón se lleva a cabo mediante tecnología Servlet por lo que, toda aplicación Web Java EE que utilice JavaMVC, tendrá declarar el Servlet **FrontController** en su descriptor de despliegue (web.xml).

```
<servlet>
	<display-name>FrontController</display-name>
	<servlet-name>FrontController</servlet-name>
	<servlet-class>edu.uoc.framework.core.FrontController</servlet-class>
	<init-param>
		<param-name>config</param-name>
		<param-value>/WEB-INF/classes/resources/JavaMVC.xml</param-value>
	</init-param>
	<init-param>
		<param-name>encoding</param-name>
		<param-value>UTF-8</param-value>
	</init-param>
	<load-on-startup>1</load-on-startup>
</servlet>

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

Figura 44 - Declaración de FrontController en el fichero web.xml

La figura anterior muestra un ejemplo de declaración del Servlet **FrontController** en el descriptor de despliegue de una aplicación Web Java EE**.** De esta declaración cabe destacar tres puntos:

- Parámetro obligatorio de inicialización **config** '<param-name>config</param-name>' que indica la ubicación del fichero que configurará el framework.
- Parámetro opcional de inicialización **encoding** '<param-name>encoding</param-name>' que indica la codificación que el Servlet **FrontController** utilizará por defecto a la hora de recuperar los datos enviados en una petición.
- El patrón de URLs atrapadas por el Servlet **FrontController** '<url-pattern>*.do</url-pattern>', que permitirá un único mapeo que podrá hacerse o bien por extensión o bien por path.

Configuración del framework

JavaMVC utiliza un fichero de configuración en formato XML mediante el cual se establece el vínculo de unión entre la aplicación Web y el framework. En este fichero el desarrollador declara aquellos componentes de la aplicación Web que deberán ser gestionados por el framework. Con el fin de facilitar tanto la creación de este fichero como la validación del mismo, JavaMVC proporciona el esquema **JavaMVC.xsd** que define el contrato que todo XML de configuración debe cumplir.

```
<?xml version="1.0" encoding="UTF-8"?>
<jmvc:framework-config xmlns:jmvc="http://www.resources.com/jmvc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.resources.com/jmvc JavaMVC.xsd ">

	<jmvc:forms>
		<jmvc:form name="CreateUserForm"
		class="edu.uoc.phonebook.presentation.forms.CreateUserForm"/>
	</jmvc:forms>

	<jmvc:interceptors>
	   <jmvc:interceptor name="int1"
	class="uoc.edu.phonebook.interceptors.Interceptor1"/>
	</jmvc:interceptors>

	<jmvc:exceptions>
	   <jmvc:exception jsp="/jsp/error.jsp" type="java.lang.Exception"
	handler="edu.uoc.phonebook.presentation.exceptions.PhonebookExceptionH
	andler"/>
	</jmvc:exceptions>

	<jmvc:actions>
	   <jmvc:action name="CreateUser"
	class="edu.uoc.phonebook.presentation.actions.CreateUser"
	actionForm="CreateUserForm" input="/jsp/createUser.jsp">
		<jmvc:interceptor-ref name="int1"/>
		<jmvc:forward name="ok" path="/Login.do"/>
		<jmvc:forward name="error" path="/jsp/createUser.jsp"/>
	   </jmvc:action>
	</jmvc:actions>

	<jmvc:i18n parameter="edu.uoc.phonebook.i18n.messages"/>
</jmvc:framework-config>
```

Figura 45 - Ejemplo fichero de configuración

La figura anterior muestra un ejemplo de XML de configuración, en el que aparecen todos los elementos permitidos por el framework y que, a continuación, serán descritos al detalle.

Todo XML que siga el esquema **JavaMVC.xsd** y que por tanto será válido como fichero de configuración para el framework, contendrá una etiqueta <**framework-config**> que englobará un máximo de 5 secciones que a continuación pasan a describirse.

1. <**forms**>: Sección que declara los formularios disponibles en la aplicación. Un documento XML de configuración podrá opcionalmente contener una etiqueta de este tipo como máximo.
    - <**form**>: Etiqueta que describe un formulario y que, únicamente, podrá aparecer como etiqueta hija de <**forms**>. Su número de apariciones no está limitado y cada aparición declarará obligatoriamente los siguientes atributos:
        - **name**: Nombre que identifica al formulario.
        - **class**: Clase que contiene la implementación de la interfaz Form.

2. <**interceptors**>: Sección que declara los interceptores disponibles en la aplicación. Un documento XML de configuración podrá, opcionalmente, contener una etiqueta de este tipo como máximo.
    - <**interceptor**>: Etiqueta que describe un interceptor y que, únicamente, podrá aparecer como etiqueta hija de <**interceptors**>. Su número de apariciones no está limitado y cada aparición declarará obligatoriamente los siguientes atributos:
        - **name**: Nombre que identifica al interceptor.
        - **class**: Clase que contiene la implementación de la interfaz Interceptor.

3. <**exceptions**>: Sección que declara las excepciones que serán tratadas según la lógica definida por el usuario. Un documento XML de configuración podrá, opcionalmente, contener una etiqueta de este tipo como máximo.
    - <**exception**>: Etiqueta que describe al encargado de manejar una excepción y que, únicamente, podrá aparecer como etiqueta hija de <**exceptions**>. Su número de apariciones no está limitado y cada aparición declarará obligatoriamente los siguientes atributos:
        - **jsp**: Jsp de destino mostrada tras la captura de la excepción.
        - **type**: Nombre de la Exception que desea capturarse.
        - **handler**: Clase que implementa la interfaz ExceptionHandler y que será la encargada de llevar a cabo la lógica definida por el usuario para el tratamiento de la excepción.

4. <**actions**>: Sección que declara las acciones disponibles en la aplicación. Un documento XML de configuración contendrá una etiqueta de este tipo como máximo.
    - <**action**>: Etiqueta que describe una acción y que, únicamente, podrá aparecer como etiqueta hija de <**actions**>. Su número de apariciones no está limitado y cada aparición podrá declarar los siguientes atributos:
        - **name**: Nombre asociado al Action. (Obligatorio).
        - **class**: Clase que implementa la interfaz Action. (Obligatorio).
        - **actionForm:** Nombre del formulario. (Opcional).
        - **input**: Path de la página que enviará el formulario de datos. (Opcional).
    - <**interceptor**-**ref**>: Etiqueta que referencia a un interceptor y que, únicamente, podrá aparecer como etiqueta hija de <**action**>. Su número de apariciones no está limitado y cada aparición declarará obligatoriamente el siguiente atributo:
        - **name**: Nombre del interceptor.
    - <**forward**>: Etiqueta que describe un recurso de destino vinculado al Action. Su número de apariciones será de cómo mínimo una sin que exista limitación en su número máximo de apariciones. Cada aparición podrá declarar los siguientes atributos:
        - **name**: Nombre asociado al recurso. (Obligatorio).
        - **path**: Ubicación del recurso destino. (Obligatorio).
        - **redirect**: Flag que indica si el recurso destino se alcanzará con o sin redirección. (Opcional).

5. <**i18n**>: Etiqueta que define el paquete de recursos utilizado para la internacionalización de mensajes. Un documento XML de configuración podrá contener una etiqueta de este tipo como máximo.
    - **parameter**: paquete de recursos que se utilizará para la internacionalización de la aplicación.

Tal y como se ha visto en el fichero de configuración, cinco son los componentes que el framework pone a disposición del desarrollador para la implementación de una aplicación Web que siga el patrón MVC y sea gestionada por JavaMVC.

Formularios

JavaMVC llama formulario a aquellas clases que implementan la interfaz Form. El cometido de los formularios es el de recuperar y/o transformar los datos recibidos del usuario (normalmente resultado del envío de un formulario HTML) permitiendo, además, llevar a cabo una serie de comprobaciones que permitan verificar la validez de los mismos antes de entregarlos al Action.

Todo formulario en JavaMVC debe declarar tantos atributos de clase como parámetros desee recuperar de la petición entrante. El nombre de estos atributos debe coincidir con el nombre del componente HTML que recoge su valor en el formulario enviado. Es obligatorio que, para todos estos atributos, existan métodos de get/set siguiendo la convención JavaBean, de esta forma, el framework será capaz de poblar el objeto con los valores recibidos y el desarrollador podrá consultar estos valores para llevar a cabo las acciones que considere oportunas.

La interfaz Form declara la siguiente firma para el método validate.

```
public FormErrors validate(ContextObject contextObject) throws Exception;
```

Como resultado de la validación de un formulario, el método validate podrá devolver o no una instancia de la clase FormErrors. Si la validación devuelve un objeto FormErrors sin errores o null, el framework considerará que la validación ha sido correcta y, por tanto, continuará el proceso de la petición delegando su tratamiento en el Action solicitado en la misma. Si por el contrario la validación no es correcta, el objeto FormErrors estará poblado con los errores detectados durante la validación de los datos y el proceso finalizará devolviendo los datos y los errores encontrados a la vista desde la que se enviaron.

Interceptores

JavaMVC llama interceptor a aquellas clases que implementen de la interfaz Interceptor. El cometido de los interceptores es permitir la ejecución de cierta lógica antes y después de llevar a cabo la acción solicitada en una petición.

La interfaz Interceptor declara los métodos doPreprocess y doPostprocess con las siguientes firmas:

```
public void doPreprocess(ContextObject contextObject) throws Exception;
public void doPostprocess(ContextObject contextObject) throws Exception;
```

La implementación del método doPreprocess permitirá llevar a cabo cierta lógica antes de que la acción se lleve a cabo, antes incluso de ejecutar el formulario asociado a la acción si ésta

definiese uno. Por otra parte, el método doPostprocess permitirá la ejecución de cierta lógica después de haber llevado a cabo la lógica de la acción solicitada.

Si una acción define más de un interceptor, éstos ejecutarán su método doPreprocess en el orden en el que fueron declarados en el fichero de configuración del framework. Por el contrario, la ejecución del método doPostprocess se llevará a cabo en orden inverso.

Acciones

Las acciones en JavaMVC son el componente más importante e imprescindible que toda aplicación Web necesitará para justificar el uso del framework.

JavaMVC entiende por acción aquellas clases que implementan la interfaz Action y cuyo cometido es el de llevar a cabo cierta lógica como respuesta a una petición. En una aplicación Web basada en el patrón arquitectónico MVC implementada bajo JavaMVC, una acción forma parte del controlador de la capa de presentación y su funcionalidad será la de recibir las peticiones del usuario, ejecutar la lógica de negocio necesaria para dar respuesta a dicha petición y determinar qué recurso se encargará de presentar el resultado.

La interfaz Action declara la siguiente firma para el método execute:

```
public ActionResponse execute(ContextObject contextObject, Form form) throws Exception;
```

La implementación de este método permitirá la invocación de la lógica de negocio (correspondiente al modelo en una arquitectura MVC) que en JavaMVC queda abierta a cualquier tecnología o implementación que interese al desarrollador.

La ejecución del método execute devolverá un objeto ActionResponse que encapsulará tanto los datos a enviar como el nombre del recurso al que deben enviarse.

Tratamiento de excepciones

JavaMVC entiende por manejador de excepciones aquellas clases que implementan la interfaz ExceptionHandler y cuyo cometido es el de informar al framework sobre la lógica que debe ejecutarse cuando, durante la ejecución de la aplicación Web, se produzca una determinada excepción.

La interfaz ExceptionHandler declara la siguiente firma para el método execute:

```
public Data execute(ContextObject contextObject, Form form, Exception exception) throws Exception;
```

La implementación de este método permite al desarrollador llevar a cabo la lógica de tratamiento de la excepción que considere necesaria, permitiendo el envío de datos encapsulados en la clase Data de forma que la JSP de respuesta vinculada a la implementación de ExceptionHandler se encargue de mostrarlos al usuario.

Etiquetas de soporte

JavaMVC proporciona una pequeña colección de etiquetas que ayudarán a eliminar de la vista toda aquella lógica que no esté directamente relacionada con la presentación, proporcionando de esta forma, un código más limpio que facilite la participación de diseñadores o maquetadores en el proyecto.

Las etiquetas proporcionadas por JavaMVC se organizan alrededor de dos librerías:

- I18N: Librería de etiquetas destinada a la internacionalización de aplicaciones.
- HTML: Librería de etiquetas destinada a la generación de código HTML.

La librería I18N contiene las siguientes etiquetas:

- **Error** - Etiqueta destinada a mostrar los mensajes de error internacionalizados enviados en respuesta a una petición. Esta etiqueta permite su configuración a través de los siguientes atributos.
    - **bundle**: Indica la ubicación del paquete de recursos que la etiqueta utilizará para internacionalizar el mensaje. En ausencia de este atributo, la etiqueta utilizará el paquete de recursos definido en el fichero de configuración del framework.
    - **locale**: Indica el nombre del atributo de sesión que contendrá el Locale que la etiqueta utilizará para internacionalizar el mensaje. En ausencia de éste, el Locale utilizado será el definido en la propia petición.
    - **name**: Indica el nombre del atributo de la Request que contiene el diccionario de errores internacionalizables (colección de pares <String, LocalizedMessage>). En ausencia de este atributo, la etiqueta buscará los errores capturados durante la validación de un formulario.
    - **property**: Identificador en la colección de errores que permite recuperar un mensaje de error concreto.
    - **header, prefix suffix y footer**: Propiedades que permiten definir porciones de HTML que permitirán la personalización del HTML de salida que mostrará el mensaje de error.

- **Message** - Etiqueta destinada a la recuperación de mensajes internacionalizables de dos formas:
    1. Utilizando directamente la clave que identifica el mensaje a recuperar del fichero de recursos.
    2. Utilizado las propiedades 'attribute' y 'scope' para determinar qué atributo de qué ámbito transporta el identificador del mensaje que desea recuperarse del paquete de recursos.

  Message permite su configuración a través de los siguientes atributos:
    - **key**: Clave que identifica el mensaje a recuperar del paquete de recursos.
    - **bundle**: Indica la ubicación del paquete de recursos que la etiqueta utilizará para internacionalizar el mensaje. En ausencia de este atributo, la etiqueta utilizará el paquete de recursos definido en el fichero de configuración del framework.
    - **locale**: Indica el nombre del atributo de sesión que contendrá el Locale que la etiqueta utilizará para internacionalizar el mensaje. En ausencia de éste, el Locale utilizado será el de la Request.
    - **attribute**: Indica el nombre del atributo que debe recuperarse del ámbito indicado por 'scope' para obtener la clave que identificará el mensaje.
    - **scope**: Indica el ámbito que debe utilizarse para localizar el atributo que contiene la clave del mensaje. Los ámbitos permitidos son: Session, Page, Application y Request siendo este último el ámbito por defecto.
    - **param1**, **param2**, **param3** y **param4**: Indican los valores que deberán sustituir respectivamente las marcas {1}, {2}, {3} y {4} en el mensaje original una vez recuperado del paquete de recursos.

I18n - Etiqueta destinada a la internacionalización de números, monedas, fechas y horas que permite su configuración a través de los siguientes atributos:

- **name**: Indica el nombre del bean que desea recuperarse del ámbito indicado por scope.
- **property**: Indica el nombre de la propiedad del bean que contiene el dato que desea internacionalizarse. Esta propiedad permitirá encadenar sub-propiedades separándolas mediante puntos.

- **scope**: Indica el ámbito que debe utilizarse para localizar el bean que contiene el dato a internacionalizar. Los ámbitos permitidos son: Session, Page, Application y Request siendo este último el ámbito por defecto.
- **locale**: Indica el nombre del atributo de sesión que contendrá el Locale que la etiqueta utilizará para internacionalizar el mensaje. En ausencia de éste, el Locale utilizado será el de la Request.
- **type**: Indica el tipo de internacionalización que desea llevarse a cabo, siendo válidos los siguientes tipos: {Number, Decimal, Currency, Date, Time y Date_Time}.
- **dateFormat**: Indica el formato que se utilizará al internacionalizar una fecha, siendo válidos los tipos {Default, Short, Medium, Long y Full}.
- **timeFormat**: Indica el formato que se utilizará al internacionalizar horas, siendo válidos los tipos {Default, Short, Medium, Long y Full}.

La librería HTML define la siguiente etiqueta:

- **Form** - Etiqueta destinada a la construcción de un formulario HTML que quede enlazado con la acción declarada en el fichero de configuración teniendo en cuenta el mapeo de urls definido para el servlet FrontController. Esta etiqueta permite su configuración a través de los siguientes atributos:
  - **action**: Nombre del Action que recibirá el formulario tal y como se declaró en el fichero de configuración del framework.
  - **method**: Método utilizado para el envío del formulario. Por defecto los formularios se enviarán por POST.
  - **enctype**: Indica la codificación utilizada para enviar el formulario. En caso de omitirse su valor por defecto será 'application/x-www-form-urlencoded'.
  - **target**: Indica dónde se mostrará el resultado de la acción especificada en el campo 'action'. Por defecto su valor es '_self'.
  - **style**: Indica los estilos CSS que quieren aplicarse al formulario.
  - **styleClass**: Indica la clase CSS que se utilizará para aplicar estilo al formulario.
  - **id**: Indica el identificador que tendrá asociado el formulario en el documento HTML.

Documentación y recursos

JavaMVC pone a disposición del desarrollador una serie de recursos destinados a facilitar la compresión y utilización del framework, estos recursos son:

- **Archivos TLD**. Archivos describen las librerías de etiquetas proporcionadas por el framework. Todo desarrollo que requiera el uso de alguna de las etiquetas proporcionadas, deberá incluir como parte de su aplicación, el archivo tld que describa dicha etiqueta para así hacerla accesible a la aplicación. JavaMVC proporciona dos librerías de etiquetas:
  - JavaMVC-html.tld
  - JavaMVC-i18.tld
- **JavaMVC.xsd**. Esquema utilizado para describir la estructura y restricciones de los contenidos del XML que configura el framework.
- **Javadoc**. Documentación de la API de JavaMVC en formato HTML. Esta documentación además de describir clases, interfaces, métodos, etc. presenta ejemplos que ayudan al desarrollador a relacionar los componentes del framework mejorando así su compresión sobre el funcionamiento del mismo.

Capítulo V – Agenda Telefónica On-Line

Con el objetivo de poner a prueba la implementación del framework, se ha llevado a cabo el análisis, diseño e implementación de una sencilla agenda telefónica on-line. El objetivo de ésta es el de permitir, a cualquiera de sus usuarios registrados, el acceso a su agenda personal desde cualquier explorador web que disponga de acceso a internet. Como es obvio para asegurar esta funcionalidad, la aplicación almacenará de forma persistente toda esta información.

Dado que una aplicación de esta índole puede abarcar un gran número de funcionalidades, y éste no es el objetivo de este proyecto, se ha seleccionado un pequeño conjunto de funcionalidades básicas que servirán, por una parte, para probar el funcionamiento del Framework y, por otra, para llevar a cabo el diseño de una aplicación web utilizando el patrón MVC a través del uso del framework.

Requisitos funcionales de la agenda on-line

- Un usuario podrá registrarse en el sistema.
- Un usuario podrá iniciar sesión en el sistema.
- Un usuario podrá cambiar el idioma de visualización de sus contenidos.
- Un usuario validado podrá crear nuevos contactos.
- Un usuario validado podrá modificar sus contactos.
- Un usuario validado podrá visualizar los datos de un contacto.
- Un usuario validado podrá eliminar un contacto.
- Un usuario validado podrá realizar búsquedas sobre sus contactos.
- Un usuario validado podrá cerrar su sesión en el sistema.
- Un usuario validado podrá modificar su clave de acceso.

A la vista de los requisitos identificados, se elabora el diagrama de casos de uso mediante el cual se identifican las funciones de un sistema software desde el punto de vista de sus interacciones con el exterior.

1. Modelo de Casos de Uso

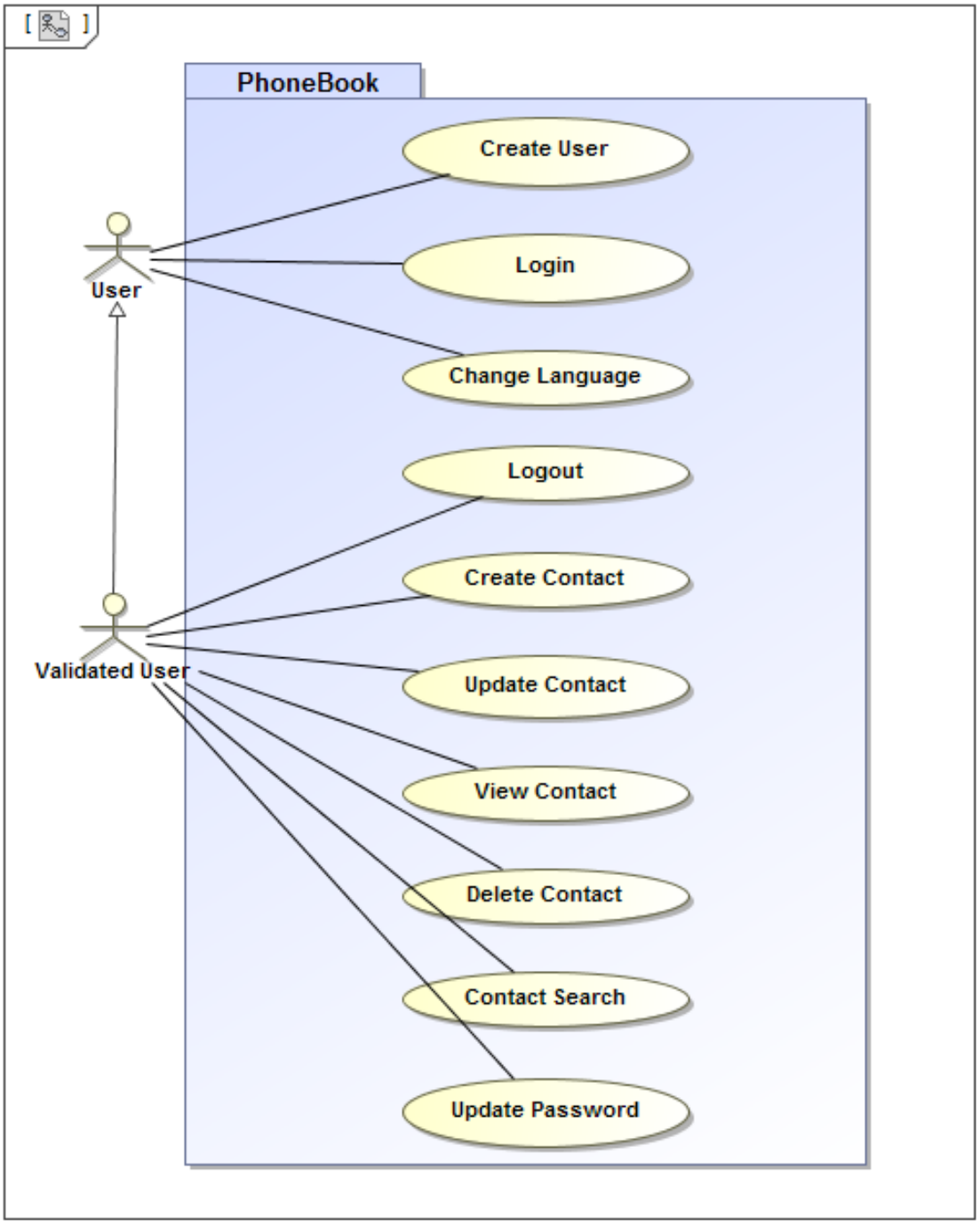

Figura 46 - Modelo de casos de uso de la agenda On-line

Caso de uso <<Create User>>

- **Resumen de la funcionalidad:** Un usuario accede a la aplicación y proporciona los datos de registro para crear un nuevo usuario.
- **Actor(es)**: User.
- **Casos de uso relacionados**: Login.
- **Pre-condición**: No pueden existir dos usuarios con el mismo nombre de usuario y/o correo electrónico en el sistema.
- **Post-condición**: Se crea un nuevo usuario en el sistema.
- **Propósito**: Crear un nuevo usuario en el sistema.
- **Flujo de acontecimientos.**
    1. Un usuario desea registrarse como usuario en la agenda on-line.
    2. El sistema le muestra la pantalla de alta de usuario.
    3. El usuario proporciona los datos de registro en el sistema.
    4. El sistema registra el usuario en el sistema y ejecuta automáticamente el caso de uso Login.
- **Flujo de acontecimientos alternativo.**

  3.a) Los datos proporcionados no permiten el alta de un nuevo usuario en el sistema.
    - El sistema muestra en la pantalla de alta de usuario los errores sucedidos.
- **Pantallas.**
    - Alta de usuario
        - Datos solicitados
            - Nombre de usuario, Clave de acceso, Correo electrónico.

Caso de uso <<Login>>

- **Resumen de la funcionalidad:** Un usuario proporciona unas credenciales para acceder al sistema.
- **Actor(es)**: User.
- **Casos de uso relacionados**: Ninguno.
- **Pre-condición**: Ninguna.
- **Post-condición**: Si las credenciales de acceso son correctas el usuario accede a la agenda vinculada con las credenciales proporcionadas, pasando a ser un usuario validado.
- **Propósito**: Permite a un usuario acceder a su agenda.
- **Flujo de acontecimientos.**
    1. Un usuario mediante un explorador web accede a la URL de la agenda online.
    2. El sistema le muestra la pantalla de login.
    3. El usuario proporciona las credenciales de acceso.
    4. El sistema valida las credenciales, registra la identificación del usuario en su sesión y le muestra su agenda.
- **Flujo de acontecimientos alternativo.**

    4.a) Las credenciales de acceso no son válidas.
        - El sistema muestra en la pantalla de login el error sucedido.
- **Pantallas.**
    - Login
        - Datos solicitados
            - Nombre de usuario.
            - Clave de acceso.

Caso de uso <<Change Language>>

- **Resumen de la funcionalidad:** Un usuario cambia el idioma de visualización de la aplicación.
- **Actor(es)**: User y Validated User.
- **Casos de uso relacionados**: Ninguno.
- **Pre-condición**: Ninguna.
- **Post-condición**: Se ha cambiado el idioma utilizado por la interfaz de la aplicación.
- **Propósito**: Cambiar el idioma utilizado por la aplicación.
- **Flujo de acontecimientos.**
    1. El usuario selecciona el idioma en el que quiere visualizar la aplicación.
    2. El sistema cambia los textos de la interfaz de forma que se adapten al idioma seleccionado.

Caso de uso <<Logout>>

- **Resumen de la funcionalidad:** Un usuario validado en el sistema cierra su sesión activa.
- **Actor(es)**: Validated User.
- **Casos de uso relacionados**: Ninguno.
- **Pre-condición**: El usuario validado debe tener activa su sesión en la aplicación.
- **Post-condición**: La sesión vinculada con el usuario se ha cerrado.
- **Propósito**: Abandonar la aplicación destruyendo la sesión vinculada con el usuario.
- **Flujo de acontecimientos.**
    1. El usuario validado desea cerrar su sesión.
    2. El sistema le muestra un mensaje en el que se le solicita confirmación.
    3. El usuario validado confirma su acción.
    4. El sistema cierra la sesión del usuario validado.
- **Flujo de acontecimientos alternativo.**

    3.a) El usuario validado Cancela el cierre de la sesión.
        - El caso de uso finaliza.

Caso de uso <<Create Contact>>

- **Resumen de la funcionalidad:** Un usuario validado crea un nuevo contacto en su agenda.
- **Actor(es)**: Validated User.
- **Casos de uso relacionados**: Ninguno.
- **Pre-condición**: No podrá crearse el contacto si ya existe previamente otro contacto con el mismo teléfono o email.
- **Post-condición**: El usuario validado dispone de una agenda con un nuevo contacto.
- **Propósito**: Crear un nuevo contacto en la agenda de un usuario validado.
- **Flujo de acontecimientos.**
    1. El usuario validado desea crear un nuevo contacto.
    2. El sistema muestra el formulario de creación de contacto.
    3. El usuario validado rellena el formulario de creación de contacto.
    4. El sistema registra el nuevo contacto y muestra un mensaje informando sobre la creación del contacto.
- **Flujo de acontecimientos alternativo.**

  3.a) El usuario cancela la creación del nuevo contacto.
    - El caso de uso finaliza.

  4.a) Los datos proporcionados para el alta de un nuevo contacto no son válidos o están incompletos.
    - El sistema muestra en la pantalla de creación del contacto los errores sucedidos.
- **Pantallas.**
    - Datos contacto
        - Datos solicitados
            - Nombre, Apellidos, Teléfono, Correo electrónico, Dirección, Código Postal, Ciudad y Fecha nacimiento.

Caso de uso <<Update Contact>>

- **Resumen de la funcionalidad:** Un usuario validado en el sistema quiere actualizar los datos de un contacto.
- **Actor(es)**: Validated User.
- **Casos de uso relacionados**: View Contact.
- **Pre-condición**: El contacto a actualizar debe existir previamente en la agenda del usuario validado.
- **Post-condición**: El usuario ha actualizado los datos del contacto.
- **Propósito**: Modificar los datos de un contacto en la agenda de un usuario validado.
- **Flujo de acontecimientos.**
    1. El usuario validado quiere modificar los datos de un contacto.
    2. Se ejecuta el caso de uso View Contact.
    3. El usuario validado indica las modificaciones deseadas sobre los datos del contacto.
    4. El sistema modifica los datos y muestra una pantalla con los datos actualizados del contacto.
- **Flujo de acontecimientos alternativo**.

  3.a) El usuario Cancela la actualización.
    - El caso de uso finaliza.

  4.a) El usuario validado no ha proporcionado todos los datos necesarios para llevar a cabo la modificación o éstos no son correctos.
    - El sistema advierte sobre la invalidez de los datos.
- **Pantallas.**
    - Datos contacto.
        - Datos mostrados/solicitados.
            - Nombre, Apellidos, Teléfono, Correo electrónico, Dirección, Código Postal, Ciudad y Fecha nacimiento.

Caso de uso <<View Contact>>

- **Resumen de la funcionalidad:** Un usuario validado en el sistema quiere visualizar los datos de un contacto.
- **Actor(es)**: Validated User.
- **Casos de uso relacionados**: Contact Search.
- **Pre-condición**: El contacto a visualizar debe existir en la agenda del usuario validado.
- **Post-condición**: Recuperar y mostrar los datos de un contacto de la agenda.
- **Propósito**: Visualizar los datos de un contacto de la agenda.
- **Flujo de acontecimientos.**
    1. Un usuario validado quiere visualizar los datos de un contacto.
    2. Se ejecuta el caso de uso Contact Search.
    3. El usuario validado selecciona la opción “visualizar” sobre el contacto deseado.
    4. El sistema muestra una pantalla con los datos del contacto.
- **Pantallas.**
    - Datos contacto.
        - Datos mostrados.
            - Nombre, Apellidos, Teléfono, Correo electrónico, Dirección, Código Postal, Ciudad y Fecha nacimiento.

Caso de uso <<Delete Contact>>

- **Resumen de la funcionalidad:** Un usuario validado en el sistema elimina un contacto de su agenda.
- **Actor(es)**: Validated User.
- **Casos de uso relacionados**: Contact Search.
- **Pre-condición**: El contacto a eliminar debe existir previamente en la agenda del usuario validado.
- **Post-condición**: El usuario validado ha eliminado el contacto seleccionado de su agenda.
- **Propósito**: Eliminar un contacto de la agenda.
- **Flujo de acontecimientos.**
    1. Un usuario validado desea eliminar un contacto de su agenda.
    2. Se ejecuta el caso de uso Contact Search.
    3. El usuario selecciona el contacto que desea eliminar.
    4. El sistema le muestra una pantalla de confirmación.
    5. El usuario validado acepta la eliminación.
    6. El sistema elimina el contacto de la agenda.
- **Flujo de acontecimientos alternativo.**

    5.a) El usuario validado Cancela la eliminación.

    - El caso de uso finaliza.

Caso de uso <<Contact Search>>

- **Resumen de la funcionalidad:** Un usuario validado en el sistema realiza una búsqueda de contactos de su agenda.
- **Actor(es)**: Validated User.
- **Casos de uso relacionados**: Ninguno.
- **Pre-condición**: Ninguna.
- **Post-condición**: El usuario validado recupera los contactos que cumplen con los criterios de búsqueda utilizados.
- **Propósito**: Localizar aquellos contactos de la agenda que cumplan ciertas condiciones.
- **Flujo de acontecimientos.**
    1. Un usuario validado desea buscar contactos.
    2. El sistema le muestra la pantalla de búsqueda.
    3. El usuario validado completa los campos del formulario que le permitirán filtrar los resultados de la búsqueda.
    4. El sistema muestra los resultados de la búsqueda.
- Flujo de acontecimientos alternativo.

    3.a) El usuario validado Cancela la búsqueda.
        - El caso de uso finaliza.

    4.a) El sistema muestra una pantalla de error si el usuario ha proporcionado datos inválidos para la búsqueda.
- **Pantallas.**
    - Búsqueda de contactos.
        - Datos solicitados.
            - Nombre, Teléfono, Ciudad, Código Postal, Fecha Inicio y Fecha Fin.
    - Resultados de la Búsqueda.
        - Datos mostrados.
            - Nombre, Teléfono, Email, Fecha Nacimiento.

Caso de uso <<Update Password>

- **Resumen de la funcionalidad:** Un usuario validado modifica su clave de acceso al sistema.
- **Actor(es)**: Validated User.
- **Casos de uso relacionados**: Ninguno.
- **Pre-condición**: La nueva clave debe cumplir las normas de seguridad estipuladas.
- **Post-condición**: El usuario validado ha modificado su clave de acceso al sistema.
- **Propósito**: Actualizar la clave de acceso al sistema de un usuario validado.
- **Flujo de acontecimientos.**
    1. El usuario validado quiere actualizar su clave de acceso.
    2. El sistema le muestra la pantalla de actualización de clave de acceso.
    3. El usuario validado proporciona los datos necesarios para modificar su clave de acceso.
    4. El sistema actualiza la clave de acceso del usuario y muestra un mensaje informando sobre la correcta realización de la actualización.
- **Flujo de acontecimientos alternativo.**

  3.a) El usuario validado Cancela la actualización.
    - El caso de uso finaliza.

  4.a) Los datos proporcionados para actualizar la clave de acceso no son válidos o están incompletos.
    - El sistema muestra una pantalla de error que informa sobre el problema sucedido.
- **Pantallas.**
    - Actualización de clave de acceso.
        - Datos solicitados.
            - Antigua clave de acceso, Nueva clave de acceso, Confirmación de la nueva clave de acceso.

2. Interfaz gráfica

Para realizar el análisis de la interfaz gráfica de usuario se elaborarán una serie de diagramas de estado para cada uno de los casos de uso anteriormente desarrollados. Estos diagramas representarán las diferentes pantallas y transiciones que podrán suceder durante la ejecución del caso de uso.

Diagrama de estados <<Create User>>

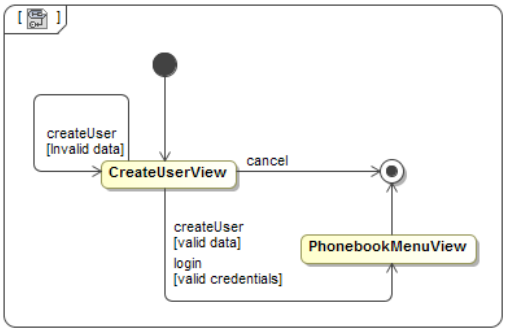

Figura 47 - Diagrama de estados – CU Create User

Diagrama de estados <<Login>>

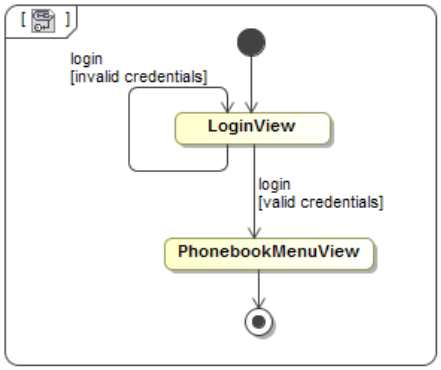

Figura 48 - Diagrama de estados – CU Login

Diagrama de estados <<Logout>>

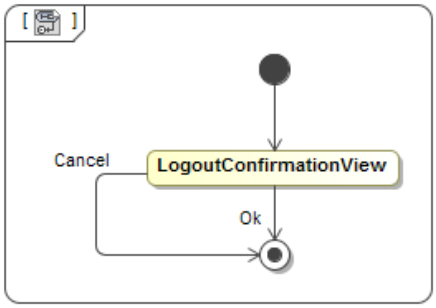

Figura 49 - Diagrama de estados - CU Logout

Diagrama de estados <<Update Password >>

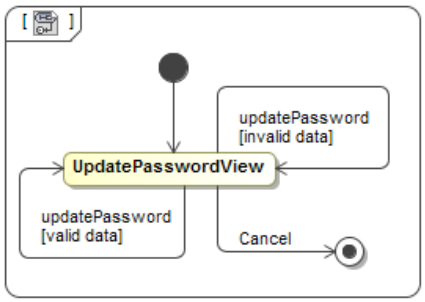

Figura 50 – Diagrama de estados - CU Update Password

Diagrama de estados <<New Contact >>

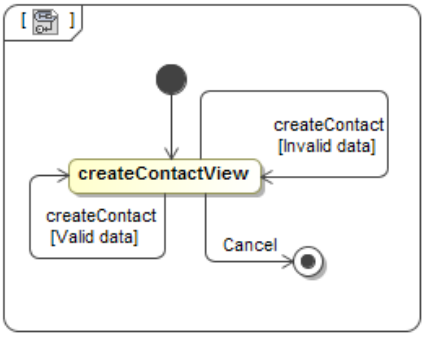

Figura 51 – Diagrama de estados - CU New Contact

Diagrama de estados <<Contact Search, View Contact, Update Contact y Delete Contact>>

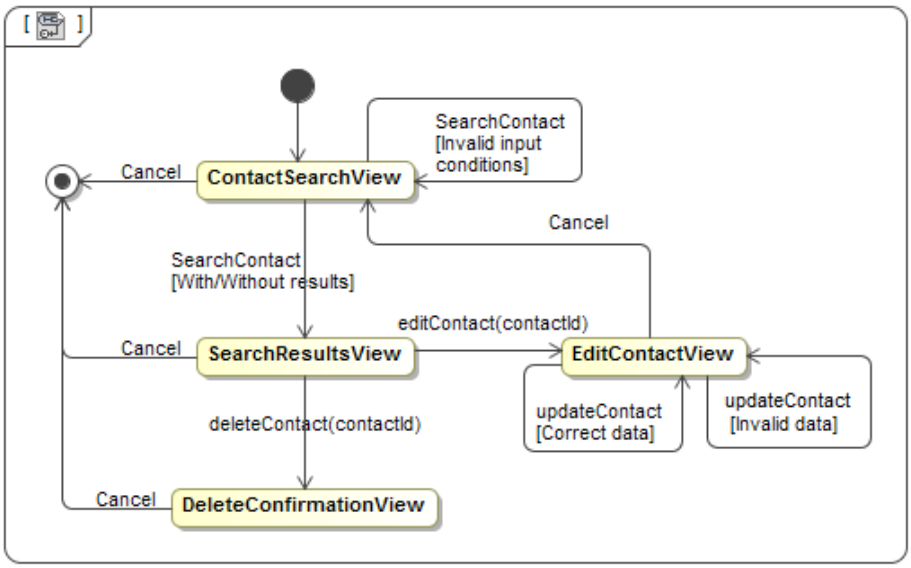

Figura 52 - Diagrama de estados - CU Buscar, Visualizar, Modificar y Eliminar Contacto

3. Diagrama estático de análisis

Dada la poca información que conoce el sistema sobre mundo real, desde un punto de vista del análisis, el diagrama de clases UML mostrado a continuación pone de manifiesto los conceptos con los que tratará el sistema así como las relaciones existentes entre ellos.

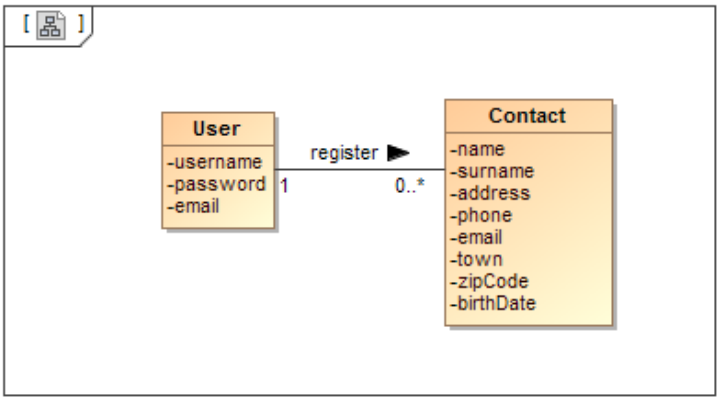

Figura 53 - Diagrama estático de análisis

4. Diseño

Para llevar a cabo el diseño de esta aplicación, el primer paso consistirá en definir la arquitectura del sistema. En este diseño se tomará el patrón de arquitectura en capas que, en el caso de la aplicación que nos ocupa, estará formado por tres capas: **presentación**, **dominio** e **integración**.

Utilizar una arquitectura en tres capas permite:

1. Que el sistema trabaje con diferentes niveles de abstracción.
2. Que los cambios en el código de una parte de la aplicación no se propaguen a otras.
3. Que la lógica de negocio quede desacoplada de la interfaz de usuario y de los servicios técnicos de modo que sean sustituibles por diferentes implementaciones con un mínimo impacto.

Esta división en capas no obstante podría llegar a reducir la eficiencia del sistema debido al aumento en los niveles de indirección existentes entre objetos a la hora de ejecutar una tarea.

5. Arquitectura de la capa de presentación.

Para la capa de presentación, obviamente, se utilizará el patrón arquitectónico Modelo-Vista-Controlador en su variante adaptada para una arquitectura en capas. Esta parte concreta del sistema se implementará mediante el framework de presentación **JavaMVC** analizado, diseñado e implementado a lo largo de esta memoria.

Se dividirá, por tanto, este subsistema en tres tipos de componentes: modelos, vistas y controladores.

Para los **controladores** se utilizarán clases **Action** que se encargarán de invocar las operaciones adecuadas de la capa de dominio y seleccionarán la siguiente pantalla a mostrar, delegando la generación de la respuesta en la vista de la pantalla a mostrar.

Para las **vistas** se utilizarán páginas JSP que generarán dinámicamente la página HTML que se enviará al navegador del usuario.

Por último el **modelo**, que en una arquitectura en capas estará formado por el conjunto de clases de la capa de dominio. Por tanto, durante el diseño de la capa de presentación, sólo se definirán las interfaces que es necesario que implemente la capa de Dominio.

6. Diseño de la capa de dominio

Tal y como se ha determinado anteriormente la capa de dominio constará de una serie de clases controladoras que implementarán las operaciones que la capa de presentación necesita. Estas operaciones estarán definidas en términos de la capa de dominio y serán independientes de cualquier tecnología utilizada en la capa de presentación.

Por otra parte, habrá un conjunto de clases que representarán las clases conceptuales detectadas durante el análisis y que implementarán la lógica del sistema.

Por último, se definirán los servicios esperados antes de implementarlos evitando la tendencia a que la capa de servicios técnicos tenga dependencias hacia la capa de dominio.

7. Diseño de la capa de servicios técnicos

En último lugar, dado que la aplicación requerirá el almacenamiento de datos persistentes en una base de datos, se ha decidido utilizar el paradigma de las bases de datos relacionales. Partiendo del diagrama estático del análisis se ha obtenido el modelo lógico de la base de datos.

A continuación, se muestra el diseño lógico resultante, destacando en negrita los campos obligatorios, mediante un subrayado continuo las claves primarias y mediante un subrayado discontinuo las claves foráneas.

- User (**userId**, **username**, **password, email**)
  {username} es clave alternativa.
  {email} es clave alternativa.
- Contact (**contactId**, **name**, surname, address, zipCode, **phone**, email, town, birthdate, **userId**)
  {userId es clave foránea a User{userId}.
  {userId, phone} son clave compuesta alternativa.
  {userId, email} son clave compuesta alternativa.

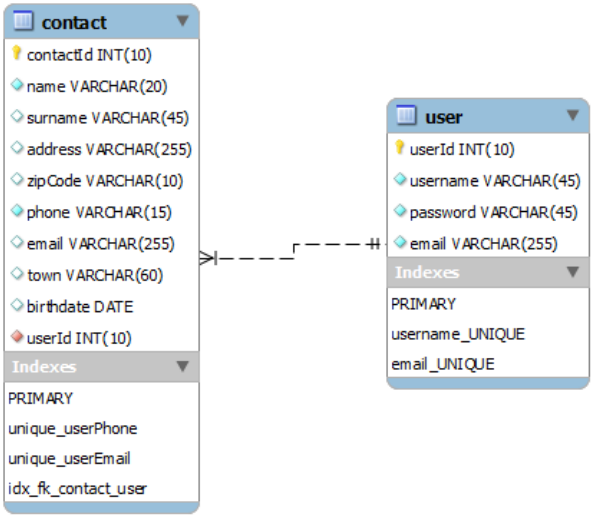

Figura 54 - Modelo Entidad-Relación

8. Diseño del subsistema de persistencia

Para el diseño del subsistema de persistencia ha decidido utilizarse el framework Hibernate que no sólo ayudará a simplificar el diseño sino que también facilitará la implementación.

En cuanto al diseño, ya que será el framework quien se encargue de conectarse a la base de datos, crear las consultas SQL, detectar cambios en memoria, etc. se ha decidido que la capa de persistencia conste de dos clases que implementen las interfaces previamente definidas en la capa de dominio. La correspondencia objeto-relacional entre las clases del modelo del dominio que han de ser persistentes y las tablas de la base de datos se realizará mediante un fichero descriptor de Hibernate en XML.

9. Diseño de la funcionalidad en una arquitectura en tres capas

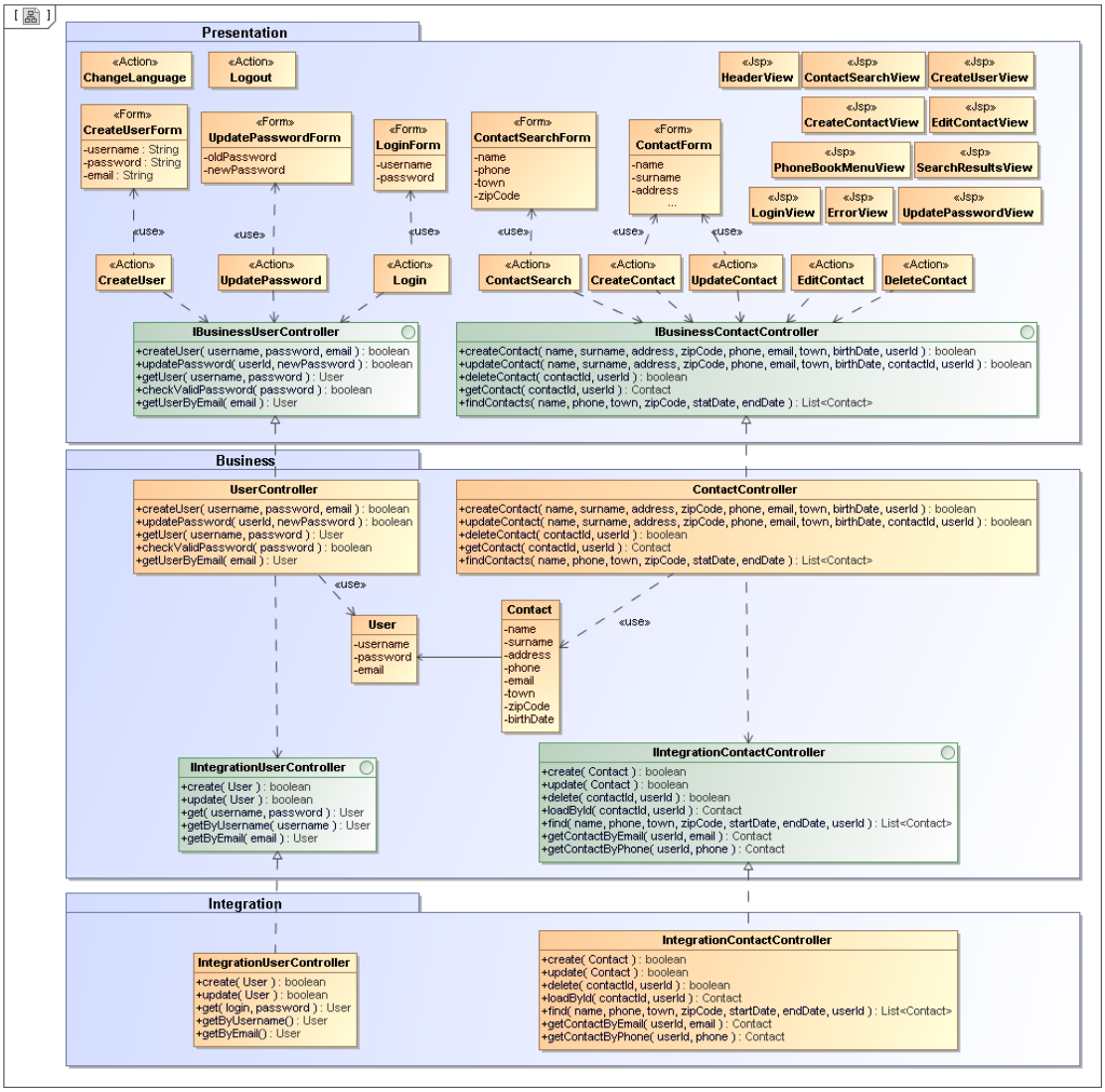

Figura 55 - Diseño en una arquitectura en tres capas

En el diagrama anterior se ha representado el diseño de la funcionalidad del sistema utilizando una arquitectura en tres capas. En este diseño se han obviado algunas clases de soporte para la aplicación que no aportan información relevante al lector, sin embargo, si es necesario destacar de este diseño que:

- Las necesidades de información identificadas en cada capa se han modelado mediante interfaces que serán implementadas en las capas inferiores.

- Los estereotipos <<Form>> y <<Action>> hacen referencia a las interfaces implementadas por los componentes de la capa de presentación como requisito de la utilización del framework JavaMVC.
- Las relaciones existentes entre Actions, Forms y JSPs en la capa de presentación no quedan reflejadas en el diagrama dado que éstas lo complicarían en exceso, dificultando su comprensión, sin embargo estas relaciones podrán consultarse en el XML de configuración del framework.

- Dada la naturaleza web de la aplicación, ha sido necesario incluir ciertas funcionalidades no contempladas durante el análisis de requisitos y, que por tanto, no fueron incluidas en el diseño, pero que resultan obligadas en una aplicación de este tipo.
    - La necesidad de una página de error que capture las condiciones de error no contempladas.
    - Identificar si un usuario dispone de una sesión activa o no, evitando que éste vuelva a hacer login.

Manual de usuario de la aplicación

Una vez desplegada la aplicación e introducida su URL en el navegador, el sistema mostrará la pantalla de login. Desde esta pantalla un usuario puede, o bien acceder al sistema mediante sus credenciales de acceso, o bien acceder al formulario de alta a través del enlace "Regístrate".

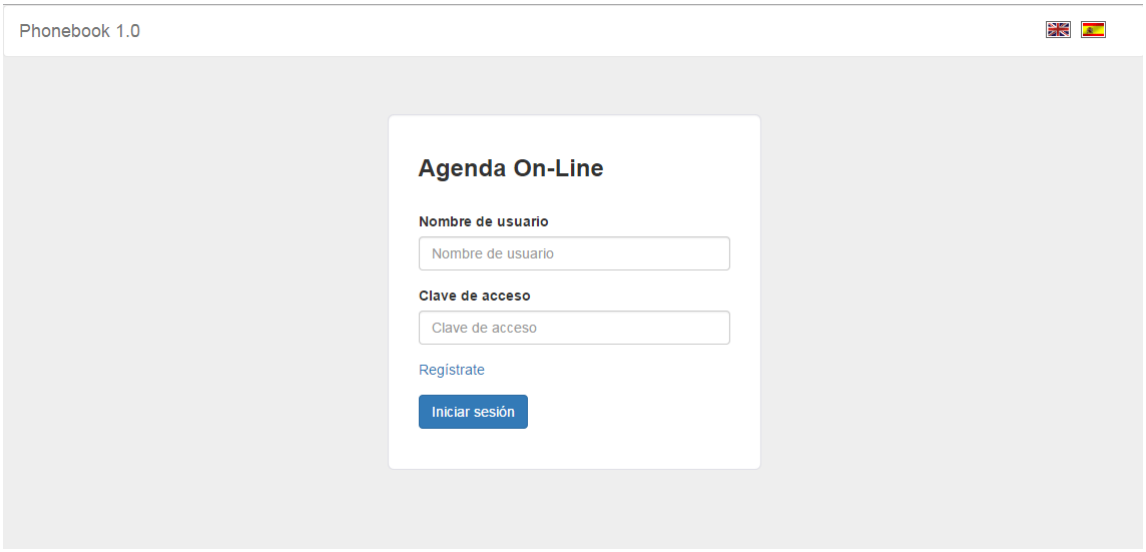

Figura 56 - Pantalla de Login

Si el usuario no dispone de una cuenta y decide darse de alta en el sistema a través del enlace "Regístrate", el sistema le mostrará la pantalla de alta de un nuevo usuario.

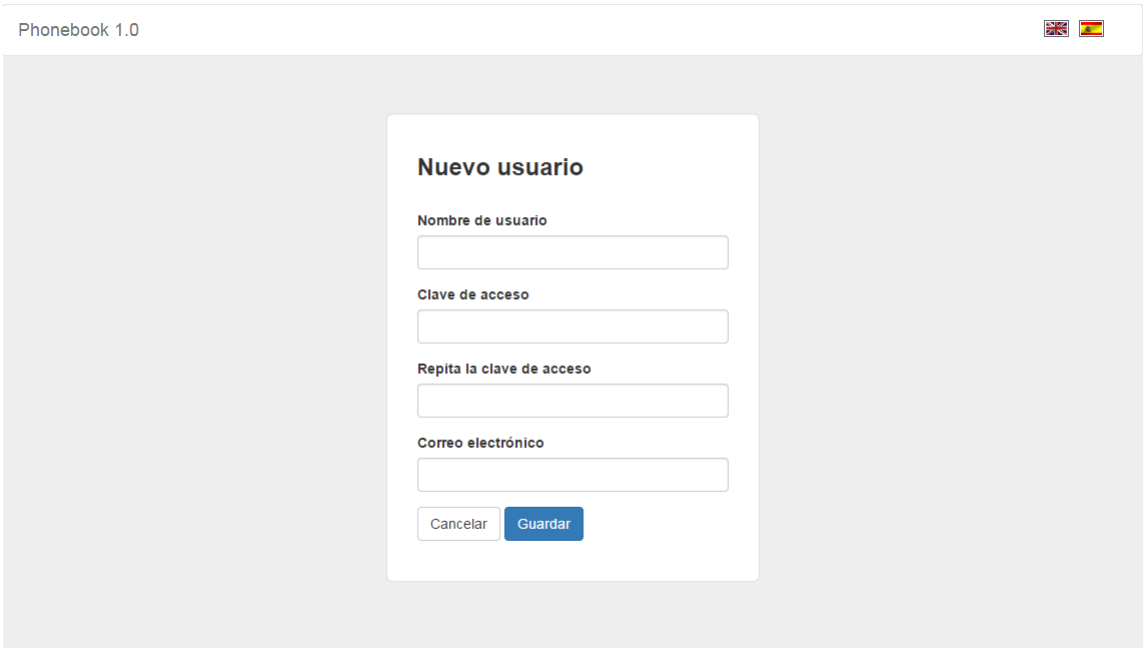

Figura 57 - Alta Usuario

Desde esta pantalla el usuario proporciona al sistema las que serán sus credenciales de acceso y éste, si confirma que no colisionan con las de otro usuario, lo registrará en el sistema y lo redirigirá al menú principal.

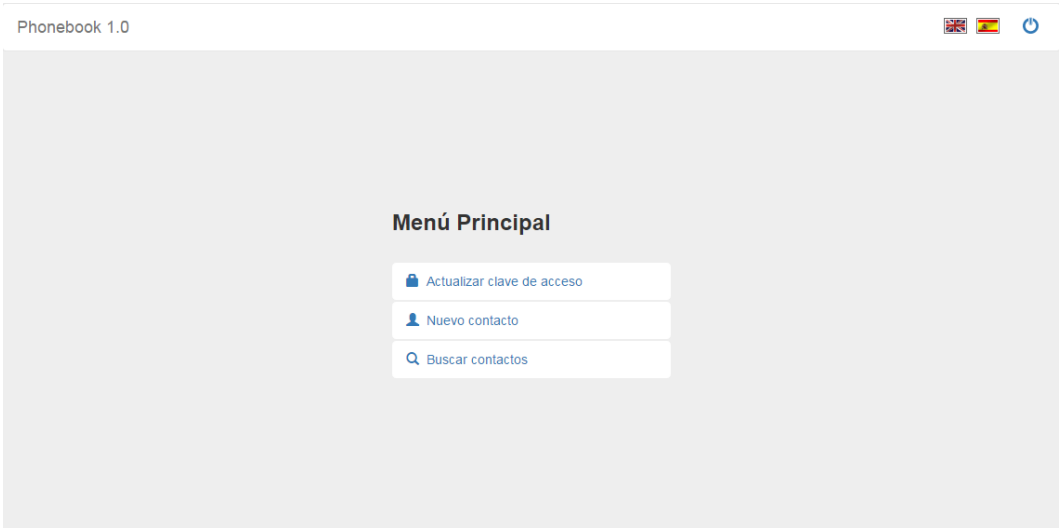

Figura 58 - Menú principal

Un usuario desde el menú principal podrá realizar diferentes operaciones como: Actualizar su clave de acceso, crear nuevos contactos o realizar búsquedas en su agenda (desde donde podrá visualizar, actualizar o eliminar contactos).

Si el usuario decide modificar su contraseña de acceso al sistema haciendo clic en el enlace del menú “Actualizar clave de acceso”, el sistema le mostrará el formulario de actualización dónde, además de la nueva clave se le solicitará, por seguridad, su clave de acceso actual antes de proceder al cambio.

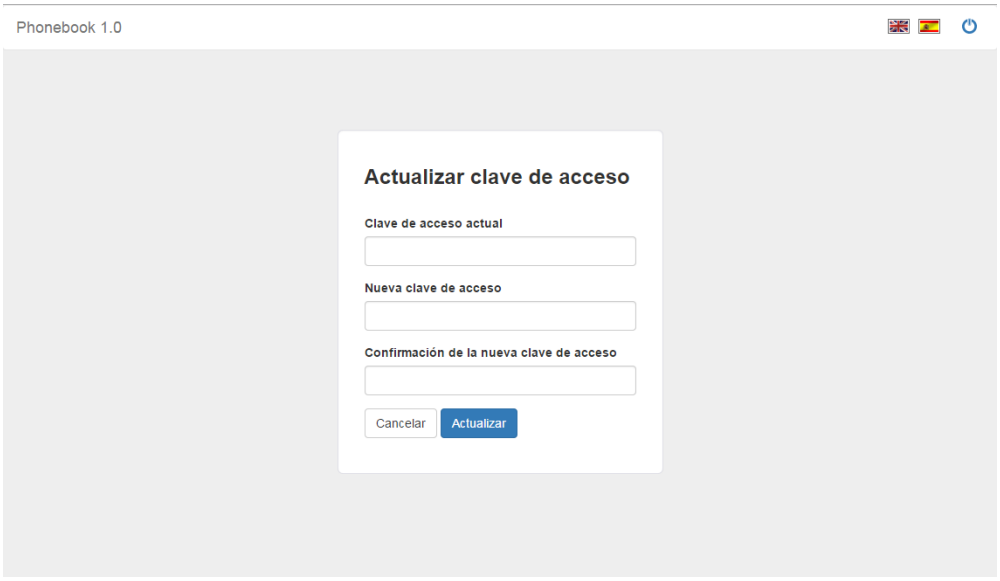

Figura 59 - Pantalla de actualización de clave de acceso

Otra opción disponible desde el menú principal es la de crear contactos. Desde ella el usuario proporcionará los datos del contacto a crear y el sistema tras unas validaciones básicas realizará el alta si todo es correcto.

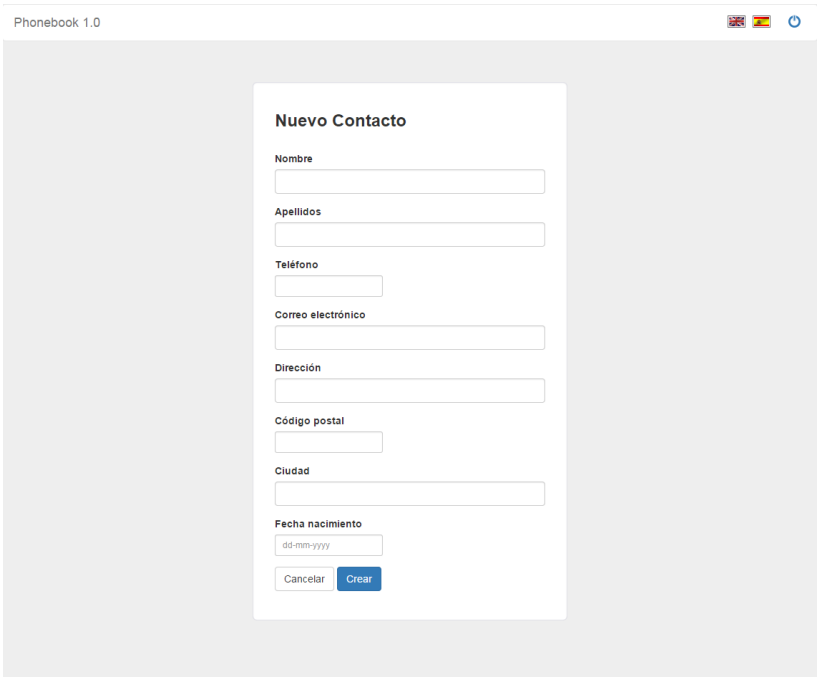

Figura 60 - Alta de nuevo contacto

Un usuario podrá realizar búsquedas sobre su agenda de forma que obtenga aquellos contactos que cumplan con una serie de criterios. El sistema permite filtrar las búsquedas por campos como: el nombre, el teléfono, la ciudad, el código postal o incluso mediante intervalos de fechas que aplicarán sobre la fecha de nacimiento del contacto.

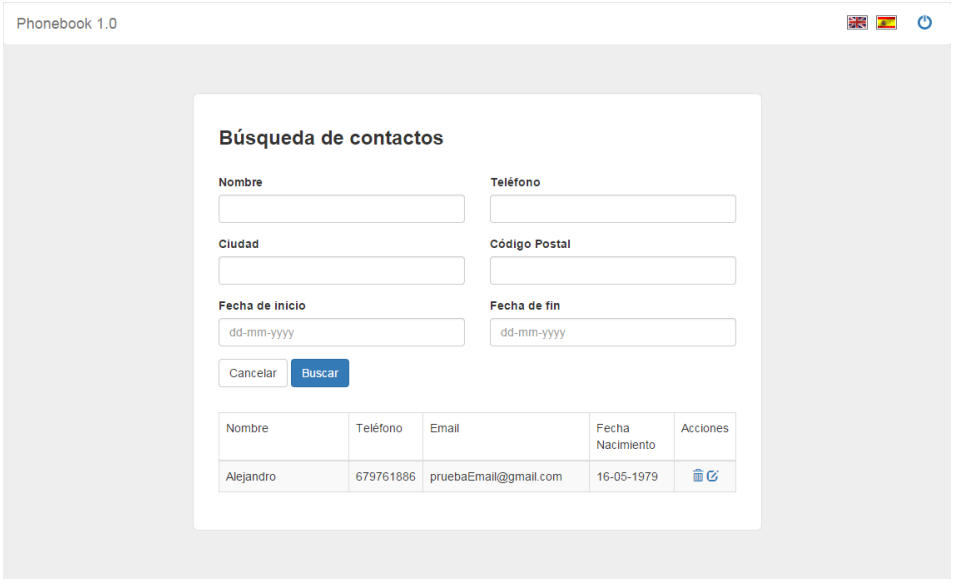

Figura 61 - Búsqueda de contactos

Una vez determinados los criterios de búsqueda, el sistema mostrará un listado en forma de tabla dónde aparecerán aquellos contactos que cumplan los criterios de búsqueda.

Sobre estos resultados, el usuario podrá llevar a cabo diversas acciones tales como: acceder a los datos del contacto o eliminarlo definitivamente de la agenda.

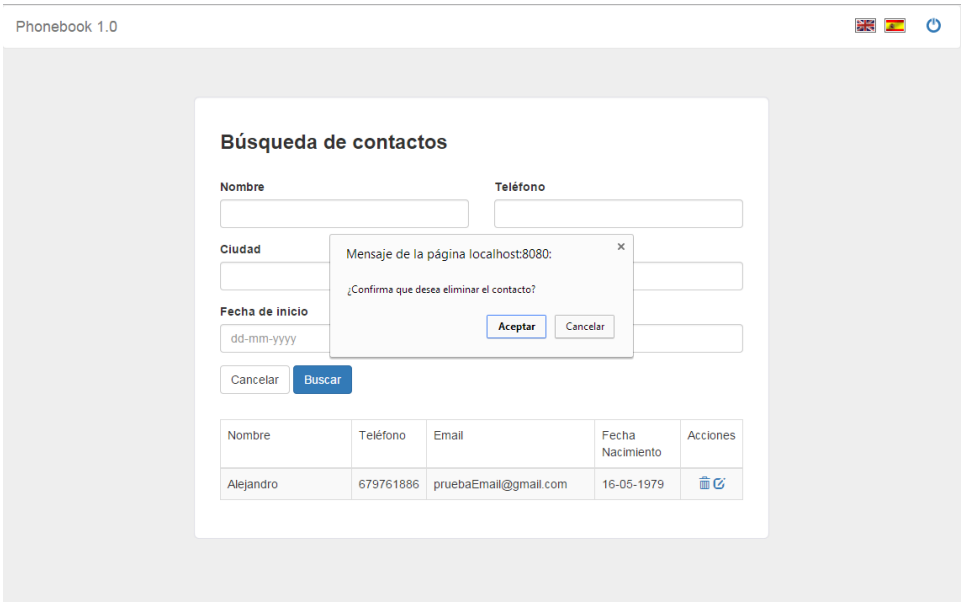

Figura 62 - Eliminación de contacto

Si sobre los resultados obtenidos el usuario decide editar un contacto en particular, el sistema le mostrará una pantalla en la que aparecerá toda la información registrada del contacto.

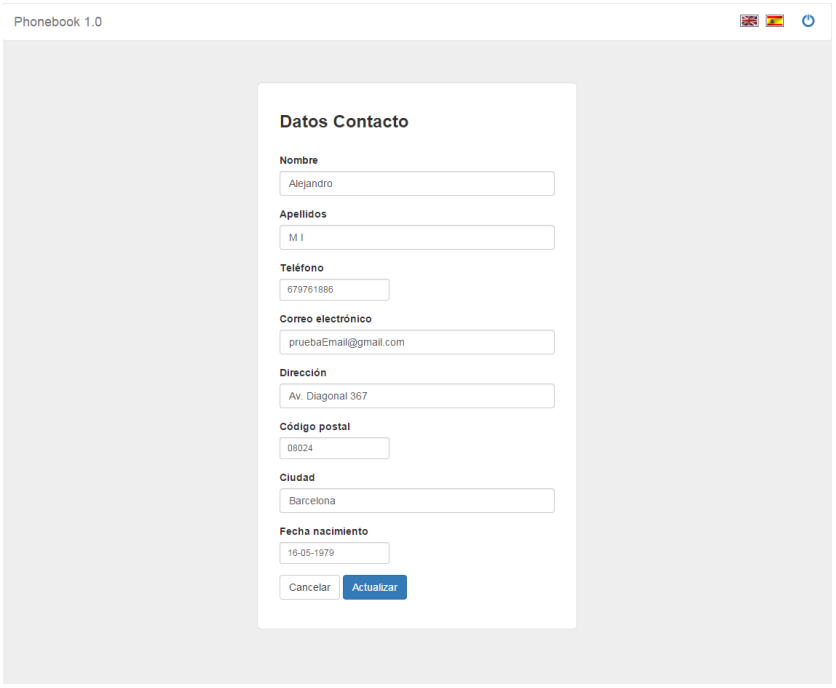

Figura 63 - Edición de contacto

Si el usuario desea modificar los datos del contacto, el sistema se lo permitirá a través del botón actualizar desde el que se actualizarán sus datos con las modificaciones realizadas.

Por último, cabe destacar la cabecera que aparece en todas las pantallas de la aplicación a lo largo de su ejecución. En ésta se pone a disposición del usuario la posibilidad de cambiar el idioma con el que se le muestran las pantallas, o cerrar su sesión y abandonar la aplicación (opción únicamente disponible cuando el usuario se ha validado previamente en la aplicación).

Figura 64 - Cabecera de la aplicación

Capítulo VI – Conclusiones

1. Sobre los frameworks

A lo largo del estudio y desarrollo de este proyecto se ha podido comprobar que, pese a existir un mercado con soluciones fuertemente asentadas que han demostrado su robustez y fiabilidad, este hecho no ha frenado en absoluto la aparición de nuevos frameworks que tratan de dar una mejor solución a un mismo problema.

Desde la aparición de frameworks como Struts (el más antiguo de todos los que se han sometido a estudio en esta memoria) que marcó de alguna forma un antes y un después en el desarrollo de la capa de presentación de aplicaciones Web Java EE, sus predecesores han tratado de mejorar, con distintas implementaciones, sus carencias y dificultades. Entre las mejoras adoptadas, la utilización de anotaciones para llevar a cabo la configuración de la aplicación, así como la posibilidad de utilizar clases POJO que aparten al desarrollador de la tecnología Servlet, han permitido facilitar la realización de pruebas evitando las complicaciones intrínsecas derivadas de la utilización de esta tecnología.

En conclusión los frameworks viven una evolución constante que los lleva a implementaciones cada vez más audaces y complejas que faciliten la tarea del desarrollador. Sin embargo, un punto que resulta diferenciador a la hora de escoger o descartar definitivamente un framework (más allá de lo eficiente o moderno que éste sea) ante un nuevo desarrollo, es la cantidad de documentación disponible así como la calidad de la misma. Este hecho ha podido comprobarse durante el estudio de los frameworks analizados. En ocasiones ha resultado más aclarador recurrir a bibliografía externa de terceros que a la documentación del propio framework. No hay que olvidar que en entornos empresariales el tiempo es oro y, es por tanto, donde la facilidad y rapidez para acceder y comprender la documentación de un framework puede marcar la diferencia.

2. Trabajo personal

En el terreno personal, el estudio y desarrollo de un framework me ha llevado a aprender tecnologías que han resultado de gran utilidad a la hora de implementar las funcionalidades ofrecidas por el framework. De entre todas ellas destacaré la API de Java Reflection, la librería JAXB, la creación de librerías de etiquetas y el soporte para la internacionalización entre otras.

Este proyecto me ha proporcionado una idea más clara sobre el panorama existente en el desarrollo Web y en cómo el patrón arquitectónico MVC se ha convertido en un estándar de facto a la hora de abordar el desarrollo de la capa de presentación de una aplicación Web.

El estudio de la evolución de las arquitecturas JSP hasta converger en una versión del patrón MVC, adaptada a aplicaciones Web, ha resultado muy útil para comprender cómo ha encajado este patrón, inicialmente pensado para aplicaciones con una interfaz de usuario más rica y capaz, en un tipo de aplicaciones con una interfaz más limitada.

Ha resultado interesante observar la gran cantidad de frameworks existentes que dan solución a un mismo problema, así como haber podido comparar implementaciones actuales como Struts[2], Spring MVC y JSF con otras ya obsoletas como Struts.

3. Posibles mejoras

Una vez finalizado el desarrollo y conociendo algunas de las implementaciones del mercado, es evidente que JavaMVC no puede competir con soluciones que llevan años evolucionando y en las que participan decenas de desarrolladores. Sin embargo, sin desmerecer el trabajo realizado existe una serie de funcionalidades que añadirían valor a su implementación inicial.

- Soporte para el envío de ficheros y creación de wizards.
- Creación de formularios dinámicos de forma declarativa.
- Configuración del framework mediante anotaciones.
- Librería de etiquetas más extensa que proporcione un abanico de funcionalidades más amplio y diverso entre los que destacaría el soporte AJAX.
- Proporcionar un sistema de plantillas que implemente el patrón *Composite-View* del estilo de Tiles mediante el que se proporcione la capacidad de crear vistas compuestas.
- Evolucionar el framework hacia una implementación que permita al desarrollador trabajar con objetos POJO que no necesiten ni implementar interfaces ni extender la funcionalidad de otras clases del framework.
- Juegos de pruebas unitarias que permitan la realización de test exhaustivos sobre la totalidad del framework. De esta forma, se podrá comprobar rápidamente si una actualización o extensión ha tenido efectos colaterales negativos sobre la funcionalidad existente.

Glosario

Ajax (Asynchronous JavaScript And Xml): Técnica de desarrollo web para crear aplicaciones interactivas que se ejecutan en el navegador manteniendo una comunicación asíncrona con el servidor. Permite la carga de secciones de página web sin necesidad de recargar la página entera.

API (Application Programming Interface): Conjunto de funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

Applet: Componente de una *aplicación* que se ejecuta en el contexto de otro programa como por ejemplo en un navegador web.

BluePrints: Guía de buenas prácticas de Sun Microsystems (ahora Oracle) para el desarrollo de aplicaciones en la plataforma Java EE.

Descriptor de despliegue: Componente en aplicaciones J2EE que describe como debe desplegarse (o implantarse) una aplicación web.

EJB (Enterprise Java Bean): API para Java EE implementada por Sun Microsystems que ofrece un modelo de componentes del lado servidor para desarrollar la lógica de negocio en aplicaciones distribuidas solventando la problemática de gestión de la concurrencia, transacciones, seguridad, etc.

GoF (Gang of four): Nombre con el que se conoce, comúnmente, a los autores del libro *Design Patterns*, de referencia en el campo del diseño orientado a objetos. La banda de los cuatro se compone de: Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.

IoC (Inversion of Control). Técnica de programación mediante la cual los frameworks o bibliotecas que utiliza una aplicación controlan el flujo de ésta.

JAR (Java ARchive): Formato de empaquetamiento Java para distribuir clases, recursos y metadatos relacionados. Las bibliotecas Java se distribuyen en este formato.

Jasper Reports: Librería para la creación de informes escrita completamente en lenguaje Java.

Java: Lenguaje de programación de propósito general, concurrente y orientado a objetos. Su objetivo es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, para ello el código se compila a un lenguaje intermedio que es interpretado por una máquina virtual Java que sí es específica de cada plataforma. Su sintaxis deriva en gran medida de C y C++.

JavaBean: Modelo de componentes en Java creado por Sun Microsystems. Los JavaBeans son clases que encapsulan a otros objetos. Para que una clase actúe como JavaBean debe ser serializable, tener un constructor por defecto y permitir el acceso a sus propiedades usando métodos get y set para cada uno de ellos.

JavaServer Pages (JSP): Documento de texto que se ejecuta como un Servlet pero permite una aproximación más natural a la creación de contenido estático.

Java Community Process (JCP): Proceso formalizado que permite, a las partes interesadas, involucrarse en la definición de futuras versiones y características de la plataforma Java. El proceso JCP conlleva el uso de **Java Specification Request** (JSR) que son documentos formales que describen las especificaciones y tecnologías propuestas para que sean añadidas a la plataforma Java.

JAXB (Java Architecture for XML Binding): Permite mapear clases Java a archivos XML. Tiene dos funciones: una que serializa los objetos Java a XML; y otra que de serializa XML en objetos Java.

JDBC (Java DataBase Connectivity): API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice.

JSTL: Extensión de la tecnología JSP que permite incorporar funcionalidades adicionales en una página JSP haciendo uso de etiquetas.

Locale (Configuración regional): Conjunto de parámetros que define el idioma, país y cualquier otra preferencia especial que el usuario desee ver en su interfaz de usuario.

Log4j: Biblioteca open-source desarrollada en Java por la Apache Software Fundation que permite a los desarrolladores escoger la salida y el nivel de granularidad de los mensajes de log en tiempo de ejecución.

Maven: Aplicación para la automatización en la creación de proyectos Java. Permite definir las dependencias en XML y automáticamente se encarga de importarlas al proyecto.

MVC (Modelo Vista Controlador): Patrón arquitectónico que divide un sistema en tres tipos de componentes: Modelos, Vistas y Controladores. El **Modelo** encapsula el estado del sistema (donde se implementa la lógica de negocio), la **Vista** presenta los datos al usuario y el **Controlador** establece la correspondencia entre las acciones del usuario y los eventos del sistema.

OGNL (Object-Graph Navigarion Language): Lenguaje de expresiones de código abierto para Java que permite obtener y establecer propiedades así como ejecutar métodos de clases Java.

Patrón arquitectónico: Patrón que ofrece soluciones a problemas de arquitectura de software en ingeniería del software. Estos patrones resuelven problemas que afectarán al conjunto del diseño del sistema.

Patrón de diseño: Solución general a un problema común y recurrente en el diseño de software. Un patrón de diseño es una descripción o plantilla para resolver un problema que se puede utilizar en muchas situaciones diferentes.

POJO (Plain Old Java Object): Clases simples que no dependen de un framework en especial.

Resource bundle: Fichero java de extensión '.properties' que contiene datos específicos de un locale. Proporciona la forma de internacionalizar aplicaciones Java haciendo que el código sea independiente del locale.

Servlet: Clase java ejecutada para ampliar las capacidades de un servidor que responde comúnmente a peticiones HTTP.

Scriptlet: Partes de código java incrustadas entre los elementos estáticos de una página JSP.

Tag: Etiqueta o marca en un lenguaje basado en XML como por ejemplo HTML. Las etiquetas permiten asociar atributos y propiedades que configuran el elemento. En lo que respecta a las librerías de tags entendemos como un conjunto de etiquetas reutilizables y encapsuladas para ser utilizadas en el desarrollo de las vistas de la capa de presentación.

Thread-safe: Concepto de programación aplicable al contexto de programas multihilos. Se considera que un código es Thread-safe si éste funciona correctamente durante la ejecución simultánea de múltiples hilos.

Thin client: Equipo con unos requerimientos hardware mínimos que permite conectar a internet y realizar peticiones web, puede ser un ordenador personal, portátil o dispositivo de bajo rendimiento.

TLD (Tag Library Descriptor): Documento XML que define la funcionalidad de los Tags de una librería.

URI (Uniform Resource Identifier): Cadena de caracteres que identifica un recurso.

URL (Uniform Resource Locator): Cadena de caracteres que identifica un recurso y permite su localización en Internet.

Velocity: Proyecto de código abierto dirigido por Apache Software Foundation. Se trata de un motor de plantillas basado en Java que proporciona un lenguaje de plantillas sencillo cuyo objetivo es asegurar una separación limpia entre la capa de presentación y la capa de negocio en aplicaciones Web.

WAR (Web application Archive): Formato de empaquetamiento de Java EE para distribuir y desplegar aplicaciones web.

XML (eXtensible Markup Language): Lenguaje de marcas desarrollado por el W3C utilizado para almacenar datos de forma legible. XML se propone como estándar para el intercambio de información estructurada entre distintas plataformas.

XML Schema: Descripción de la sintaxis de un determinado subconjunto de XML.

XSLT: Estándar de la organización W3C que presenta una forma de transformar documentos XML en otros, e incluso a formatos que no son XML.

Bibliografía

Understanding JavaServer Pages Model 2 architecture - Exploring the MVC design pattern. [En línea]. [Fecha de consulta: 2 de Octubre de 2014]. Disponible en la Web: http://www.javaworld.com/article/2076557/java-web-development/understanding-javaserver-pages-model-2-architecture.html

Struts1 framework architecture – MVC. [En línea]. [Fecha de consulta: 6 de Octubre de 2014] Disponible en la Web: http://gopaldas.org/struts/struts1/struts1-framework-architecture-mvc/

Struts 2 - Home Site. [En línea]. [Fecha de consulta: 9 de Octubre de 2014] Disponible en la Web: http://struts.apache.org/docs/home.html

Struts 2 Tutorial – Tutorials Point. [En línea]. [Fecha de consulta: 9 de Octubre de 2014] Disponible en el Web: http://www.tutorialspoint.com/struts_2/index.htm

Comparing Struts1 and Struts^2. [En línea]. [12 de Octubre de 2014] Disponible en la Web: http://struts.apache.org/release/2.3.x/docs/comparing-struts-1-and-2.html

Introduction to Spring Framework. [En línea]. [Fecha de consulta: 15 de Octubre de 2014]. Disponible en la Web: http://docs.spring.io/spring-framework/docs/3.0.x/reference/overview.html

Top 4 Java Web Frameworks Revealed. [En línea]. [Fecha de consulta: 15 de Octubre de 2014] Disponible en la Web: http://zeroturnaround.com/rebellabs/top-4-java-web-frameworks-revealed-real-life-usage-data-of-spring-mvc-vaadin-gwt-and-jsf/

Spring MVC Framework Tutorial – Tutorials Point. [En línea]. [Fecha de consulta: 15 de Octubre de 2014]. Disponible en la Web: http://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm

JavaServer Faces (JSF) Tutorial – Tutorials Point. [En línea]. [Fecha de consulta: 21 de Octubre de 2014] Disponible en la Web: http://www.tutorialspoint.com/jsf/index.htm

Java Platform, Enterprise Edition: The Java EE Tutorial - JavaServer Faces Technology. [En línea]. Disponible en la Web: http://docs.oracle.com/javaee/7/tutorial/jsf-intro.htm#BNAPH

Jérôme LAFOSSE. *Struts 2: El framework de desarrollo de aplicaciones Java EE*. Ediciones ENI. 2010.

Ian Roughley.*Practical Apache Struts2 Web 2.0 Projects*. Appress. 2007.

Clarence Ho and Rob Harro. *Pro Spring 3*. Apress. April 17, 2012.

Ed Burns, Chris Schalk with Neil Griffin. *The complete Reference JavaServer Faces 2.0: The Complete Reference*. MC Graw Hill. 2010.

David Geary, CAY Horstmann. *Core JavaServer Faces*. Third Edition. Prentice Hall. 2010.

Bill Dudney, Jonathan Lehr, LeRoy Mattingly. *Mastering JavaServer Faces*. Wiley Publishing, Inc. 2004.

Arnold Doray, *Beginning Apache Struts: From Novice to Professional*, 2006, Apress.

Anexo I

1. Instalación y ejecución del framework

El framework JavaMVC se distribuye en un único fichero jar de nombre "javaMVC-1.0.jar" con su código compilado con la versión 1.6 de la JDK.

Para su utilización es necesario añadir el jar como librería adicional de un proyecto Web en Java EE y asegurar que sus dependencias (javax.servlet-api 3.0.1, jsp-api 2.1 y log4j 1.2.17) quedan resueltas.

El fichero Zip de entrega del framework "**javaMVC-framework.zip**" consta de los siguientes directorios:

- **javaMVC**. Contendrá los ficheros fuente del framework listos para ser importados como proyecto de eclipse. Entre todos los ficheros que forman el framework cabe destacar los siguientes:
  - javadoc/**index.html**. Documento HTML que contiene el índice de la documentación Javadoc del Framework.
  - src/main/resources/**javaMVC.xsd**. Esquema XML que define las reglas que debe cumplir el XML de configuración de JavaMVC.
  - src/main/java/META-INF/**JavaMVC-html.tld** – TLD de etiquetas HTML.
  - src/main/java/META-INF/**JavaMVC-i18n.tld** – TLD de etiquetas de internacionalización.
- **jar**. Librería compilada (JDK 1.6) y lista para ser integrada en cualquier proyecto web Java EE.

Consideraciones

- Dada la naturaleza de las aplicaciones a las que va destinado el framework, éste no podrá probarse en solitario, necesitando en todo momento una aplicación web que haga uso de él.

- El framework JavaMVC utiliza Maven2 para la resolución de dependencias y construcción de la librería, por tanto, será conveniente disponer de esta herramienta debidamente instalada y configurada en el sistema en caso de que quiera compilarse la aplicación utilizando esta herramienta.

Anexo II

1. Instalación y ejecución de la aplicación de prueba

La aplicación de prueba 'Agenda On-Line' se distribuye en un único fichero de despliegue WAR de nombre **phonebook.war** que incluye los recursos necesarios para su correcto funcionamiento en un entorno con Java 1.6, servidor Apache Tomcat 7 y visibilidad a un SGBD MySql 5.0.

El Zip con la entrega de esta aplicación "**Phonebook_Web_App.zip**" consta de los siguientes directorios:

- **database** → Contiene el script de base de datos necesario para crear el entorno de persistencia (en un SGBD MySQL 5.0) que la aplicación de prueba necesita.
- **war** → Contiene el WAR de la aplicación de prueba lista para su despliegue en un servidor Apache Tomcat 7.
- **phonebook** → Contiene el proyecto phonebook (aplicación de prueba) listo para ser importado en un entorno de desarrollo Eclipse.

Consideraciones

- La configuración del pool de conexiones utilizado por la aplicación se encuentra definido en el fichero **context.xml** de la carpeta **META-INF** del proyecto. Este pool de conexiones se construye considerando que MySQL se encuentra en la máquina local, por tanto, en caso de realizarse pruebas con una instancia de MySQL en una máquina distinta, será necesario modificar este fichero adaptando la URL de conexión a la configuración disponible.

- Phonebook utiliza log4j como sistema de mensajes de log. En caso de ser necesario realizar alternaciones en la configuración de Log4j, éstas se harán a través de la modificación del fichero **phonebook/src/log4j.properties** incluido en la aplicación.

- El proyecto phonebook utiliza Maven2 para la resolución de dependencias y construcción del proyecto, por tanto, será conveniente disponer de esta herramienta debidamente instalada y configurada en el sistema en caso de que quiera compilarse la aplicación utilizando esta herramienta.

Instalación

1. Se ejecuta el script 'phonebook.sql' existente en la carpeta **database**.

   Este script llevará a cabo las siguientes tareas:

   - Crear el esquema 'phonebook'.
   - Crear las tablas necesarias en el esquema 'phonebook'.
   - Crear el usuario que utilizará la aplicación para conectar con la base de datos creada.

2. Se copia el WAR de la aplicación en la carpeta {**directorio_instalacion_tomcat**}/webapps del servidor Tomcat 7.
3. Se arranca el servidor.
4. Si la aplicación se ha desplegado correctamente, ésta pasará a estar accesible desde la url: http://localhost:{puerto_escucha_tomcat}/phonebook