## Licencia GPL

Puede copiar y distribuir el Programa (o un trabajo basado en él, según se especifica en el apartado 2, como código objeto o en formato ejecutable según los términos de los apartados 1 y 2, suponiendo que además cumpla una de las siguientes condiciones:

1. Acompañarlo con el código fuente completo correspondiente, en formato electrónico, que debe ser distribuido según se especifica en los apartados 1 y 2 de esta Licencia en un medio habitualmente utilizado para el intercambio de programas, o 2. Acompañarlo con una oferta por escrito, válida durante al menos tres años, de proporcionar a cualquier tercera parte una copia completa en formato electrónico del código fuente correspondiente, a un coste no mayor que el de realizar físicamente la distribución del fuente, que será distribuido bajo las condiciones descritas en los apartados 1 y 2 anteriores, en un medio habitualmente utilizado para el intercambio de programas, o 3. Acompañarlo con la información que recibió ofreciendo distribuir el código fuente correspondiente. (Esta opción se permite sólo para distribución no comercial y sólo si usted recibió el programa como código objeto o en formato ejecutable con tal oferta, de acuerdo con el apartado 2 anterior).

Proyecto Final de Carrera
Ingeniería Informática

# Diseño e implementación de un marco de trabajo de presentación para aplicaciones J2EE "MTP"

## Memoria

*Junio de 2012*

*UOC*

*Autor: José Alonso de Motta*
*alonsomotta@gmail.com*

*Consultor: Óscar Escudero Sánchez*

“El agradecimiento es la memoria del corazón”

J.B. Massieu

---

A Astrid, Santiago y Laura.

A mi familia, a ambos lados del atlántico.

A mis amigos de siempre.

A mis amigos de la “urba” y del colegio.

A mis compañeros de trabajo.

# 1 Resumen ejecutivo

El proyecto consiste en la elaboración de un marco de trabajo de presentación, denominado MTP con las siguientes características: está basado en el patrón Modelo-Vista-Controlador (MVC); incorpora las principales características de otros marcos de trabajo representativos de mercado; y las mejores prácticas ofrecidas por los patrones de diseño. Después se demuestra su utilidad en una aplicación de ejemplo. La tecnología objetivo es la de las aplicaciones Web con la plataforma J2EE.

En primer lugar se analizan los frameworks JavaServer Faces 2, Struts² y Spring MVC, obteniendo sus características principales, la forma de implementar el patrón MVC, identificando las fases por las que pasan las peticiones de los usuarios y comprobando cómo se realiza la configuración en aplicaciones.

A la par que el punto anterior, se profundiza en los patrones de diseño, comenzando por el propio patrón MVC, un conjunto de patrones de diseño específicos de J2EE (*Core J2EE Patterns*) y finalmente por el conjunto de patrones de diseño de programación orientada a objetos del conocido como *Gang of Four*.

Con las bases sólidas se acomete el análisis, diseño e implementación del marco de trabajo MTP. Se ha realizado un producto sencillo, que no necesita librerías de terceros, apoyándose por completo en los API de Java 1.6 y J2EE 5. El framework presenta a las aplicaciones un interfaz (RequestData) para realizar llamadas desde el model (la clase de la aplicación que representa el modelo), así como una librería de etiquetas (TLD) para la integración con páginas JSP.

Finalmente, se presenta una aplicación de ejemplo que aprovecha las funcionalidades de MTP, y que utiliza también los patrones de diseño del grupo *Core J2EE Patterns* y los del *Gang of Four*. La aplicación se instala en un servidor de aplicaciones J2EE y utiliza la base de datos MySQL.

# 2 Índice de contenidos y figuras

## 2.1 Tabla de contenido

## 2.2 Índice de figuras

# 3 Memoria

## 3.1 Introducción

Se presenta una primera aproximación al trabajo realizado, comenzando por la justificación para realizar el trabajo; una exposición de objetivos a alcanzar; las alternativas planteadas y la solución escogida; la planificación de alto nivel del proyecto realizado; y finalmente una relación de productos obtenidos.

### 3.1.1 Justificación

En los últimos años la tecnología J2EE ha cosechado mucho éxito, siendo una de las más utilizadas en el ámbito de las aplicaciones con interfaz de usuario web, que a su vez suponen una mayoría de las aplicaciones realizadas en los últimos tiempos.

Este éxito ha venido aumentado gracias a la apuesta de muchos fabricantes que han puesto en el mercado servidores de aplicaciones con ese estándar, a la comunidad open source que ha puesto a disposición de los equipos de desarrollo servidores de aplicaciones de código abierto y librerías que cubren los problemas básicos a los que se enfrentan los programadores: servicios web, tratamiento de ficheros XML, envío de correo electrónico…

También hay otras librerías que son marcos de trabajo completos especializados en una función: persistencia, seguridad, presentación... Estos marcos de trabajo se apoyan en patrones de diseño que recogen las mejores prácticas para resolver problemas comunes a las aplicaciones.

Cada vez más, las aplicaciones web J2EE aprovechan todos estos elementos: librerías especializadas, marcos de trabajo de las distintas capas del servidor de aplicaciones y uso de patrones de diseño. Con ello se benefician de un tiempo inferior de desarrollo; de aplicaciones más robustas, al aplicar soluciones bien probadas; y de una mayor comunidad de técnicos que aplican soluciones similares, reconocidas por la generalidad.

El proyecto se apoya en todos estos aspectos, siendo el eje central el diseño y construcción de un marco de trabajo especializado en la capa de presentación, que se apoya en la experiencia de otros frameworks analizados en primer lugar para adoptar las mejores soluciones de cada uno de ellos. También se apoya en los patrones de diseño más reconocidos: el patrón Modelo-Vista-Controlador; el conjunto de patrones de diseño J2EE del libro “Core J2EE Patterns” y los patrones recopilados por el grupo conocido como “Gang of Four”.

La aplicación construida utiliza el framework desarrollado, pero también utiliza el framework de persistencia Hibernate. En el diseño de la arquitectura de la aplicación se han aplicado un conjunto de los patrones de diseño de J2EE mencionados, así como los del grupo Gang of Four. El producto obtenido es una aplicación que puede servir como base para la construcción de otras aplicaciones más complejas, ya utilicen estos dos marcos de trabajo u otros similares.

### 3.1.2 Objetivos

Para la primera etapa, de análisis de los framework existentes, se van a evaluar tres de los principales, dos de código abierto con licencia Apache y otro incorporado en la especificación J2EE:

- JSF 2
- Struts[2]
- Spring MVC

Durante la etapa inicial también se van a analizar los patrones J2EE utilizados por estos frameworks con la intención de aplicar los que se consideren adecuados en el framework a desarrollar.

A continuación se elabora el framework de presentación, etapa que se subdivide a su vez en varias actividades:

- Análisis.
- Diseño.
- Implementación.

Las funcionalidades que va a incorporar el framework son:

- Implementación del patrón MVC:
    - Elaboración de la clase *controller* que va a procesar las peticiones de clientes y decidir la navegación.
    - Método para implementar el modelo (interfaz o clase raíz para las acciones, identificación automática de acciones o mapeo en fichero de configuración…)
    - Implementación del elemento vista del patrón. Se utilizan páginas JSP, por lo que solo es necesario implementar algún tipo de ayudas en el framework, como una librería de etiquetas.
- Capa de seguridad. Autenticación y control de acceso a recursos.
- Validación de datos procedentes del cliente.
- Lectura de datos recibidos (lectura e incorporación en objetos Java).
- Internacionalización.

Finalmente, para probar y demostrar la utilidad del framework, se elabora una guía de uso y se desarrolla una aplicación de ejemplo que utiliza el framework implementado en la etapa anterior.

### 3.1.3 Alternativas y solución escogida

En las primeras fases del proyecto se ha decidido la arquitectura a seguir, en base al conocimiento adquirido durante el estudio del funcionamiento de otros marcos de trabajo, diferentes entre ellos. Se han escogido las funcionalidades más relevantes para un marco de trabajo de estas características, teniendo en cuenta el tiempo disponible.

En todo caso, se ha tratado de hacer un trabajo limpio, apoyado en todo momento por las buenas prácticas que ofrecen los patrones de diseño. El código es sencillo de interpretar y

se ha documentado mediante Javadoc, disponible para su consulta a través de navegador web.

En cuanto al uso de recursos adicionales, se ha elaborado una librería que no tiene otras dependencias, excepto Java 1.6 y J2EE 5. La solución se ha desarrollado utilizando el servidor de aplicaciones Tomcat 7, certificado con el perfil web de J2EE 1.5 y con el JDK 1.6.0_31, pero también se ha probado con el servidor de aplicaciones JBoss AS 7.1, aunque debería poder ser utilizada sin problema con otros servidores de aplicaciones compatibles con estos dos estándares.

### 3.1.4 Planificación

El diagrama de Gantt muestra la temporalización de las etapas principales y los hitos, que se corresponden con las entregas del proyecto.

| Id | Nombre de tarea | Duración |
|---|---|---|
| 1 | **Framework Presentación** | **95 días** |
| 2 | Elaboración propuesta PFC | 8 días |
| 3 | Entrega PEC 1 | 0 días |
| 4 | **Investigación** | **14 días** |
| 5 | Investigación de frameworks del mercado | 14 días |
| 6 | Estudio patrones | 14 días |
| 7 | Análisis | 14 días |
| 8 | Entrega PEC 2 | 0 días |
| 9 | Diseño | 14 días |
| 10 | Implementación Framework | 21 días |
| 11 | Guía de uso del framework | 3 días |
| 12 | Implementación ejemplo | 7 días |
| 13 | Entrega PEC 3 | 0 días |
| 14 | Elaboración memoria | 14 días |
| 15 | Entrega TFC | 0 días |

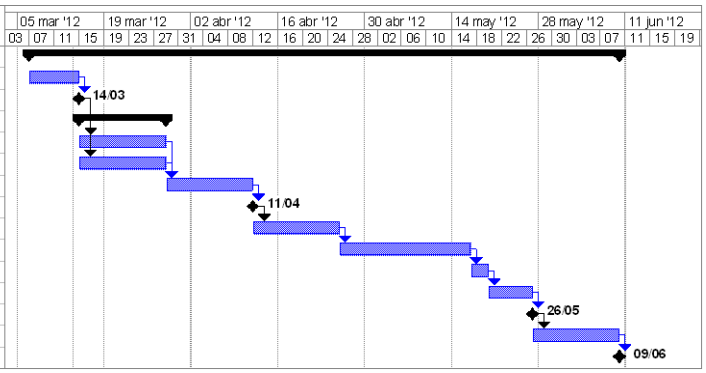



Ilustración 3:1 Diagrama de Gantt con la planificación del proyecto

### 3.1.5 Productos obtenidos

Como resultado del proyecto se han obtenido los siguientes entregables:

- **Framework "MTP"**. Se entrega lo siguiente:
  - Librería para incorporarla a las aplicaciones.
  - Proyecto Eclipse para poder evolucionar el framework utilizando esta herramienta de desarrollo.
  - Fuentes, para su análisis o posterior evolución con otras herramientas de desarrollo.
  - Javadoc. Documentación generada para estudiar tanto el API público como el resto de clases.
- **Aplicación "AMPA"**. La aplicación ejemplo que demuestra el uso del framework:
  - Desplegable. WAR. Se entrega sin las librerías de terceros, que habrá que añadir para su uso.
  - Proyecto Eclipse, para poder modificar la aplicación.
  - Fuentes, para su análisis o posterior evolución con otras herramientas de desarrollo.
  - Script de creación de la base de datos.
- **Memoria**. Este documento, que contiene la información del trabajo realizado, y las instrucciones para la instalación de la librería y la aplicación ejemplo.
- **Presentación** del proyecto, en formatos Power Point y PDF con notas.

## 3.2 Marcos de trabajo de presentación del mercado

En los siguientes puntos se analiza la arquitectura de tres marcos de trabajo (frameworks) de mercado, los que se han considerado más representativos en el momento actual.

Se analizan varios puntos, siempre con el objetivo en mente de obtener el conocimiento para afrontar el posterior diseño de un nuevo framework de presentación: las principales características funcionales; la implementación que hace cada uno del patrón Modelo-Vista-Controlador; los componentes principales; el ciclo de vida de una petición; y la configuración que es necesario realizar.

Finalmente se hace una comparativa de las formas de afrontar distintos problemas en los tres frameworks estudiados.

### 3.2.1 JavaServer Faces 2

Creado por una serie de fabricantes de software, su intención es establecer un estándar en la gestión de la interfaz de usuario (IU) de aplicaciones web para clientes ligeros con J2EE, siguiendo el patrón Modelo-Vista-Controlador (MVC).

#### *3.2.1.1 Características*

- Estándar extensible, que permite añadir funcionalidad a los componentes básicos que oferta o crear otros componentes nuevos. De hecho hay varios frameworks que extienden JSF, como MyFaces o ICEfaces.
- Incluye un conjunto de clases para representar los componentes del IU, para gestionar su estado y para administrar los eventos de entrada. Con ello se consigue independencia del tipo de cliente, pudiendo generar por ejemplo código adecuado a un navegador estándar o a un teléfono con un navegador limitado.
- API para validación y conversión de datos de entrada a tipos Java.
- Internacionalización y regionalización (locale) del IU.
- Soporta estándares de accesibilidad (WAI), JavaScript y AJAX.
- Utiliza librerías de etiquetas para la creación de componentes HTML y para otras operaciones como validación, conversión, balanceo de carga.
- Uso opcional de un mapa de navegación centralizado.
- Mantenimiento de estado a diferentes niveles (petición, sesión, aplicación y vista).

#### *3.2.1.2 Implementación de MVC*

Los elementos del patrón MVC en JavaServer Faces 2 son:

- Controlador: Faces Servlet. Siguiendo el patrón FrontController, este servlet es la parte del framework que gobierna el ciclo de vida de la petición delegando las responsabilidades al resto de componentes.
- Modelo: Managed Beans. Son clases normales que utilizan la nomenclatura estándar de los Java Beans (métodos get y set, constructor vacío y serializables). Estos objetos son referenciados por el framework bien en el fichero faces-config.xml o bien mediante anotaciones Java en el código de los propios Managed Beans.
- Vista: Típicamente se implementa con ficheros XHTML (denominados facelets) aunque también soporta JSP. Las vistas deben importar las directivas para el

namespace de JSF para las librerías de etiquetas Core y HTML (patrón View Helper). La navegación entre vistas se puede hacer de forma explícita en las vistas, pero también se puede crear un mapa de navegación centralizado en el faces-config.xml.

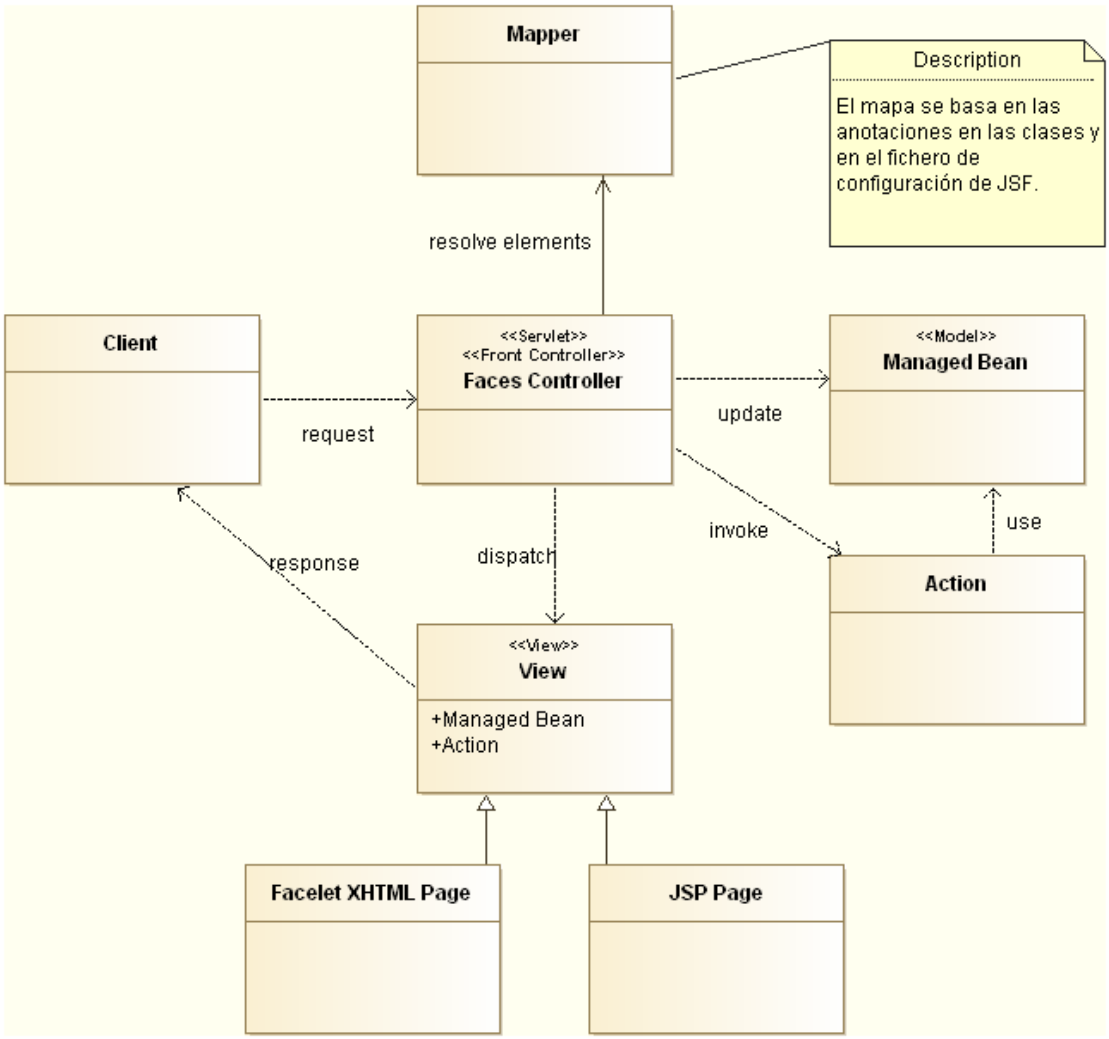



Ilustración 3:2 Diagrama de clases en JavaServer Faces

El Faces Controller aplica el patrón Front Controller, estando implementado con un servlet (FacesServlet). Se apoya en un mapa de recursos (patrón Mapper) que carga en función del fichero de configuración y de anotaciones en las clases que realizan las distintas funciones.

Los Managed Bean representan los datos del modelo, que puede comportarse como un Transfer Object, mientras que las Action (patrón Command) representan las distintas acciones que pueden ser invocadas para implementar los procesos de negocio.

View representa a los tipos de vistas que pueden ser invocados para representar la respuesta a los usuarios, que consiste principalmente por páginas Facelet XHTML, aunque también están soportadas las páginas JSP.

#### 3.2.1.3 Ciclo de vida de las peticiones

En JavaServer Faces las peticiones tienen las siguientes etapas:

1. Crea una estructura de datos (vista) o recupera los valores de la petición anterior (si se trata de un proceso con varios pasos).
2. Lee los valores de la petición y los aplica a la vista creada/recuperada.
3. Realiza las validaciones correspondientes.
4. Efectúa las actualizaciones pertinentes en el modelo. En este punto realiza las llamadas correspondientes a los Managed Beans.
5. Llama a los métodos de negocio (Actions) para realizar las acciones necesarias.
6. Recupera la página de respuesta, la formatea con los datos necesarios para el cliente oportuno.

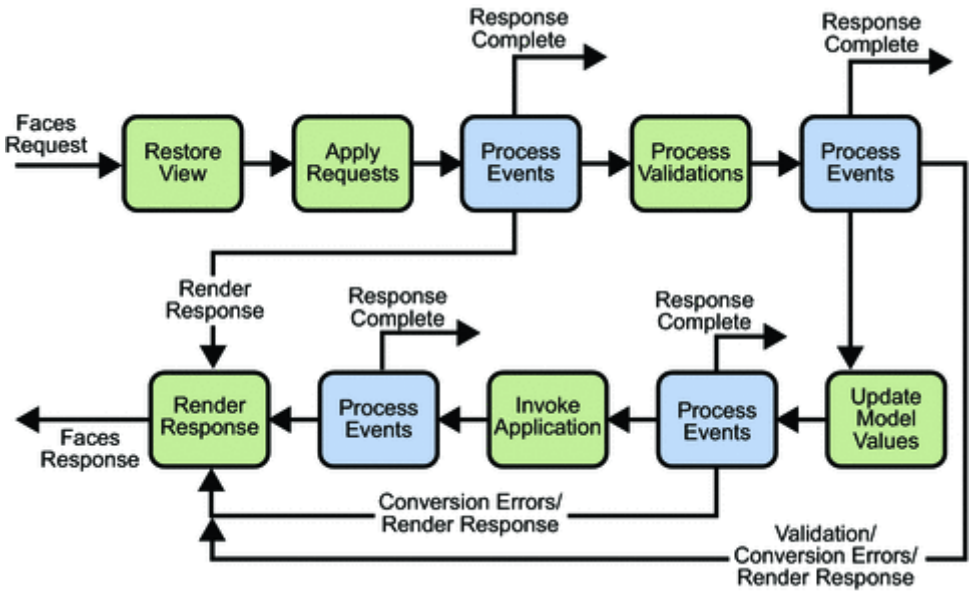



Ilustración 3:3 Java Server Faces Standard Request Life Cycle

Fuente: The Java EE 5 Tutorial. JavaServer Faces Technology

#### 3.2.1.4 Configuración

La configuración comprende los siguientes elementos:

- web.xml. Una entrada en el web.xml activa el Servlet que actúa como controlador de Faces y se especifica un patrón de URL, como /faces/* (normalmente se mapea a /faces/*, *.jsf o *.faces).
- faces-config.xml. Este fichero de configuración se sitúa en el directorio WEB-INF/. Contiene el mapa de navegación, los managed beans y otros componentes, como validadores o conversores. El uso es opcional, ya que la mayoría de elementos se pueden definir mediante anotaciones en el código.
- Librerías de JSF. Si el contenedor no soporta JSF, las librerías con la implementación de JSF se ubican en el directorio de librerías de la aplicación (WEB-INF/lib) o bien se definen las librerías en el servidor J2EE.

### 3.2.2 Struts$^2$

Struts es el framework de presentación de Apache. La segunda generación de Struts surge del producto WebWork, que a su vez fue una escisión del primer Struts. Implementa una arquitectura MVC basada en Servlet Filters.

#### *3.2.2.1 Características*

- Validación de datos de entrada y conversión.
- Internacionalización y regionalización (locale) del interfaz de usuario.
- Soporte de AJAX.
- El modelo MVC permite una separación clara entre el código de negocio y la interfaz de usuario.
- Soporta varias tecnologías de visualización como JSPs, plantillas Velocity, XSLT y permite incorporar nuevas tecnologías manteniendo la separación de responsabilidades de la vista.
- Configuración a través de fichero XML o de anotaciones Java. La configuración asume como valores por defecto los más adecuados para un uso general, lo que reduce drásticamente las necesidades de configuración.
- Definición del mapa de navegación a través del fichero de configuración o de anotaciones Java.
- Implementación por defecto de interceptores y resultados.
- Almacén de datos de aplicación accesible desde todo el proceso de la petición (ValueStack), que facilita aislar de manera natural el código de la vista y del modelo.
- Componentes de la interfaz de usuario predefinidos a través de una librería de etiquetas.
- Extensible a través de plugins.

#### *3.2.2.2 Implementación de MVC*

Los elementos del patrón MVC en Struts$^2$ son:

- Controlador: Implementado por el FilterDispatcher, es un "Servlet Filter" que procesa las peticiones entrantes y, en función de estas peticiones decide qué acción es la que debe ejecutar.
- Modelo: El modelo se implementa mediante clases Java denominadas acciones (Action). Una acción (patrón Command) se dispara con motivo de una petición recibida para ejecutar la lógica de negocio y para servir como vehículo para intercambiar información entre el usuario y la aplicación. Con la cadena de texto retornada por la acción el framework sabe a qué vista/resultado ha de redirigir la salida hacia el usuario. La clase de acción contiene el código de negocio en sí -método execute()- pero también funciona como un Transfer Object para ofrecer información a la vista.
- Vista: Se representa con un elemento result que normalmente consiste en páginas JSP y plantillas de Velocity, aunque se soportan otras tecnologías como XSLT. La vista a representar se elige según el resultado de la acción ejecutada. Permite aplicar el patrón Composite View a través de Apache Tiles.

#### 3.2.2.3 Ciclo de vida de las peticiones

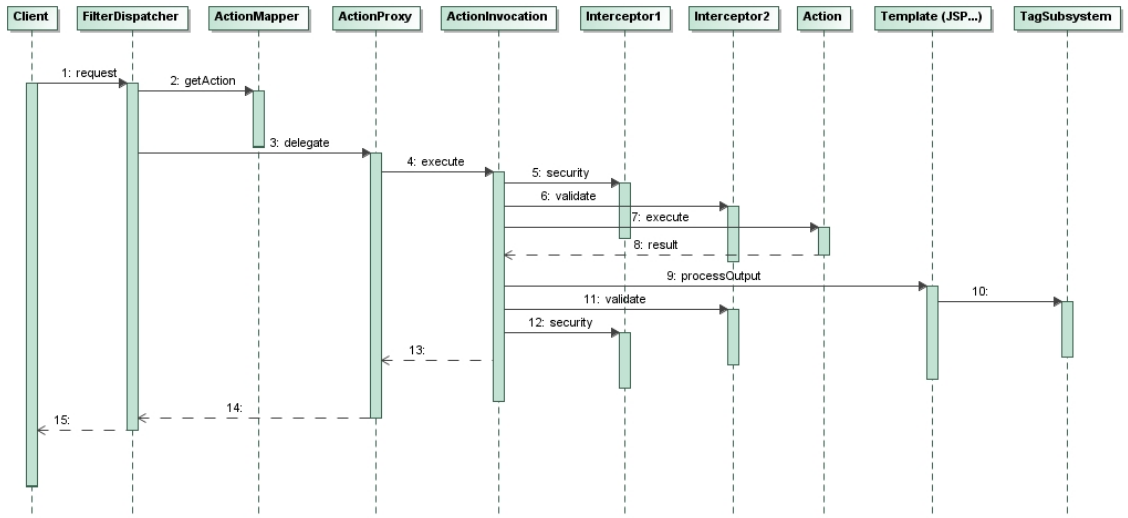



Ilustración 3:4 Diagrama de secuencia de una petición con Struts 2

Cuando se recibe una petición del cliente, la recoge el FilterDispatcher (patrón Intercepting Filter) y escoge la acción a ejecutar apoyándose en el ActionMapper (patrón Mapper).

Para invocar la acción se apoya en el ActionProxy (patrón Abstract Factory) que encapsula cómo obtener la acción, y éste a su vez en el ActionInvocation que encapsula la ejecución de la acción. Para ello, el ActionInvocation llama primero a una pila de interceptores que realizan tareas que no son propias de la acción, dejándola limpia de tareas administrativas y fácilmente reaprovechables como validación y conversión de datos, upload de ficheros o logging. Después de los interceptores, el ActionInvocation llama a la acción en sí (patrón Command).

Con el resultado de la acción se decide qué vista hay que ejecutar (se configura la relación entre resultados de la acción y plantilla a cargar), invocando al adecuado (JSP, FreeMarker, Velocity…). Estas plantillas pueden hacer uso del subsistema de etiquetas (patrón View Helper).

Después, el ActionInvocation vuelve a llamar a los mismos interceptores, pero en sentido inverso. Los interceptores pueden realizar operaciones en uno o en los dos momentos en que son invocados. Finalmente se devuelve la vista al cliente.

El framework guarda la información de la aplicación en el ValueStack (patrón Context Object), un objeto accesible mediante el lenguaje de expresiones OGNL y que está almacenado en el hilo de ejecución (thread) que procesa la petición, por lo que estos datos están disponibles durante todas las etapas de procesamiento de la petición. La información también está accesible en la propia acción como variables locales, al poder almacenar campos del formulario de entrada en atributos de la acción, la cual a su vez es almacenada en el ValueStack.

#### 3.2.2.4 Configuración

La configuración se realiza con los siguientes elementos:

- web.xml. Se define el FilterDispatcher de Struts2, el mapeo de direcciones que debe procesar y los paquetes donde Struts2 debe buscar clases con anotaciones.
- Declaración de elementos (ficheros de configuración o anotaciones). Se declaran los objetos que se van a utilizar como acciones (action), resultados (result) e interceptores (interceptor), bien sea a través de los ficheros de configuración o bien a través de anotaciones en el código Java. El fichero de configuración struts.xml, situado en WEB-INF/classes/, puede apuntar a otros ficheros con el fin de distribuir la configuración en módulos.
- Configuración del framework. A través del fichero de configuración struts.xml se pueden definir propiedades del framework para cambiar los valores por defecto.
- Librerías. La librería principal del framework es struts2-core.jar, pero también es necesario instalar otras librerías adicionales para soportar los plugins así como las dependencias a otras librerías. Según la documentación de Struts 2 actualmente, son necesarias además las librerías: xwork.jar, ognl.jar, freemarker.jar, commons-logging.jar, commons-fileupload.jar y commons-io.jar.

### 3.2.3 Spring MVC

El framework Spring incluye varios módulos que cubren todas las áreas de una aplicación J2EE, y se integran tanto con los servicios ofrecidos por el contenedor J2EE como con otros frameworks. El paquete Spring MVC se encuadra en la capa de presentación proporcionando una implementación del patrón modelo-vista-controlador.

Spring presenta varias particularidades que también son aplicables al paquete MVC, entre las que cabe destacar el desacoplamiento entre componentes de aplicación, que quedan unidos a través de recursos que presenta el propio framework; el trabajo con POJOs, ya que no es necesario extender a las clases del framework ni importar interfaces, solo requiere que las clases de la aplicación cumplan algunas características para integrarlas en el framework; el mecanismo de inversión de control consiste en que es el framework el que se encarga de instanciar e invocar al código y no al revés.

#### *3.2.3.1 Características*

- Configuración del framework y del mapa de navegación a través ficheros y/o anotaciones. Resolución dinámica de controladores y vistas por varios mecanismos a través de componentes del framework.
- Separación de roles en varios tipos de componentes (DispatcherServlet, Controller, View…).
- Permite la reutilización de código existente al no requerir que las clases del usuario extiendan o implementen clases o interfaces del framework.
- Soporte de múltiples tipos de salida (JSP, Velocity, Tiles, FreeMarker…).
- Componentes de la interfaz de usuario predefinidos y soporte de temas mediante librerías de etiquetas.
- Integración con otros frameworks de presentación.
- Validación de parámetros y upload de ficheros.
- Integración con el resto de componentes del framework Spring que incluye funcionalidades en las capas de presentación, negocio e integración.
- Soporta flujos de navegación a través de la extensión Spring Web Flow.

#### 3.2.3.2 Implementación de MVC

Los elementos del patrón MVC en Spring MVC son:

- Controlador: El DispatcherServlet actúa con el patrón Front Controller, es un Servlet implementado por el framework que se apoya en uno o varios handler mappings (patrón Mapper) para decidir a qué Controller debe entregar cada petición.
- Modelo: La aplicación puede implementar varios Controller (patrón Application Controller) que se encargan de ejecutar el código de negocio y de recuperar los datos del modelo para que después el framework los entregue a la vista correspondiente.
- Vista: Las vistas son archivos JSP normalmente, pero pueden ser otro tipo de recursos como plantillas Velocity. Para decidir a qué vista debe entregar el control, el DispatcherServlet se apoya en los ViewResolver que transforman los nombres lógicos de vistas devueltos por el Controller a nombres físicos de recursos utilizando varios métodos (resolución por nombre, fichero de propiedades o XML…).

#### 3.2.3.3 Ciclo de vida de las peticiones

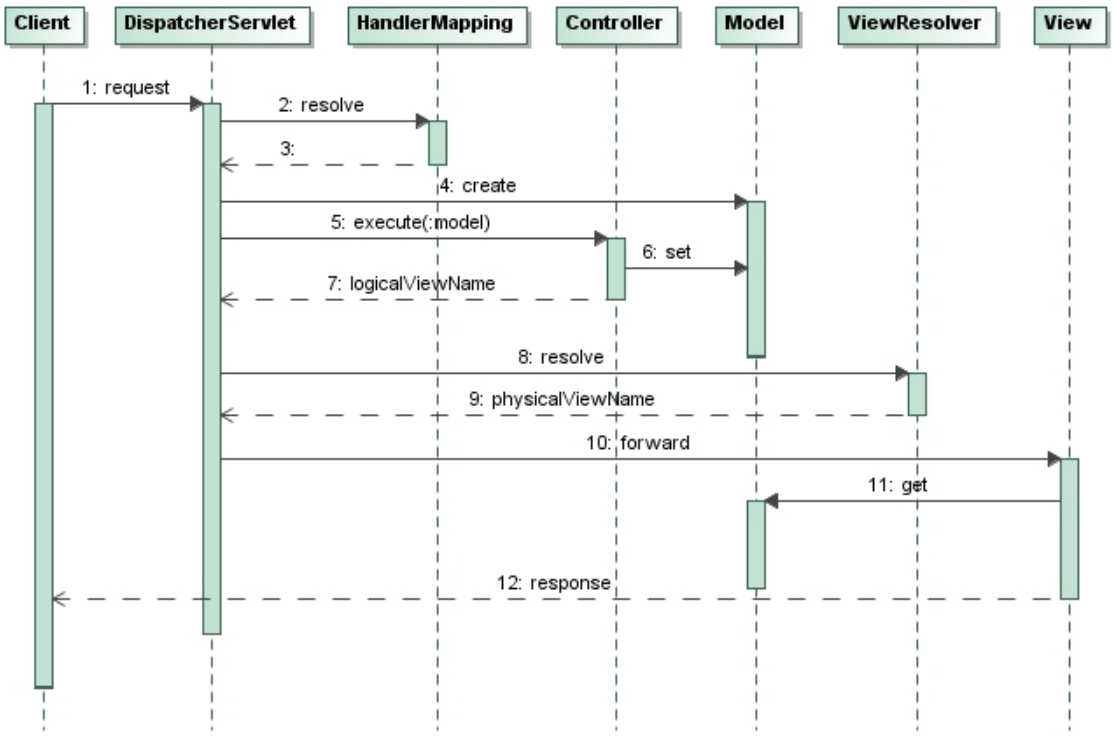



Ilustración 3:5 Diagrama de secuencia del ciclo de vida de una petición en Spring MVC

Cuando el cliente envía una petición, la recibe en primer lugar el DispatcherServlet, que hace el papel de FrontController, el cual consulta los handler mappings (patrón Mapper) para decidir a qué Application Controller (Controller) debe enviar la petición. Hay varios handler mappings que permiten mapear los Controller con varios métodos, como anotaciones o asociación de URLs a nombres concretos de clases entre otros.

Una vez el DispatcherServlet entrega el mando al Controller, éste realiza las acciones de negocio correspondientes y carga los datos del modelo en un parámetro recibido por el Controller (patrón Transfer Object), que puede consistir en un mapa de objetos (Map<String, Object>) o un objeto Model. El método retorna el nombre lógico de la vista a procesar que recibirá el DispatcherServlet.

Después, el DispatcherServlet convierte el nombre lógico de la vista recibida del Controller a un nombre real utilizando un View Resolver (patrón Mapper).

El último paso del DispatcherServlet consiste en entregar el control a la vista de salida, poniendo a su disposición el conjunto de datos del modelo recibido del Controller para que cargue el contenido y devolver al cliente el resultado.

#### *3.2.3.4 Configuración*

La configuración se realiza en los siguientes elementos:

- web.xml. Hay que declarar el DispatcherServlet en el descriptor de despliegue e indicar las URLs que procesará el Servlet.
- Fichero de configuración. Ubicado en el directorio WEB-INF, el nombre del fichero se deriva del nombre asignado al Servlet en el web.xml, por ejemplo si el Servlet se nombra como springmvc, el fichero de configuración sería springmvc-servlet.xml. La configuración vía anotaciones se puede configurar en éste fichero.
- Librerías. Se deben incorporar las librerías de Spring utilizadas en el directorio de librerías de la aplicación (WEB-INF/lib) o en el classpath del contenedor. Las librerías básicas requeridas son spring-aop.jar y aspectjweaver.jar, pero será necesario añadir sus dependencias así como las correspondientes a los módulos que se utilicen.

### 3.2.4 Comparación entre los marcos de trabajo

A continuación se describen las diferentes formas en las que los framework estudiados han implementado las funcionalidades que serán necesarias en el framework a elaborar.

**Implementación de MVC**. Todos los marcos de trabajo analizados se decantan por utilizar el patrón Service to Worker.

**Configuración**. La configuración de los componentes del framework puede venir dada por tres partes: explícita a través de ficheros de configuración; a través de anotaciones Java o bien de forma implícita por el nombre de las clases, que pertenecen a un paquete determinado previamente prefijado por configuración en ficheros.

**Controlador**. En los casos de JSF2 y Spring MVC se ha implementado con un Servlet (Front Controller), mientras que en el caso de Struts[2] se ha implementado con un servlet filter (Intercepting Filter). La diferencia es que el servlet filter puede actuar sobre otros recursos servlets o páginas JSP y se pueden encadenar varios filtros, mientras que el servlet no se puede encadenar ni puede actuar sobre otros contenidos.

**Selección del modelo a ejecutar**. En todos los casos se permite introducir algún tipo de mapeo entre la URL o acción (en un parámetro) solicitada y la clase del modelo a ejecutar.

Para ello el controlador se apoya en una clase Mapper que convierte la URL o nombre de acción en una clase a ejecutar.

**Visibilidad de los datos en el modelo**. Hay que dar visibilidad a los datos generados en el request y a los datos de la sesión, pero evitando que el modelo tenga que acceder a objetos propios de la tecnología (objetos request y sesión) (patrón Context Object). En un caso se almacenan mediante una variable de tipo ThreadLocal para estar accesibles durante la vida de la petición, en otros casos se pasan a la clase del modelo, bien por parámetro al método que ejecuta la acción o bien a través de atributos de la instancia del modelo que son cargados por el framework.

**Visibilidad del modelo en la vista**. El framework debe hacer visibles a la vista los datos cargados por el modelo (patrón Transfer Object). Generalmente se utilizan librerías de etiquetas para facilitar esta operación (patrón View Helper) y evitar en lo posible programar en la vista código específico del framework, más allá de las etiquetas de las librerías.

**Validación de parámetros**. Los parámetros del request son validados bien en la parte cliente mediante algún tipo de código de scripting (como Javascript) o en servidor, habiendo introducido en los componentes de la vista las reglas de validación o bien mediante indicación de las validaciones a realizar antes de ejecutar la acción del modelo en un fichero de configuración.

**Modelo**. En el caso de JSF el código de negocio está representado por clases acción, mientras que los datos se gestionan con managed beans. En el caso de Struts2 se utilizan clases acción que a su vez pueden contener los datos, accesibles a través del ValueStack para su tratamiento posterior en la vista. Finalmente, en Spring MVC el modelo está representado por clases con el patrón Application Controller, que reciben los datos de la petición en un parámetro Map, que a su vez utiliza para dejar los datos del modelo para la vista.

**Selección de la vista a representar**. La selección de vista a ejecutar viene dada en función del resultado de la ejecución del modelo, que puede retornar el nombre de la vista a utilizar o bien retornar un valor lógico que es convertido por una clase Mapper del framework utilizando un mapa de navegación.

**Vista**. La vista queda representada por distintas tecnologías de visualización, como páginas JSP, plantillas Velocity o XHTML entre otros. Se dispone en las vistas de librerías de etiquetas (patrón View Helper) para introducir los componentes de la interfaz de usuario y para recuperar los datos del modelo.

**Internacionalización**. Para resolver los problemas de traducción de páginas, se apoyan en librerías de etiquetas y en ResourceBundles para obtener las traducciones. El framework se encarga de obtener estas traducciones para la aplicación.

**Autenticación y control de accesos**. Con Struts2 la autenticación se puede implementar a través de un Interceptor (los interceptores se ejecutan antes de la acción y después del resultado). Con Spring la autenticación se realiza a través del módulo Spring Security, que permite definir una página de login y un patrón de URLs que requieren al usuario estar

autenticado. Finalmente, con JSF, se especifican en el descriptor de despliegue los roles admitidos para cada conjunto de URLs.

La tabla muestra el resumen de las características más relevantes de los frameworks analizados:

| Característica | JSF2 | Struts2 | Spring MVC |
|---|---|---|---|
| **Implementación MVC** | Service to Worker | Service to Worker | Service to Worker |
| **Configuración** | Fichero XML o anotaciones Java | Fichero XML o anotaciones Java | Ficheros o anotaciones |
| **Controlador** | Servlet | Servlet Filter | Servlet |
| **Selección del modelo** | Mapa de recursos<br>Invocación desde la vista | Action Mapper sustituibles o parámetro action mapeando opcionalmente por nombre de clase de acción | Handler mapping (patrón Mapper). Usa anotaciones, asociación de URL a nombres de clases… |
| **Visibilidad de los datos de la petición y sesión en el modelo** | Envío desde la vista | Value Stack (almacén de datos visible en todo el proceso del request) | Entrega de datos al modelo por parámetro (Map) o un objeto Model |
| **Visibilidad del modelo en la vista** | A través de Java Beans | Value Stack | Los entrega el framework a la vista |
| **Parámetros** | API para validación y conversión a tipos Java | Validación<br>Conversión a tipos Java<br>Upload de ficheros | Validación<br>Upload de ficheros |
| **Modelo** | Clases acción (Command)<br>Managed beans (Java beans) (Transfer Object) | Clases acción (Command), que sirven como Transfer Object | Application controller (ejecutan código de negocio y recuperan datos para la vista) |
| **Selección de la vista** | Mapa de navegación centralizado navegación entre vistas | Mapa entre resultado de la acción y vista a cargar.<br>Posibilidad de que la acción retorne el nombre físico o lógico. | View Resolver. Varios métodos que convierten el nombre lógico del Application Controller por nombre, fichero de propiedades, XML…<br>Flujos de navegación con Spring Web Flow |
| **Apoyos vista** | Componentes de UI con clases, control estado, eventos, independientes al tipo de cliente<br>WAI, JavaScript, AJAX<br>Taglibs (componentes HTML, validación en cliente, conversión…) | Taglibs, AJAX, Componentes UI con taglib | Componentes UI con taglib |
| **Tipo vista** | XHTML, JSP… | JSP, Velocity, FreeMarker, XSLT, abierto a más tecnologías, también patrón Composite View (Tiles) | JSP, Velocity, FreeMarker, Tiles… |
| **Internacionalización** | Internacionalización y regionalización | Resource Bundle + Taglib | Properties file<br>Locale Change Interceptor<br>Taglib |
| **Autenticación y control de accesos** | Roles en descriptor de despliegue | Interceptor | Spring security |
| **Otros** | Extensible (hay extensiones de varios fabricantes disponibles) | Valores más comunes en configuración por defecto.<br>Extensible con plugins<br>Mapa de navegación<br>Interceptores programables para gestionar cosas comunes (logging, seguridad…) | Integración con Spring (framework que cubre todas las capas de aplicación) y con otros frameworks de presentación<br>Desacoplamiento entre componentes (los acopla el framework).<br>POJOs (no hay que importar interfaces del framework)<br>Inversion of control. El framework instancia el código de aplicación |

Ilustración 3:6 Tabla comparativa de características entre frameworks de mercado.

## 3.3 Patrones

Los patrones se plantean como soluciones probadas a problemáticas concretas durante las fases de la ingeniería del software. En este apartado se describen patrones de varios tipos que van a ayudar a comprender los frameworks de presentación estudiados y a construir uno nuevo. También serán de utilidad durante la construcción de la aplicación de ejemplo.

Se han dividido en tres partes: en primer lugar se presenta el patrón arquitectónico modelo-vista-controlador que es el eje principal sobre el que se basan este tipo de frameworks; después se presenta un conjunto de patrones de diseño de J2EE, algunos de utilidad para el framework y otros de utilidad para el diseño del resto de aplicación; y finalmente se presentan otros patrones del reconocido grupo Gang Of Four, también de utilidad para determinados aspectos de la construcción del framework.

### 3.3.1 Patrón Modelo-Vista-Controlador

El patrón MVC es un patrón que se utiliza en la capa de presentación de una aplicación para separar la lógica de negocio de la presentación al usuario y del flujo de la aplicación. Los tres componentes principales son el modelo, la vista y el controlador:

- Controlador: Su misión es controlar la lógica de interacción entre el usuario y la aplicación, gestionando la entrada de datos de la aplicación y seleccionando el modelo y la vista adecuados.
- Modelo: Representa la lógica de negocio de la aplicación.
- Vista: Es la interfaz entre la aplicación y el usuario.

Cuando llega una acción del usuario, el controlador la recoge y la gestiona, accediendo después al modelo, probablemente también para cambiar el estado. Después entrega el control a la vista y le entrega una referencia al modelo para que lo pueda consultar antes de devolver al usuario la interfaz actualizada.

En J2EE se habla tradicionalmente de dos tipos de implementación del patrón MVC:

- Tipo 1 (patrón Dispatcher View): El usuario invoca directamente a una página JSP que actúa como controlador y como vista y accede al modelo que está representado por otro objeto (un EJB, POJO…).
- Tipo 2 (Patrón Service to Worker): En este caso el controlador está implementado por un Servlet, que es el que gobierna el flujo de la petición. El modelo se representa igualmente por un objeto Java (EJB, POJO…) y la vista se representa por una JSP.

El tipo 2 es más extensible, pues queda separada la presentación (páginas JSP) del control del flujo de la aplicación y de la invocación al modelo.

### 3.3.2 Patrones de diseño J2EE

Se describen a continuación los patrones de diseño de J2EE[1], divididos en las capas de presentación, negocio e integración. Los de la capa de presentación, dada su naturaleza, son los más utilizados en los marcos de trabajo analizados.

#### *3.3.2.1 Capa de presentación*

**Intercepting Filter**: Es un filtro que se ubica entre el usuario y el servidor para interceptar y modificar las peticiones y respuestas al cliente. Permite realizar de forma centralizada funciones de logging, gestión de juegos de caracteres, compresión, inserción de componentes desacoplados en la respuesta… Se pueden encadenar varios filtros que realicen acciones diferentes. Estos filtros se pueden utilizar, a partir de la especificación Servlet 2.3, declarándolos en el descriptor de despliegue.

**Front Controller**: Es un punto único de entrada a la aplicación que permite centralizar el ciclo de vida de las peticiones de los clientes. Se puede ubicar aquí el código común a todas las peticiones (control de acceso, selección de modelo y vista…), aunque normalmente se delega cada una de estas funciones en otras clases. Normalmente se implementa con un Servlet.

**Context Object**: Permite encapsular la información de la petición mostrando un API independiente de la tecnología del elemento que la genera con la intención de que las partes de la aplicación que van a utilizar esta información sean independientes de dicha tecnología. Como ejemplo, se puede encapsular la información que llega en el request en un Context Object para que luego sea procesada por otros objetos de las capas de presentación o de negocio.

**Application Controller**: Se encarga de realizar las acciones de negocio correspondientes según el tipo de petición recibida. Después puede ser responsable de elegir la vista a utilizar en la respuesta. También puede apoyarse en un Mapper para elegir la acción a invocar y otro para seleccionar la vista a mostrar.

**View Helper**: Su misión es limpiar de código de negocio la vista. Al utilizar vistas basadas en plantillas (JSP, Velocity…), éstas deben evitar incorporar código de negocio, por lo que delegan el procesamiento del modelo en clases Helper. Estos Helper pueden presentarse en forma de POJOs, custom tags o tag files. Sirven como adaptadores entre el modelo y la vista en particular.

**Composite View**: Permite construir vistas basadas en la composición de otras, permitiendo reutilizar componentes comunes entre varias de ellas, como cabeceras o menús.

**Service to Worker**: Obtiene el control de la ejecución e invoca a un comando o proceso de negocio con el objetivo de ejecutar la acción que corresponda y recuperar la información del modelo, pasando después el control a la vista que puede generar una respuesta dinámica basada en esta información del modelo. Se apoya en el Front Controller para

[1] Fuente: Core J2EE Patterns: Best Practices and Design Strategies, Second Edition

recuperar el control de las peticiones, en el Application Controller para procesar el Modelo y en el View Helper para la creación de las vistas.

**Dispatcher View**: Se pasa el control a la vista que invoca directamente al procedimiento de negocio. Puede ser útil para vistas con contenido estático y para aplicaciones sencillas con poca integración entre la vista y el negocio. También se puede utilizar para vistas que trabajan con un modelo de presentación previamente cargado. El mantenimiento de las vistas es más costoso, pues pueden incorporar directamente llamadas a procedimientos de negocio. La invocación a la vista puede ser directamente desde el cliente o a través del Front Controller o el Application Controller que actúan como intermediarios para arrancar la vista.

#### *3.3.2.2 Capa de negocio*

**Business Delegate**: Usado como intermediario de acceso a la capa de negocio, este delegado se sitúa en la parte cliente del negocio (p. ej. la capa de presentación) para: separar los problemas de invocación a las clases de negocio (que pueden estar situadas en una ubicación remota); actuar como caché evitando accesos repetidos si es necesario; ofrecer un acceso uniforme a la capa de negocio consiguiendo un desacoplamiento entre ambos.

**Service Locator**: Permite localizar un servicio de negocio independientemente de dónde esté ubicado, de forma que el cliente solo se tiene que ocupar de pedirle al Service Locator que le indique cómo llamar al servicio de negocio.

**Session Façade**: Presenta en una interfaz único varios servicios de la capa de negocio, que de otra forma serían expuestos individualmente. Unificando la publicación de servicios de negocio: se consigue independizar a los clientes de las implementaciones particulares de los métodos de negocio reduciendo el acoplamiento; permite también que el acceso de clientes remotos se realice siempre a través de esta interfaz centralizada; oculta las interdependencias entre componentes de negocio y servicio; presenta solo los servicios necesarios para el cliente.

**Application Service**: Sirve para agregar varios objetos de negocio en un objeto global que ofrece este servicio como una unidad. Se consigue evitar la lógica de negocio en los objetos fachada y centralizar el funcionamiento de un servicio de negocio. También permite encapsular la lógica específica de cada caso de uso en un objeto individual y aislado de los objetos de negocio.

**Business Object**: Separa la lógica de negocio de los datos mediante un modelo de objetos. Se consigue encapsular los datos, su comportamiento y persistencia, lo cual resulta conveniente cuando el modelo de datos y el modelo conceptual difieren. Estos objetos se suelen representar como POJO o como entity beans.

**Composite Entity**: Implementan la persistencia de objetos de negocio con objetos locales (entity beans o POJO). Permite la agregación de objetos relacionados en entidades de grado mayor, como un objeto principal y varios objetos relacionados (p. ej. una factura y sus líneas de detalle).

**Transfer Object**: Almacena varios elementos de datos en un solo objeto que puede ser transferido entre capas. Se consigue reducir el tráfico entre capas al no ser necesario hacer múltiples llamadas remotas a métodos get y set, pues el objeto es una copia local del original. El objeto puede viajar en ambos sentidos, conteniendo información a leer o información a actualizar.

**Transfer Object Assembler**: Este objeto construye una parte del modelo de aplicación en un Transfer Object compuesto a su vez por otros Transfer Object, de forma que solo viaja un objeto entre las distintas capas de la aplicación.

**Value List Handler**: Se encarga de realizar una búsqueda de datos a través de un Data Access Object, guardando los resultados en una memoria intermedia y permitiendo al cliente el acceso a los mismos a través de un Iterator.

#### *3.3.2.3 Capa de integración*

**Data Access Object**: Implementan el acceso a los datos desde su fuente (RDBMS, LDAP, ficheros…), de forma que abstraen al resto de la aplicación de la implementación concreta del sistema de persistencia utilizado. Mediante un API propio consiguen independizar la aplicación de la tecnología utilizada.

**Service Activator**: Permite acceder a servicios de negocio de forma asíncrona. A través de mensajes JMS permite recibir peticiones asíncronas que después se convierten en llamadas a servicios de negocio.

**Domain Store**: Permite dar persistencia a un modelo completo de objetos. Esta persistencia se realiza a través de un framework de persistencia de mercado o bien desarrollando uno propio.

**Web Service Broker**: Publica y gestiona servicios web utilizando XML y protocolos web.

### 3.3.3 Otros patrones

En este punto se presentan otros patrones para la programación orientada a objetos, recopilados por el cuarteto conocido como *Gang Of Four*[2]. Sirven de apoyo para los patrones descritos, así como para la construcción de los framework. Se dividen en tres grandes conjuntos:

- Creational. Su objetivo es abstraer el proceso de creación de instancias.
- Structural. Se utilizan para crear estructuras de objetos complejas.
- Behavioral. Se utilizan para asignar responsabilidades a los objetos y para construir algoritmos.

#### *3.3.3.1 Creational patterns*

**Abstract Factory**: Permite crear objetos similares pero de diferentes clases, delegando su creación en una factoría abstracta, y así evitando la necesidad conocer las clases concretas de los objetos que hay que instanciar en cada momento. El cliente de la factoría después utiliza los objetos creados por ella utilizando un interface común.

[2] Design Patterns: Elements of Reusable Object-Oriented Software

**Builder**: Se independiza la creación de un objeto complejo en otra clase que se encarga de construirlo y devolver el resultado. Permite también crear especializaciones del proceso de creación para distintos tipos de objetos con un interface común.

**Factory Method**: Una clase (abstracta) delega la creación de objetos relacionados a una subclase. De esta forma, la subclase es la que decide la clase que debe utilizar para crear el objeto relacionado y crearlo así con su misma especialización.

**Prototype**: Permite crear objetos clonando una instancia que sirve de prototipo.

**Singleton**: Asegura que solo hay una instancia de una clase. Esta única instancia es obtenida a través de un método estático de la propia clase, que la instancia solo la primera vez que es accedida.

#### *3.3.3.2 Structural patterns*

**Adapter**: Modifica la interfaz de una clase a una forma diferente para que otra clase cliente pueda ser compatible.

**Bridge**: Separa la implementación de una tarea de la clase que la representa. Así, un objeto que dice poseer determinada función, cuando se invoca dicha acción, la ejecución se realiza en otro objeto. De esta manera, el objeto principal abstrae la forma de implementar la acción que puede ser decidida al construir el objeto o en un momento posterior (se le pasa de alguna manera el nombre del objeto en el que debe delegar la acción). Con este patrón se evitan estructuras de herencia que pueden llegar a ser muy complejas.

**Composite**: Se crea un objeto que es representante de una jerarquía de objetos. Así un cliente interactúa solo con el objeto principal pudiendo interactuar de la misma forma con un objeto sencillo que con un conjunto de objetos de la misma jerarquía.

**Decorator**: Asigna responsabilidades adicionales a un objeto de forma dinámica. El objeto Decorator se convierte en una subclase del objeto que decora, implementando las nuevas responsabilidades.

**Facade**: Presenta al cliente un interface que actúa como fachada de un subsistema, ocultándole los varios interfaces que pueda presentar ese subsistema y desacoplando a la vez ambos extremos.

**Proxy**: Es un objeto que actúa en nombre de otro para controlar el acceso al mismo desde terceros.

#### *3.3.3.3 Behavioral patterns*

**Chain of Responsibility**: Desacopla la llamada entre dos objetos, emisor y receptor, de forma que entre medias se puedan ubicar otros objetos y que alguno de ellos pueda decidir asumir el papel del receptor.

**Command**: Encapsula una llamada en un objeto. El objetivo es permitir a un cliente hacer llamadas a distintas operaciones sin necesidad de conocer la operación en sí a través de una interfaz común de invocación.

**Flyweight**: Para evitar crear muchas instancias de una misma clase, un objeto de este tipo actúa como todos ellos. Para lograrlo se extrae el estado de las instancias que varía según contexto. El cliente debe enviar el estado cada vez que requiera al objeto Flyweight actuar.

**Iterator**: Permite acceder a una lista de objetos de forma secuencial sin necesidad de conocer su implementación concreta.

**Mediator**: Actúa como intermediario en la interacción entre un conjunto de objetos, de forma que no es necesario que se conozcan entre ellos. Todos interactúan con el intermediario que sabe cómo conectar a unos con otros.

**Memento**: Permite capturar el estado de un objeto fuera de él para que pueda ser restaurado en un momento posterior.

**Observer**: Crea una dependencia entre varios objetos de forma que, cuando uno cambie el estado, sus objetos dependientes sean notificados del cambio de estado.

**State**: Permite cambiar el comportamiento de un objeto en función de su estado. Para ello se dispone de una clase que contiene lo común a todos los estados, además de un objeto que representa el estado, que es a su vez una jerarquía donde cada estado está representado por una subclase con la implementación particular del algoritmo adecuado al estado.

**Strategy**: Varios algoritmos alternativos son encapsulados en una interfaz de invocación para que el cliente que los utiliza los invoque de la misma forma. Nuevas implementaciones del algoritmo no requieren un cambio en el cliente.

**Template Method**: Ofrece en una clase abstracta el esqueleto de un algoritmo. Este esqueleto se compone de un conjunto de llamadas a otros métodos abstractos del mismo objeto que después son resueltos por subclases con implementaciones concretas del algoritmo.

**Visitor**: Se dispone de un conjunto de clases estable, pero se necesita añadir nuevas funciones. El Visitor permite realizar estas operaciones sin necesidad de modificar las clases existentes.

## 3.4 El Marco de Trabajo de Presentación "MTP"

El framework o marco de trabajo de presentación elaborado, en adelante MTP, recoge algunas de las mejores prácticas proporcionadas por el patrón arquitectónico modelo-vista-controlador, los patrones de diseño y los frameworks de presentación de mercado estudiados.

El objetivo es proporcionar a las aplicaciones J2EE con interfaz de presentación web una infraestructura básica para gestionar dicha interfaz, de forma que los desarrolladores se centren en diseñar el modelo de negocio y las páginas de presentación, liberándoles por tanto de tareas de programación que se repiten en múltiples proyectos similares como el control del flujo de navegación, las validaciones y conversión a tipos Java de los valores de entrada o la presentación de la interfaz en múltiples idiomas.

En todo el proceso se ha tratado de conseguir un marco de trabajo sencillo de utilizar por las aplicaciones, sin grandes artificios y que permita la reutilización de los objetos de negocio. Durante la implementación se ha tratado de utilizar las API propias de Java y J2EE, pero sin utilizar librerías de terceros que pueden causar problemas de versionado a las aplicaciones usuarias a la hora de integrar otros productos de terceros que sí pueden necesitar determinadas versiones de esas librerías.

### 3.4.1 Análisis

A continuación se exponen las características generales del que se han tenido en cuenta para diseñar MTP, así como una descripción detallada de las funcionalidades implementadas.

#### *3.4.1.1 Características*

MTP se basa en el patrón Modelo-Vista-Controlador para facilitar el desarrollo de la capa de presentación en aplicaciones J2EE basadas en páginas web, aunque también se utilizan otros patrones que se irán describiendo a medida que se describe la arquitectura del framework.

El componente principal desarrollado en el framework es el controlador, sobre el que giran el resto de componentes, que van tomando protagonismo a lo largo del ciclo de vida de las peticiones (request).

El controlador tiene dos eventos, que se describirán en detalle más adelante:

- Inicialización (configuración): Se dispara una vez durante el arranque de la aplicación en el contenedor J2EE. En esta etapa se configura el entorno de ejecución.
- Request: Se dispara cada vez que se recibe una petición de un cliente web. Este evento pasa por diferentes etapas para conformar el "ciclo de vida de una petición".

El componente modelo es implementado por la aplicación, utilizando Java Beans y sin necesidad de implementar interfaces o extender clases del framework. Solamente se necesita programación específica del framework cuando se desea consultar o modificar expresamente información de la sesión o del request, para lo cual se facilita un API.

La vista, implementada con JSP tampoco tiene necesariamente dependencias del framework, aunque para obtener la información del modelo utiliza librerías de etiquetas (taglibs) o llamadas a métodos de un objeto almacenado en el request.

Para implementar la capa de seguridad, el framework ofrece un API basado en roles que permite a la aplicación fijar los roles de las sesiones (las funciones de negocio de la aplicación deben decidir qué roles se asignan a cada sesión). Mediante configuración del framework se asignan roles a los modelos, de forma que pueda autorizar el acceso los modelos en función de los roles activos para cada sesión. Permite definir un conjunto de roles por defecto que son asignados a todas las sesiones en el momento de su creación.

La validación de datos se basa en formularios asociados a los modelos, que son verificados en varios niveles: conversión a tipos de datos básicos de Java; campos obligatorios; validación específica de campos individuales; y validación del formulario en conjunto. La lectura y carga de los datos de la petición se realiza en el mismo proceso de validación.

El soporte de internacionalización se realiza a través de un API facilitado para traducir los textos a diferentes idiomas, que también queda accesible a la vista a través de la librería de etiquetas.

#### *3.4.1.2 Descripción de las funciones del framework*

A continuación se describe el funcionamiento en detalle las características de MTP.

##### 3.4.1.2.1 Configuración

MTP requiere configurar el funcionamiento de algunas características por cada aplicación usuaria. El siguiente modelo de clases representa la configuración posible:

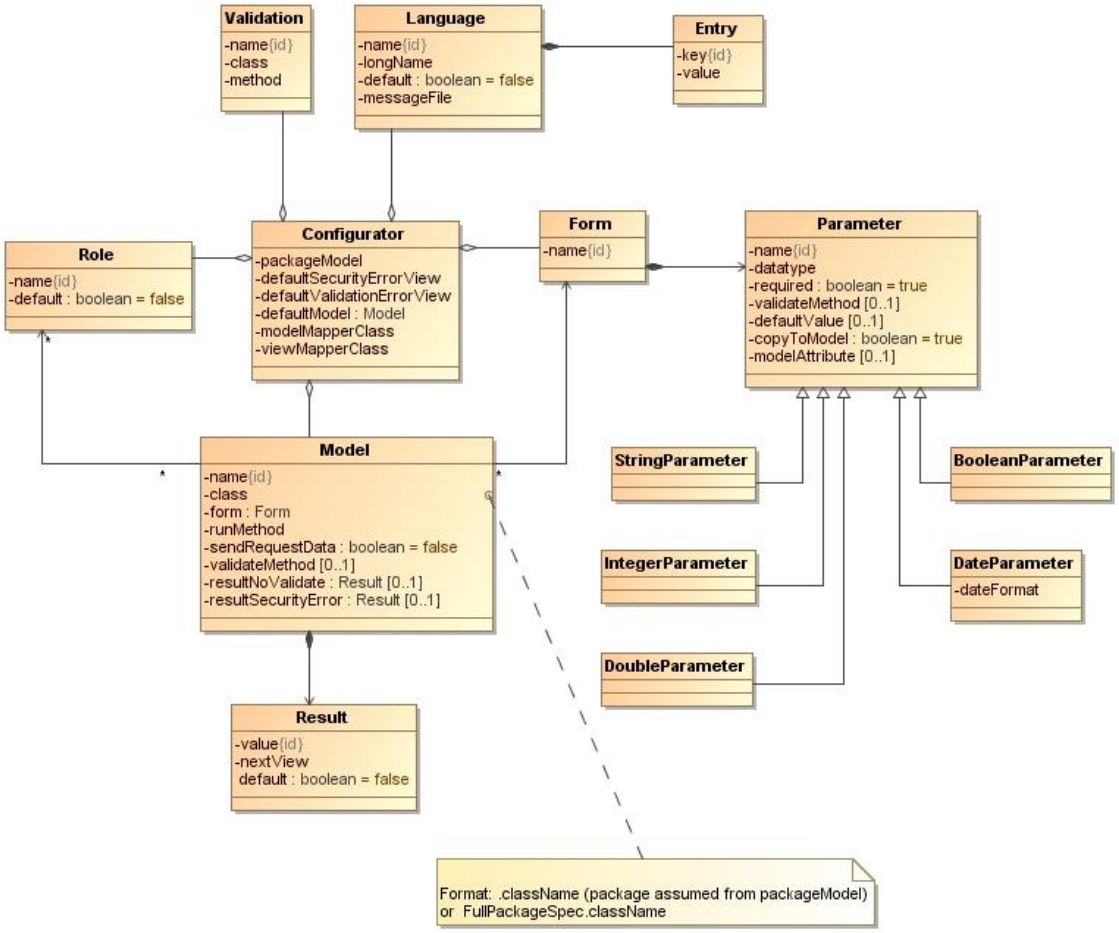



Ilustración 3:7 Diagrama de clases: Configuración del framework

**Configurator**: Referencia principal de configuración. Hay una instancia por aplicación (patrón Singleton).

**Language**: Cada uno de los idiomas soportados en las funciones de internacionalización. Cada idioma tiene asociado un fichero de mensajes (Resource Bundle).

**Entry**: Una entrada de un mensaje en un idioma. Es equivalente a una línea en un fichero de mensajes.

**Validation**: Función de validación, utilizada por parámetros individuales (p.ej. validación de códigos postales, NIF…).

**Role**: Rol de la aplicación, que se asocia a los modelos para dar permisos de acceso a estos últimos (un modelo sin roles es de acceso libre).

**Form**: Conjunto de parámetros de entrada que son recuperados del request, validados y opcionalmente copiados al modelo.

**Parameter**: Cada uno de los parámetros de un Form, puede ser de varios subtipos, según el tipo de datos Java equivalente.

**Model**: Cada tipo de modelo diferente que se ejecuta como resultado de una petición de un cliente. El tipo de modelo es un elemento de configuración que contiene los metadatos de la ubicación y comportamiento del modelo de aplicación que después se instancia durante la ejecución.

**Result**: Resultados que puede devolver modelo y que quedan asociados a vistas.

#### 3.4.1.2.2 Creación del RequestData y acceso a la información almacenada

El interface RequestData sirve para definir el API que va a permitir acceder en modo lectura y escritura (get y set) a la información durante la petición. Esta información comprende el modelo, el request, la sesión y demás información del framework accesible por la aplicación.

El framework implementa la clase HttpRequestData con este interface y que está adaptada para el protocolo http. HttpRequestData guarda la información generada por el framework durante la petición, así como una referencia al objeto HttpServletRequest y a través de él también tiene acceso al objeto HttpSession. El objetivo del interface es, aplicando el patrón ContextObject independizar a los consumidores y productores de ésta información de la implementación concreta (en este caso http).

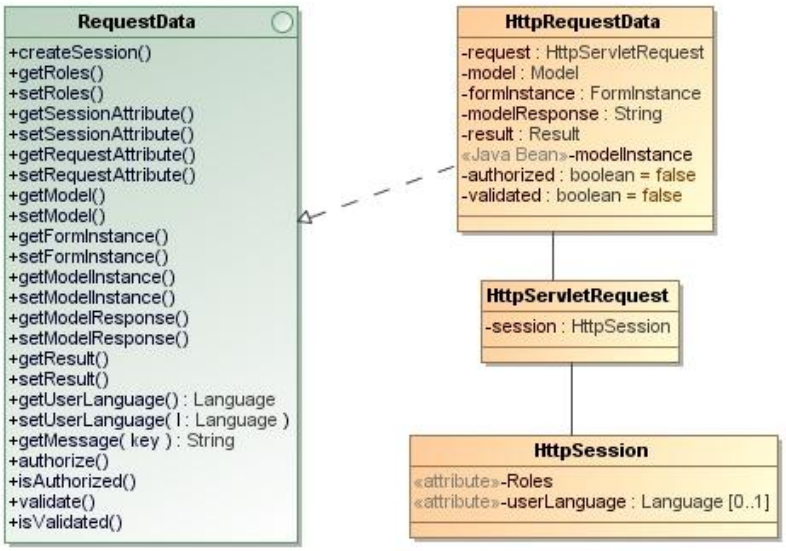



Ilustración 3:8 Diagrama de clases del interface RequestData

#### 3.4.1.2.3 Elección del modelo a cargar

Para seleccionar el modelo a instanciar y ejecutar se utiliza un Mapper que, basado en la configuración elige un modelo. La implementación de ModelMapper ofrecida por el framework busca el valor del parámetro MODEL del request en la lista de modelos disponibles. El framework permite sustituir este Mapper por otro algoritmo que implemente el interfaz ModelMapperImpl en una clase proporcionada por la aplicación definiéndolo en la configuración.

#### 3.4.1.2.4 Autorización

La autorización de acceso a modelos desde los clientes se hace a través de roles. Cada modelo que active la seguridad debe indicar los roles autorizados, alguno de los cuales

debe poseer la sesión que intenta acceder al modelo. En caso de que un modelo no tenga ningún rol asociado se asume que es de acceso libre.

Si no se cumplan los requisitos de seguridad, se redirige a una vista indicada por el modelo para estos casos, o, en su defecto a una vista definida a nivel global en la configuración para casos de denegación de acceso.

##### 3.4.1.2.5 Carga de datos y validación

El proceso de carga de datos de las peticiones se realiza a través de formularios. Después de haber seleccionado un modelo, el framework conoce qué formulario de entrada tiene asociado ese modelo. El formulario se compone de un conjunto de parámetros que el framework carga en variables del tipo Java equivalente y posteriormente valida. Cada parámetro individual del formulario puede ser cargado en el modelo de forma opcional.

La validación se produce secuencialmente en varios niveles:

1º Validación de conversión al tipo de datos Java adecuado.
2º Comprobación de valores requeridos.
3º Validación mediante función de verificación del parámetro individual.
4º Validación del modelo completo, a través de un método de validación del objeto modelo.

El siguiente diagrama muestra la relación entre un tipo de formulario junto con sus parámetros leídos de la configuración y una instancia de formulario con sus valores que se crea al recibir una petición. La clase Parameter delega la creación de instancias de parámetros a la subclase correspondiente (patrón Factory Method).

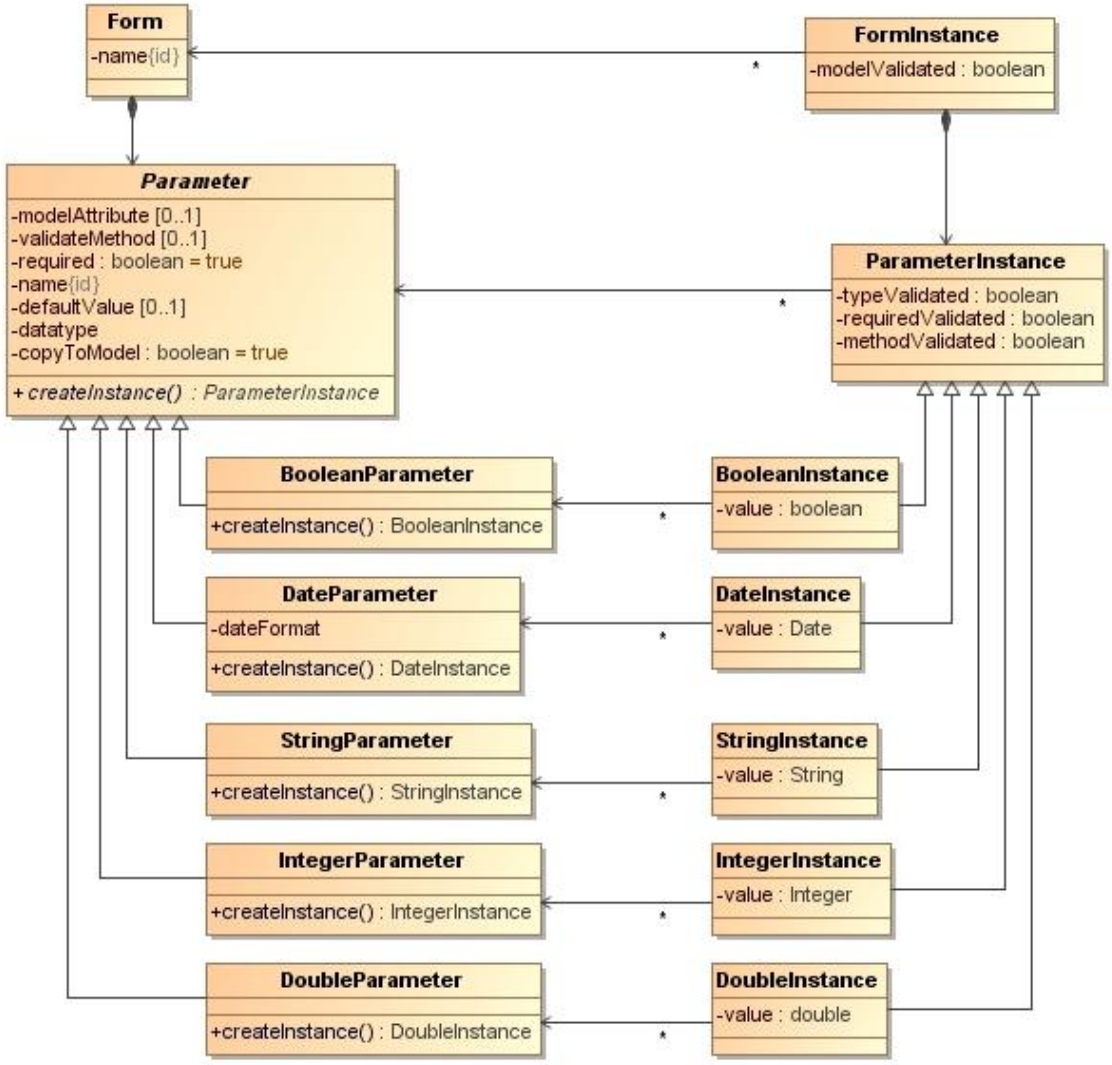



Ilustración 3:9 Diagrama de clases: Formularios, parámetros y sus instancias

#### 3.4.1.2.6 Ejecución del modelo

La ejecución del modelo se realiza ya con los datos cargados y validados. Esta ejecución consiste en la invocación a un método del objeto modelo instanciado que se encarga de invocar a los procesos de negocio y devuelve una cadena de texto como resultado de la operación.

El procedimiento puede opcionalmente recibir el objeto RequestData. El ejemplo siguiente muestra dos instancias tipo de modelo que implementan las aplicaciones usuarias del framework. El tipo1 no recibe el RequestData, mientras que el tipo 2 sí lo recibe. En el primero se han definido 2 atributos uno de cada tipo y una función de validación, pero en el segundo se han definido 3 atributos de otros tantos tipos.

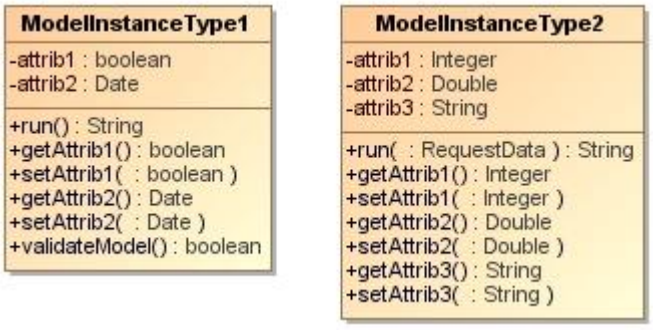



Ilustración 3:10 Diagrama de clases: Clases modelo de ejemplo

#### 3.4.1.2.7 Elección de la vista a cargar

Se utiliza otro Mapper que, en función del resultado de la ejecución del modelo selecciona la vista que debe cargar utilizando un mapa de resultados por modelo y vistas. También se tiene en cuenta el posible error en las fases de autorización y de carga/validación de parámetros, en cuyo caso puede haberse definido una vista a nivel de modelo o a nivel global. La clase ViewMapper implementada en el framework puede ser sustituida fácilmente por otra implementación diferente, que debe ser referenciada en la configuración.

#### 3.4.1.2.8 Carga y ejecución de la vista

La vista es un archivo JSP aportado por la aplicación usuaria u otro recurso: una redirección a otro modelo, una página HTML estática… Puede acceder a una librería de etiquetas para acceder al modelo, objetos del request, sesión y a procedimientos de internacionalización.

#### 3.4.1.2.9 Internacionalización

La internacionalización consiste en dar soporte multilenguaje a textos. El acceso a las funciones de internacionalización se realiza a través del objeto RequestData. Este objeto proporciona un API para fijar un idioma para la sesión, obtener el idioma actual, la lista de idiomas disponibles y obtener textos en el idioma actual. En caso de que no se haya fijado un idioma de forma expresa, se utilizará el idioma indicado por el navegador (si estuviera soportado) o en último caso el idioma por defecto definido en el framework.

El acceso a estas funciones de internacionalización también se puede realizar desde las JSP a través de una librería de etiquetas.

### 3.4.2 Diseño

En este apartado se presentan los distintos componentes que se han creado para implementar las funcionalidades descritas, pasando después a describir cómo interviene cada uno durante las distintas fases del ciclo de vida de las peticiones.

#### 3.4.2.1 Arquitectura y componentes

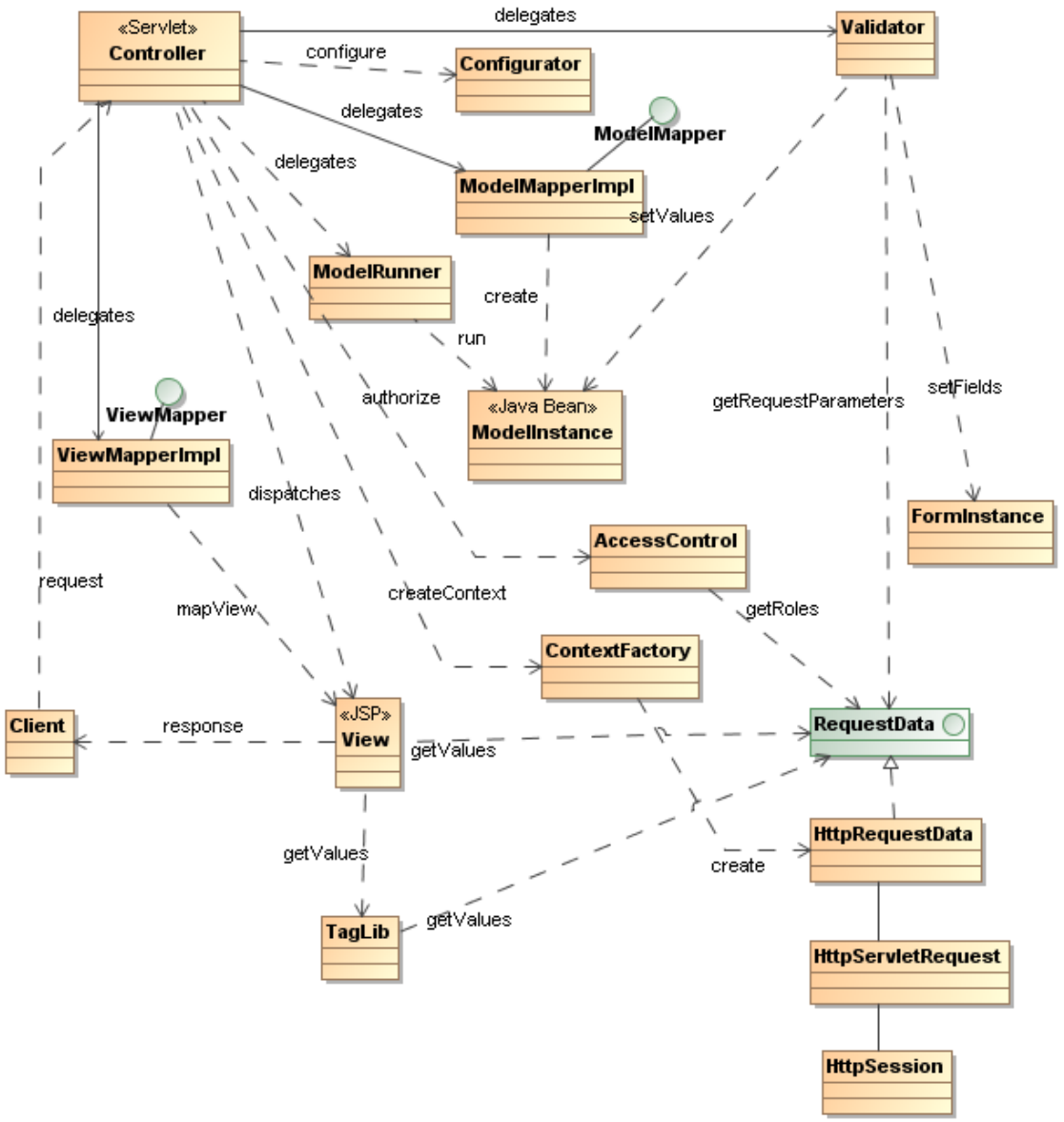



Ilustración 3:11 Diagrama de clases de los componentes del framework

**Controller** (patrones Front Controller/Application Controller, Mediator). Recibe las peticiones de los clientes y se encarga de dirigir el curso del request que va pasando por las distintas fases utilizando otros componentes (creación de los datos de contexto, mapeo del modelo, autorización, validación, ejecución del modelo, mapeo de la vista y ejecución de la vista). En la fase de inicialización carga la configuración.

**Configurator** (patrón Singleton). Se encarga de leer y almacenar la configuración del framework que pone a disposición del resto de componentes. Utiliza una sola instancia para leer, guardar y servir la configuración a los demás componentes.

**ContextFactory** (patrón Context). Su misión es crear el objeto con el contexto y asignarle la sesión. La implementación aportada solo contempla el protocolo HTTP, pero de esta manera se puede extender más adelante a otras tecnologías con un esfuerzo menor.

**RequestData** (patrones Context y Façade). Oculta a los usuarios del framework los detalles específicos para guardar objetos en el request y en la sesión, actuando además como fachada para ofrecer a las aplicaciones usuarias un API unificado. Permite obtener y guardar valores en el request y la sesión, facilita el acceso a otros datos que el framework guarda en estos ámbitos de vida (roles, modelo, formulario) y también ofrece las funciones de internacionalización. La información guardada en el request existe hasta que finaliza la petición del cliente, mientras que lo que se guarda en la sesión tiene vida durante toda la sesión del cliente con la aplicación, que puede comprender varias peticiones.

**HttpRequestData** (Patrones Context y Façade). Implementación del interface RequestData para el protocolo HTTP.

**ModelMapper** (patrón Mapper). Selecciona el modelo a cargar/ejecutar en función de la acción solicitada. El Mapper implementado busca el valor del parámetro MODEL del request para buscar en la configuración el Model coincidente, y obtener así la clase Model que se corresponde. Este componente podría ser sustituido fácilmente por otro Mapper que utilice otro criterio para seleccionar el modelo, por ejemplo basándose en la URL recibida.

**AccessControl** (Patrón Iterator). Da autorización al cliente a utilizar el modelo elegido. La implementación se basa en los roles asociados a cada modelo y los roles asociados a la sesión actual. Un modelo que implementa seguridad tiene asociados roles, alguno de los cuales debe tener la sesión que intenta acceder, mientras que un modelo que no implementa seguridad no tiene ningún rola asociado, por lo que es de acceso libre.

**Validator**. La validación de parámetros de entrada del formulario asociado al modelo seleccionado se realiza en las siguientes etapas:

- En primer lugar se hace una validación de conversión del parámetro al tipo de datos objetivo.
- Después se comprueban los valores requeridos.
- Se ejecuta la función de validación del parámetro individual. Estas funciones se declaran opcionalmente en los parámetros y utilizan métodos estáticos a los que se les pasa el valor. Deben retornar un valor lógico verdadero si valida y falso si no valida el valor. Como ejemplo, una función de validación de código postal puede recibir un valor entero para verificar que se trata de un código postal válido.
- Carga los parámetros necesarios a la instancia del modelo.
- Función de validación global. Cada modelo puede incluir opcionalmente una función de validación general, que no recibe parámetros y permite comprobar la consistencia global del modelo antes de ejecutar el método. Esta función retorna un valor lógico verdadero si valida y falso si no valida el conjunto de valores.

En caso de fallar cualquiera de los pasos de validación, no se ejecuta la acción del modelo y se invoca a una vista especificada opcionalmente para esa acción del modelo o si no se ha definido se invoca a una definida a nivel global.

**FormInstance**. Representa el conjunto de parámetros de entrada que son leídos y cargados en un formulario.

**ModelRunner**. Se encarga de invocar al modelo de aplicación, pasando si es necesario el objeto RequestData, encapsular los errores que pudiera provocar y capturar la respuesta de su ejecución.

**ModelInstance** (patrón Context – POJO). Las aplicaciones usuarias del framework implementan el modelo a través de Java Beans. De esta forma el modelo no tiene dependencias directas con el framework permitiendo la reutilización de código existente. El framework puede cargar en atributos del modelo los parámetros del request que se hayan mapeado en la configuración, mientras que el propio modelo puede cambiarlos o añadir más atributos como resultado de su ejecución. Este objeto al final queda expuesto a la vista a través del Framework. En la implementación del modelo se ha aprovechado el patrón Context Object en su implementación "Request Context POJO Strategy".

Estas clases también tienen un método que ejecuta la acción requerida y que puede recibir opcionalmente el objeto RequestData como parámetro para interactuar a través de esta fachada con el framework, los objetos de la petición y de la sesión. Al terminar, el método devuelve una cadena que representa el resultado de la acción, que el framework puede utilizar para decidir la vista a mostrar al cliente.

**ViewMapper** (patrón Mapper). Selecciona la vista a cargar/ejecutar en función del resultado de la ejecución del modelo, o del error de validación/seguridad. El componente puede ser sustituido por otra implementación que implemente el interface adecuado.

**View**. Representa a la vista que se ejecuta para retornar el resultado al cliente. Es un archivo JSP u otro recurso web implementado por la aplicación. Puede hacer uso de la librería de etiquetas implementada para acceder al modelo ejecutado y a otros objetos del RequestData (request, sesión, roles…), pero también puede acceder directamente a través del RequestData almacenado en el request.

**Librería de etiquetas (taglib)**. Implementa las funciones del framework para acceder desde las JSP a los datos almacenados en el RequestData sin necesidad de incluir código Java, sino solo con etiquetas. También permite utilizar las funciones de internacionalización.

### 3.4.2.2 Ciclo de vida de una petición

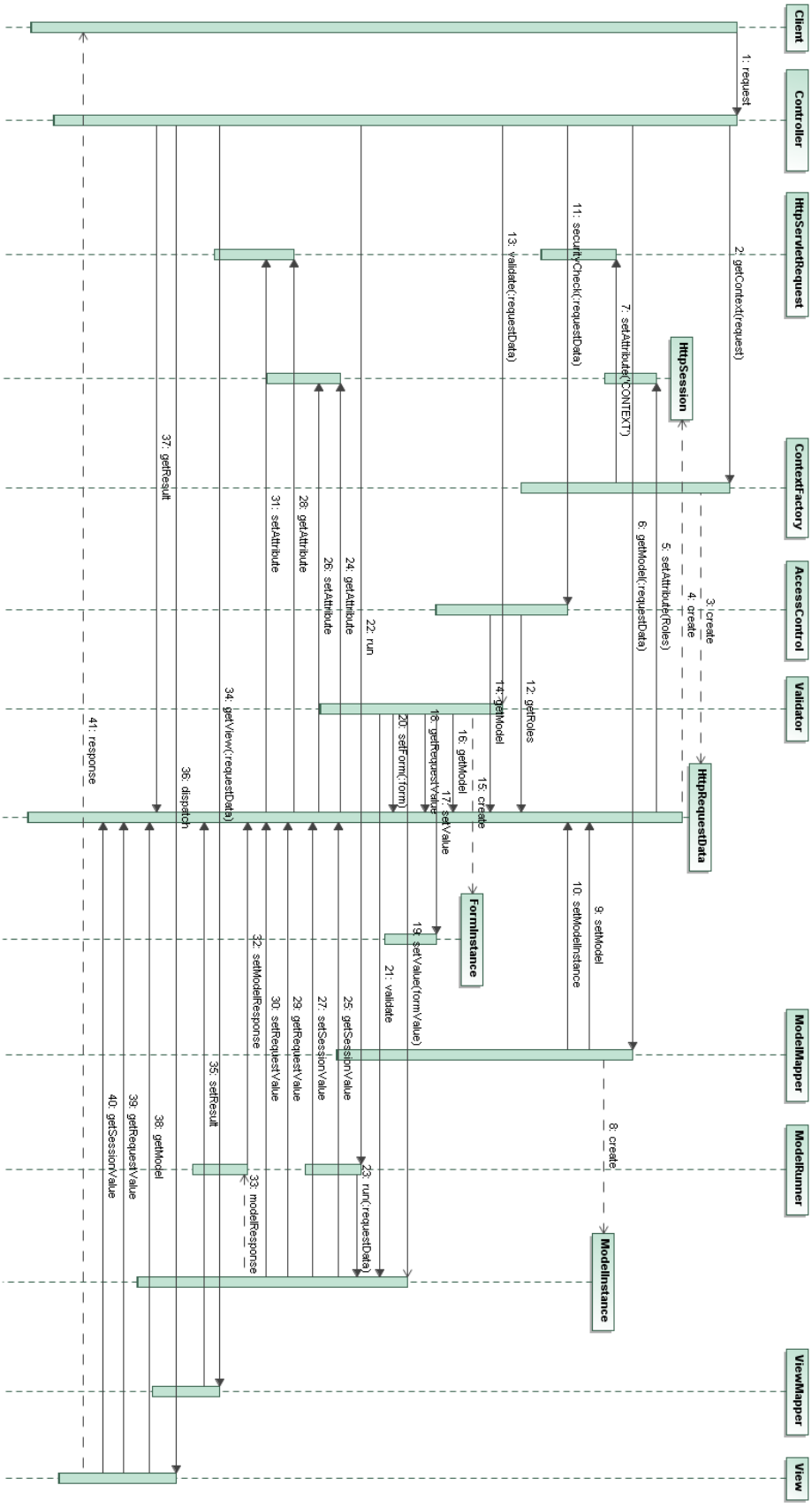



Ilustración 3:12 Diagrama de secuencia. Ciclo de vida de una petición

La petición del cliente llega al Controller y éste la procesa en siete pasos:

**Paso 1: Creación de RequestData y Sesión**. El Controller llama al ContextFactory que crea el entorno, instanciando un objeto de tipo RequestData (en la implementación del framework el objeto es HttpRequestData) al que pasa el objeto Request con la información de la petición. Si no tiene sesión, el RequestData se encarga de crearla y fijarle los roles por defecto. El objeto RequestData es guardado como un atributo más del Request, para que esté accesible durante toda la petición.

**Paso 2: Elección del modelo a cargar**. El Controller delega al ModelMapper la selección del modelo a cargar y ejecutar. Como resultado, carga en el RequestData una instancia de la clase del modelo seleccionado. Carga también la definición del tipo de modelo (instancia de Model) para su posterior uso (validación y autorización).

**Paso 3: Comprobación de seguridad**. Después le pide al AccessControl que compruebe si la sesión tiene permiso de acceso al modelo seleccionado. En caso de no autorizar, el Controller no continúa con la validación y ejecución del modelo y salta directamente a la elección de vista a cargar, avisando al ViewMapper el error de seguridad.

**Paso 4: Validación y carga de parámetros de entrada**. Se llama al Validator para que cargue los parámetros del request y haga las validaciones oportunas, de tipo de datos, campo obligatorio, validación personalizada de parámetro y/o validación en conjunto. En caso de fallar alguna validación, el Controller no continúa con la ejecución del modelo y salta directamente a la elección de vista a cargar, avisando al ViewMapper el error de validación.

**Paso 5: Ejecución del modelo**. El Controller delega la ejecución del modelo de aplicación al ModelRunner, que invoca al método correspondiente y opcionalmente le pasa por parámetro el RequestData. El valor retornado como resultado de la ejecución indica el resultado de la operación y queda guardado en el RequestData.

**Paso 6. Elegir vista a cargar**. En función del resultado de las fases anteriores, especialmente de la autorización, validación y resultado de la ejecución del modelo, el ViewMapper elige la vista que se ha de cargar.

**Paso 7. Ejecutar vista**. El Controller invoca a la vista seleccionada que devuelve el resultado al usuario.

### 3.4.3 Implementación

En lo que respecta al nivel de implementación, el framework se ha dividido en los siguientes elementos, agrupados en la librería mtp.jar:

- Código del framework. Conjunto de clases Java con el código del framework.
- MTPTags.tld. Librería de etiquetas para las páginas JSP.
- Fichero de definición del schema de validación del fichero de configuración.

Las versiones requeridas por el framework se mencionan en el anexo de “Configuración, referencia de uso y extensibilidad del framework”

La librería JAR presenta los elementos divididos de la forma siguiente:

- Paquete edu.uoc.pfc.j2ee.jalonsod.mtp: Contiene el Servlet que actúa como controlador y gestiona el ciclo de vida, la excepción de aplicación MTPException y una clase auxiliar.
- Paquete edu.uoc.pfc.j2ee.jalonsod.mtp.config: Contiene todos los elementos de la configuración y conserva el estado de la misma leído del fichero XML. Asimismo contiene el *schema* que se utilizará para validar los ficheros de configuración de las aplicaciones.
- Paquete edu.uoc.pfc.j2ee.jalonsod.mtp.helper: Contiene las clases auxiliares en las que se apoya el controlador para gobernar las diferentes fases del ciclo de vida de la petición.
- Paquete edu.uoc.pfc.j2ee.jalonsod.mtp.request: Contiene el API RequestData representado por un interfaz, su implementación para el protocolo HTTP y las clases que almacenarán información adicional del request (los parámetros).
- Paquete edu.uoc.pfc.j2ee.jalonsod.mtp.request.taglib: Contiene la implementación de las funciones de la librería de etiquetas para acceder al API del framework, a través del RequestData.

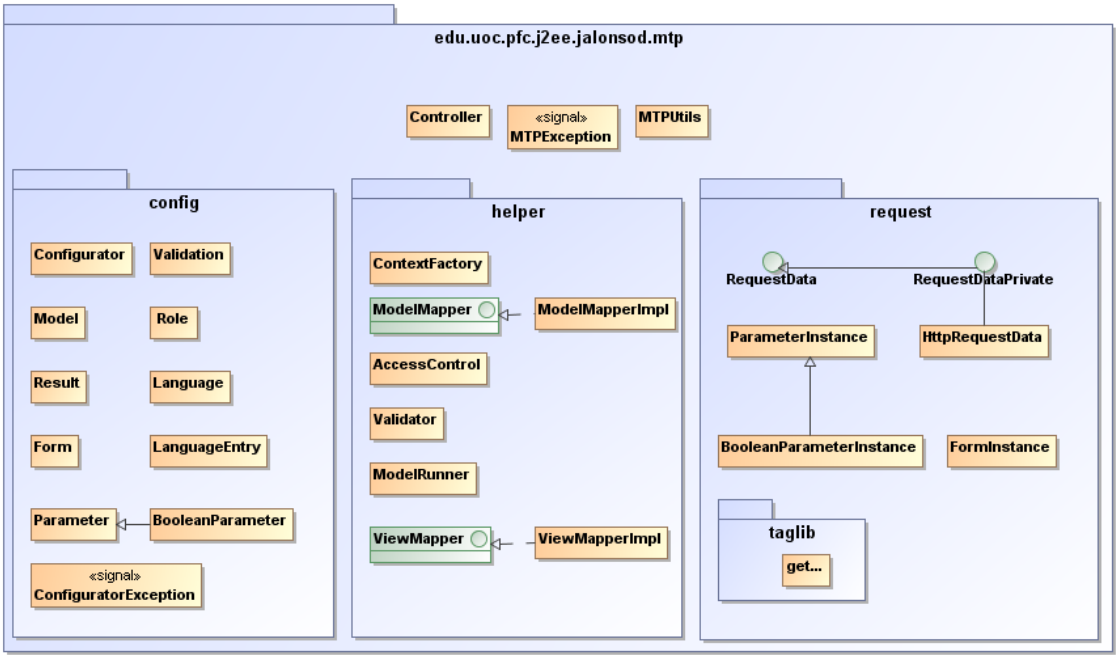



Ilustración 3:13 Diagrama de paquetes de "MTP"

### 3.4.4 Posibles mejoras del framework

Durante la elaboración del framework y de la aplicación de ejemplo, se han identificado algunas mejoras que quedan fuera del alcance del proyecto, pero que podrían servir para una futura ampliación del mismo.

- Configuración a través de anotaciones Java.
- Integración con otras librerías de etiquetas o ampliación de la propia de MTP para incorporar componentes del interfaz de usuario, lo que permitiría realizar validaciones en cliente o incluir funcionalidad AJAX.

- Incorporación de una librería de funciones de validación predefinidas para las operaciones más frecuentes.
- Ampliación de los tipos de datos soportados para la carga de parámetros.
- Upload de ficheros.
- Soporte de regionalización: diferentes representaciones de fecha/hora, moneda, separadores decimales y de millar…
- Incorporación de forma nativa de retroceso, para recargar la última página con los datos enviados y presentando los errores en caso de fallos de validación o autorización.
- Inclusión de varias líneas de detalle en formularios (carga y validación de líneas de detalle en arrays de parámetros).
- Carga de formularios de entrada directamente en objetos tipo Java Beans.
- Carga de atributos de la sesión en propiedades del objeto Model.

## 3.5 Aplicación de ejemplo "AMPA"

Esta parte del proyecto consiste en realizar una aplicación sencilla para probar y demostrar el uso del framework, así como para definir una arquitectura de componentes en tres capas, extrapolable a otras aplicaciones web con J2EE. La implementación de toda la funcionalidad de la aplicación propuesta excede los requisitos del proyecto, por lo que se va a implementar una parte, que sirva de utilidad para los dos objetivos indicados.

La gestión de una asociación de madres y padres de alumnos (AMPA), como muchos otros tipos de asociaciones, implica una carga administrativa en la interacción con los asociados. El objetivo de la aplicación es ofrecer un sistema de autoservicio para que los socios puedan gestionar su membresía y la participación en los eventos que organiza la asociación.

### 3.5.1 Requisitos

**Conceptos**

- **Fichero de socios**. Contiene la información personal de los socios. Un socio es una familia, compuesta de uno o dos padres/tutores y uno o varios alumnos matriculados. Un socio se identifica con la dirección de email y una password.
- **Actividades**. La asociación realiza actividades periódicas (servicio de desayuno, actividad extraescolar…) y otras puntuales (excursiones...).
    - Las actividades tienen: un nombre, una descripción larga, una fecha o periodo de realización de la actividad (inicio, fin) o varios periodos (para las actividades periódicas) y pueden tener un coste por periodo. La inscripción en algunas actividades puede realizarse automáticamente y en cambio para otras se requiere la validación de un administrador de la asociación.
- **Usuarios**. Se han identificado tres tipos de usuarios de la aplicación:
    - Sin identificar. Solo pueden solicitar el alta.
    - Socios: Manejan sus datos personales como socios y la inscripción en actividades.
    - Administradores. Tienen acceso a las funciones administrativas de la aplicación.

**Funciones a implementar**

- Fichero de socios:
    - **Solicitud de alta** en la asociación. El socio va a solicitar, mediante la cumplimentación de un formulario, ser miembro de la asociación. Un administrador de la asociación va a aceptar o denegar la solicitud. El socio utiliza una dirección de email y una contraseña para identificarse. Cada socio puede tener varios hijos matriculados.
    - **Validación de socios**. Un administrador de la asociación valida o anula las solicitudes de alta.
    - **Consulta/actualización de datos**. El socio puede consultar y actualizar sus datos personales.

  - **Bajas**. El socio puede darse de baja por deseo expreso, pero el administrador de la asociación también puede realizar la baja por otros motivos (impago o devolución de las cuotas, no renovación).
  - **Consulta de socios**. El administrador puede consultar la relación de socios y obtener los detalles de cada uno de ellos.
  - **Fin de curso**. Al finalizar el curso, el administrador provoca la invalidación de todos los socios, que quedan en suspenso y deben confirmar la renovación.
  - **Renovación**. Cuando finaliza el curso y se invalidan los datos de socio, el socio entra a renovar los datos para el siguiente curso. Una vez renovado, el socio continúa activo, sin necesidad de que el administrador le vuelva a validar.
- **Actividades**:
  - **Inscripción**. Un socio inscribe a uno de los hijos matriculados en una actividad. Si es periódica (tiene varios periodos), debe indicar en qué periodo desea comenzar. Las actividades también pueden ser para padres, por lo que se pueden inscribir también ellos.
  - **Baja**. Un socio decide dar de baja a uno de los hijos matriculados en una actividad. En caso de ser una actividad periódica, debe elegir el periodo en el que se debe producir la baja.

**Funciones no implementadas**

Debido al alcance limitado de la aplicación, como prueba que es, no se han implementado algunas funcionalidades, que podrían ser implementadas como mejora del sistema:

  - Cambio y reseteo de contraseña de socio.
  - Confirmación de inscripción en actividades que requieran validación.
  - Envío de emails a los socios: Todos los eventos que implican cambios en el fichero de socios y en las actividades conllevan el envío de un email de confirmación.
  - Mantenimiento de inscripciones y periodos (se proporciona una carga inicial).
  - Listados de inscritos en actividades (listado en pantalla y CSV para manipulación posterior).

### 3.5.2 Navegación

La página inicial de la aplicación permite solicitar el alta como socio o identificarse como usuario. Una vez identificado como usuario aparece el menú de opciones, según sea un administrador o un socio. Desde ese punto se accede a las diferentes funciones implementadas para cada rol.

El interfaz de usuario está disponible en español e inglés. Para probar los dos idiomas, se puede cambiar el idioma del navegador entre ambos.

Los siguientes diagramas muestran el flujo de la aplicación.

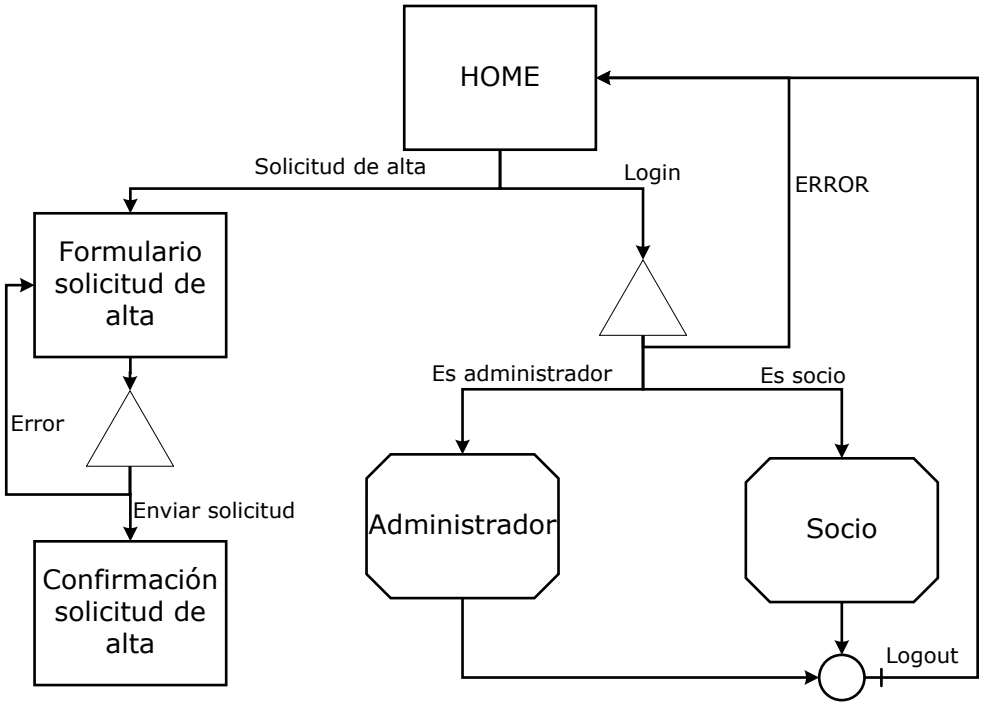



Ilustración 3:14 Aplicación AMPA. Flujo de navegación principal.

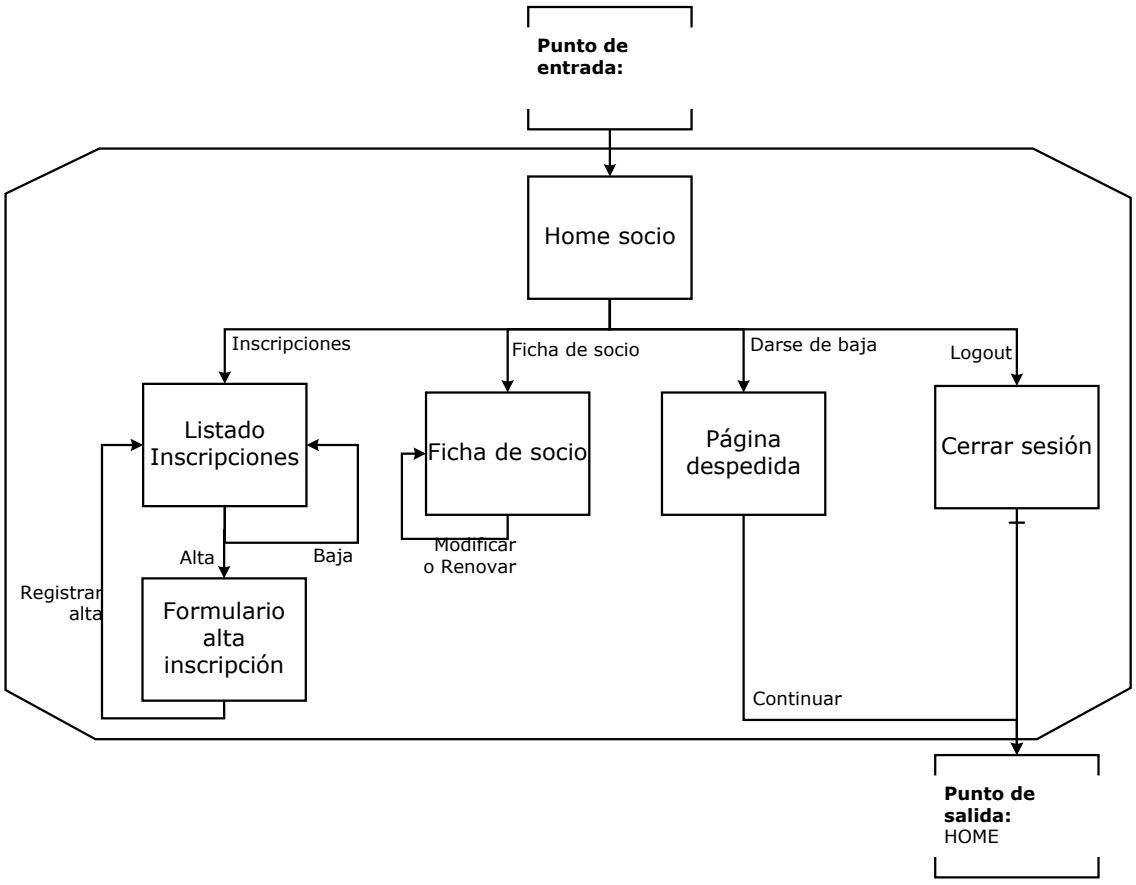



Ilustración 3:15 Aplicación AMPA. Flujo de navegación del menú de socios.

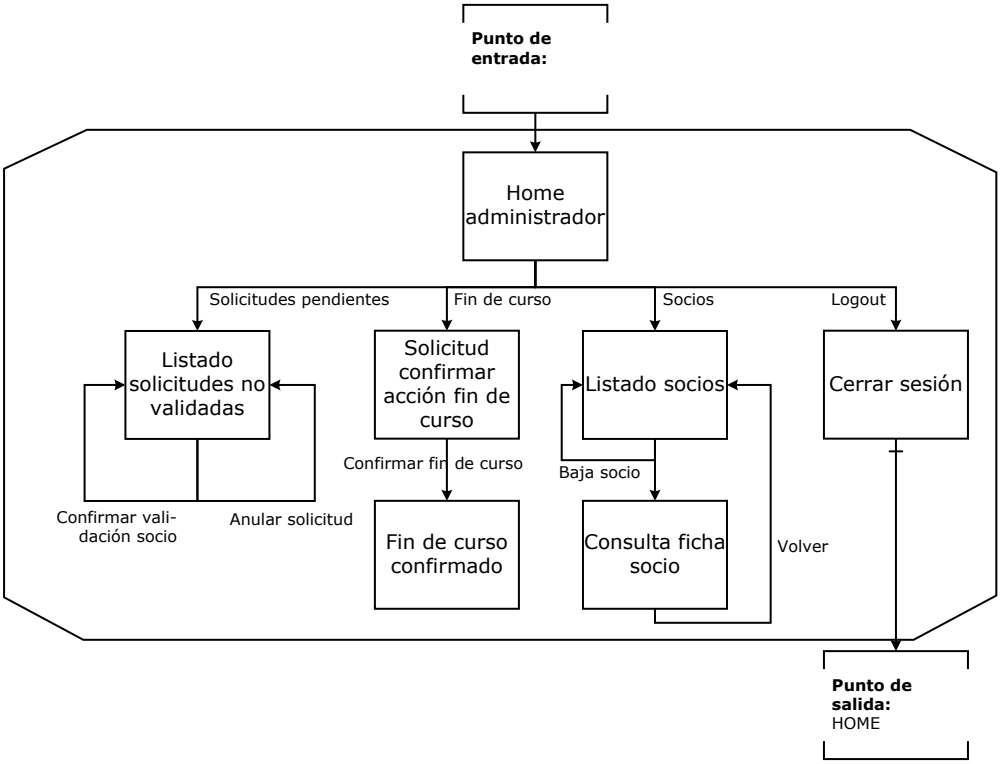



Ilustración 3:16 Aplicación AMPA. Flujo de navegación del menú de administradores.

### 3.5.2.1 Modelos (acciones)

| MODEL | Roles | Descripción |
|---|---|---|
| HOME | - | Carga la página home según el tipo de usuario:<br>- No identificado: carga la página de login.<br>- Socio: Home de socio (menú).<br>- Administrador: Home de administrador (menú). |
| SOLICITUDALTAFORM | - | Redirige a la página de solicitud de alta de formulario. |
| SOLICITUDALTACONFIRM | - | Recibe el formulario de solicitud de alta. Si es correcto, va a la página de confirmación del alta, si no, vuelve al formulario mostrando el error. |
| LOGIN | - | Valida al usuario y carga la página home (menú), de usuario o administrador según el usuario proporcionado. |
| LOGOUT | SOCIO<br>ADMINISTRADOR | Cierra la sesión actual y redirige a la página de login. |
| INSCRIPCIONESLIST | SOCIO | Redirige a la página principal con el listado de inscripciones del usuario. También se muestra una lista de inscripciones posibles a las que se puede dar de alta. |
| INSCRIPCIONALTAFORM | SOCIO | Redirige a la página de solicitud de alta en una inscripción seleccionada previamente. |
| INSCRIPCIONALTACONFIRM | SOCIO | Registra la solicitud de alta de los miembros del socio en una inscripción. |
| INSCRIPCIONBAJA | SOCIO | Da de baja a los miembros del socio en una actividad inscrita previamente. |
| FICHASOCIOFORM | SOCIO | Redirige a la página donde se carga la ficha de socio para modificarlo o renovar como socio. |
| FICHASOCIOCONFIRM | SOCIO | Actualiza los datos de socio. También se utiliza para renovar como socio, si estaba invalidado. |
| SOCIOBAJA | SOCIO | Realiza la baja del socio. Después hace logout y vuelve a la página principal. |
| SOLICITUDALTAPENDIENTES | ADMINISTRADOR | Muestra una lista de las solicitudes pendientes de confirmar. Junto a cada solicitud aparecen dos opciones para validarla o cancelarla. |
| SOLICITUDALTAVALIDAR | ADMINISTRADOR | Valida una solicitud de alta y registra el nuevo socio. Elimina la solicitud anterior. |

| MODEL | Roles | Descripción |
|---|---|---|
| SOLICITUDALTAANULAR | ADMINISTRADOR | Anula una solicitud de alta, eliminándola. |
| FINCURSO | ADMINISTRADOR | Marca todos los socios como inválidos, para que renueven o cancelen su membresía. |
| SOCIOSLISTADO | ADMINISTRADOR | Muestra una lista de socios. Por cada socio aparece una marca para darlo de baja, y un link para consultar la ficha. |
| SOCIOVERFICHA | ADMINISTRADOR | Muestra una página con todos los datos del socio, sus padres y alumnos. |
| SOCIOBAJAADMIN | ADMINIISTRADOR | Da de baja al socio y vuelve al listado de socios. |

Ilustración 3:17 Modelos (acciones) de la aplicación AMPA

#### 3.5.2.2 *Diagrama conceptual de clases*

El diagrama muestra las clases de negocio de la aplicación. Se pueden agrupar en:

- **Solicitudes**. Contiene la información de solicitudes de socios.
- **Socios**. Contiene la ficha de los socios.
- **Usuarios**. Contiene la información de acceso a la aplicación (socios y administradores).
- **Actividades**. Contiene la información de las actividades e inscripciones en las mismas.

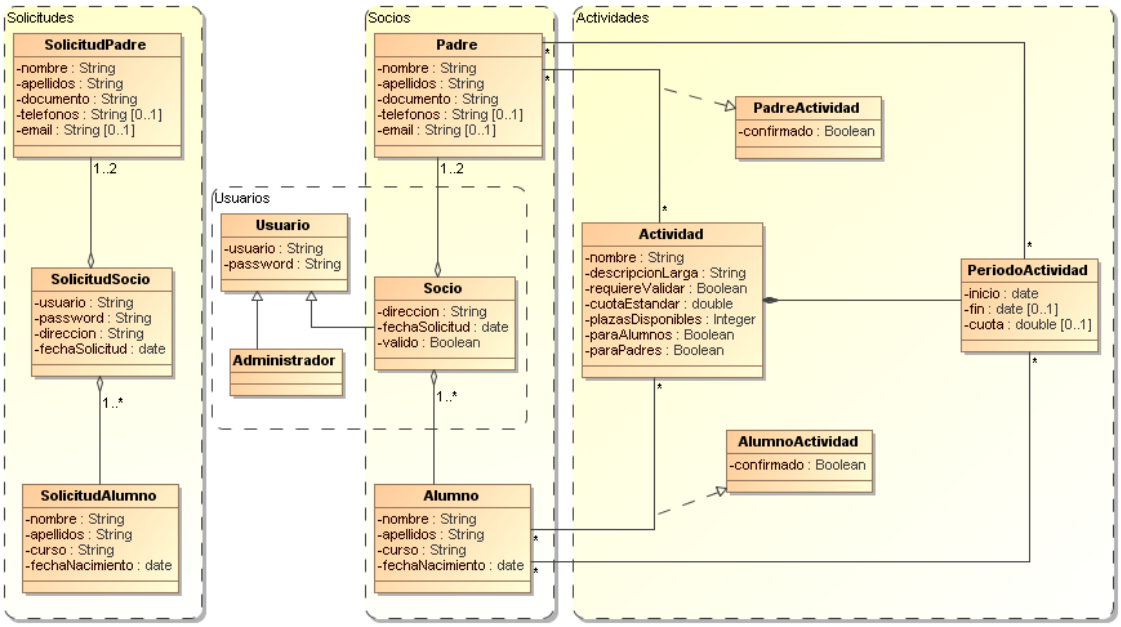



Ilustración 3:18 Aplicación AMPA. Modelo conceptual de clases.

### 3.5.3 Diseño

#### 3.5.3.1 *Arquitectura*

La aplicación está dividida en las capas típicas de las aplicaciones Web J2EE:

- La **capa de presentación**, está gobernada por el framework MTP objetivo principal del proyecto. Se apoya en los siguientes elementos:
  - **Model**. Son objetos de aplicación que manejan, dentro de las posibilidades configuradas en MTP el flujo de navegación, ejecutan la acción solicitada y proporcionan a las vistas los datos necesarios para la presentación. Estos objetos tienen la forma requerida por el framework para actuar como tales.

  - **JSP**. Son las vistas que se presentan al usuario. Se apoyan en la librería de tags de MTP para obtener información del modelo y del estado (sesión y request) y en las librerías de etiquetas de JSTL como apoyo. Se han ubicado en un subdirectorio dentro de WEB-INF para que no estén publicadas directamente, sino que deben ser accedidas desde el propio servlet controlador del framework MTP.
- La **capa de negocio** está formada por tres tipos de elementos fundamentalmente:
  - **ApplicationService**. Es un conjunto de clases que aplican este patrón de diseño. Su misión es servir como fachada para la capa de presentación, haciendo de intermediario con los objetos de negocio. También gobiernan la transacción de la base de datos y controlan las posibles excepciones que se provoquen, devolviendo un resultado apropiado en ese caso.
  - **Business Objects**. Los objetos de negocio contienen la lógica propiamente dicha de la aplicación. Contienen algoritmos y ordenan las operaciones a realizar al motor de persistencia. No incluyen el nivel de control de la transacción, que como se ha explicado, pertenece a los objetos ApplicationService.
  - **Beans** (Transfer Object). Estos objetos realmente están a caballo entre las capas de integración, negocio e integración. Son los que contienen la información de negocio propiamente dicha. Tienen formato de Java Beans para obtener y fijar los valores, actuando como Transfer Objects para pasar la información de negocio de una capa a otra. También se utilizan desde el motor de persistencia Hibernate como entity class para interactuar con la base de datos, por lo que se han incluido anotaciones para definir el comportamiento y el mapeo entre los objetos Java y las tablas de la base de datos.

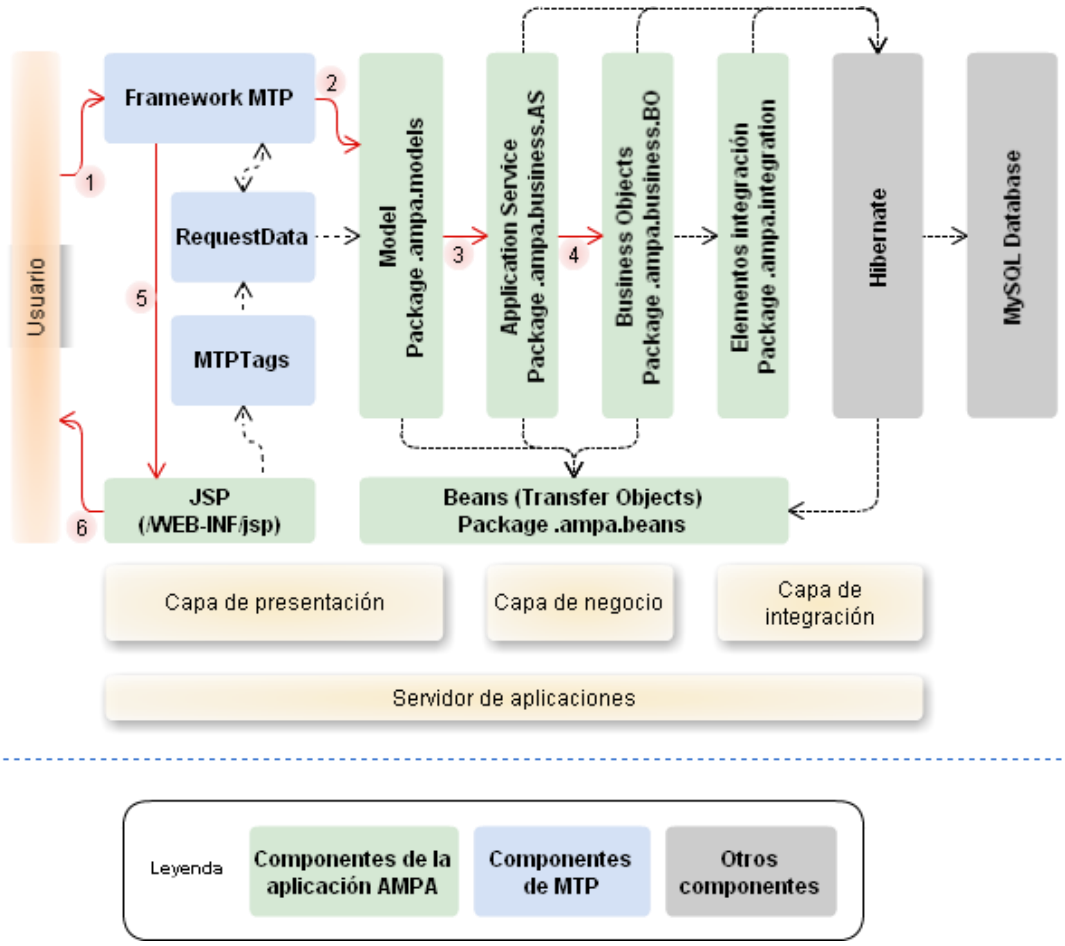



Ilustración 3:19 Aplicación "AMPA". Arquitectura de capas.

- La **capa de integración** está formada por los objetos que interactúan con sistemas externos. En la aplicación que se entrega está solamente la clase que configura el motor de persistencia y proporciona las sesiones. Como la funcionalidad de la aplicación también contempla el envío de correos electrónicos, en esta capa también se incluirían las clases que interactúen con el sistema de mensajería.

#### *3.5.3.2 Modelo de datos*

Se proporciona script de creación de los objetos de la base de datos y de una carga inicial. El siguiente diagrama muestra las relaciones entre las diferentes tablas.

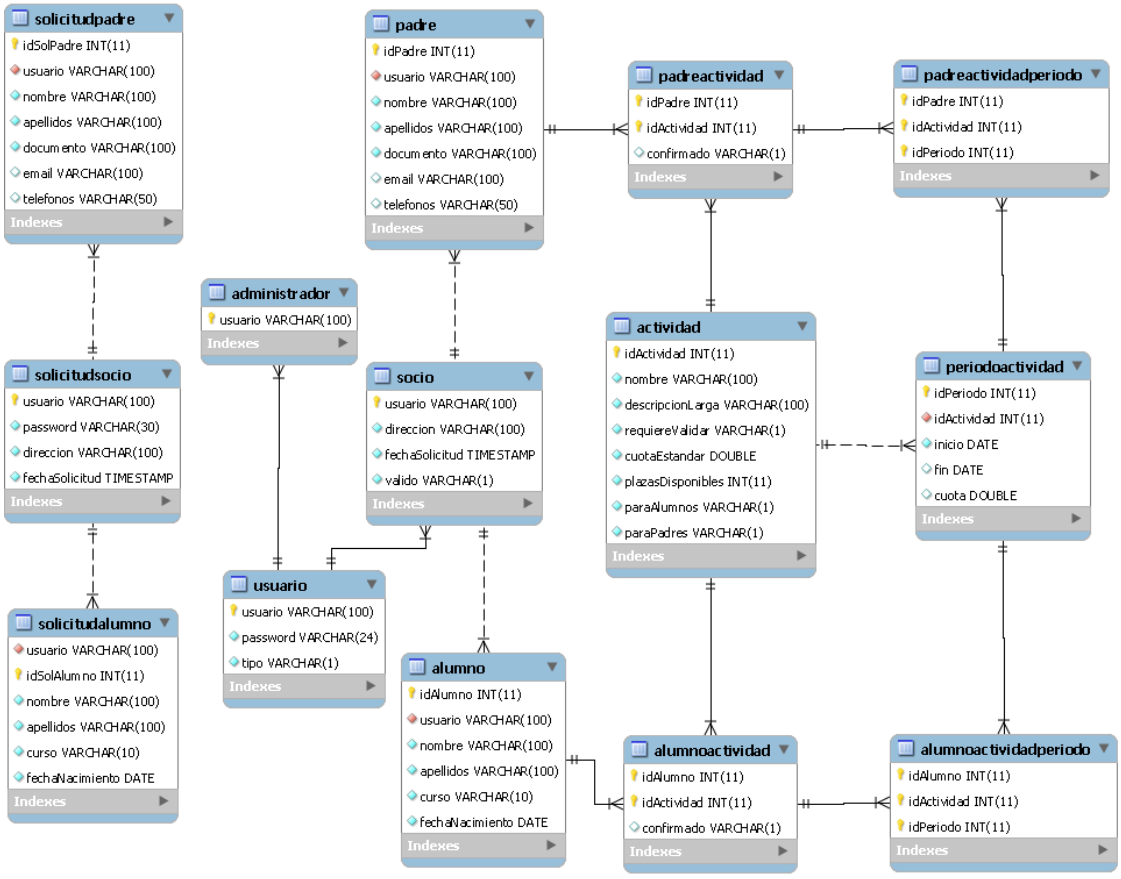



Ilustración 3:20 Aplicación AMPA. Modelo de datos.

### 3.5.3.3 *Diseño del interfaz de usuario*

Para representar el interfaz de usuario, se ha escogido un diseño sencillo, basado en una hoja de estilos css para unificar la presentación.

A continuación se presentan algunas páginas representativas del interfaz.


Ilustración 3:21 Aplicación AMPA. Página inicial de acceso


Ilustración 3:22 Aplicación AMPA. Menú de socio


Ilustración 3:23 Aplicación AMPA. Actividades inscritas

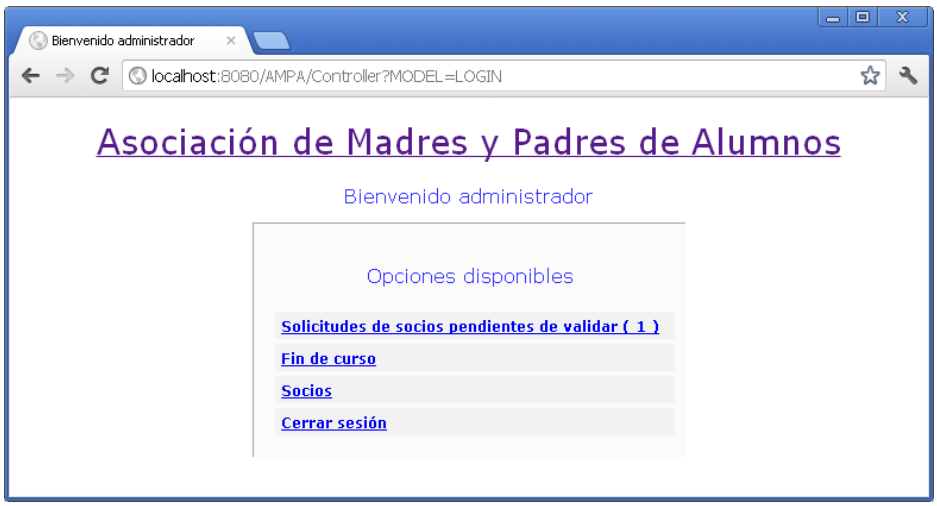



Ilustración 3:24 Aplicación AMPA. Menú del administrador


Ilustración 3:25 Aplicación AMPA. Listado de socios

# 4 Conclusiones

Durante las distintas fases del proyecto he podido comprobar la utilidad y la necesidad de una buena definición arquitectónica de las aplicaciones J2EE que se apoye en soluciones probadas. Para obtener un diseño satisfactorio, es importante el apoyo que ofrecen tanto los marcos de trabajo, en sus diferentes especializaciones (presentación, persistencia…) como las buenas prácticas que proveen los patrones de diseño.

Para la elaboración del marco de trabajo y de la aplicación ejemplo, ha sido necesario profundizar en esas dos materias:

- Los marcos de trabajo, concretamente los de la capa de presentación.
- Los patrones de diseño: el MVC, los especializados en J2EE y los patrones generales de la programación orientada a objetos.

El proyecto a su vez ha servido para aplicar materias de distintas áreas de la Ingeniería Informática, como son: la gestión de proyectos informáticos; la ingeniería del software orientado a objeto y de componentes; las bases de datos; y por supuesto la programación de aplicaciones J2EE.

En las primeras fases se ha tenido que tomar decisiones sobre el alcance del proyecto, extrayendo las características más útiles para el marco de trabajo desarrollado, pero teniendo en cuenta el tiempo disponible. Las decisiones se han basado en obtener un producto ligero, que no requiera de librerías de terceros, pero que ofrezca la funcionalidad básica para evitar que las aplicaciones tengan que realizar implementaciones propias del patrón MVC y que presente soluciones eficaces, basadas en la experiencia de otros marcos de trabajo de presentación y en la de los patrones de diseño.

Una vez obtenido un diseño del framework, durante la construcción ha habido que enfrentarse a varios problemas, como el tratamiento del fichero XML de configuración, su schema de definición o la implementación de la librería de etiquetas.

En el diseño arquitectónico de la aplicación se ha vuelto al conocimiento adquirido sobre los patrones de diseño, para definir las responsabilidades de los objetos de las distintas capas de la aplicación. Se ha recurrido al marco de trabajo realizado en el proyecto y al framework Hibernate para manejar la persistencia de datos.

Durante la implementación de la aplicación, además de aprender el manejo de Hibernate, se han descubierto pequeños errores y se han realizado mejoras en el framework desarrollado para hacerlo más útil.

El marco de trabajo junto con la aplicación de ejemplo pueden servir como guía arquitectónica para el desarrollo de nuevas aplicaciones web con J2EE.

# 5 Glosario

**API**: Application Programming Interface. Es la especificación o interfaz que provee un software para comunicarse con otros programas.

**Capas J2EE**: Las aplicaciones J2EE se suelen dividir en las capas de presentación, negocio e integración, siendo responsable, cada una de ellas de una parte de las responsabilidades.

**Framework**: En el ámbito del proyecto, es un software que aporta una arquitectura estandarizada para ser aprovechado por otras aplicaciones, resolviendo una serie de problemáticas, como puede ser la implementación del patrón MVC.

**Entity class**: Clase que representa una entidad de datos de negocio de la aplicación, y que puede tener su imagen paralela en el modelo de datos.

**Hibernate**: Framework de persistencia utilizado en la aplicación de ejemplo.

**J2EE/JEE**: Especificación Java Enterprise Edition. Es una especificación para plataformas de ejecución de aplicaciones Java en arquitectura multicapa que pone a disposición de las aplicaciones, además del API de Java, una serie de APIs como el Servlet para aplicaciones web.

**JAR**: Java Archive. Es un contenedor de recursos de Java para distribuir varios recursos, como clases Java u otros ficheros. El framework MTP se distribuye en el archivo mtp.jar.

**Java Bean**: Son clases Java que se caracterizan por tener un constructor sin parámetros, tienen métodos con nomenclatura estándar get/set para acceder/grabar los atributos y son serializables.

**JDK**: Java Development Kit. Se refiere al software y la especificación Java, que incluye el cliente (JRE – Java Runtime Environment) y otras utilidades como el compilador (javac).

**MTP**: Marco de Trabajo de Presentación. Es el nombre que se ha dado al framework construido en este proyecto.

**MVC**: Model-View-Controller. Patrón para la capa de presentación que pretende separar la lógica de negocio de la presentación del usuario (ver capítulo 3.3.1 - Patrón Modelo-Vista-Controlador).

**Patrón**: Solución probada y reconocida para una problemática concreta de la ingeniería del software u otras áreas de ingeniería.

**POJO**: Plain Old Java Objects. Se refiere a clases Java sencillas, que no requieren la importación de interfaces o superclases.

**Request** (petición): Es cada una de las solicitudes que se envían desde el navegador del usuario al servidor de aplicaciones para que realice una acción.

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

**Taglib**: Librería de etiquetas. Se utilizan para sustituir código Java por etiquetas similares al HTML en las páginas JSP.

**WAR**: Web Application Archive. Es un contenedor de una aplicación web de J2EE, contiene todos los recursos, como clases Java, librerías, ficheros de configuración, un descriptor de despliegue… necesarios para su despliegue en un servidor de aplicaciones. La aplicación de ejemplo AMPA se distribuye en el archivo AMPA.war.

# 6 Bibliografía

## 6.1 Referencias

**Oracle**. "Java EE Compatibility" [artículo en línea]. [Fecha de consulta: 10 de marzo de 2012]. http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html

**Wikipedia**. "Comparision of web application frameworks" [artículo en línea]. [Fecha de consulta: 10 de marzo de 2012]. http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks#Java_2

**Varios autores**. (2007, 17 de mayo). "JSF Tutorials" [artículo en línea]. [Fecha de consulta: 11 de marzo de 2012]. http://www.roseindia.net/jsf/

**Oracle**. "Chapter 10. Java Server Faces Technology" en *"The Java EE 5 Tutorial"* [libro en línea]. [Fecha de consulta: 12 de marzo de 2012]. http://docs.oracle.com/javaee/5/tutorial/doc/bnaph.html

Struts² [web producto]. [Fecha de consulta: 10 de marzo de 2012]. http://struts.apache.org/2.x/

Spring MVC [web producto]. [Fecha de consulta: 11 de marzo de 2012]. http://www.springsource.org/

**Burns, Ed**; **Schalk, Chris** (2009). *JavaServer Faces 2.0. The Complete Reference*. New York: McGraw-Hill.

**Geary, David; Horstmann, Cay S.** (2010). *Core JavaServer Faces, Third Edition*. Upper Saddle River: Prentice Hall.

**Oracle**. *The Java EE 5 Tutorial* (2010). "JavaServer Faces Technology". En: *The Java EE 5 Tutorial*. [Fecha de consulta: 20 de marzo de 2012]. http://docs.oracle.com/javaee/5/tutorial/doc/bnaph.html.

**Varios autores** (2012). "JavaServer Faces". *Wikipedia* [Fecha de consulta: 20 de marzo de 2012]. http://en.wikipedia.org/wiki/JavaServer_Faces.

**Brown, Don; Davis, Chad Michael; Stanlick, Scott** (2008). *Struts 2 in Action*. Greenwich: Manning Publications.

**Apache**. *Struts 2 Documentation*. [Fecha de consulta: 3 de abril de 2012]. http://struts.apache.org/2.x/docs/home.html.

**Alur, Deepak; Crupi, John; Malks, Dan** (2003). *Core J2EE Patterns: Best Practices and Design Strategies, Second Edition*. Upper Saddle River: Prentice Hall.

**Varios autores**. *Spring Framework Reference Manual* (2011). [Fecha de consulta: 4 de abril de 2012]. http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/.

**Walls, Craig** (2011). "Chapter 7. Building web applications with Spring MVC". En: *Spring in Action, Third Edition*. Greenwich: Manning Publications.

**Mak, Gary; Long, Josh; Rubio, Daniel** (2010). *Spring Recipes*. New York: Apress.

**Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John** (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley Professional.

**Oracle**. *Java SE 6 API*. http://docs.oracle.com/javase/6/docs/api/

**Oracle**. *Java EE 5 SDK*. http://docs.oracle.com/javaee/5/api/

## 6.2 Herramientas utilizadas

### 6.2.1 Herramientas de desarrollo

- Java JDK .6.0_31.
  http://www.oracle.com/technetwork/java/javase/downloads/index.html
- IDE Desarrollo Java/J2ee. Eclipse. Distribución JEE Indigo SR2.
  http://www.eclipse.org/downloads/
- Servidores J2EE:
  - Tomcat 7.0.27
    http://tomcat.apache.org/download-70.cgi
  - JBoss AS 7.1.0
    http://www.jboss.org/jbossas/downloads/
  - Apache Geronimo 2.2.1 (with Tomcat 6).
    http://geronimo.apache.org/downloads.html
- Plugins Servidores J2EE para Eclipse
  - Plugin Tomcat para Eclipse (incorporado en la distribución de Eclipse).
  - Geronimo Eclipse Plugin
    http://geronimo.apache.org/development-tools.html
  - JBoss Tools 3.3.
    http://www.jboss.org/tools/download.html
- Servidor de datos: MySQL 5.5.21.0
  http://dev.mysql.com/downloads/installer/
- Framework de persistencia: Hibernate 4.1.1
  http://sourceforge.net/projects/hibernate/files/hibernate4/
- Diseño de páginas web: Kompozer 0.7.10
  http://www.kompozer.net/download.php
- Librerías de etiquetas para JSP: Jakarta taglibs-standard
  http://archive.apache.org/dist/jakarta/taglibs/standard/binaries/jakarta-taglibs-standard-1.1.2.zip
- Iconos aplicación AMPA: FatCow web hosting.
  http://www.fatcow.com/free-icons

### 6.2.2 Herramientas para la documentación

- Documento de memoria: Microsoft Word.
- Presentación: Microsoft Power Point.
- Planificación: Microsoft Project.
- Diagramas UML: MagicDraw UML.
- Diagramas de navegación:
  - Microsoft Visio.
  - Stencil file for Visio 2000 (diagramas de navegación).
    http://www.jjg.net/ia/visvocab/
- Otros diagramas: Cacoo (herramienta online).
  www.cacoo.com

# 7 Anexos

## 7.1 Entorno de desarrollo

Para el desarrollo del framework (MTP) y de la aplicación ejemplo (AMPA), se utilizan los siguientes productos, a los que se acompaña una pequeña explicación de la instalación necesaria:

- **Java SDK** (MTP y AMPA):
  - Descargar e instalar por defecto.
- **IDE Eclipse** (MTP y AMPA):
  - Descargar y descomprimir en C:\
    La instalación queda en C:\eclipse
  - Cambiar la máquina virtual por defecto: Abrir eclipse e ir a Window / Preferences / Java / Installed JREs. Añadir el JDK y marcarlo por defecto. Para que arranquen los servidores J2EE es necesario seleccionar por defecto el JDK, no el JRE6.
  - Para añadir un servidor J2EE al workspace, ir a File / New / Server y añadir el servidor del tipo indicado (ver instrucciones a continuación para instalar el plugin e integrar el servidor con Eclipse). Elegir la ubicación del servidor según el directorio en el que se haya instalado.
- **Servidor de aplicaciones** (MTP y AMPA). Se ha utilizado de forma primaria el servidor Tomcat, pero se ha probado la aplicación también con JBoss, Geronimo se descartó en las primeras fases del proyecto por problemas con el plugin de Eclipse:
  - Apache Tomcat (servidor J2EE web profile)
    - Server: Descargar y descomprimir en C:\
      La instalación queda en C:\apache-tomcat-7.0.27
    - Plugin para Eclipse: La distribución de Eclipse utilizada ya incorpora soporte para Tomcat.
  - JBoss (servidor J2EE full profile)
    - JBoss Server: Descargar y descomprimir en C:\
      La instalación queda en C:\jboss-as-7.1.0.Final
    - JBoss Tools. Acceder a la web de JBoss Tools, arrastrar el icono Install de la versión 3.3 al menú de Eclipse y seguir las instrucciones.
  - Apache Geronimo (servidor J2EE full profile)
    - Server: Descargar y descomprimir en C:\
      La instalación queda en C:\geronimo-tomcat6-javaee5-2.2.1
    - Geronimo Eclipse Plugin: Crear un nuevo servidor en File / New / Server. Si no aparece Apache Geronimo 2.2.1 en la lista, pulsar en "Download additional server adapters", elegir "Geronimo v2.2 Server Adapter" y seguir las instrucciones adicionales.

- **Servidor de datos** (solo AMPA).
  - MySQL. Instrucciones: Descargar Installer e instalar seleccionando el tipo de instalación "Developer Default" y después "Developer Machine". Tomar nota del puerto del servidor (por defecto 3306) y la contraseña de root.
- **Librerías de terceros** (solo AMPA).
  - Librería JDBC de MySQL.
    Instrucciones: Extraer la librería JDBC de la instalación de MySQL y copiarla en el directorio WebContent\WEB-INF\lib de la aplicación de ejemplo. La ubicación original es:
    C:\Program Files\MySQL\MySQL Connector J\mysql-connector-java-5.1.15-bin.jar
  - Librerías de Hibernate
    Instrucciones: Descargar, extraer las librerías del subdirectorio lib\required y copiarlas en el directorio WebContent\WEB-INF\lib de la aplicación de ejemplo.
  - Librerías de Jakarta taglibs-standard:
    Instrucciones: Descargar y obtener los archivos standard.jar y jstl.jar del directorio lib de la distribución y copiarlos en el directorio WebContent\WEB-INF\lib de la aplicación de ejemplo.

### 7.1.1 Carga de proyectos en Eclipse

Para importar un proyecto de los suministrados en Eclipse:

- Desde eclipse, seleccionar el menú: File / Import… / General / Existing Projects into Workspace / Select archive file.
- Seleccionar el archivo mtp_project.zip o ampa_project.zip, según se esté importando el framework o la aplicación ejemplo.
- Después de importar el proyecto de la aplicación ejemplo, es necesario incluir las librerías adicionales indicadas en el apartado anterior al directorio WebContent\WEB-INF\lib del proyecto.

### 7.1.2 Creación de la distribución del framework

La distribución consiste en la librería Java mtp.jar, que se genera desde Eclipse con los siguientes pasos:

1. Seleccionar dentro de src la carpeta META-INF (contiene la librería de etiquetas) y todos los paquetes que comienzan por edu.uoc.pfc.j2ee.jalonsod.mtp.
2. Abrir el menú File / Export… (o desde el menú contextual seleccionar Export…)
3. Destino de la exportación: Java / JAR File
4. JAR File Specification:
   - En recursos para exportar habrá marcado los indicados en el punto 1.
   - Seleccionar Export Java source files and resources.
   - JAR File. Elegir una ubicación y el nombre mtp.jar
5. JAR Packaging Options. Dejar las opciones por defecto.
6. JAR Manifest Specification. Dejar las opciones por defecto.
7. Finalizar la creación del archivo mtp.jar

## 7.2 Configuración, referencia de uso y extensibilidad del framework

A continuación se describen los pasos para incorporar el framework en una aplicación, la referencia de uso tanto del interfaz RequestData como de la librería de etiquetas para JSP y finalmente los puntos en que el framework es extensible.

### 7.2.1 Configuración del framework en una aplicación

El framework presenta tres ejes fundamentales de cara a la integración con las aplicaciones:

- Configuración, basada en un fichero XML.
- Clases modelo o "model" para integrar la presentación con el resto de la aplicación. Esta integración se hace combinando en la clase POJO un Java Bean con un método opcional de validación y uno de ejecución que puede ser independiente o puede recibir información propia del marco de trabajo.
- Interfaz de usuario, basado en JSPs, que a través de una librería de etiquetas (taglib), permite la integración de estas páginas dinámicas con el marco de trabajo.

#### *7.2.1.1 Versiones requeridas*

El framework requiere para su ejecución las siguientes versiones de Java y J2EE:

- JRE. Versión 1.6 o superior. Se ha desarrollado con la versión 1.6.0_31
- J2EE. Versión J2EE 5.0 o superior. Puede utilizar cualquier servidor que cumpla el "Web Profile" o el "Full Profile". Se ha probado en los servidores Apache Tomcat 7.0.27 y JBoss AS 7.1.0.

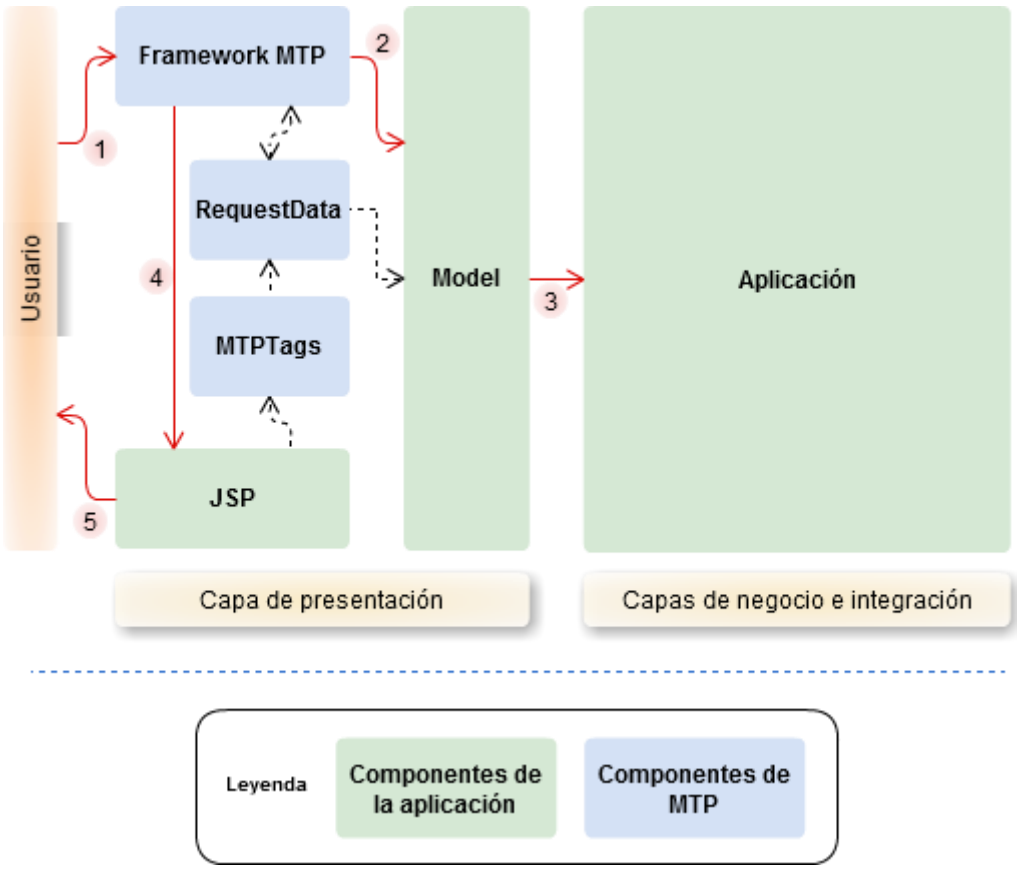
#### *7.2.1.2 Distribución*

La distribución de binarios del framework se presenta en un solo archivo que hay que incorporar a la aplicación:

- **mt.jar**: Librería JAR que contiene las clases del framework, el schema para el fichero de configuración y la librería de etiquetas MTPTags.tld para las páginas JSP.

#### *7.2.1.3 Arquitectura tipo de aplicaciones MTP y ciclo de vida de una petición*

Las aplicaciones que utilizan MTP presentan típicamente la arquitectura de componentes siguiente en la capa de presentación:




*Componentes y flujo de navegación típica de una aplicación MTP*

- **Framework MTP**: Recoge las peticiones del usuario y ejecuta el model adecuado. A la finalización de la ejecución, encamina la petición a la JSP que corresponda.
- **RequestData**: Es el objeto que facilita el framework para interactuar con él y almacenar estructuras de datos. Tiene un periodo de vida igual al de una petición (request). Ofrece información del request, la sesión, modelo de la aplicación, funciones de internacionalización y lectura de la configuración del framework.
- **Model**: Es el componente de aplicación que valida (opcionalmente) y ejecuta la acción solicitada interactuando con los objetos de negocio de la aplicación. Actúa también como medio de intercambio de datos entre la vista y la aplicación.
- **JSP**. Presenta la vista al usuario. Se apoya en la librería de etiquetas para obtener información.
- **MTPTags**. Es la librería de etiquetas que da soporte a las páginas JSP para ofrecer la información ofrecida por el RequestData.

El ciclo de vida de la petición, desde el punto de vista de la aplicación presenta las siguientes fases:

1. El usuario envía una petición que recoge el framework MTP.
2. El framework preprocesa los datos de la petición depositando la información procesada en el RequestData y delega la ejecución de la acción asociada a la petición al modelo adecuado de la aplicación.
3. El modelo se apoya en la capas de negocio/integración de la aplicación para procesar la petición.

4. Finalizada la ejecución del modelo, el framework vuelve a tomar el control y delega el control a la vista (JSP) adecuada, para que presente el resultado al usuario.
5. La JSP construye la página de respuesta al usuario, utilizando la librería de etiquetas (TLD) del framework MTP para obtener la información. Finalmente devuelve la vista construida al usuario.

#### *7.2.1.4 Instalación y configuración*

Antes de utilizar MTP en una aplicación, primero hay que poner su artefacto (librería Java) accesible por la aplicación, así como configurar los distintos componentes del marco de trabajo.

##### 7.2.1.4.1 Librería Java

La librería mtp.jar debe estar ubicada en el classpath de la aplicación. Típicamente se ubica en el directorio WEB-INF/lib de la aplicación J2EE, quedando así visible por el classloader, aunque puede añadirse al classpath de la aplicación o del servidor.

##### 7.2.1.4.2 Configuración del Servlet

El Servlet que actúa como controlador del framework debe configurarse adecuadamente en el descriptor de despliegue (web.xml) de la aplicación. Para ello, se añade el Servlet y se le asigna una ruta de acceso web (servlet-mapping).

El Servlet tiene dos parámetros de inicialización:

| Nombre | Requerido | Valor por defecto | Descripción |
|---|---|---|---|
| **configFile** | No | MTPConfig.xml | Indica la ubicación del fichero de configuración de MTP, relativa a la raíz del directorio de clases: /WEB-INF/clases |
| **loggerLevel** | No | WARNING | Indica el nivel de detalle de log del framework. Los valores posibles son los niveles indicados por la clase java.util.logging.Level |

Ejemplo de configuración:

```
<servlet>
  <description>Controlador del framework MTP</description>
  <display-name>Controller</display-name>
  <servlet-name>Controller</servlet-name>
  <servlet-class>edu.uoc.pfc.j2ee.jalonsod.mtp.Controller</servlet-class>
  <init-param>
    <description>Configuration File Name</description>
    <param-name>configFile</param-name>
    <param-value>MTPConfig.xml</param-value>
  </init-param>
  <init-param>
    <description>Logger level</description>
    <param-name>loggerLevel</param-name>
    <param-value>CONFIG</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Controller</servlet-name>
  <url-pattern>/Controller</url-pattern>
</servlet-mapping>
```

#### 7.2.1.4.3 Fichero de configuración

El fichero de configuración de la aplicación permite que ésta declare al framework la estructura que va a utilizar. Este fichero debe estar basado en el schema proporcionado por el Framework (está situado dentro de la librería en la ruta edu/uoc/pfc/j2ee/jalonsod/mtp/config/MTPConfig.xsd). Para recibir ayuda del IDE de desarrollo en la construcción del XML, puede ser necesario extraer el XSD de la librería y copiarlo en el proyecto.

También se puede copiar el XSD a la misma carpeta donde se ubica el MTPConfig.xml y referenciarlo para que la herramienta de desarrollo.

En caso de que no se indique otra ruta en el parámetro de arranque del Servlet controlador en el descriptor de despliegue (web.xml), la ubicación del descriptor es /MTPConfig.xml, por lo que debe situarse en el directorio raíz del directorio de clases (/WEB-INF/classes/).

Los elementos principales de la configuración son:

- **MTPConfig**. Elemento raíz. Contiene algunos atributos de configuración general del framework y los elementos de los distintos tipos de conceptos que maneja la configuración: model, form, validation, language, role.
  - **Form**: 0..* ocurrencias. El form representa el conjunto de parámetros recibidos en el request que son de significado para el modelo o la vista. Estos parámetros son cargados en objetos Java del tipo de datos adecuado, validados y opcionalmente cargados en el model. El form por tanto consiste en un conjunto de parámetros.
    - **Parameter**: 1..* ocurrencias. Es cada uno de los parámetros recibidos en el request y de significado para el modelo o la vista. El parámetro tiene tres fases de validación: la carga al tipo de datos Java, la verificación de valores requeridos y la función opcional de validación.
  - **Language**: 1..* ocurrencias. El language (idioma) se utiliza para la traducción de mensajes en el interfaz de usuario de la aplicación. Cada idioma tiene asociado un fichero de mensajes (puede utilizarse el mismo fichero de mensajes para varios idiomas). El idioma utilizado por defecto en el framework se describe en el elemento raíz.
  - **model**: 0..* ocurrencias. Cada modelo representa una acción de la aplicación. El modelo apunta a una clase de la aplicación en la cual primero se cargan los parámetros de entrada, después se validan en conjunto (opcional) y finalmente se ejecuta la acción. La vista puede acceder después a este modelo para consultar los parámetros de entrada cargados por el framework u otros parámetros depositados por el mismo modelo durante la ejecución de la acción. Cada modelo a su vez consta de varios resultados posibles (result) y permisos otorgados (grantedRole). El modelo utilizado por defecto en el framework se describe en el elemento raíz.
    - **result:** 1..* ocurrencias. Cada resultado equivale a una respuesta posible de la ejecución del modelo y tiene asociada una vista (JSP).
    - **grantedRole:** 0..* ocurrencias. Un modelo puede incorporar, opcionalmente, varios roles, en cuyo caso, solo las sesiones que tengan alguno de estos roles pueden acceder al modelo. Si no se especifica ningún grantedRole, se considera que el modelo es de acceso libre (ver role).

- **role**: 0..* ocurrencias. Enumera los roles de aplicación que se utilizarán en los modelos y sesiones para implementar la seguridad (ver grantedRole).
- **validation**: 0..* ocurrencias. Describe cada una de las funciones de validación (facilitadas por la aplicación) que pueden ser utilizadas para verificar parámetros de entrada individuales.

En general, los elementos tienen un atributo "name" que debe ser único en su clase, y se utiliza como referencia entre elementos (por ejemplo, el form de un modelo). Para más información sobre el significado concreto de los atributos, se remite al schema XSD, que está autodocumentado.

Adicionalmente al fichero de configuración, en los elementos language se ha referido a unos ficheros de mensajes. Estos ficheros planos contienen las distintas traducciones, se ubican de forma predeterminada en el mismo directorio que el fichero de configuración, son cargados al inicializar la configuración y tienen el formato de los ficheros de propiedades, es decir, contienen entradas del tipo:

```
Clave1=Mensaje localizado1
Clave2=Mensaje localizado2
```

#### 7.2.1.4.4 Librería de etiquetas

Para utilizar la librería en las páginas JSP, hay que incluir la siguiente directiva taglib en la parte inicial de la JSP:

```
...
<%@ taglib uri="http://jalonsod.edu.uoc/pfc/j2ee/mtp/MTPTags.tld"
    prefix="mtp" %>
...
<mtp:getModelInstance>
username: ${requestModelInstance.username}<BR>
e-mail: ${requestModelInstance.email}<BR>
</mtp:getModelInstance>
```

La referencia de uso de las etiquetas de la librería se proporciona más adelante.

### *7.2.1.5 Navegación*

En la implementación de la aplicación, el desarrollador debe tener en cuenta las siguientes consideraciones:

- Para que el framework tome el control de las peticiones (request) de los clientes, éstas deben ir dirigidas a la ruta indicada para el Servlet controlador.
- La implementación del ModelMapper facilitada utiliza el parámetro MODEL para seleccionar el modelo a cargar y ejecutar (el valor del parámetro MODEL debe coincidir con el atributo "name" del modelo y es sensible a mayúsculas y minúsculas).
- Los nombres de los parámetros enviados desde el cliente deben coincidir con los indicados en el "name" de los parameter indicados en los form del fichero de configuración y es sensible a mayúsculas y minúsculas.

- El framework crea automáticamente una sesión Java a los clientes, en la que almacena los roles asignados, el idioma del usuario (en caso de que la aplicación lo fije) y los demás atributos que la aplicación almacene a nivel de sesión.
- El framework redirige las peticiones a la vista seleccionada con forward, por lo que no hay redirecciones de URL en el navegador.
- En la aplicación de ejemplo, las páginas JSP se han incorporado fuera del directorio context-root para evitar que se pueda acceder a ellas sin acceder a través del controller. Concretamente se han ubicado en el directorio WEB-INF\jsp para que puedan accederse desde el controller, pero no directamente desde una URL. Esto no es necesario, pero aconsejable si se desea que el usuario no cargue páginas JSP sin pasar por el controller de MTP.
- Se ha indicado que el modelo por defecto es home, por lo que cualquier error de navegación redirige a esa página que además cierra la sesión actual. La página inicial indicada en el descriptor de despliegue corresponde al controller, por lo que se puede acceder a la aplicación directamente utilizando el context root de la aplicación.
- En la implementación del ViewMapper, la vista se selecciona con el siguiente algoritmo:

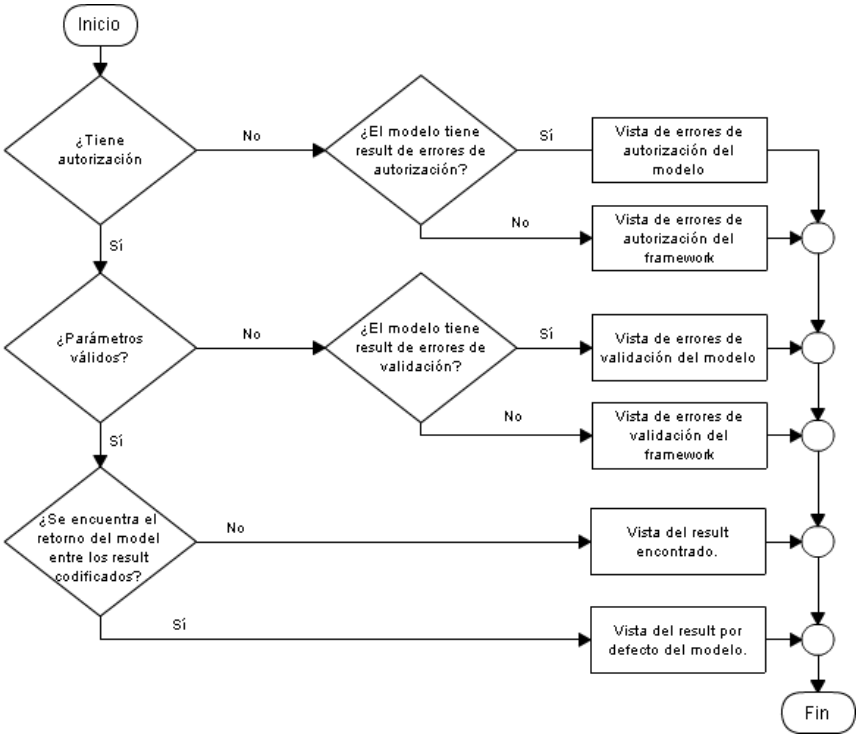



Ilustración 7:1 Algoritmo de selección de vista del ViewMapper implementado en MTP.

#### *7.2.1.6 Modelos (model)*

Los modelos son uno de los dos tipos de componentes de la aplicación con los que interacciona el framework, siendo su misión invocar a la lógica de negocio de la aplicación.

Como parte de la capa de presentación, los modelos ejecutan las acciones solicitadas desde el interfaz de usuario, por lo que hacen de puente entre el usuario y el corazón de la aplicación. En la arquitectura del marco de trabajo MTP, el modelo tiene las siguientes responsabilidades:

- Validar en conjunto los parámetros de entrada (opcional).
- Ejecutar la acción solicitada por el usuario.
- Servir de contenedor de datos de entrada que viajan del interfaz de usuario a la aplicación.
- Servir de contenedor de datos de salida de la aplicación necesarios para presentar la siguiente vista.

Para poder integrar los modelos con el framework, éstos deben ser declarados en el fichero de configuración de MTP (típicamente MTPConfig.xml), y deben cumplir una serie de características:

- El framework crea una instancia del modelo durante el proceso de selección de modelo, que será utilizada en el resto de la vida del request, por lo que deben poseer un constructor público sin parámetros.
- Si se define un método de validación en la configuración del modelo, es necesario implementar un método público de instancia sin parámetros que devuelva un valor Boolean, true cuando la validación es correcta y false en caso contrario.
- Debe tener forma de Java Bean con métodos públicos para guardar en la instancia los parámetros del form de entrada. Los métodos setter deben tener la forma adecuada, por ejemplo setParamName(String) para el parámetro paramName de tipo String. Los tipos de datos de entrada pueden ser: Boolean, java.util.Date, Double, Integer y String.
- Debe poseer un método de ejecución de la acción. Este método tiene dos formas posibles: String runMethod(RequestData) o String runMethod(), según se haya indicado en la configuración del modelo si éste debe recibir el objeto requestData o no. En cualquiera de los casos debe devolver una cadena (string) que será evaluada por el view mapper para la selección de la vista.
- El modelo puede cargar atributos para que sean accedidos por la vista, bien sean los propios parámetros recibidos que se han modificado u otros valores nuevos. Para estos atributos el modelo también debe poseer forma de Java Bean, con los correspondientes métodos públicos getter: por ejemplo Double getAttribute() para el atributo attribute de tipo Double.

### 7.2.2 RequestData. El interfaz de la librería mtp.jar

Para la interacción entre el framework, el request, la sesión, el modelo y la vista, MTP ofrece un objeto, denominado "requestData" cuya vida comienza en las primeras fases del ciclo de vida del request y finaliza a la vez que éste.

Su objetivo es ofrecer un punto único de interacción a todos los elementos implicados, para lo que necesita ir almacenando aquellos objetos creados por el framework para el request: language, model, form, result y view seleccionados, respuesta del modelo, referencias a la instancia de modelo, al request, la sesión y el configurador y finalmente los estados de autorización y validación.

El API que ofrece el RequestData se puede consultar en la documentación Javadoc, pero a modo de resumen, ofrece a las aplicaciones los siguientes métodos:

- Acceso al configurador e idiomas disponibles:
```
Configurator getConfigurator()
Hashtable<String, Language> getLanguages()
```
- Manipulación de la sesión (creación, finalización y consulta de validez):
```
void createSession()
boolean hasValidSession()
void logoutSession()
```
- Manipulación de atributos de la sesión (lectura, escritura y borrado):
```
void deleteSessionAttribute(String attributeName)
Object getSessionAttribute(String attributeName)
void setSessionAttribute(String attributeName, Object attributeObject)
```
- Manipulación de los roles fijados a la sesión (lectura, escritura):
```
Hashtable<String, Role> getRoles()
void setRoles(Hashtable<String, Role> sessionRoles)
```
- Funciones de internacionalización: lectura y fijación del idioma y obtención de mensajes en el idioma del request:
```
String getMessage(String key)
Language getUserLanguage()
void setUserLanguage(Language userLanguage)
```
- Manipulación de atributos del request (lectura, escritura):
```
Object getRequestAttribute(String attributeName)
void setRequestAttribute(String attributeName, Object attributeObject)
```
- Lectura de parámetros del request:
```
String getRequestParameter(String parameterName)
```
- Obtención de objetos propios del MTP para el request (modelo, instancia del modelo, respuesta del modelo, instancia del formulario de parámetros, result y nombre de la vista seleccionados):
```
FormInstance getFormInstance()
Model getModel()
Object getModelInstance()
String getModelResponse()
Result getResult()
String getView()
```
- Estado de la autorización del modelo y validación de parámetros.
```
int getValidationStatus()
boolean isAuthorized()
boolean isValidated()
```

Existe un API interno para los objetos que se van creando en el framework. Este API no está mostrado al modelo y la vista, y debe ser utilizado exclusivamente por los elementos del framework.

### 7.2.3 MTPTags.tld. La librería de etiquetas para las páginas JSP

La librería de etiquetas para páginas JSP contempla las funciones del requestData, para poder incluir la funcionalidad de forma más natural en las páginas JSP, sin necesidad de incluir código Java en las mismas. Las etiquetas disponibles son las siguientes:

| Tag | Detalles |
|---|---|
| **getProperty** | **Descripción**: Lee una propiedad de un Java Bean almacenado en el contexto de la página y la escribe en la salida.<br>**Cuerpo**: No<br>**Atributos**:<br>property: Propiedad a leer del Java Bean. |

| | |
|---|---|
| | attribute: Nombre del atributo del contexto que contiene el Java Bean. |
| **getLanguages** | **Descripción**: Itera sobre la lista de idiomas soportados por el framework.<br>**Cuerpo**: JSP<br>**Variables**:<br>language: Objeto language del framework. |
| **createSession** | **Descripción**: Crea una nueva sesión para el usuario.<br>**Cuerpo**: No. |
| **hasValidSession** | **Descripción**: Escribe en la salida true si el request tiene una sesión válida o false en caso contrario.<br>**Cuerpo**: No. |
| **logoutSession** | **Descripción**: Invalida la sesión actual.<br>**Cuerpo**: No. |
| **getSessionAttribute** | **Descripción**: Obtiene a través del RequestData un atributo de la sesión.<br>**Cuerpo**: JSP.<br>**Atributos**:<br>attribute: Nombre del atributo del contexto de sesión a leer.<br>**Variables**:<br>sessionAttribute: Atributo obtenido del contexto de sesión. |
| **deleteSessionAttribute** | **Descripción**: Elimina a través del RequestData un atributo de la sesión.<br>**Cuerpo**: No.<br>**Atributos**:<br>attribute: Nombre del atributo del contexto de sesión a eliminar. |
| **getRoles** | **Descripción**: Itera sobre la lista de roles de la sesión.<br>**Cuerpo**: JSP.<br>**Variables**:<br>sessionRole: Objeto role de la sesión. |
| **getUserLanguage** | **Descripción**: Obtiene una propiedad del objeto language del request.<br>**Cuerpo**: No.<br>**Atributos**:<br>property: Nombre de la propiedad del atributo language de la sesión. Los valores posibles son name, longName y messageFile |
| **setUserLanguage** | **Descripción**: Fija el idioma asignado a la sesión.<br>**Cuerpo**: No.<br>**Atributos**:<br>name: Nombre del idioma a asignar a la sesión. Debe ser uno de los name indicados en la configuración de la aplicación. |
| **msg** | **Descripción**: Escribe en la salida un mensaje en el idioma indicado (o por defecto en el idioma del request.<br>**Cuerpo**: No.<br>**Atributos**:<br>key: Clave del mensaje a mostrar.<br>language: Opcional. Idioma en el que se quiere mostrar el mensaje. Si no se indica se utiliza el idioma del request. |
| **getRequestAttribute** | **Descripción**: Obtiene a través del RequestData un atributo del request.<br>**Cuerpo**: JSP.<br>**Atributos**:<br>attribute: Nombre del atributo del contexto de request a leer.<br>**Variables**:<br>requestAttribute: Atributo obtenido del contexto del request. |
| **getRequestParameter** | **Descripción**: Obtiene a través del RequestData un parámetro del request.<br>**Cuerpo**: No.<br>**Atributos**:<br>name: Nombre del parámetro del request a leer. |
| **getModel** | **Descripción**: Obtiene el objeto de configuración model seleccionado para el request.<br>**Cuerpo**: JSP.<br>**Variables**:<br>requestModel: Objeto model seleccionado. |
| **getModelInstance** | **Descripción**: Obtiene la instancia del modelo de aplicación creada para el request.<br>**Cuerpo**: JSP. |

| | |
|---|---|
| | **Variables**:<br>requestModelInstance: Instancia del modelo de aplicación. |
| **getFormInstanceParameter** | **Descripción**: Obtiene el objeto parameterInstance del framework con el parámetro indicado (devuelve el objeto parameterInstance, no el valor).<br>**Cuerpo**: JSP.<br>**Atributos**:<br>name: Nombre del parámetro del form.<br>**Variables**:<br>requestFormInstanceParameter: Objeto parameterInstance con el parámetro indicado. |
| **getFormInstanceParameterValue** | **Descripción**: Escribe en la salida el valor del objeto parameterInstance del framework con el parámetro indicado.<br>**Cuerpo**: No.<br>**Atributos**:<br>name: Nombre del parámetro del form. |
| **getModelResponse** | **Descripción**: Obtiene la respuesta del método run de la instancia del modelo de aplicación.<br>**Cuerpo**: JSP.<br>**Variables**:<br>requestModelResponse: Respuesta del método run de la instancia del modelo de aplicación. |
| **getResult** | **Descripción**: Obtiene el objeto de configuración result seleccionado para el request.<br>**Cuerpo**: JSP.<br>**Variables**:<br>requestResult: Objeto result seleccionado. |
| **getView** | **Descripción**: Escribe en la salida el nombre de la vista seleccionada en el request.<br>**Cuerpo**: No. |
| **isAuthorized** | **Descripción**: Escribe en la salida si el request ha sido autorizado (true) o no (false).<br>**Cuerpo**: No. |
| **getValidationStatus** | **Descripción**: Escribe en la salida el estado de validación del request (ver constantes VALIDATION* del RequestData).<br>**Cuerpo**: No. |
| **isValidated** | **Descripción**: Escribe en la salida si el request ha sido validado (true) o no (false).<br>**Cuerpo**: No. |

Ilustración 7:2 Etiquetas de la librería de tags para JSP.

### 7.2.4 Extensibilidad del framework

El framework se ha implementado dejando abiertas o preparadas una serie de funciones del mismo, con el objetivo de extenderlo a nuevas necesidades de las aplicaciones usuarias. Los aspectos más destacados en este sentido son:

- **ModelMapper**. El framework implementa un ModelMapper por defecto, que permite elegir e instanciar el modelo a cargar en cada petición. Pero deja abierto a utilizar, indicándolo en el fichero de configuración, una implementación particular del mismo, que debe implementar el interfaz ModelMapper.
- **ViewMapper**. El framework implementa un ViewMapper por defecto, que permite elegir la vista a cargar en cada petición. Pero deja abierto a utilizar, indicándolo en el fichero de configuración, una implementación particular del mismo, que debe implementar el interfaz ViewMapper.

- **Protocolo**. El framework implementa el protocolo http, pero está preparado para ampliarse fácilmente a otros protocolos, sustituyendo tan solo el Servlet principal y un ContextFactory que permite crear el contexto adecuado a cada petición.
- **Tipos de datos**. El marco de trabajo utiliza los tipos de datos Boolean, java.util.Date, Double, Integer y String, pero está preparado para recibir otros tipos de datos con pequeñas modificaciones.

## 7.3 Instalación y ejecución aplicación de ejemplo AMPA

Para ejecutar la aplicación ejemplo, se necesita disponer de:

- Distribución de la aplicación (fichero AMPA.war)
- Librerías de terceros.
- Servidor de aplicaciones.
- Servidor de base de datos.

La instalación básica de estos productos se ha cubierto en el anexo "Entorno de desarrollo". A continuación se explica cómo preparar la aplicación AMPA para su ejecución.

### 7.3.1 Creación de la base de datos

Se entrega en el directorio AMPA\database el script CreacionEsquema.sql que se debe ejecutar desde el interfaz de comandos "SQL Workbench" de MySQL. El script tiene varias fases: creación del esquema (database) ampa; creación del usuario ampauser y asignación de permisos; creación de tablas; y finalmente inserción de datos de prueba.

**Conexión a la base de datos**. Si la base de datos no está instalada en el mismo equipo que el servidor de aplicaciones o no está atendiendo al puerto por defecto (3306), será necesario indicarlo en la propiedad "connection-url" del fichero de configuración de Hibernate situado en el directorio WEB-INF/classses del fichero WAR.

### 7.3.2 Instalación en un servidor de aplicaciones

Se entrega en el directorio AMPA\bin el archivo AMPA.war. Por limitaciones de espacio en la entrega, se han eliminado del fichero WAR las librerías de terceros, por lo que antes de desplegarlo en el servidor de aplicaciones, deben incluirse las librerías externas que se mencionan en el anexo "Entorno de desarrollo" en el directorio WEB-INF\lib del fichero WAR.

Verificar que la variable JAVA_HOME apunta al directorio principal de la instalación de Java (JDK) indicado.

#### *7.3.2.1 Despliegue en Tomcat 7.0.*

Copiar el fichero AMPA.war al directorio <TOMCATHOME>\webapps.

Arrancar el servidor Tomcat (<TOMCATHOME>\bin\startup.bat o startup.sh para Linux).

La aplicación se despliega automáticamente.

#### 7.3.2.2 Despliegue en JBoss AS 7.1.0

Copiar el fichero AMPA.war al directorio <JBOSSHOME>\standalone\deployments.

Arrancar el servidor JBoss (<JBOSSHOME>\bin\standalone.bat o standalonse.sh para Linux).

La URL de acceso en la instalación por defecto es http://localhost:8080/AMPA/

### 7.3.3 Ejecución

La URL de acceso en la instalación por defecto en los servidores Tomcat y JBoss es http://localhost:8080/AMPA/.

Los usuarios creados en la carga inicial de la base de datos para acceder una vez en la aplicación son:

| Perfil | Usuario | Contraseña |
|---|---|---|
| Administrador | admin | manager |
| Socio | user@example.com | userpass |

Ilustración 7:3 Usuarios predefinidos en la aplicación AMPA

La información sobre las opciones disponibles para cada perfil de usuario se pueden obtener en el apartado “Aplicación de ejemplo AMPA” / “Navegación” de este documento.

La aplicación se ha probado con los navegadores

- Internet Explorer 8
- Mozilla Firefox 12
- Google Chrome 19