Diseño e implementación de un marco de trabajo de presentación para aplicación J2EE

Felipe Benavente Cabrera

Ingeniería en Informática

Josep Maria Camps Riba

20/06/2011

A continuación se presenta la memoria del proyecto final de carrera “Diseño e implementación de un marco de trabajo de presentación para aplicaciones J2EE” el cual pretende definir y crear un conjunto de utilidades y pautas de programación que permitan al programador implementar aplicaciones J2EE siguiendo el patrón de diseño MVC.

En el contexto actual, el desarrollo de la capa de presentación en aplicaciones J2EE es muy costoso, sobretodo si son de una cierta complejidad; es aquí donde el patrón MVC ayuda a aislar la lógica de la vista y la gestión de peticiones. Además, si tenemos un marco de trabajo (o *framework*) que proporcione una serie de utilidades y pautas, el desarrollo de estas aplicaciones será más eficiente y estandarizado.

Para la definición y desarrollo del marco de trabajo se definen tres objetivos básicos: Estudiar las alternativas existentes en el mercado, diseñar e implementar el marco y finalmente desarrollar una aplicación de prueba que muestre las funcionalidades implementadas.

Como podremos observar en esta memoria, se ha dividido el proyecto en diferentes entregas, planificadas al inicio del curso para así hacer un seguimiento de su evolución y llegar a los objetivos en los plazos adecuados. Como es propio de cualquier proyecto, dicha planificación ha sufrido cambios de fechas, dedicación o alcance para responder a las incidencias que han ido produciéndose.

Índice

1. Introducción

1.1. Justificación

En el momento en el que nos encontramos, el estándar J2EE es posiblemente el estándar de desarrollo de aplicaciones más usado. Todo ello es gracias a que es multiplataforma, abierto, contiene numerosa documentación –la propia del estándar, además de la liberada por la comunidad de programadores–, permite una gran escalabilidad y distribución gracias a los diferentes componentes que define –característica importante en un momento en el que las aplicaciones distribuidas tienen gran relevancia–, etcétera.

Una de las partes que este estándar define son los componentes propios de la capa de presentación; los más comunes son los Servlets y las JSP (Java Server Pages). No obstante, implementar aplicaciones mediante el uso exclusivo de estos componentes y sin seguir una serie de pautas, puede hacer que las aplicaciones generadas tengan la lógica de control dispersa entre los diferentes elementos de control y la vista; por consiguiente, el programa generado será complejo y difícil de mantener lo que amplía los tiempos de programación y reduce la calidad del producto.

Además, si estudiamos los tiempos de desarrollo dedicados a cada una de las capas de una aplicación comprobaremos, que en la mayoría de los casos, la mayor parte corresponde a la presentación.

Por otro lado, no podemos menospreciar que cada vez es más común la integración y comunicación entre aplicaciones, así como presentar una aplicación para diferentes dispositivos y en diferentes formatos. Por ello es importante aislar lo máximo posible la vista del control y de la lógica de negocio.

Con tal de aislar cada uno de los componentes de la presentación surgió el patrón MVC (Model View Controller). Siguiendo dicho patrón de diseño conseguiremos aislar componentes, lo que permitirá tener un código más organizado y sencillo, y por tanto, más fácil de mantener y programar. Además, nos permitirá, cambiar alguno de los componentes sin necesidad de cambiar el resto de módulos.

Si a parte de seguir este patrón, disponemos un marco de presentación que no sólo lo implemente sino que además nos aporte una serie de utilidades para el desarrollo, conseguiremos que los tiempos se reduzcan ya que el programador se podrá centrar en la configuración e implementación de cada una de las partes, para las que además dispondrá una serie de utilidades que agilizarán la programación.

1.2. Objetivos

Por todo ello, el principal objetivo de este proyecto es proporcionar un marco que implemente el patrón MVC y que además, si es posible, proporcione utilidades al programador que reduzcan los tiempos de desarrollo de la capa de presentación de una aplicación J2EE.

Para ello, según el enunciado, el proyecto tiene tres objetivo básicos para alcanzar el objetivo principal:

- Estudiar y evaluar las diferentes alternativas existentes para implementar la capa de presentación de aplicaciones J2EE.
- Hacer un diseño e implementación de un marco de presentación.
- Construir una aplicación de ejemplo que muestre el uso del marco implementado.

1.3. Método seguido

Con tal de llegar a los objetivos en los plazos adecuados hemos seguido un proceso de desarrollo secuencial. Dado que el proyecto tiene unos requisitos muy concretos y no necesitamos un soporte continuo del consultor, parece adecuado seguir este sistema.

Con este método la salida de cada una de las partes será la entrada de la siguiente. El problema que tiene este método es que los errores de cada fase se detectarán al final, lo que puede suponer un impacto temporal importante; dado que los requisitos son suficientemente claros, entendemos que los posibles errores detectados en cada entrega no deberían suponer un impacto importante sobre el proyecto.

Sobre esta metodología hemos hecho una variación; debido a que inicialmente queríamos ser ambiciosos con el proyecto, dividimos la fase de implementación en dos partes, una primera con funciones básicas y una segunda con una serie de mejoras adicionales que se irían analizando según los tiempos y necesidades. Como veremos posteriormente esta fase de mejoras acabará desapareciendo.

1.4. Planificación

Siguiendo la metodología comentada se entregó el día 16 de marzo de 2011 la planificación a seguir para el desarrollo del PFC. En ella se dividía el proyecto en cuatro entregas:

1. 8 de abril, entrega del estudio y evaluación de marcos de presentación disponibles en el mercado y del diseño del marco implementado. El diseño deberá permitir, dentro de lo posible, un conjunto de mejoras para implementar en fases posteriores.

2. 29 de abril, entrega del marco de trabajo básico –sólo funciones esenciales– y una pequeña aplicación de prueba.

3. 17 de mayo, entrega del marco de trabajo mejorado y una pequeña aplicación de prueba que emplee dichas mejoras.

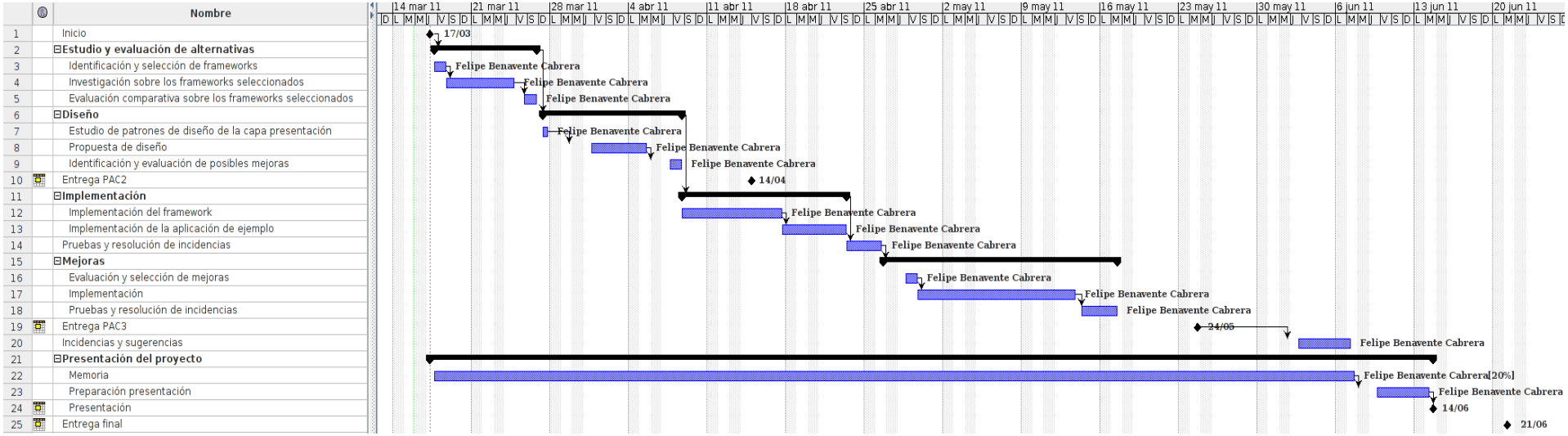
4. 20 de junio, entrega de la memoria y presentación del Proyecto Final de Carrera.

Estas fechas surgieron de la estimación de horas y el diagrama de Gantt que mostramos a continuación:

1.4.1. Estimación de horas

Tarea			Tiempo
1	**Estudio y evaluación de alternativas**		
	1.1	Identificación y selección de frameworks	3 horas
	1.2	Investigación sobre los frameworks seleccionados	21 horas
	1.3	Evaluación comparativa de frameworks seleccionados	6 horas
2	**Diseño**		
	2.1	Estudio de patrones de diseño de la capa presentación	6 horas
	2.2	Propuesta de diseño	21 horas
	2.3	Identificación y evaluación de posibles mejoras	3 horas
3	**Implementación**		
	3.1	Implementación del framework	30 horas
	3.2	Implementación de una aplicación de ejemplo	15 horas
4	**Pruebas y resolución de incidencias**		**15 horas**
5	**Mejoras**		
	5.1	Evaluación y selección de mejoras	3 horas
	5.2	Implementación de las mejoras	45 horas
	5.3	Pruebas y resolución de incidencias	15 horas
6	**Incidencias y sugerencias**		**15 horas**
7	**Presentación del proyecto**		
	7.1	Memoria	50 horas
	7.2	Preparación de la presentación	15 horas
	7.3	Presentación	3 horas

1.4.2. Diagrama de Gantt



Esta planificación se hizo teniendo en cuenta un esfuerzo semanal de 15 horas.

Como veremos en capítulos posteriores, esta planificación sufrirá cambios tanto de fechas, esfuerzo, como alcance; con tal de solucionar las incidencias que fueron produciéndose durante el desarrollo.

1.5. Productos obtenidos

Los productos obtenidos finalmente han sido:

- Un estudio de *frameworks* de presentación disponibles en el mercado.
- El diseño de un marco de presentación.
- Implementación de un marco de presentación.
- Una aplicación de prueba que funcione sobre el marco implementado.

1.6. Estructura de la memoria

Hemos estructurado la memoria según el desarrollo de los productos entregados. Dentro de cada apartado incluiremos las incidencias producidas, para así mostrar la evolución temporal.

En primer lugar tenemos un apartado de aspectos técnicos, donde hablamos de toda la información recopilada para el posterior desarrollo del marco. Incluye la evaluación de los *frameworks* actuales del mercado.

Una segunda parte que contiene el diseño del marco implementado, donde detallaremos las funcionalidades a implementar y de como se estructurarán.

A continuación hablaremos de lo referente a la implementación del marco de presentación, describiendo de forma detallada los módulos o componentes más importantes. También se detallarán los cambios respecto al diseño inicial.

Finalmente detallaremos brevemente la aplicación de prueba implementada la cual muestra como funcionan cada uno de los componentes del marco.

2. Desarrollo de un marco

Antes de empezar a desarrollar un marco de presentación que siga el patrón de diseño MVC es importante conocer en que consiste cada uno de estos conceptos, así como conocer otros productos del mercado para hacernos una idea de las tendencias y aprender de los defectos y virtudes de cada una de las opciones.

Para ello nos hemos ayudado de internet para recopilar información y en otros casos, nos hemos aprovechado de la experiencia laboral personal.

2.1. Aspectos técnicos

En los próximos puntos detallaremos que son los patrones de diseño y para que sirven, explicando de forma más detallada el patrón MVC; introduciremos que son los marcos de trabajo, o también conocidos como *frameworks*; finalmente, evaluaremos tres de los marcos de presentación más importantes: Struts, Spring MVC y Java Server Faces. Este último apartado se corresponde con uno de los productos de este proyecto.

2.1.1. Patrones de diseño, patrón MVC

Los patrones son básicamente plantillas que ofrecen soluciones concretas para problemas típicos del análisis, diseño y/o implementación de programas. El uso de patrones garantizaran la calidad del producto siempre que se usen en las condiciones para las cuales se definió, por ello será importante conocer para que problemas, en que condiciones deben aplicarse, así como cual es el resultado de aplicarlos.

Uno de los patrones más conocidos es el patrón MVC, el cual se diseñó con tal de aislar la interfaz gráfica del resto del sistema, para ello se definen tres componentes básicos: el Modelo, la Vista y el Controlador. El Modelo encapsula la lógica del sistema, la Vista presenta la información al usuario y recogen la información de éste, finalmente, el Controlador se encarga de gestionar los eventos producidos por el usuario, interactuando con el modelo y mostrando las vistas adecuadas.

2.1.2. Marcos de trabajo (*frameworks*)

Un marco de trabajo o *framework*, es un conjunto de librerías, clases, XML, etcétera, que definen e implementan un comportamiento genérico y que suelen incluir un conjunto de utilidades para agilizar el trabajo del programador que sólo deberá centrarse en los problemas propios de su negocio.

En el caso que nos aplica, hay numerosos frameworks que implementan el patrón de diseño MVC, pero sólo nos hemos centrado en los que consideramos más importantes en la actualidad:

- **Struts**, hemos seleccionado este debido a que es uno de los marcos más consolidados y posiblemente el más usado si tenemos en cuenta sus dos versiones (Struts 1 y Struts 2).

- **Spring MVC**, con una gran difusión gracias a que forma parte de un framework muy potente (**Spring**) para el desarrollo de aplicaciones en java.

- **JSF**, también muy difundido gracias a sus diferentes implementaciones y a que forma parte del estándar J2EE.

En el **anexo I**, se encuentra disponible el estudio detallado de estos *frameworks*. De dicho estudio se extraen las siguientes conclusiones –estas conclusiones serán la base de nuestro diseño–

Conclusiones del estudio de marcos de trabajo de presentación

Actualmente existen dos mecanismos básicos para la implementación de un marco de trabajo para la capa de presentación: orientado a peticiones (el que podríamos determinar como el método clásico) y el orientado a componentes y eventos. En el primero encontraríamos a Struts (1 y 2) y Spring MVC; en el segundo estaría JSF.

Struts ha sido pionero en marcos de trabajo que implementan el patrón MVC. De su experiencia nacieron otros marcos que corregían los errores de éste, de hecho, la segunda versión de Struts surgió de otro diferente y no de la primera versión como cabría esperar.

Hay una serie de características que siempre encontramos en todos los marcos de presentación:

- Un controlador principal que se encarga de gestionar las peticiones y que el programador debe configurar, ya sea mediante ficheros XML o anotaciones sobre las clases programadas.

- Un conjunto de interfaces para la definición de los objetos encargados de controlar la lógica y los datos necesarios para ésta, aunque los más modernos permiten usar las llamadas comúnmente clases POJO (éstas siglas hacen referencia a clases que no implementan ninguna interfaz o extienden de una clase genérica).

- Una serie de utilidades de validación y conversión de datos, así como opciones internacionalización. Éstas pueden ser ampliadas por el programador y en algunos casos reemplazadas por las de terceros.

- Una serie de *taglibs* que facilitan la implementación de la vista, o sino, permiten el uso de *taglibs* estándares o de terceros.

A continuación se muestra una tabla que ayudará a entender las diferencias entre las soluciones analizadas:

	Struts 1	Struts 2	Spring	JSF
Basado en peticiones	Sí	Sí	Sí	No
Basado en componentes	No	No	No	Sí
Soporte para AJAX	No	Bajo	Alto	Alto
Uso de anotaciones	No	Sí	Sí	Sí
Limitaciones del diseño	Altas	Medias	Bajas	Bajas

2.1.3. Incidencias durante la fase de estudio y evaluación

Durante esta fase, la dedicación al proyecto fue inferior a la prevista, en lugar de 15 horas semanales, la dedicación pasó a ser de 5 horas durante 4 semanas; lo que supone un retraso de 40 horas.

A pesar de tener un margen considerable en la planificación inicial, no era posible absorber el total de la desviación, por ello, debíamos tomar una decisión, o ampliar la dedicación, o bien, reducir el alcance. En este punto decidimos mantener el alcance del proyecto y ampliar la dedicación, ya que todavía teníamos capacidad suficiente.

2.2. Diseño

Fruto del estudio y evaluación anterior se detectaron una serie de componentes y funcionalidades que según nuestro parecer deberían ser imprescindibles para el marco implementado:

- Un controlador encargado de gestionar las peticiones del cliente.
- Un sistema de configuración que permita especificar el comportamiento del controlador en función de la petición, los datos, resultados internos, etcétera.
- Un conjunto de interfaces que ayuden al programador a definir los componentes necesarios para el control de datos y lógica de la capa de presentación.
- Un conjunto de utilidades de validación, conversión de datos e internacionalización. Sería preferible que estos elementos puedan ser configurados y ampliados por el usuario.
- Una estructura que permita el uso de *taglibs* estándares.

Teniendo en cuenta estos aspectos, lo primero que debemos decidir es que mecanismo usamos: orientado a peticiones o a componentes y eventos. En nuestro caso escogemos un modelo orientado a peticiones, ya que entendemos que no estamos en condiciones de implementar lo necesario para la definición de componentes y eventos.

2.2.1. Detalles del diseño

El marco de trabajo estará centrado en un controlador principal (**PFCController**) encargado de gestionar las peticiones y que al igual que en los ejemplos anteriores será un Servlet. Éste se configurará mediante un fichero XML llamado **pfc-configuration.xml** en el que se podrá indicar:

- Conjunto de objetos que gestionarán el comportamiento de las acciones y los datos de la presentación que a partir de ahora los llamaremos **Managers**. Para cada Manager se definirá el nombre y clase. Inicialmente sólo aceptaremos que los métodos devuelvan una cadena con el resultado para determinar acción o bien nada.

  Para cada una de las propiedades de los objetos, así como para el objeto en sí, se puede definir que validación usará; para cada validación será necesario indicar las variables que debe usar, siempre que estas hayan sido definidas para el validador.

  Del mismo modo, para cada objeto y cada atributo se puede definir que conversor usar.

  Estos objetos serán únicos por sesión, aunque está previsto añadir la gestión de hilos de ejecución en la fase de mejoras.

- **Mapeo de peticiones**. Para cada petición se deberá indicar la ruta de la llamada, donde además permitiremos añadir variables de la forma ${NOMBRE_VARIABLE}; la ruta de destino por defecto; además si fuera necesario se especificará el Manager sobre el cual se ejecutará la acción y el método del objeto a ejecutar, si no se indica método se ejecutará el método *execute*; además para cada método se deberá especificar los parámetros de entrada que pueden sacarse de variables de la url, de parámetros de la petición, de atributos de sesión, o de otros Managers; también se podrá definir para cada acción el destino final de las acciones en función de los resultados del método ejecutado; de la misma manera se podrá configurar el destino en función de si se ha producido una excepción.

  Para facilitar la configuración se podrán mapear las peticiones por niveles, es decir, en un primer nivel podremos indicar que todas las direcciones que empiecen por una cierta cadena usarán cierto objeto para, a continuación, indicar que método se ejecuta en función del resto de la cadena.

  También será posible definir un conversor y validador para cada una de las variables usadas en los métodos anteriormente comentados de forma similar a como lo haríamos con los Managers.

- **Tratamiento de excepciones.** Se podrá definir el comportamiento por defecto en caso de detectar errores en el controlador, indicando el destino al que debe navegar en función de la excepción capturada.

- **Definición de validadores**, donde el usuario indicará el nombre de la clase encargada de la validación y un conjunto de parámetros de configuración (para la construcción del validador). Además, el usuario podrá definir cual es el método de validación así como sus variables de entrada.

- **Definición de conversores**, que en este caso sera equiparable a la declaración de validadores y tendrá las mismas opciones.

Para la internacionalización usaremos librerías estándar y por ahora no estarán incluidas dentro de nuestro marco.

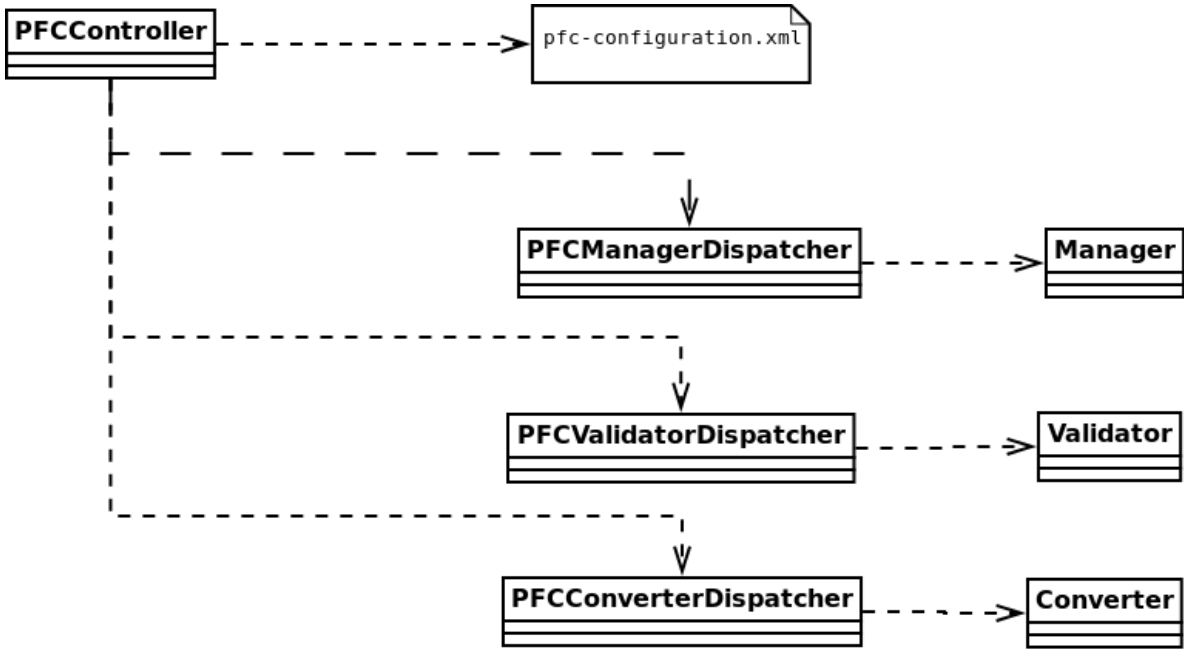
Todas las clases que debe definir el usuario podrán ser clases simples (POJO), aunque definiremos una serie de interfaces para los componentes que el programador puede definir y que le puedan servir como ayuda:

- **PFCManager**, Interfaz para la definición de objetos encargados de ejecutar acciones del controlador principal.

- **PFCValidator**, Interfaz para la definición de un validador simple que valida las variables de una petición.

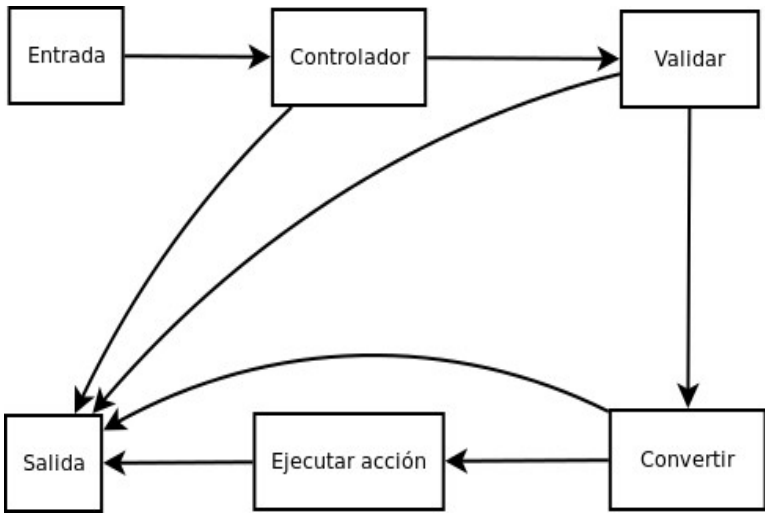
- **PFCConversor**, Interfaz para la definición de conversores de datos encargados de transformar una petición a un objeto cualquiera.

A continuación mostramos un diagrama de clases que refleja los componentes que tendrá nuestro diseño:



Como podemos observar hemos delegado parte de la lógica del controlador principal a unos **disparadores** encargados de gestionar los componentes que anteriormente hemos detallado.

El diagrama de flujo de nuestro del controlador principal será el siguiente:



De este diagrama se desprende que el primer paso es ejecutar las validaciones, para pasar a continuación a la transformación de datos, una vez que tenemos todos los datos se ejecuta la acción del manager y con el resultado de éstas procedemos a la mostrar la salida. Claro está que si hay errores el controlador se dirigirá inmediatamente a la salida mostrando el error adecuado en función de la definición que hayamos hecho.

2.2.2. Mejoras propuestas

Tal como estaba previsto en el proyecto, sobre el diseño se definen una serie de mejoras que podrán implementarse durante la fase final. Entre esas mejoras éstas son las que parecen más interesantes:

- Configuración mediante anotaciones y con múltiples ficheros XML.
- Integración con AJAX.
- El control de hilos de ejecución.
- Ampliar las opciones de validación y conversión de datos permitiendo al programador definir cuales usar para cada objeto, así como permitir definir validadores y conversores propios.
- Integrar la internacionalización dentro del marco de trabajo.

El número de mejoras propuestas no es elevado, pero entendemos que son puntos suficientemente complejas como para que ocupen el tiempo previsto.

Por otro lado no hemos añadido la implementación de *taglibs* propios porque entendemos que existen ya numerosas librerías y no forman parte del alcance del proyecto.

2.2.3. Incidencias durante el diseño

Durante esta fase no se produjeron nuevas incidencias, por lo que no fue necesario hacer una replanificación, simplemente introdujimos las horas reales y entregamos la nueva planificación junto al estudio y diseño.

Los detalles de la nueva planificación teniendo en cuenta las incidencias del estudio y evaluación se pueden consultar en el **anexo II**.

2.3. Implementación

2.3.1. Entorno de desarrollo

El primer paso antes de proceder a la implementación ha sido preparar el entorno de desarrollo que consta de los siguientes elementos:

- Sistema Operativo Ubuntu 11.04
- Plataforma JAVA 1.6
- Herramienta de desarrollo Netbeans 6.9
- Servidor Apache Tomcat 6 (también sobre Ubuntu 11.04)

Todas estas herramientas están disponibles de forma libre y gratuita en internet por lo que para preparar el entorno sólo hay que seguir los siguientes pasos.

Descargar el SO de la página oficial de Ubuntu (http://www.ubuntu.com/), nosotros hemos usado la versión 11.04, la más actual en el momento de desarrollo. Existen diferentes versiones en función de la máquina y del tipo de instalación que se desee, en nuestro caso hemos descargado la versión de 64 bits para instalar mediante un CD (esta misma versión se puede usar para la instalación USB).

La instalación es bastante sencilla si se siguen los pasos que se indican tanto en el CD como en la web, el punto más complejo de la instalación es la creación de particiones, aunque también viene bien explicado en el propio menú de instalación como en la web. En caso de dudas, los foros oficiales contienen soluciones para numerosos problemas típicos.

Una vez que se tiene Ubuntu instalado se puede instalar el resto de herramientas mediante el comando *aptitude*:

- **aptitude search NOMBRE_PROGRAMA** para buscar el programa que deseamos instalar.
- **aptitude install NOMBRE_PROGRAMA** para instalar un programa concreto.

Estas instrucciones se deben ejecutar en modo **superadministrador**.

Es probable que la versión oficial de java no esté disponible por defecto, por ello es necesario añadir las fuentes disponibles de los llamados “Canonical Partner Repositories”.

Una vez instalados cada uno de los programas no es necesario realizar ninguna acción más para usarlos, aunque es recomendable tener a mano la documentación de todos ellos por los posibles problemas de configuración que puedan surgir; es típico que la instalación por defecto del servidor Apache Tomcat 6 no se integre correctamente con Netbeans.

2.3.2. Paso del diseño a la implementación

Definición del fichero de configuración del controlador

El primer paso para pasar del diseño a la implementación ha sido crear el documento que contendrá la especificación del fichero XML de configuración pfc-configuration.xml. La definición se ha hecho mediante un fichero DTD [15] el cual servirá tanto para guiar al programador como para la validación de los ficheros generados. El fichero de configuración pfc-configuration.dtd se encuentra en el anexo III.

El fichero ha sido bastante fiel con el diseño, con la salvedad de la definición de conversores y uso de managers:

- Los conversores estaban definidos para cada manager y cada petición y durante la implementación nos ha parecido adecuado que la conversión de datos debería ser siempre la misma para cada tipo de objeto, por ello se definen los conversores para toda la aplicación y por cada uno se indica la clase que se encarga de convertir.

- En el diseño hablábamos de que se podría definir las acciones de un manager en bloque, pero dado que definir una llamada a un método es bastante simple hemos decidido simplificarlo, por lo que en cada acción se deberá indicar el mánager usado.

Lectura de la configuración

Una vez especificada las opciones de configuración se ha procedido a crear el sistema que se encargará de procesar la configuración y que usará el controlador para gestionar las peticiones.

Para que dicho proceso sea lo más sencillo posible se han creado un conjunto de clases que contendrán la configuración y una serie de métodos para facilitar su lectura.

Para ejecutar la carga de dichas clases se usa lo que el estándar J2EE define como **Listener [4]:** una clase que se ejecuta al iniciar y destruir la aplicación. Esta clase es muy adecuada para nuestro problema ya que nos interesa leer la configuración una vez al inicio y tenerla siempre cargada para cuando la necesitemos. Para que funcione será necesario que la aplicación que use nuestro marco declaré dicho Listener.

A su vez, el Listener, hará uso de la utilidad XStream [16] que permite transformar el contenido de un fichero XML a una estructura de objetos y viceversa de forma sencilla.

Controlador

El controlador principal, PFCController, se alimentará de la configuración anteriormente cargada y que tal como se definió en diseño sigue las siguientes fases: procesado de entrada, validación, conversión, ejecución de acción y salida. Las fases de validación, conversión y ejecución son opcionales. Además, si se produce un error en cualquiera de esas fases se producirá un error.

- En la fase de procesado de entrada se extrae la URL de la petición y con ella se extrae de la configuración el mapeo correspondiente, si no se encuentra mapeo alguno, se activa un error. En esta fase se extrae el manager y se instancia si fuera necesario.

- En la fase de validación, se obtiene del mapeo las validaciones a aplicar y gracias al PFCValidatorDispatcher (definido en diseño) extraerá los parámetros necesarios para ejecutar el validador.

- En la fase de conversión se procesa en primer lugar cada una de las entradas posibles: parámetros del método a ejecutar y atributos del manager; generando por cada una una estructura propia que llamaremos entrada (PFCEntry). Una vez que tenemos la entrada y conocemos la clase a la que debemos convertir, delegamos la conversión en el PFCConverterDispatcher (definido en diseño), el cual determinará que conversor aplicar en función de la configuración actual y ejecutará la conversión.

- En la fase de ejecución se ejecutará un método concreto del manager usado, usando los parámetros definidos. La ejecución del método puede devolver una cadena con el resultado, o nada.

- En la fase de salida, en función del resultado, la posible excepción, y la configuración de la petición se determinará la vista a mostrar. Si no se detecta ninguna salida se activará un error.

Manager o gestores

Son los encargados de la lógica de la aplicación. Tal como se indicó en diseño éstos pueden ser cualquier tipo de clase, aunque se ha definido una interfaz básica para acciones más simples que sólo tendrán un método de ejecución. La interfaz se llama PFCManager.

El único requisito que deben cumplir los métodos de gestión del manager es que no tengan ninguna salida o que la salida sea una cadena con el resultado, el cual indicará el destino de la acción. Está previsto permitir otro tipo de salidas en el futuro para integrar con AJAX.

El controlador se encargará de instanciar estos gestores, por ahora habrá uno por sesión y se registrarán con el mismo nombre que se le haya asignado en configuración.

Validaciones

Las validaciones se hacen mediante objetos POJO, o bien seguir la interfaz que hemos creado. La única restricción que deben cumplir estos validadores es que deben devolver una lista de errores, donde cada error es una cadena con la clave del error. Devuelve una clave para que el sistema soporte multi-idioma.

Actualmente sólo hemos definido un validador de fechas, pero para fases posteriores añadiremos nuevos validadores.

La ejecución de validadores se hace mediante el intermediario PFCValidatorDispatcher, que recupera los parámetros de validación, obtiene el validador y lo ejecuta. Además por cada error genera una estructura que nos ayudará a identificar las propiedades que tienen errores.

Para la extracción de estos parámetros no se ha previsto ninguna conversión de datos, por lo que, si el parámetro se extrae del Request o de la URL será texto y si se extrae de sesión o del manager dependerá del atributo seleccionado. Cada validación devolverá un conjunto de errores que se apilarán, al final del proceso se activará una excepción de validación si ha habido errores con todos los errores detectados.

Conversiones

En el caso de los conversores también se definieron como clases simples, pero por la complejidad que conlleva la conversión de datos se ha decidido definir una interfaz para los conversores (PFCConverter). Éstos tendrán como parámetro una entrada, que contendrá los datos a convertir, la clase a la que deseamos transformar la entrada y el disparador (PFCConverterDispatcher), previamente iniciado.

Que tengamos como parámetro el disparador es debido a que un objeto puede ser tan complejo que requiera usar diferentes conversores para cada atributo, por ello el conversor delegará al disparador la conversión de cada uno de los atributos que no sean propios.

2.3.3. Incidencias durante la implementación

Durante el mes de abril el proyecto avanzó según la previsión, pero cuando se acercaba la entrega de la primera parte empezamos a tener problemas con la conversión de datos, lo que provocó unos días de bloqueo.

Para evitar ese bloqueo empezamos a trabajar en las mejoras, para así avanzar trabajo y cumplir la previsión final. Por ello el día 10 de mayo se envió una propuesta de mejoras poco ambiciosa, ya que debíamos reservar parte del tiempo a solucionar los problemas encontrados.

Aún así, por enfermedad no se pudo dedicar ni una hora de trabajo desde el día 12 hasta el día 17, por lo que se acercaba la fecha de entrega y el avance del proyecto era escaso. Por ello nos centramos en solucionar los problemas del proyecto base y dejar las mejoras para la fase final.

Con el trabajo de la presentación y memoria estas mejoras al final no se han entregado y pasan a formar parte del apartado “Aspectos a mejorar”.

2.4. Aplicación de test

La aplicación de pruebas consiste en un portal de compra con las siguientes funcionalidades:

- Selección de productos, donde aparecerá una lista de productos que se pueden seleccionar indicando la cantidad.

- Consulta y modificación del carro de la compra. Donde aparecerá la selección de productos actuales y se permitirá eliminar productos seleccionados.

- Confirmación de compra. Donde se confirma el pedido y se da por finalizado. Si no se han introducido los datos del cliente se procederá previamente a introducirlos.

- Introducción de datos de cliente. Pantalla donde se introducirán los datos del cliente.

3. Conclusiones

Tal como se comentaba en la introducción de esta memoria, el desarrollo de la capa de presentación es uno de los puntos más complejos y costosos en aplicaciones J2EE, por diferentes motivos:

- Es la parte que entra en contacto con el usuario y por tanto debe adaptarse a diferentes perfiles y ser capaz de tolerar los fallos que éstos cometen.

- Si no se tiene una estructura bien definida se tiende a tener la lógica dispersa, en diferentes componentes.

En consecuencia, por un lado es necesario seguir un patrón de diseño que defina correctamente las responsabilidades de cada elemento de la presentación como el patrón MVC, donde M representa el modelo de datos, V las vistas y C el controlador que gestiona las peticiones del usuario.

Por otro, es preferible tener un marco de trabajo que implemente dicho patrón, de esta forma el programador se podrá centrar exclusivamente en el desarrollo del programa. Además, si dicho marco dispone de utilidades propias de la capa de presentación se agilizarán mucho más los procesos de desarrollo.

En relación a estos apartados es importante destacar que en el mercado actual existen numerosos marcos, o *frameworks*, que implementan el patrón MVC y que además ofrecen numerosas utilidades que agilizan los procesos relacionados como el pintado de pantallas, la validación y conversión de datos, internacionalización, etcétera.

Además, usando estos marcos nos aprovechamos de su experiencia en proyectos de diferente índole; implementar un marco no es tarea fácil, tal como hemos podido observar, hay muchos elementos y tecnologías a tener en cuenta y el marco debe ser capaz de adaptarse.

Un ejemplo de aplicación en J2EE sería una aplicación web, donde la lógica de negocio se encuentra en EJBs remotos y web services; donde además de las típicas vistas el usuario interactúa mediante tecnología AJAX. Sería interesante que nuestro marco gestione ambas peticiones y facilite, dentro de lo posible, la interacción con la capa de negocio. Implementar todas estas funciones llevaría mucho tiempo y difícilmente estaría libre de errores.

No obstante, es interesante observar lo que hemos conseguido desarrollando un marco propio con un conjunto reducido de funciones:

- Se aísla la lógica de aplicación en managers definidos por el programador. Además, gracias al diseño se permite que usemos cualquier tipo de objeto, lo que da mayor flexibilidad.

- Tenemos un controlador central que gestiona las peticiones de los clientes y que tiene la capacidad de determinar que manager y método del manager usar, validar, obtener los datos y una vez hecho ejecutar el método adecuado y en función de su salida determinar la pantalla a mostrar. Todo ello mediante un fichero de configuración bastante sencillo.

- Tenemos un conjunto de utilidades de obtención y validación de datos reutilizables que hacen transparente la vista al modelo. Además, como se permiten definir validaciones y conversores propios hace que el marco sea fácilmente escalable.

En definitiva, a pesar de las múltiples carencias, tenemos un buen punto de partida para implementar una aplicación sencilla con una estructura bastante más limpia, donde las responsabilidades de cada elemento son claras, por lo que será más fácil desarrollarla y mantenerla.

4. Aspectos a mejorar

Tal como hemos comentado, el desarrollo de aplicaciones J2EE y en concreto la capa de presentación pueden contener muchas funciones, implementadas de diferentes formas y con diferentes tecnologías, por tanto, las utilidades a incluir dentro de un marco de presentación pueden ser infinitas.

No obstante, hemos dejado muchas características –que pueden considerarse básicas– sin implementar. Algunas de ellas estaban incluidas dentro del diseño básico del marco, otras se plantearon como mejoras para la fase final y otras ,simplemente, son funciones implementadas pero en un estado bastante primitivo:

- Validación y conversión de datos incluidos dentro de la URL. Estaba incluida en el diseño básico, pero al final no se ha podido implementar.

- Juego de validadores y conversores para datos básico. Estaba previsto en el diseño, aunque en el producto entregado los conversores son bastante sencillos y no son configurables y no se han incluido funciones de validación básicas.

- Permitir la subida de ficheros. Se detectó durante la implementación y parece una función esencial para un marco de presentación.

- Integración con AJAX. Se definió como mejora y se tuvo en cuenta durante el diseño inicial para que la integración fuera sencilla, pero no ha habido tiempo.

- Configuración mediante múltiples ficheros XML, o mediante anotaciones. Se planteó también como mejora.

- Permitir diferentes hilos de ejecución de un mismo Manager, se sugirió como mejora, pero por su complejidad se ha descartado.

Todas estas mejoras hacen referencia al marco de presentación, pero también sería interesante mejorar la aplicación de pruebas, porque a pesar de que cada una de las funcionalidades del marco se han probado, la aplicación entregada usa pocas funciones y de manera muy simple. Debido a esto es difícil garantizar que las funciones implementadas sean correctas.

5. Bibliografía

[1] Apache Software Foundation. *Apache Struts*. [en línea]. http://struts.apache.org [fecha de consulta: 05/04/2011].

[2] Apache Software Foundation. *DTD for the Struts Application Configuration File.* [en línea]. http://struts.apache.org/1.x/struts-core/dtddoc/struts-config_1_2.dtd.org.html [fecha de consulta: abril de 2011]

[3] Apache Software Foundation. *Apache Tomcat* [en línea]. http://tomcat.apache.org/ [fecha de consulta: 21/04/2011].

[4] API Java Platform SE6. http://download.oracle.com/javase/6/docs/api/ [fecha de consulta: abril-junio 2011]

[5] ELOY A, Esteban. *El API Struts*. [en línea]. http://www.programacion.com/articulo/el_api_struts_150 [fecha de consulta: 05/04/2011].

[6] FERNÁNDEZ GONZÁLEZ , Jordi; PRADEL I MIQUEL , Jordi; RAYA MARTOS , José Antonio (Septiembre 2005). *Enginyeria del programari orientat a l'objecte* . Barcelona: Universitat Oberta de Catalunya .

[7] GEARY, David (29/11/2002). *A first look at JavaServer Faces, Part 1*. [en línea]. http://www.javaworld.com/javaworld/jw-11-2002/jw-1129-jsf.html [fecha de consulta: 31/03/2011].

[8] GEARY, David (27/12/2002). *A first look at JavaServer Faces, Part 2*. [en línea]. http://www.javaworld.com/javaworld/jw-12-2002/jw-1227-jsf2.html [fecha de consulta: 31/03/2011].

[9] jcarreira; DECCICO, Adrian. *Struts 1 vs Struts 2*. [en línea]. Java Samples. http://www.java-samples.com/showtutorial.php?tutorialid=200 [fecha de consulta: 05/04/2011].

[10] Netbeans [en línea]. http://netbeans.org/ [fecha de consulta: 21/04/2011].

[11] Oracle. *JavaServer Faces Technology - Documentation*. [en línea]. http://www.oracle.com/technetwork/java/javaee/documentation/index-137726.html [fecha de consulta: 31/03/2011].

[12] Spring Source Community. *Spring Framework*. [en línea]. http://static.springsource.org/ [fecha de consulta: 14/04/2011].

[13] SpringHispano.org. *Spring MVC* [en línea]. http://www.springhispano.org/?q=taxonomy/term/28 [fecha de consulta: 14/04/2011].

[14] Ubuntu [en línea]. http://www.ubuntu.com [fecha de consulta: abril-junio].

[15] W3schools. *DTD Tutorial* [en línea]. http://www.w3schools.com/dtd/default.asp [fecha de consulta: abril 2011]

[16] XStream. [en línea]. http://xstream.codehaus.org/ [fecha de consulta: abril 2011]

6. Anexo I: Estudio y evaluación de alternativas

6.1. Struts

Struts es uno de los frameworks MVC más consolidados ya que, posiblemente, ha sido uno de los primeros de este tipo, sino el primero, y aún en la actualidad sigue usándose. Fue donado a la Apache Software Foundation durante el año 2000 y en la actualidad sigue totalmente vivo. Existen dos grandes versiones de este gran proyecto, Struts 1 y Struts 2; la segunda nació para suplir algunos errores de diseño que impedían al programador realizar algunas tareas.

En primer lugar, describiremos las características de la primera versión, para a continuación pasar a describir que mejoras ha introducido la segunda.

6.1.1. Struts 1

Para seguir el patrón MVC Struts dispone de un controlador (ActionServlet) encargado de gestionar las peticiones y redirigir a las acciones (Actions) concretas que interactuarán con el modelo –el cual puede estar implementado con cualquier tecnología java– y también, en función de los resultados, mostrar las vistas adecuadas mediante Java Server Pages (JSP). Además, para la interacción con la vista, struts ofrece unas clases formulario (ActionForm) que permiten tanto establecer como obtener datos.

En primer lugar, el controlador consiste en un Servlet implementado por Struts que permite gestionar las peticiones de los usuarios. Para ello es necesario que el desarrollador defina en un fichero de configuración (struts-config.xml) como debe responder para cada petición: que acción debe ejecutarse, que formulario debe usar, donde debe redirigirse en función de la salida y posibles errores, etcétera. En dicho fichero también se definen formularios, tratamiento global de excepciones, redirecciones globales, tratamientos de mensajes, así como otras configuraciones del controlador.

En segundo lugar, tenemos las acciones que son unos de los elementos más importantes de este marco y que deben ser desarrolladas por el programador. En ellas se implementa la respuesta de la aplicación en función de los siguientes parámetros: el mapeo de acciones, el formulario, la petición (request) y el objeto de respuesta (response). Estas acciones son las que interactúan con el modelo (ya sea por JDBC, EJBs u otros mecanismos) y el formulario (que interactúa a su vez con la vista) y una vez finalizadas devuelven una respuesta en forma de ActionForward –normalmente esta respuesta depende directamente de la configuración del controlador–. Además estas acciones pueden acabar también con errores, que serán gestionados por el controlador.

En tercer lugar, hablaremos de los formularios, que es el mecanismo que usa este framework para comunicar datos entre la vista y las acciones. Son básicamente contenedores de datos y se pueden definir de dos formas, con clases que extiendan de ActionForm o bien dentro del fichero struts-config.xml. Usando estos formularios no deberemos preocuparnos de la gestión de los datos recibidos de la vista, no sólo porque Struts ofrece mecanismos para rellenar y establecer dichos datos, sino porque también proporciona una serie de componentes predefinidos para validar estos formularios, que además el programador puede modificar si lo desea.

Finalmente hablaremos de la vista, como hemos comentado, la vista se genera con páginas JSP. Para acceder a los datos sólo es necesario acceder a la sesión, ya que es allí donde se registran los formularios; en el otro caso, para escribir datos de la vista a los formularios, sólo es necesario conocer la nomenclatura de campos de Struts y éste ya se encargará de establecerlos cuando confirmemos la acción. Aún así, no será necesario programar la vista de esta forma tan rudimentaria, ya que, también ofrece una serie de taglibs que permiten rellenar datos y definir formularios usando unas etiquetas propias con una forma similar al código HTML.

Adicionalmente Struts ofrece otras extensiones, como la definición de páginas en trozos mediante lo llamado tiles, facilitando así la modularidad. También permite el uso de expresiones propias de JSTL así como el conjunto de taglibs definidos por estas.

6.1.2. Struts 2

La segunda versión de Struts no nace propiamente de la primera, como cabe esperar, sino de un proyecto conocido como WebWork 2. En esta nueva versión se corrigen unos ciertos errores de diseño de Struts 1, los cuales dificultaba el trabajo de los programadores en ciertas acciones propias del desarrollo de aplicaciones web, incluso en algunos casos no permitían el desarrollo con Struts.

La mejora más importante la podemos apreciar en la definición de las acciones. Éstas ya no extienden de clases abstractas, ahora implementan una interfaz y no de forma obligatoria. Además, ya no dependen del contenedor, por lo que no se requiere el acceso a las variables request y response. Todo esto hace que podamos hacer pruebas automáticas, ya que sólo necesitamos instanciar las acciones, definir las propiedades y ejecutarlas. A parte, las instancias de las acciones ya no son únicas por aplicación, como lo eran anteriormente, y permiten ejecución multihilo así como la gestión de estos hilos.

Otra de las mejoras son las taglibs, donde no sólo integran JSTL, sino que incluyen más opciones que facilitan la implementación de las vistas. Además, Struts 2 no necesita ya las ActionsForm, se pueden usar directamente las propiedades de las acciones o cualquier otro objeto simplemente definiendo los métodos necesarios para establecer y obtener valores.

A parte, no podemos olvidar otros avances tecnológicos, como el uso de anotaciones, que permiten que el programador no tenga que introducir toda la configuración dentro de ficheros XML, sino dentro de las propias clases.

Todas estas mejoras hacen que el desarrollo de aplicaciones sea más simple, más dinámico y más rápido.

6.2. Spring MVC

Spring Framework es un marco de trabajo para el desarrollo de aplicaciones java y da soporte para la implementación de numerosos elementos de la arquitectura de una aplicación. Dentro de su implementación dispone de un apartado para el control de la capa de presentación siguiendo el patrón MVC, llamado Spring MVC. Gracias a la difusión de este marco este módulo ha tenido gran difusión y se usa tanto conjuntamente con el resto del marco, como de forma independiente.

De forma similar al anterior, Spring MVC tiene un controlador central en forma de Servlet que es el encargado de gestionar las peticiones del cliente al servidor. Este controlador es el llamado DispatcherServlet.

La configuración del controlador puede hacerse mediante ficheros XML y mediante anotaciones, de hecho en la documentación casi todos los ejemplos son con anotaciones dentro de cada uno de los componentes. Mediante la configuración se pueden definir los controladores, mapeo de peticiones (donde se pueden definir hasta variables dentro de la petición), el tipo de resultado de las llamadas, el uso de atributos de sesión, interceptores, vistas, etcétera.

En el caso de Spring, la implementación de la lógica de la capa de presentación está implementada en los llamados Controllers (controladores). Estos ocuparían el lugar que ocupaban las acciones en Struts. Al igual que en la segunda versión de Struts, éstos no tienen porque implementar una interfaz concreta o extender de una clase genérica, puede ser cualquier tipo de clase. De la misma manera, para la gestión de datos no se necesitan clases específicas, se puede hacer dentro de los mismos controladores, en sesión, en otros objetos etcétera. La respuesta de estos controladores puede ser desde una petición en forma de cadena, un objeto de la vista, del modelo, de ambas, etcétera.

Las vistas son uno de los aspectos interesantes de este framework, pueden ser de cualquier tipo, no sólo JSP como lo eran en Struts. Por lo tanto, dentro de Spring MVC se puede especificar: cuantas salidas hay, de que tipo se permiten, etcétera.

Como en Struts, también hay soporte para la internacionalización, validación de objetos, conversión de datos y control de errores. Además, también dispone un conjunto de taglibs propios, sin contar que gracias a su diseño es bastante fácil usar taglibs de terceros.

6.3. JavaServer Faces (JSF)

JavaServer Faces es una especificación propia de JEE. Una de las cosas más interesantes es la diferencia en el trato de la presentación como si fueran aplicaciones de escritorio o de dispositivos móviles. Esto contrasta considerablemente respecto al modelo tradicional de aplicaciones web y es gracias a que está orientada a componentes y eventos. El hecho de que sea una especificación permite que haya diferentes implementaciones, lo que facilita desarrollar una aplicación sin atarse a una implementación concreta.

Al igual que el resto de propuestas dispone de un controlador central encargado de gestionar las peticiones, en este caso se llama FacesServlet. Éste también se podrá configurar mediante un fichero XML, en él se indicará la navegación, los componentes, los llamados Managed Beans (objetos usados para el modelo), validaciones, conversores, a parte de otras propiedades de JSF.

Los componentes son uno de los elementos más interesantes, representan las partes que se muestran en la interfaz de usuario: tablas, listas, botones, campos, …, a parte de los que el propio programador puede definir. Para cada uno de ellos se puede definir como se pintan, que eventos disparan, que atributos tienen, entre otros.

Los ManagedBeans contienen los datos que se gestionan en la capa de presentación, los componentes acceden y manipulan los datos de estos objetos. Para la gestión de los datos el controlador hará uso de conversores y validadores previamente configurados.

El funcionamiento de todo esto se basa en la definición de componentes en cada página, donde éstos se vinculan a ManagedBeans y atributos, además, los componentes pueden disparar eventos, que a su vez lanzarán acciones concretas. A todo esto debemos añadir la navegación, similar al resto de controladores anteriormente comentados.

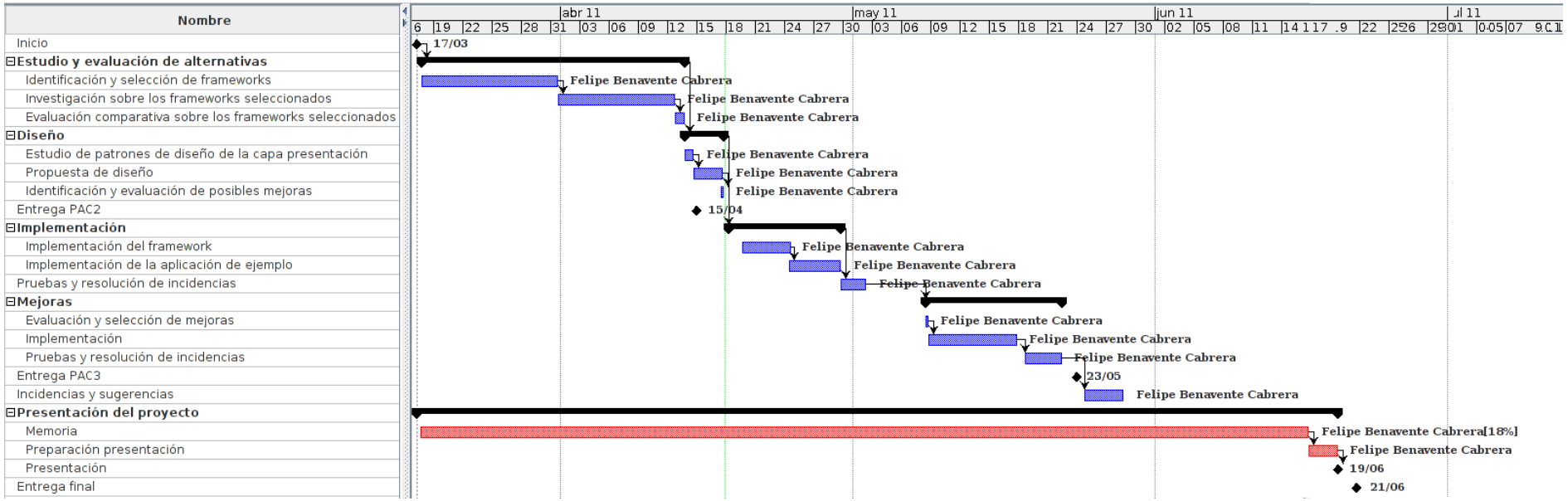
Finalmente, al ser una especificación existe un mercado amplio, que ofrece un número amplio de componentes que pueden agilizar la implementación de diferentes interfaces. Además, existen diversas implementaciones que incluyen AJAX, lo que ayuda a definir interfaces ágiles desde el punto de vista del usuario.

7. Anexo II: Planificación revisada al final del diseño

En azul aparecen aquellas que han sido modificadas en tiempo y subrayadas las tareas que ya se han realizado y por lo tanto su duración ya no es estimada sino real.

Tarea			Tiempo
1	**Estudio y evaluación de alternativas**		
	1.1	Identificación y selección de frameworks	3 horas
	1.2	Investigación sobre los frameworks seleccionados	30 horas
	1.3	Evaluación comparativa de frameworks seleccionados	5 horas
2	**Diseño**		
	2.1	Estudio de patrones de diseño de la capa presentación	2 horas
	2.2	Propuesta de diseño	8 horas
	2.3	Identificación y evaluación de posibles mejoras	3 horas
3	**Implementación**		
	3.1	Implementación del framework	30 horas
	3.2	Implementación de una aplicación de ejemplo	15 horas
4	**Pruebas y resolución de incidencias**		**15 horas**
5	**Mejoras**		
	5.1	Evaluación y selección de mejoras	3 horas
	5.2	Implementación de las mejoras	45 horas
	5.3	Pruebas y resolución de incidencias	15 horas
6	**Incidencias y sugerencias**		**15 horas**
7	**Presentación del proyecto**		
	7.1	Memoria	50 horas
	7.2	Preparación de la presentación	15 horas
	7.3	Presentación	3 horas

Diagrama de Gantt con la planificación revisada del proyecto.



Tal como se desprende del diagrama, en el punto en el cual se presentó todavía se tenía cierto margen de maniobra para conseguir entregar cada una de las partes en los plazos marcados, aunque muy inferior al inicial; más teniendo en cuenta que hemos ampliado la dedicación al máximo, por tanto sólo nos quedan dos estrategias para corregir desviaciones:

- Reducir el alcance del proyecto, en nuestro caso lo haríamos quitando horas del apartado de mejoras.
- O retrasar las fechas de entrega, siempre teniendo en cuenta que la fecha final debe mantenerse.

8. Anexo III: DTD para la definición del fichero de configuración del controlador.

```
<?xml version="1.0" encoding="UTF-8"?>
    <!--
    DTD Con la especificación del documento de configuración del marco de
    presentación creado para el Proyecto Final de Carrera de Felipe Benavente
    Cabrera
    -->

    <!-- Definimos los tipos básicos -->
    <!-- Nombre de clases -->
    <!ENTITY % ClassName "CDATA">
    <!-- Nombre de elementos -->
    <!ENTITY % Name "CDATA">
    <!-- Tipos básicos permitidos -->
    <!ENTITY % Type "(int|double|string)">
    <!--
    Ámbitos de aplicación
    - El ámbito path sólo es aplicable a elementos que contienen rutas
    -->
    <!ENTITY % Scope "(REQUEST|SESSION|MANAGER|PATH)">
    <!ENTITY % Path "CDATA">
    <!ENTITY % Value "CDATA">

    <!--
    A continuación mostraremos los elementos que compondrán este fichero
    -->
    <!--
    Definición central del elemento raíz (pfc-configuration) para el cual se
    indica el conjunto de:
    - controladores que utilizará (managers)
    - Tipos de validaciones (validators)
    - Conversores de datos (converters)
    - Mapeo de peticiones (requestMappings)
    - Forwards en función de errores (errorForwards)
    -->
    <!ELEMENT pfc-configuration (managers?,validators?,converters?,requestMappings?,errorForwards?)>

    <!--
    Definición de controladres (managers), éstos contendrán la lógica de la
    aplicación y para cada uno se indicará la clase que lo implementa así como
    el conjunto de validadores (validate) y conversores (convert) que usará.
    Si no se indica nada usará los conversores y validadores por defecto para
    cada tipo de atributo
    -->
    <!ELEMENT managers (manager*)>
    <!ELEMENT manager (validations?)>
    <!ATTLIST manager className %ClassName; #REQUIRED>
    <!ATTLIST manager id ID #REQUIRED>
    <!--
    Para la obtención de datos de cada manager se puede definir una serie de
    validaciones
    -->
    <!ELEMENT validations (validation*)>
    <!ELEMENT validation (validationParameters?)>
    <!ATTLIST validation idValidator IDREF #REQUIRED>
    <!--
    Parámetro de validación, puede tener un valor concreto o la referencia a una
    variable
    -->
    <!ELEMENT validationParameters (validationParameter*)>
    <!ELEMENT validationParameter EMTPY>
    <!-- Nombre del parámetro (Definido en el validador) -->
    <!ATTLIST validationParameter name %Name; #IMPLIED>
    <!-- Propiedad usada para la validación -->
    <!ATTLIST validationParameter property %Name; #IMPLIED>
    <!-- Ámbito donde entronar la variable, por defecto el request -->
    <!ATTLIST validationParameter scope %Scope; #IMPLIED>
    <!-- Valor -->
    <!ATTLIST validationParameter value %Value; #IMPLIED>

    <!--
    Definición de validadores (validators), éstos son los encargados de validar
    la información enviada en cada petición.
    En esta sección se declaran el conjunto de validadores a usar, para ello se
    indica la clase encargada de controlar la validación y los parámetros de
    configuración, es decir, los parámetros de construcción del objeto.
    Adicionalmente se podrá especificar que método es el encargado de la
    validación y cuales son sus parámetros de entrada.
    -->
    <!ELEMENT validators (validator*)>
    <!ELEMENT validator (initParams*, validatorParameters?)>
    <!ATTLIST validator id ID #REQUIRED>
    <!ATTLIST validator validatorClass %ClassName; #REQUIRED>
    <!ATTLIST validator methodName %Name; #IMPLIED>
    <!-- Parámetros para la llamada del validador (definidos según el orden de aparición) -->
    <!ELEMENT validatorParameters (validatorParameter*)>
    <!ELEMENT validatorParameter EMPTY>
    <!ATTLIST validatorParameter type %ClassName; #REQUIRED>
    <!ATTLIST validatorParameter name %Name; #IMPLIED>
```

```
<!--
Definición de conversores (converter), éstos son los encargados de convertir
la información enviada en cada petición a un conjunto de objetos.

En esta sección se declaran el conjunto de conversores a usar, para ello se
indica la clase encargada de controlar el conversor y los parámetros de
configuración, es decir, los parámetros de construcción del objeto.

Para facilitar su declaración se especificará que clases convertira cada
conversor
-->
<!ELEMENT converters (converter*)>
<!ELEMENT converter (initParams*,converterTypes?)>
<!ATTLIST converter class %ClassName; #REQUIRED>
<!ATTLIST converter id ID #REQUIRED>

<!--Tipos que convierte: paquetes, clases o tipos básicos-->
<!ELEMENT converterTypes (converterType*)>
<!ELEMENT converterType (#PCDATA)>

<!--
Una vez definidos todos los componentes, solo queda pendiente definir el
mapeo de peticiones, para cada petición se puede redirigir a una ruta
concreta o ejecutar simplemente una acción de uno de los managers.

El resultado de estas acciones determinarán donde nos llevará la acción

También se pueden definir una serie de validaciones concretos para la
petición, ya que los valores de ésta pueden ser erróneos
-->
<!ELEMENT requestMappings (requestMapping*)>
<!ELEMENT requestMapping (managerParameters?, validations?, managerForwards?, errorForwards?)>
<!ATTLIST requestMapping path %Path; #REQUIRED>
<!ATTLIST requestMapping managerId IDREF #IMPLIED>
<!ATTLIST requestMapping methodName %Name; #IMPLIED>
<!ATTLIST requestMapping forward %Path; #IMPLIED>
<!--
Parámetros para la ejecución de la acción
-->
<!ELEMENT managerParameters (managerParameter*)>
<!ELEMENT managerParameter EMPTY>
<!ATTLIST managerParameter type %ClassName; #REQUIRED>
<!ATTLIST managerParameter property %Name; #IMPLIED>
<!ATTLIST managerParameter scope %Scope; #IMPLIED>
<!ATTLIST managerParameter value %Value; #IMPLIED>
<!-- Forwards en función de los resultados -->
<!ELEMENT managerForwards (managerForward*)>
<!ELEMENT managerForward EMPTY>
<!ATTLIST managerForward result %Name; #REQUIRED>
<!ATTLIST managerForward forward %Path; #REQUIRED>
<!-- Forwards en función de los errores -->
<!ELEMENT errorForwards (errorForward*)>
<!ELEMENT errorForward EMPTY>
<!ATTLIST errorForward exceptionClass %ClassName; #REQUIRED>
<!ATTLIST errorForward forward %Path; #IMPLIED>

<!-- Parámetros para el inicio  de elementos -->
<!ELEMENT initParams (#PCDATA)>
```