Universitat Oberta de Catalunya

UOC

Ingeniería en Informática

Proyecto Fin de Carrera

Creación de un framework de presentación para aplicaciones JEE

Alumno: Alberto Díaz Martínez

Consultor: Oscar Escudero Sánchez

Resumen

En el presente Proyecto Fin de Carrera (en adelante PFC) vamos a realizar un estudio amplio sobre el uso de *frameworks* de la capa de presentación en el desarrollo de aplicaciones JEE.

JEE es ya una tecnología madura, de la que se posee una amplia experiencia. Esta ya (en términos informáticos) larga historia de aciertos y errores ha permitido que se establezcan una serie de patrones y de buenas prácticas que son ampliamente reconocidos y aceptados por la comunidad.

De estos patrones han ido surgiendo multitud de *frameworks* para proporcionar apoyo a todas las tecnologías incluidas dentro de la especificación JEE. De todos estos marcos de trabajo nos centraremos, como ya se ha dicho, en aquellos que se ocupan de la capa de presentación.

Comenzaremos dando un breve repaso a la historia de JEE. Nos aproximaremos después a los conceptos de patrón y de *framework*, continuaremos con un estudio profundo de los más utilizados del momento y por último abordaremos la construcción de nuestro propio – y modesto – *framework* de presentación para el desarrollo de aplicaciones JEE.

El *framework* que se pretende crear tendrá un ámbito de aplicación limitado: la creación de aplicaciones sencillas orientadas a la presentación de formularios simples dirigidos a las administraciones públicas[1], tales como pago de impuestos o tasas, instancias, solicitudes de información, etc.

Palabras clave proyecto, carrera, uoc, jee, framework, formwork, AJAX, patrones

[1] Se puede ver un ejemplo del tipo de aplicaciones al que nos referimos en `https://www1.agenciatributaria.gob.es/wcl/PAGE-M250/main.zul` (se precisa de certificado de usuario).

Índice general

Índice de figuras

Índice de cuadros

Índice de Listados

Capítulo 1

Plan de trabajo

1.1. PLANIFICACIÓN

La planificación en este proyecto es relativamente sencilla, ya que viene marcada a fuego por las fechas de entrega tanto de cada PEC como de la entrega final, fechas que se pueden considerar como inamovibles. Podemos ver el diagrama de Gantt de esta planificación en el apéndice C.

La primera tarea, PEC1, se corresponde con la elaboración de la propuesta de proyecto. Cumple estrictamente con sus fechas de publicación (29-9-2012) y entrega (3-10-2012).

La tarea PEC2 es una de las más importantes. Incluye dos subtareas, el estudio de los frameworks y patrones y el análisis y diseño del *framework*. Comienza al día siguiente de la entrega de la PEC1 y tiene que estar terminada con la fecha límite de entrega, el 9 de noviembre. Se considera suficiente con cinco días para la tarea de estudio indicadas.

La implementación comenzará inmediatamente después de la entrega anterior, es decir el 12 de noviembre (el programa ProjectLibre cuenta los fines de semana como no útiles, aunque es previsible que se trabaje en sábados y domingos si hay apuros) y debe estar concluida para la fecha de entrega, el 17 de diciembre.

La última tarea, denominada entrega final, se extiende durante toda la vida del proyecto y consiste en la elaboración de la memoria, que se habrá ido escribiendo durante todas las fases anteriores. En realidad,cada una de las entregas de PEC serán una parte de la memoria.

A la vez que la memoria, se obtendrán el resto de productos que componen el *framework* en sí: binarios, código fuente, documentación javadoc y aplicación de ejemplo.

Parte I

JEE

Capítulo 2

JEE

"¿Internet? ¿Todavía anda eso por ahí?"

Hommer Simpson.

Teniendo en cuenta que el objetivo de este PFC es la creación de un *framework* de presentación para aplicaciones web hechas con tecnología JEE no está demás profundizar antes en el conocimiento de lo que es JEE y de su historia. A esta tarea dedicaremos este primer capítulo.

2.1. Una historia de JEE

Podemos situar el origen (o más correctamente, la necesidad) de JEE casi al mismo momento del nacimiento de la plataforma Java, allá por el año 1995. Entonces aparecieron los *applets* que trajeron gran popularidad hacia Java gracias a que Netscape, por aquel entonces el navegador de Internet líder, incluyó soporte para esta plataforma.

Con la novedad se creó una fiebre por convertir todas los sitios web (incluso aquellos que eran estáticos) a *applets* Java y se puede decir que este gran éxito inicial fue también la causa de su derrumbe. La tecnología no estaba madura y pronto surgieron las carencias: problemas de comunicación con el servidor, con el acceso a bases de datos, con la gestión de la concurrencia, con la compatibilidad entre distintos navegadores, ausencia de un API de transacciones etc.

Así, se cambió de paradigma y se decidió trasladar toda la lógica de negocio y la persistencia de la información al servidor; el cliente sería tan ligero como fuese posible y sus únicas misiones serían mostrar información al usuario e interaccionar con él. Había nacido el API de *servlets*.

2.1.1. Servlets

El API de Servlets, aparecida en 1997, junto con el JWS (el antepasado de los servidores de aplicaciones actuales) fueron los primeros ladrillos en la construcción de lo que hoy es JEE. Los *servlets* eran pequeñas piezas de software que se ejecutaban dentro de un servidor. Proporcionaban soporte multihilo (aunque no eran *thread-safe*) y gestión del ciclo de vida.

JWS: *Java Web Server*

Junto con el API de acceso a bases de datos (JDBC), desde el *servlet* se ejecutaba la lógica de negocio y se generaba la salida HTML[1].

EJB: *Enterprise JavaBeans*

JTA: *Java Transaction API*

En el año 1998 aparecieron EJB y JTA . Con estas dos tecnologías, se dotó de transacciones y de seguridad a las aplicaciones empresariales Java y se las acercó a las necesidades de la industria.

2.1.2. EJB

EJB introdujo un modelo de componentes de servidor que vinieron a solucionar el problema de la concurrencia y la consistencia de las transacciones. Cada EJB se ejecuta en un hilo dedicado y no compartido por ningún otro hilo, lo que da la apariencia de *single threaded*.

CORBA: *Common Object Request Broker Architecture.*

Un EJB consiste en una clase Java y dos interfaces: local y remota. Los clientes pueden acceder a la interfaz remota mediante CORBA . La especificación prohíbe expresamente el acceso concurrente a una instancia de EJB, sea éste con o sin estado. El propio contenedor de EJB inicia una transacción antes de ejecutar cualquier método del *bean*, lo que unido al acceso no concurrente garantiza la consistencia de los datos.

Pese a que EJB era una buena solución a los problemas mencionados, en sus versiones 1.x y 2.x eran tan farragosos y complicados de programar y de usar que su éxito en la industria fue muy reducido, en beneficio de otras soluciones como Spring. No obstante, con la aparición de la versión 3.x EJB experimentó un resurgir.

2.1.3. JMS, JNDI...

JMS: *Java Messagin System*

JNDI: *Java Naming and Directory Interface*

Paralelamente fueron surgiendo más y más API: de mensajería (JMS), de directorio (JNDI), de proceso de XML etc. Comenzó a plantearse la posibilidad de reunir todas estas especificaciones independientes dentro de un mismo estándar.

Fue así como surgió la primera especificación de J2EE o *Java 2 Enterprise Edition*, para diferenciarla de J2SE o *Java 2 Standard Edition*. Con el tiempo, ambas perderían el "2" para quedar en JEE y JSE respectivamente.

2.2. Qué es JEE

JCP: *Java Community Process*

JSR: *Java Specification Request*

Digamos para terminar que JEE no es más que una colección de especificaciones reunidas bajo un mismo estándar. Todas estas especificaciones las publica el JCP una organización sin ánimo de lucro integrada por empresas, fundaciones y particulares que se dedica a publicar estándares para la plataforma Java. Estos estándares se publican bajo la forma de JSR.

Para cada JSR el JCP publica una especificación, una implementación de referencia y un test de compatibilidad que deberán superar todas las implementaciones del estándar. Para JEE, la

[1] En 4.1.1 estudiaremos los modelos de arquitectura de aplicaciones con servlets

última versión publicada es la 6 —JSR 316— y la implementación de referencia es el servidor de aplicaciones Glassfish.[2]

Se puede ver una lista de las tecnologías que se incluyen con JEE 6 en el cuadro 2.1

Web Services Technologies	
Java API for RESTful Web Services (JAX-RS) 1.1	JSR 311
Implementing Enterprise Web Services 1.3	JSR 109
Java API for XML-Based Web Services (JAX-WS) 2.2	JSR 224
Java Architecture for XML Binding (JAXB) 2.2	JSR 222
Web Services Metadata for the Java Platform	JSR 181
Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101
Java APIs for XML Messaging 1.3	JSR 67
Java API for XML Registries (JAXR) 1.0	JSR 93
Web Application Technologies	
Java Servlet 3.0	JSR 315
JavaServer Faces 2.0	JSR 314
JavaServer Pages 2.2/Expression Language 2.2	JSR 245
Standard Tag Library for JavaServer Pages (JSTL) 1.2	JSR 52
Debugging Support for Other Languages 1.0	JSR 45
Enterprise Application Technologies	
Contexts and Dependency Injection for Java (Web Beans 1.0)	JSR 299
Dependency Injection for Java 1.0	JSR 330
Bean Validation 1.0	JSR 303
Enterprise JavaBeans 3.1 (includes Interceptors 1.1)	JSR 318
Java EE Connector Architecture 1.6	JSR 322
Java Persistence 2.0	JSR 317
Common Annotations for the Java Platform 1.1	JSR 250
Java Message Service API 1.1	JSR 914
Java Transaction API (JTA) 1.1	JSR 907
JavaMail 1.4	JSR 919
Management and Security Technologies	
Java Authentication Service Provider Interface for Containers	JSR 196
Java Authorization Contract for Containers 1.3	JSR 115
Java EE Application Deployment 1.2	JSR 88
J2EE Management 1.1	JSR 77
Java EE-related Specs in Java SE	
Java API for XML Processing (JAXP) 1.3	JSR 206
Java Database Connectivity 4.0	JSR 221
Java Management Extensions (JMX) 2.0	JSR 255
JavaBeans Activation Framework (JAF) 1.1	JSR 925
Streaming API for XML (StAX) 1.0	JSR 173

Cuadro 2.1: Tecnologías JEE

[2] `http://www.oracle.com/technetwork/java/javaee/downloads/index.html`

Capítulo 3

DE PATRONES Y FRAMEWORKS

> *"Es importante destacar que ningún ingeniero de software civilizado escribiría un procedimiento llamado* `DestruirBaghdad`*. Su ética le obligaría a escribir un procedimiento* `DestruirCiudad` *al que se pasaría como parámetro* `Baghdad`*."*
>
> ***Nathaniel S. Borenstein***

En este capítulo y el siguiente vamos a ofrecer un repaso de algunos de los *frameworks* de presentación existentes para el desarrollo de aplicaciones JEE. Comenzaremos analizando el concepto de **patrón** -pues es en patrones en lo que están basados los ***frameworks***-, repasaremos los patrones que se usan en la mayoría de los *frameworks* más utilizados y por último haremos un estudio comparativo en profundidad de algunos de ellos.

3.1. Concepto de patrón

Fue el arquitecto Christopher Alexander quien aportó el concepto de patrón en su libro *The timeless way of building*, en el que proponía el aprendizaje de una serie de patrones para construir edificios de mayor calidad. Sobre los patrones dice el Sr. Alexander: [1]

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Posteriormente, esta definición fue adoptada por el GoF para definir un patrón de diseño en el desarrollo del software: [10]

GoF: *se refiere a Gang of Four o banda de los cuatro: Gamma-Helm-Johnsson-Vlissides*

> "Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context."

Es decir, que un patrón es una descripción de una solución, que ha sido probada con éxito, a problemas comunes y conocidos que se reproducen una y otra vez en el ámbito de una actividad concreta (la construcción de edificios o la construcción de software por ejemplo). El patrón se describirá en un lenguaje adecuado a dicha actividad (el lenguaje UML en la construcción de software).

El libro de GoF [10] se convirtió pronto en un gran éxito y se considera como la guía de referencia del diseño orientado a objetos. El catálogo de patrones que presenta el libro se divide en tres grupos

	Propósito		
	De creación	**Estructurales**	**De comportamiento**
De clase	Factory Method	Adapter	Interpreter Template method
De objeto	Abstract factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of responsability Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Cuadro 3.1: Catálogo de patrones GoF.

De este catálogo de patrones GoF se usarán algunos en el *framework* que vamos a desarrollar en este PFC: Factory Method, Strategy, Observer y Composite.

Pero el catálogo de patrones no se reduce solo al definido en el libro de GoF. En JEE existe un catálogo de patrones específico, alguno de los cuales están basados en, o incluyen a, los patrones GoF. Este catálogo es conocido como *Core JEE Patterns* [2]. Podemos ver un resumen de estos patrones en la figura 3.1.

Muchos *frameworks* incorporan varios de estos patrones en su construcción. Daremos aquí un repaso al patrón *Front Controller*.

Además, hablaremos de un par de patrones de arquitectura, no especificamente de JEE, pero que se usan de manera generalizada. Los patrones MVC (ver 3.1.1) y, en menor medida, MVVM (ver 3.1.2).

3.1.1. El patrón MVC

El patrón MVC (Modelo-Vista-Controlador o *Model-View-Controller* en su denominación inglesa) es uno de los que más éxito han tenido. Es ya un patrón antiguo, surgido de los creadores del lenguaje SmallTalk en los años 80. Este patrón establece una clara separación entre los datos (el Modelo) la lógica de negocio (el Controlador) y la presentación (la Vista). El dibujo clásico de este patrón se puede ver en la figura 3.2

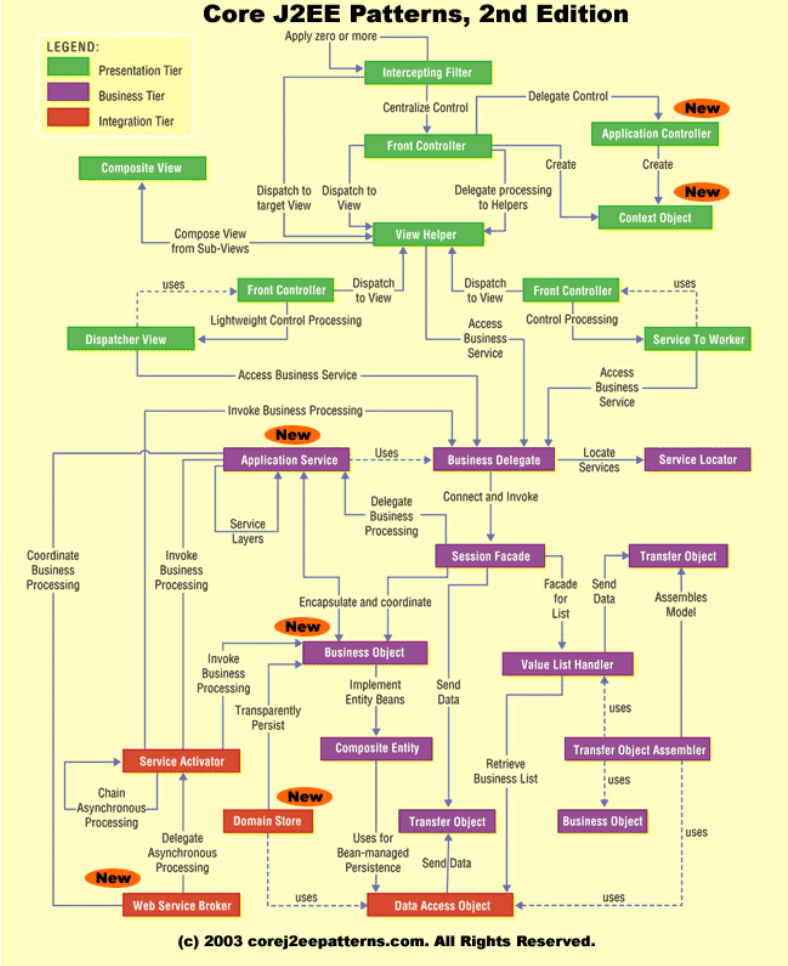

Figura 3.1: Core JEE patterns Fuente: http://corej2eepatterns.com/Patterns2ndEd/

- **La Vista** interacciona con el usuario, se encarga de trasladar los gestos del usuario de la aplicación hacia el controlador. También muestra los cambios producidos en el modelo.

- **El Controlador** recibe notificación de las acciones de usuario desde la vista, habitualmente en forma de eventos. En función del evento, decide que cambios se deben de hacer sobre el modelo. Opcionalmente, selecciona la siguiente vista.

- **El Modelo** se identifica con los datos de la aplicación. Cambia su estado en respuesta a las acciones del controlador. Puede informar a la vista de que su estado ha cambiado mediante un evento, para que ésta actúe en consecuencia.

Sobre este dibujo clásico se admiten diversas variaciones, pero en esencia, el patrón permanece inalterable. La práctica totalidad de los *frameworks* de la capa de presentación para aplicaciones JEE incorpora la infraestructura necesaria para aplicar de forma natural el patrón MVC a las aplicaciones con ellos construidas.

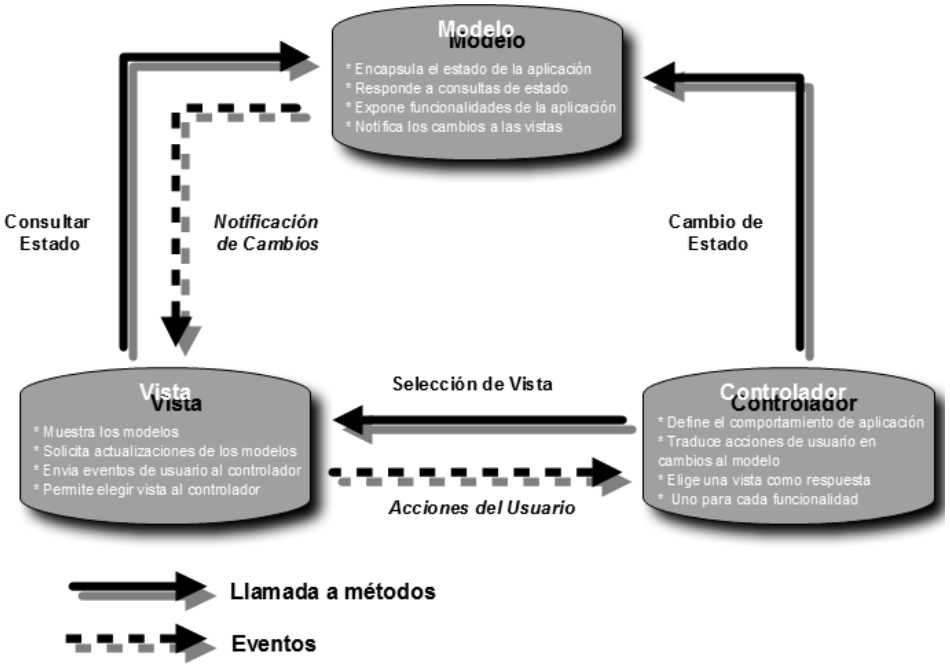

Figura 3.2: El patrón MVC Fuente: http://doviedo.rianetworks.net/PLANTILLA%20WEB/design.html

3.1.2. El patrón MVVM

El patrón MVVM (*Model-View-ViewModel*), extraído del catálogo de patrones de Microsoft[1], es una especialización del patrón *Presentation Model*[2] de Martin Fowler que a su vez es una variación del patrón MVC (ver 3.1.1).

En este patrón el modelo y la vista interpretan los mismos papeles que en el MVC clásico. La diferencia estriba en el *ViewModel*, un controlador especializado que mantiene la información y el estado de la vista, pero está absolutamente desacoplado de ella.

El *ViewModel* supone en un nivel más de abstracción sobre la vista, de modo que no conoce nada sobre los componentes gráficos que la forman, solamente almacena sus datos y sus estados, como ya se ha dicho. Mediante este nivel extra de abstracción, las vistas pueden ser diseñadas enteramente por diseñadores y los *ViewModel* por programadores de forma paralela y un mismo *ViewModel* puede ser aplicado a diferentes vistas. Dicho de otro modo, un vista se puede definir como una proyección del *ViewModel* en el interfaz gráfico de la aplicación.

Ya que la vista y su "controlador" (su *ViewModel*) están completamente desacoplados, el patrón MVVM precisa de un actor más para su correcto funcionamiento: un mecanismo de enlace de datos o *data binding* que asocie los componentes gráficos de la vista con los datos del *ViewModel*. Se puede ver un esquema del patrón en la figura 3.3.

MVVM aporta algunas ventajas con respecto al MVC tradicional:

- Se adapta bien al "diseño por contrato".
- Menor acoplamiento con la vista

[1] `http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx`
[2] `http://martinfowler.com/eaaDev/PresentationModel.html`

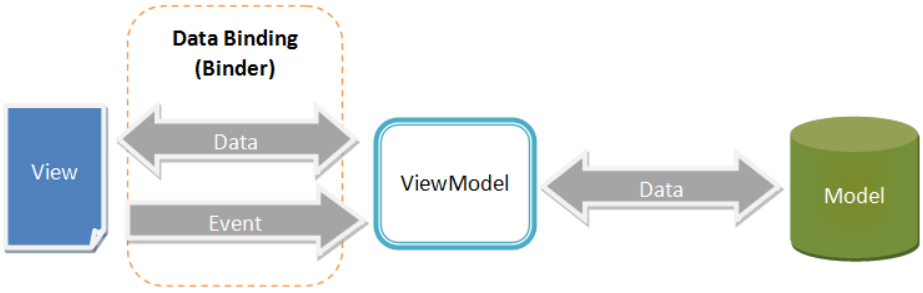

Figura 3.3: El patrón MVVM Fuente: http://books.zkoss.org/wiki/ZK%20Developer%27s%20Reference/MVVM

- Mejor reusabilidad.
- Es más fácil realizar pruebas unitarias.

Uno de los *frameworks* que estudiaremos en el siguiente capítulo incorpora soporte para MVVM, concretamente el *framework* ZK (ver 4.2.2).

3.1.3. El patrón *Front Controller*

El patrón *Front Controller* consiste en mantener un único controlador centralizado encargado de recibir las peticiones del usuario y despacharlas adecuadamente de acuerdo a reglas de navegación, servicios de autenticación etc.

Todos los *frameworks* de presentación que vamos a ver aplican este patrón y casi todos ellos lo hacen asignado ese papel de controlador centralizado a un *servlet* (ver 4.1.1).

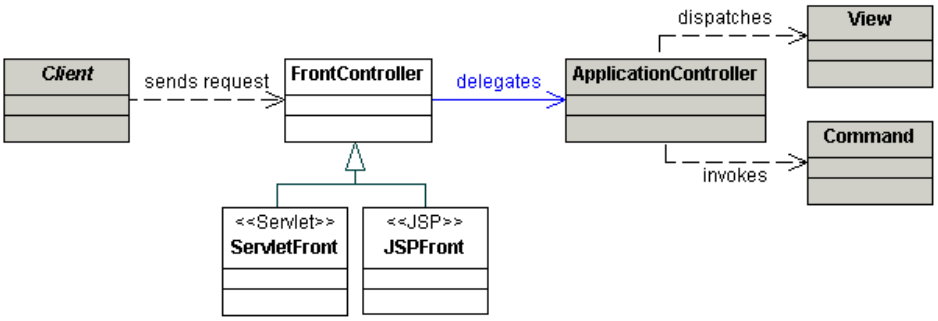

Figura 3.4: Front Controller. Diagrama de clases Fuente: http://corej2eepatterns.com/Patterns2ndEd/FrontController.htm

3.2. Concepto de *framework*

Para aproximarnos al concepto de *framework* vamos a actuar como legos en la materia y acudiremos a la popular Wikipedia:

> "La palabra inglesa «framework» define, en términos generales, un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática

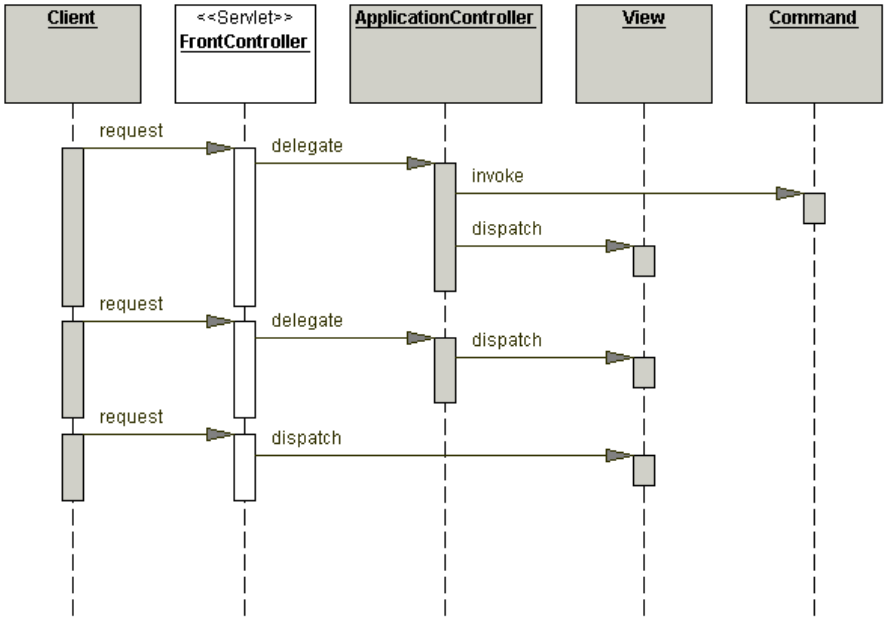

Figura 3.5: Front Controller. Diagrama de secuencias Fuente: http://corej2eepatterns.com/Patterns2ndEd/FrontController.htm

particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar."

En términos que todo el mundo entienda, un *framework* es un conjunto de herramientas y procedimientos que se han empleado con éxito en la solución de un tipo de problema determinado y que pueden reutilizarse para resolver con mayor seguridad, productividad y con un menor esfuerzo otros problemas similares.

Centrándonos ya en el ámbito que nos interesa, el desarrollo de *software*, la definición que nos proporciona la Wikipedia continúa así:

> "En el desarrollo de software, un *framework* o infraestructura digital, es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, con base a la cual otro proyecto de software puede ser más fácilmente organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.
>
> Representa una arquitectura de software que modela las relaciones generales de las entidades del dominio, y provee una estructura y una especial metodología de trabajo, la cual extiende o utiliza las aplicaciones del dominio."

Es decir, un *framework* para el desarrollo de software es un conjunto de bibliotecas (de bibliotecas de clases ya que estamos hablando de JEE y, por lo tanto, de POO) y unas reglas para su uso que siguen unos patrones concretos y que configuran una arquitectura que fuerza a las aplicaciones construidas con su apoyo al seguimiento de unas buenas prácticas que facilitan enormemente la creación y mantenimiento de aplicaciones, abstrayendo al desarrollador de los

aspectos más áridos de la tecnología y permitiendo que éste se centre más en los aspectos de negocio.

Porque en definitiva se trata de eso, de que el esfuerzo que un desarrollador tenga que emplear para aprender el uso de un determinado *framework* —y en el mercado existen decenas de ellos para cada problema común y concreto— se vea ampliamente compensado por el aumento en la productividad y en la calidad de las aplicaciones construidas con él. Un *framework* es, en definitiva, nada más —y nada menos— que una herramienta que nos facilita la vida en nuestro trabajo.

Capítulo 4

LOS FRAMEWORKS JEE

"Todas las piezas deben unirse sin ser forzadas. Recuerde que los componentes que está reensamblando fueron desmontados por usted, por lo que si no puede volver a unirlos debe existir una buena razón. Pero sobre todo, ¡no use un martillo!"

Manual de mantenimiento de IBM, año 1925.

Ya en el capítulo anterior vimos lo que era un patrón de diseño y el catálogo de patrones más utilizados en las aplicaciones JEE. Es a partir de estos patrones y de su comprobada utilidad en muchas aplicaciones creadas a lo largo del tiempo que surge la necesidad de automatizar su uso y se crean los primeros *frameworks* JEE.

En este capítulo daremos un repaso por los *frameworks* más utilizados en la capa de presentación. Algunos son estándares *de iure*, otros *de facto* y otros no han alcanzado aún ninguno de esos estatus.

4.1. Los *Frameworks* clásicos

Dentro de esta categoría incluimos los *frameworks* tradicionales de petición-respuesta síncrona. Es decir, aquellos en que el usuario interacciona con la aplicación y en algún momento se produce una petición al servidor, que la procesa, construye una página de respuesta desde cero y la devuelve al usuario. Durante todo este tiempo, la interfaz gráfica ha permanecido bloqueada.

Dentro de esta apartado estudiaremos Servlets-JSP, Struts y JSF 1.x.

4.1.1. *Servlets* - JSP

En el principio fueron los *servlets*

Sun Microsystems fue adquirida por Oracle en el año 2009

Unos años después de la creación del lenguaje Java, en la tristemente desaparecida Sun Microsystems se dieron cuenta de que su previsión inicial sobre como sería el dominio de Java en Internet – basada en el uso de *applets* – era profundamente equivocada.

De aquella primera idea de aplicaciones Java corriendo en el lado del cliente e incrustadas dentro del navegado web se pasó a crear pequeñas piezas de software que se ejecutarían en el servidor, dentro de un "contenedor"[1] de aplicaciones. Acababan de nacer los *servlets*.

La primera especificación de *servlets* (versión 1.0) aparece en el año 1997, aunque no es hasta la versión 2.3 (aparecida en el año 2002) que se convierte en estándar del JCP[2]– JSR-53 –. En el momento de escribir el presente documento la última versión publicada es la especificación de *servlets* 3.0 – JSR-315 –.

En un primer momento, la labor principal encomendada a un *servlet* era la de generar contenido dinámico, casi siempre en lenguaje HTML. Nada nuevo con respecto a lo que hacían los tradicionales programas CGI escritos en otros lenguajes como C, Perl o Python. Pero los *servlets* aportaban alguna ventaja con respecto a estos.

Por un lado las ventajas del propio lenguaje de programación Java como seguridad, *garbage collector*, etc. Por otro lado, el contenedor de *servlet* proporciona gestión del ciclo de vida (ver fig. 4.1) y control de la concurrencia. Ni que decir tiene que la práctica totalidad de los *frameworks* que veremos posteriormente están basados en *servlets*.

Desde el punto de vista de un desarrollador, un *servlet* no es más que una clase que implementará una interfaz del API de *servlets*: `javax.servlet.Servlet`

Aunque lo más habitual es heredar directamente de alguna de las clases abstractas:

- `javax.servlet.GenericServlet`: Para implementar *servlets* genéricos.
- `javax.servlet.http.HttpServlet`: Para implementar *servlets* basados en el protocolo HTTP.

El ciclo de vida de un *servlet*

Como hemos dicho en el punto anterior, una de las ventajas de los *servlets* es disponer de un ciclo de vida gestionado por el contenedor. Este ciclo de vida se puede ver en la figura 4.1.

En el momento de recibirse la primera petición se instancia el *servlet* y se ejecuta su método `init`. En este método se recogerán los parámetros de inicio y se crearán los recursos (por ejemplo, conexiones con fuentes de datos) que se necesiten. Después se dará servicio a la petición en el método `service`. Cuando el servidor decida eliminar el *servlet* del contenedor invocará a su método `destroy` para que se liberen los recursos utilizados.

En las siguientes peticiones se ejecutará directamente el método `service`. En el caso de que, como es habitual, se trate de un *servlet* HTTP este método `service` despachará la petición a los métodos `doGet` o `doPost` en función del método HTTP usado[3].

[1] La idea de contenedor de aplicaciones no es nueva, ya se venía utilizando desde hace tiempo, por ejemplo por parte de IBM con su CICS.

[2] Ver http://jcp.org

[3] tambien existen métodos como `doHead` o `doPut` para el resto de métodos del protocolo HTTP.

Cabe destacar que cada petición se ejecutará **en un nuevo hilo**, sin instanciar nuevos objetos[4] ni crear un nuevo proceso.

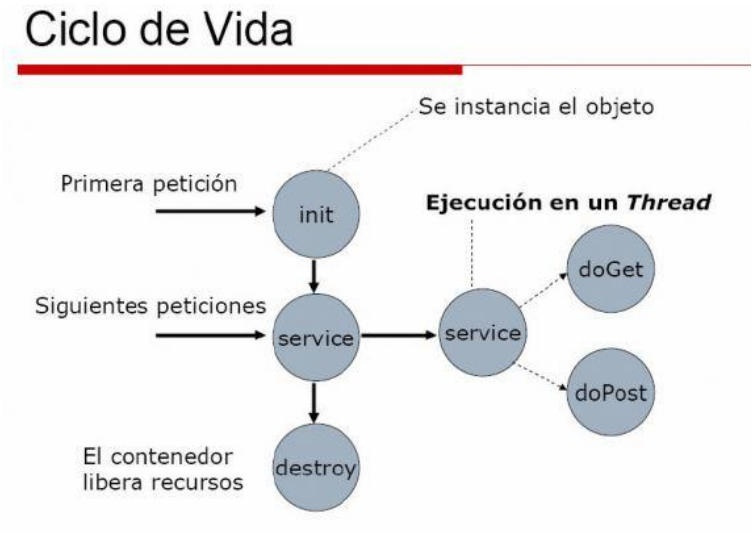

Figura 4.1: El ciclo de vida de un *servlet* Fuente: http://jtagua.wordpress.com/2010/10/31/tutorial-de-servlet-1-introduccion-ciclo-de-vida-y-ejemplo-basico/

El Modelo 1 de programación

En las primeras aplicaciones creadas con *servlets* se hacía todo el trabajo – lógica de negocio y de presentación – en el método `service` (o en los métodos `doPost` o `doGet` si se trataba de *servlets* HTTP):

Listado 4.1: Servlet doPost

```
public void doPost(HttpServletRequest req, HttpServletResponse res) {
    String nombre = req.getParameter("nombre");
    Writer w = response.getWriter();
    w.print("<html><body><h1>Hola␣" + nombre + "</h1></body></html>");
    w.close();
}
```

Pronto se vio que con esta estrategia la aplicaciones, a medida que iban aumentando en tamaño y complejidad, se volvían inmantenibles. Como respuesta a este problema se idearon las JSP (*Java Server Pages*). En esencia, se trataba de volver los *servlets* del revés. Se tenía una página con contenido estático (generalmente HTML) en la que se "incrustaba" código Java como se puede ver en el listado 4.2.

Estas páginas se compilaban en el contenedor (bien en el despliegue de la aplicación, bien con su primera ejecución) y se transformaban en un *servlet*. Esta solución es lo que se conoció como el **Modelo 1** de programación.

Sin embargo, con JSP quedaron descontentos tanto los diseñadores de páginas como los desarrolladores. Se seguía teniendo una mezcla de lógica de negocio y de presentación, los diseñadores se encontraban su precioso HTML lleno de cosas raras en un lenguaje de programación que no entendían y otro tanto podía decirse de los programadores Java.

[4] Aunque en función de la carga de la aplicación y de la propia implementación del contenedor éste podría decidir levantar otra instancia del *servlet* para distribuir la carga.

Listado 4.2: JSP

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>GlassFish JSP Page</title>
  </head>
  <body>
    <h1>Hola <%= request.getParameter("nombre") %></h1>
  </body>
</html>
```

El Modelo 2 de programación

Visto el estado de las cosas, se hacía necesario un cambio. Entonces se optó por aplicar una de los patrones de diseño que ya hemos visto: el patrón Modelo-Vista-Controlador (ver 3.1.1)

Los *servlets* eran unos magníficos candidatos para el rol de controlador. Lo misma cosa podía decirse de las páginas JSP en el papel de las vistas. Esta nueva arquitectura se denominó **Modelo 2 de programación**.

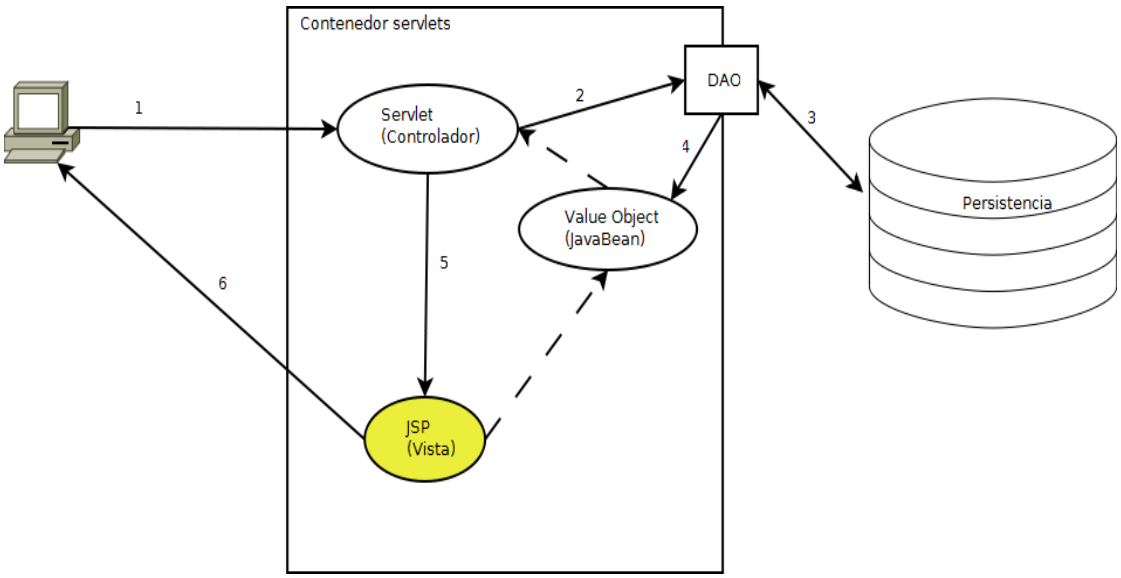

Figura 4.2: *servlet* + JSP + MVC

CRUD: *Siglas de Create, Read, Update, Delete.*

En la figura 4.2 se muestra la arquitectura de una típica aplicación CRUD a base de *servlets* y JSP donde se ha aplicado el patrón MVC (entre otros). El *servlet* (o *servlets* ya que puede haber más de uno) recibe las peticiones y las procesa actuando con la capa de persistencia mediante objetos DAO. El *servlet* construye una colección de *beans* (*value objects*) que encapsula dentro del objeto petición, selecciona la siguiente vista (página JSP) a mostrar al cliente y le pasa el control. La pagina JSP crea la salida HTML que se envía al cliente como respuesta.

Para asegurar la máxima falta de acoplamiento entre capas, se deberían de cumplir las siguientes directrices:

- El *servlet* no debería tener contacto con la capa de persistencia sino a través de objetos DAO. Al controlador no le interesa nada donde están almacenados los datos ni como se accede a ellos.

- Los objetos `Request` y `Response` no debería salir nunca del ámbito del contenedor web. Ni los DAO ni la capa de persistencia están interesados en conocer quien hace las peticiones.

- Las vistas ni siquiera deberían de tener conocimiento de la existencia de los DAO. Solo están interesadas en los *value objects*.

Esta arquitectura se hizo popular y se empezó a utilizar de forma masiva. Los desarrolladores se dieron cuenta de que había una gran cantidad de tareas tediosas que se repetían invariablemente de un proyecto a otro: reglas de navegación, despacho de peticiones etc. Comenzaron a surgir nuevos *frameworks* orientados a la aplicación del patrón MVC en las aplicaciones web. De todos ellos, el que más éxito tuvo fue sin duda Struts.

4.1.2. Struts

Struts es un *framework* basado en *servlets* que fuerza a los desarrolladores a aplicar el patrón MVC y que durante mucho tiempo se convirtió en el estándar *de facto* para el desarrollo de aplicaciones web con JEE.

Fue creado originalmente por Craig McClanahan[5], quien lo donó a la Fundación Apache en mayo del año 2000. Al hablar de Struts estamos referiéndonos en realidad a dos *frameworks* distintos, Struts 1 y Struts 2, nacido de la fusión de Struts con Webworks.

En honor a su importancia en la historia de las aplicaciones web y a las cientos de aplicaciones que con él se han construido haremos un breve repaso a Struts 1 para centrarnos posteriormente y con más detenimiento en Struts 2.

Struts 1

La versión 1 de Struts se creó con una aproximación *page centric*, basada en formularios web clásicos, en los que para cada petición se ejecuta la lógica de negocio correspondiente y se construye una página de respuesta desde cero que se envía al cliente para que la repinte entera.

Como se dijo en el punto anterior, Struts fuerza (o al menos trata de forzar) al desarrollador a seguir el patrón MVC. Para ello, nos proporciona un extenso API, un *framework* de validación (Struts Validator) un *framework* de plantillas para construir el *layout* de la aplicación (Struts Tiles) y una colección de librerías de etiquetas JSP.

Struts nos da pues soporte para la V (vistas) y la C (controlador) del patrón MVC mientras que nos deja total libertad para la implementación de la M (el modelo).

[5] Además de creador de Struts, McClanahan es el colider de la especificación de JSF y creador del contenedor de *servlets* Catalina, incluido en Tomcat que es a su vez la implementación de referencia de *servlets*

Las vistas en Struts 1 se crean mediante páginas JSP (aunque se soportan otros tipos de *frameworks* de plantilla como Velocity). Para construir estas páginas también se nos proporcionan unas librerías de etiquetas.[6]

- **Bean:** para la creación y manipulación de Java Beans
- **HTML:** para la creación de formularios HTML
- **Logic:** para generar salida de texto condicionada, recorrido de colecciones para generar texto repetitivo etc.
- **Nested:** Extensión de las etiquetas para relacionar unas con otras en una forma anidada.

Para implementar el controlador Struts utiliza el patrón *Front controller* (ver 3.1.3) mediante el *servlet* `org.apache.struts.action.ActionServlet`. Se puede decir que este *servlet* es el verdadero corazón de Struts y se debe configurar en el `web.xml` de la aplicación, tal y como se muestra en el listado 4.3.

Listado 4.3: Configuración Struts

```
<servlet>
   <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
       <param-name>config</param-name>
         <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
        <param-name>debug</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>detail</param-name>
        <param-value>2</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Una vez configurado, el *servlet* recibirá todas las peticiones y decidirá a que clase `Action` pasar el control. Estas clases tienen que heredar de la clase `org.struts.action.Action` e implementarán la lógica de negocio de la aplicación.

El controlador decide a cual de las acciones debe invocar en función de un fichero de configuración denominado `struts-config.xml` como el que se puede ver en el listado 4.4 y que entre otras muchas opciones configura las reglas de navegación de la aplicación.

Mediante este *snippet* de código se está indicando que cuando se reciba una petición a la URI `login.do` se debe trasladar la petición a la clase `myapp.action.LoginAction`.

[6]No obstante, Struts recomienda utilizar la librería estándar JSTL siempre que sea posible.

Listado 4.4: struts-config.xml

```
<form-bean name="loginForm" type="myapp.form.LoginForm"/>

<action path="/login"
          type="myapp.action.LoginAction"
        name="loginForm">
   <forward name="logged" path="/index.jsp" />
   <forward name="notlogged" path="/login.jsp" />
</action>
```

Esta clase recibirá los parámetros de la petición encapsulados en un objeto de la clase `myapp.form.LoginForm`. Si el proceso de la petición termina correctamente se pasará a la página `index.jsp` o a la página `login.jsp` en caso contrario.

Podemos considerar los objetos `Action` como componentes del controlador, a modo de *plug-ins*, que habremos de programar sobreescribiendo alguno de los métodos `execute`, generalmente el adaptado al protocolo HTTP. Estamos implementando el patrón estrategia.

Listado 4.5: Clase Action

```
public class LoginAction extends Action {

    /** Creates a new instance of LoginAction */
    public LoginAction() {
    }

    @Override
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
            throws LoginException, ServicioException {

        Servicio servicio = createServicio();


        LoginForm loginForm = (LoginForm) form;
        String usuario = loginForm.getUsuario();
        String clave = loginForm.getClave();

        UsuarioVO usuarioVo = null;

        usuarioVo = servicio.login(usuario, clave);

        if (usuarioVo == null) {
            return mapping.findForward("notlogged");
        }

            request.getSession().setAttribute("user", usuarioVo);

        return mapping.findForward("logged");
    }
}
```

Toda esto ha sido una descripción más o menos informal del funcionamiento del *framework*. En el diagrama de clases (figura 4.3 y de secuencias (figura 4.4) queda explicado de un modo mucho más formal.

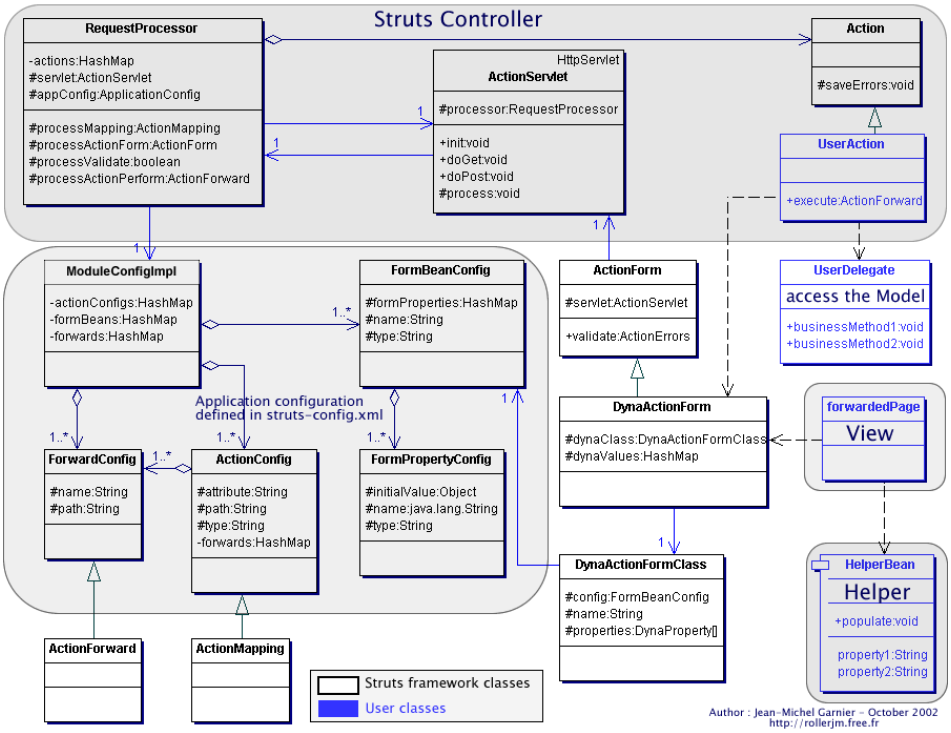

Figura 4.3: Struts. Diagrama de clases Fuente: http://rollerjm.free.fr/pro/Struts11.html

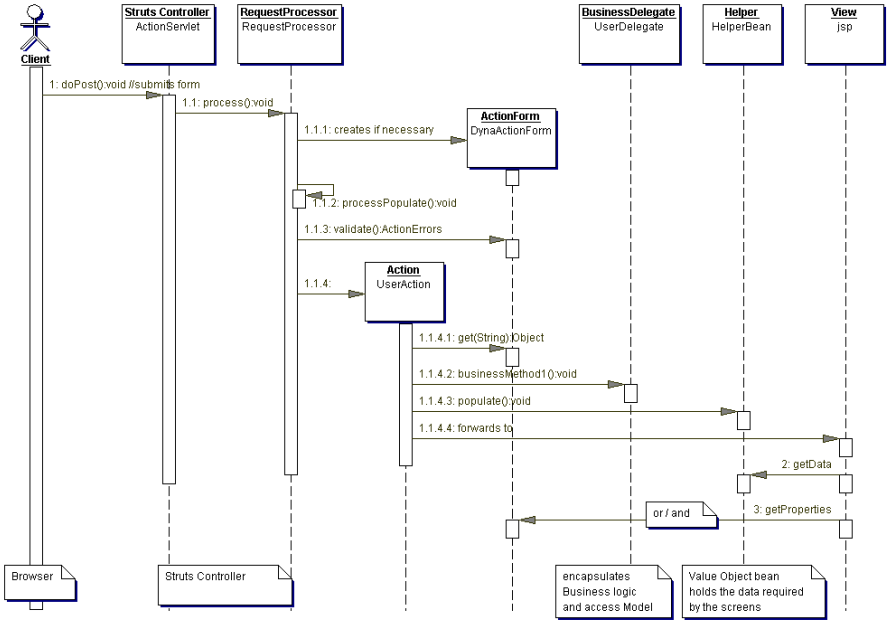

Figura 4.4: Struts. Diagrama de secuencias Fuente: http://rollerjm.free.fr/pro/Struts11.html

Struts 2

Struts 2 no es una evolución de Struts 1. Se trata en realidad de un *framework* completamente nuevo, nacido de la fusión de Struts con WebWork, *framework* que el creador de Struts consideraba que era superior al suyo.

Struts 2 no se basa en *servlets* sino en filtros, interceptores, acciones y resultados. Además incorpora algunas capacidades AJAX, inyección de dependencias (patrón IoC) y nos ofrece la posibilidad de evitar los incómodos ficheros de configuración XML de Struts 1 mediante el uso de anotaciones Java.

El primer gran cambio en Struts 2 con respecto a Struts 1 es que no tenemos un *servlet* que haga de *front controller* sino que disponemos de un filtro de *servlet* por el que pasarán todas las URL de la aplicación (patrón *filter*). En el listado 4.6 se muestra un ejemplo de archivo `web.xml` de configuración de una aplicación para usar Struts 2.

Listado 4.6: Struts 2. Configuración

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">
        <display-name>HelloStruts2</display-name>

        <filter>
                <filter-name>struts2</filter-name>
                <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
        </filter>

        <filter-mapping>
                <filter-name>struts2</filter-name>
                <url-pattern>/*</url-pattern>
        </filter-mapping>
</web-app>
```

Como cabía esperar, Struts 2 también está basado en el patrón MVC. En la figura 4.5 podemos ver como se implementa este patrón en Struts 2:

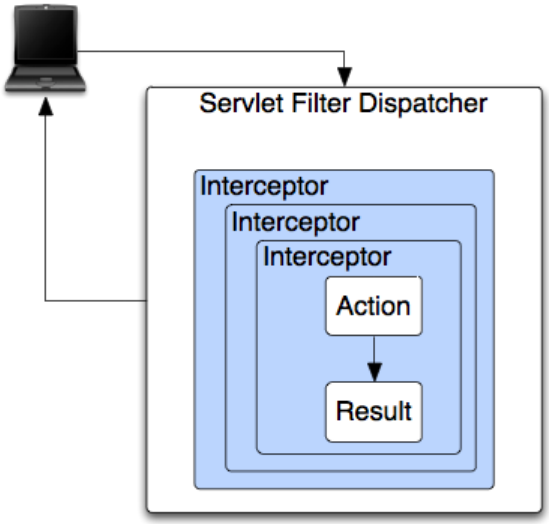

Figura 4.5: Struts 2. Arquitectura Fuente: http://struts.apache.org/2.2.1/docs/nutshell.html

En Struts 2, el controlador está formado por el filtro y las clases `Action` — que se pueden considerar como componentes del controlador —. Estas clases deben implementar la interfaz `com.opensymphony.xwork2.Action` aunque lo más habitual es que hereden de la clase `com.opensymphony.xwork2.ActionSupport`.

Pero al contrario que en la versión 1 de Struts, a estas clases no van a llegar ni un *form bean* ni los objetos `request` ni `response` y el método llamado por el *framework* podrá ser cualquier método sin parámetros que devuelva una cadena. En el listado 4.7 se puede ver un ejemplo muy sencillo de una de estas clases.

Listado 4.7: Struts 2. Clase Action

```
import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Hola extends ActionSupport {
        private String nombre;

        public String getNombre() {
            return nombre;
        }

        public void setNombre(String nombre) {
            this.nombre = nombre;
        }

        public String execute() {
            System.out.println(nombre);
            return SUCCESS;
        }
}
```

De forma similar a como se hacía en Struts 1, las acciones se configuran en un fichero, en este caso denominado `struts.xml`. Podemos ver un ejemplo en el listado 4.8

Listado 4.8: Struts 2. struts.xml

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

    <struts>
        <package name="mi-paquete" extends="struts-default">
            <action name="Hola" class="Hola">
                <result name="success">/hola.jsp</result>
            </action>
        </package>
    </struts>
```

Con esta configuración indicamos que la URL `Hola.action` se dirigirá al método `execute` de la clase `Hola`[7]. Según el valor devuelto por el método `execute` (denominado resultado) el *framework* seleccionará la siguiente vista (en el ejemplo solamente hay un posible resultado —`SUCCESS`— que redirecciona a la página `hola.jsp`).

Pero la parte más interesante quizá se encuentre en el empleo de interceptores. La petición, antes de llegar a la clase de acción y después de que ésta la procese, atravesará una serie de objetos interceptores que se encargarán de realizar diferentes tareas: extraer parámetros de la petición, validar las entradas de los usuarios etc.

Struts 2 define una pila de interceptores por defecto de forma transparente que lo más normal es que nunca tenga que ser modificada. No obstante, si fuese necesario podríamos definir nuestra propia pila de interceptores.

Debido a la gran cantidad de clases involucradas en una petición de Struts 2 no es posible ver un diagrama de clases o de secuencia que quepa en una sola imagen. No obstante el diagrama de la figura 4.6 nos da una idea bastante aproximada.

[7] En Struts 2 podemos evitarnos los ficheros de configuración XML mediante el empleo de anotaciones Java

Para acabar, diremos que las vistas en Struts 2 se implementan también como páginas JSP. aunque se pueden usar otros *frameworks* de plantillas como Velocity o FreeMarker o incluso JSF.

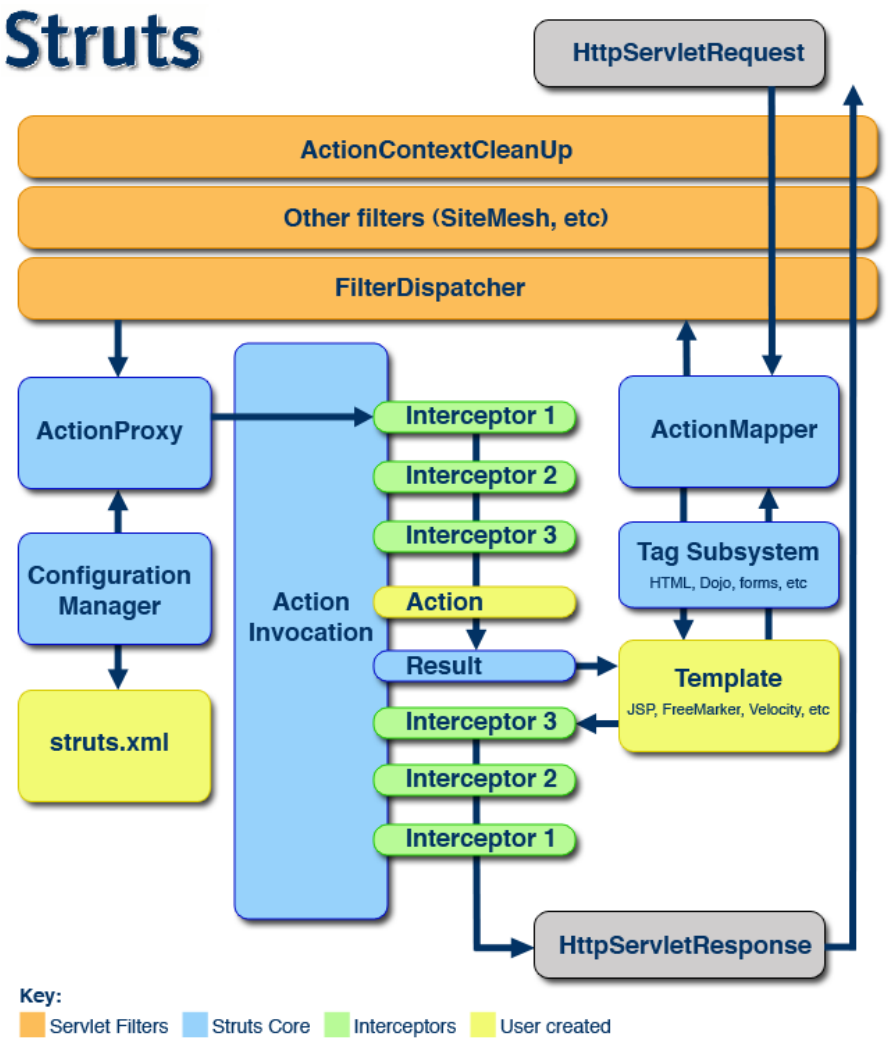

Figura 4.6: Struts 2. Arquitectura detallada Fuente: http://struts.apache.org/2.x/docs/big-picture.html

4.1.3. JavaServer Faces (JSF) 1.x

Tras el auge de los *frameworks* de presentación en general, y de Struts en particular, en el JCP creyeron que había llegado la hora de dotar a JEE de su propio marco de trabajo estándar de presentación.

El 30 de mayo de 2001 se formó el grupo de expertos encargado de desarrollar la especificación de JavaServer Faces, liderado por Craig McClanahan (ver 4.1.2) y Edward Burns. La primera especificación de JSF estuvo lista el 11 de marzo de 2004 bajo la JSR-127.

La última especificación disponible de JSF 1.x es la 1.2 (JSR-252 [7]) publicada el 27 de mayo de 2008. Será sobre esta versión sobre la que nos vamos a centrar en este análisis. Comenzamos con la definición que de JSF que se hace en su propia especificación:

> "JavaServer Faces (JSF) is a user interface (UI) framework for Java web applications. It is designed to significantly ease the burden of writing and maintaining applications that run on a Java application server and render their UIs back to a target client. JSF provides ease-ofuse in the following ways:
>
> - Makes it easy to construct a UI from a set of reusable UI components
> - Simplifies migration of application data to and from the UI
> - Helps manage UI state across server requests
> - Provides a simple model for wiring client-generated events to server-side application code
> - Allows custom UI components to be easily built and re-used"

Con JSF aparece por primera vez un concepto interesante. Aunque es todavía un *framework* claramente orientado a páginas con formularios (*page centric*), nos proporciona un juego de **componentes reutilizables** mas allá de los clásicos `<INPUT>` del lenguaje HTML. Y si hablamos de componentes, inevitablemente estamos hablando de eventos.

Pero además, al ser JSF una especificación estándar, las diferentes implementaciones de la especificación podrán aportar su propio juego de componentes, por lo general más ricos que los ofrecidos por la implementación de referencia.

POJO: *Plain Old Java Object*

JSF también es un *framework* MVC. Las vistas las forman páginas JSP con formularios construidos a base de componentes JSF. Los controladores serán POJO que siguen la convención *Java Beans* y que reciben el nombre de *managed beans*. El enlace entre unos y otros se hará mediante un fichero de configuración centralizado con el nombre de `faces-config.xml`, donde declararemos, entre otras cosas, los *beans* y las reglas de navegación de la aplicación. Podemos ver un ejemplo de fichero de configuración de JSF en el listado 4.9

Listado 4.9: faces-config.xml

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

<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">
        <managed-bean>
                <managed-bean-name>loginBean</managed-bean-name>
                <managed-bean-class>jsf1test.beans.LoginBean</managed-bean-class>
                <managed-bean-scope>session</managed-bean-scope>
        </managed-bean>
        <navigation-rule>
                <display-name>index</display-name>
                <from-view-id>/index.jsp</from-view-id>
                <navigation-case>
                        <from-outcome>logged</from-outcome>
                        <to-view-id>/logged.jsp</to-view-id>
                </navigation-case>
        </navigation-rule>
        <navigation-rule>
                <display-name>index</display-name>
                <from-view-id>/index.jsp</from-view-id>
                <navigation-case>
                        <from-outcome>rejected</from-outcome>
                        <to-view-id>/rechazado.jsp</to-view-id>
                </navigation-case>
        </navigation-rule>

</faces-config>
```

Estamos configurando un *bean* (clase `jsf1test.beans.LoginBean`) y dos reglas de navegación, una desde la página `index.jsp` a `logged.jsp` y otra desde la misma página inicial

a `rejected.jsp`. El *outcome*, que es un valor devuelto por algún método del *bean*, es quien decide que camino se toma.

Podemos seguir el camino de una petición en JSF desde la vista hasta el controlador. En el listado 4.10 tenemos el ejemplo de una vista JSF. Las vistas en JSF se construyen con JSP y librerías de etiquetas.

Listado 4.10: index.jsp

```
1  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2          pageEncoding="ISO-8859-1"%>
3  <%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
4  <%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
5  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
6                  "http://www.w3.org/TR/html4/loose.dtd">
7  <html>
8  <head>
9  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
10 <title>Insert title here</title>
11 </head>
12 <body>
13 <f:view>
14 <h:form>
15 Nombre: <h:inputText value="#{loginBean.usuario }"/><br/>
16 Contraseña: <h:inputSecret value="#{loginBean.password }"/><br/>
17 <h:commandButton value="Enviar" action="#{loginBean.login }"/>
18 </h:form>
19 </f:view>
20 </body>
21 </html>
```

Es una página sencilla, con un formulario formado por una caja de edición de texto (línea 15), una de edición de tipo contraseñas (línea 16) y un botón de *submit* (línea 17). Los dos campos de edición están enlazados con las propiedades `usuario` y `password` del *bean*. En el botón enlazamos el atributo `action` con el método `login` del *bean*, de modo que al pulsar el botón se invocará dicho método.

Lo único que se exige a los métodos `action` de los *beans* es que no tengan parámetros y devuelvan un objeto `String`. Esta cadena de caracteres será el *outcome* para las reglas de navegación. Encantadoramente sencillo. En el listado 4.11 vemos el código del *bean*.

Ciclo de vida

Y básicamente este es el funcionamiento de JSF 1.x, aunque por detrás haya muchas más cosas, por supuesto. Por ejemplo, la petición pasa por hasta seis fases distintas desde que se origina hasta que se sirve la respuesta. La figura 4.7 muestra el ciclo de vida completo de una petición *Faces Request - Faces Response*[8].

Es el momento de ver cada una de estas fases con más detenimiento:

1. **Restore view:** El objetivo de esta fase es construir la vista (el árbol de componentes) si se trata de una petición inicial, o restaurar la vista con los valores generados en una respuesta anterior.

[8] *Faces Response* se refiere a una respuesta generada en la fase *Render response*. *Faces Request* se refiere a una petición que proviene de una *Faces response* anterior.

Listado 4.11: LoginBean.java

```
1  package jsf1test.beans;
2
3  public class LoginBean {
4          private String usuario;
5          private String password;
6
7          private String usuarioValido = "alberto";
8          private String passwordValida = "123456";
9
10         public String login() {
11                 if (usuarioValido.equals(getUsuario()) &&
12                                 password.equals(passwordValida)) {
13                         return "logged";
14                 }
15
16                 return "rejected";
17         }
18
19         public String getPassword() {
20                 return password;
21         }
22
23         public void setPassword(String password) {
24                 this.password = password;
25         }
26
27         public String getUsuario() {
28                 return usuario;
29         }
30
31         public void setUsuario(String usuario) {
32                 this.usuario = usuario;
33         }
34 }
```

2. **Apply request values:** En esta fase cada componente tiene la oportunidad de actualizar su estado a partir de la información (parámetros, *cookies*, etc.) contenida en la petición. Los valores se convierten a los tipos de datos de la propiedad asociada al componente. Si la conversión no se puede realizar, se genera un mensaje de error que se mostrará en la fase 6.

   También pueden originar eventos que se despacharán en la fase 5, a menos que algún componente tenga activa su propiedad `immediate`, en cuyo caso sus eventos se despacharán al final de esta misma fase; en este caso, también se producirá la validación de los datos en este momento del ciclo de vida.

3. **Process validations:** Durante esta fase se invocarán todas las validaciones de los datos de cada componente. Estas validaciones pueden ser proporcionadas por la aplicación o validaciones estándar incluidas en JSF.

   Si alguna de las validaciones falla, el componente con el dato erróneo se marca como invalido, se genera un mensaje de error y el ciclo de vida salta directamente a la fase 6. En caso contrario, el proceso sigue hasta la siguiente fase.

4. **Update model values:** En esta fase, con los datos ya convertidos y validados, se actualizan los valores correspondientes de las propiedades del *bean* que estén enlazadas con componentes de la vista.

5. **Invoke application** En esta fase, con los datos ya convertidos, validados y aplicados a las propiedades del *bean* se invoca a la lógica de negocio. En esta fase también se produce la selección de la siguiente vista, mediante alguno de los *outcomes* definidos en las reglas de navegación del fichero `faces-config.xml`

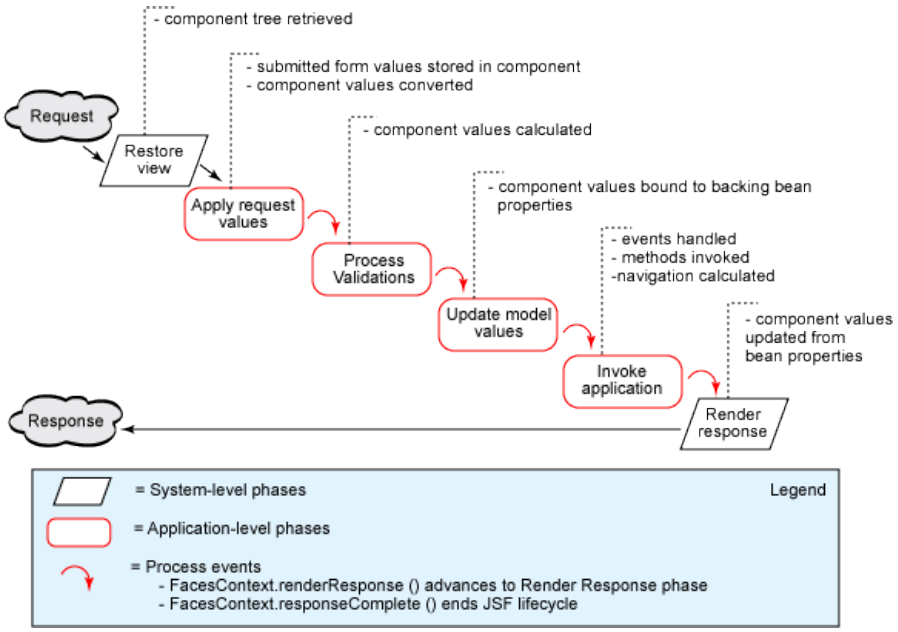

Figura 4.7: JSF - Ciclo de vida de una petición Fuente: http://www.ibm.com/developerworks/library/j-jsf2/

6. **Render response** En esta última fase se muestra la vista seleccionada en la fase anterior con todos los componentes actualizados.

4.2. *Frameworks* AJAX

AJAX: *Asynchronous Javascript And XML*

RIA: *Rich Internet Applications*

Consideraremos con el nombre genérico de *frameworks* AJAX a todos aquellos que adoptan dicha tecnología para ofrecer una experiencia RIA , más próxima al aspecto y al uso de las aplicaciones de escritorio.

Estos *frameworks* ofrecen una interacción asíncrona con el servidor. La interfaz gráfica no se queda bloqueada mientras se trata la petición y la respuesta no consiste en una página nueva que se recarga desde cero sino que se actuará directamente sobre el árbol DOM del cliente para modificar solo aquellas partes de la vista que hayan cambiado.

Estos *frameworks*, además de en todas las tecnologías propias de JEE, se apoyan también en el lenguaje Javascript en general y en el objeto `XMLHttpRequest` en particular.

De este tipo de *frameworks* vamos a estudiar tres: uno estándar que ha incorporado soporte AJAX (JSF 2), otro construido desde el principio con AJAX en mente (ZK) y un *framework* con ideas diferentes y desarrollado en nuestro país (ItsNat).

4.2.1. JSF 2.x

JavaServer Faces 2.x es una evolución de JSF 1.x (ver 4.1.3). La última versión de la especificación en el momento de escribir este documento es la 2.2, publicada bajo la JSR-344 [**?**] en noviembre de 2011.

JSF 2.x incorpora una serie de nuevas e interesantes caracerísticas, entre las que destacan:

1. Un nuevo formato para la construcción de las vistas:*facelets*.
2. Nuevas anotaciones para *managed beans*.
3. Mapeos por defecto de resultados a páginas.
4. Soporte para AJAX

Facelets

Las vistas en JSF 2.x se crean con XHTML (se denominan *facelets*) en lugar de las páginas JSP de JSF 1.x (aunque se puedan usar de forma opcional en esta versión). Básicamente un *facelet* es un página XHTML con un *namespace determinado*. En el listado 4.12 se puede ver un ejemplo.

Listado 4.12: index.xhtml

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3
4  <html xmlns="http://www.w3.org/1999/xhtml"
5           xmlns:h="http://java.sun.com/jsf/html">
6          <h:head>
7                  <title>Managed beans</title>
8          </h:head>
9          <h:body>
10         <fieldset>
11                 <legend>Saldo bancario</legend>
12                 <h:form>
13                         ID Cliente:
14                         <h:inputText value="#{bankingBean.idCliente}"/><br/>
15                         Contraseña:
16                         <h:inputSecret value="#{bankingBean.password}"/><br/><br/>
17                         <h:commandButton value="Mostrar saldo actual"
18                                 action="#{bankingBean.mostrarSaldo}"/>
19                 </h:form>
20         </fieldset>
21         </h:body>
22 </html>
```

Como se puede observar, ya no se emplean librerías de etiquetas JSP sino *namespaces* (líneas 4 y 5). Por lo demás no hay demasiadas diferencias con la página JSP que teníamos para JSF 1.x. En JSF 2.x seguimos teniendo a `FacesServlet` como *front controller*. Las URL se suelen mapear al patrón `*.jsf`, como se ve en el listado 4.13.

Managed beans anotados

En JSF 2.x se mantiene un fichero `faces-config.xml`, aunque el elemento `<faces-config>` está vacío[9]. Como novedad, indicaremos qué clases serán *beans* mediante la anotación `@ManagedBean`. En el listado 4.14 tenemos un ejemplo de *bean* anotado.

[9]No obstante, se puede configurar este fichero al antiguo estilo de JSF 1.x

Listado 4.13: Configuración JSF 2

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        version="3.0">
  <display-name>jsfsample2</display-name>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>

</web-app>
```

Listado 4.14: BankingBean.java

```
1  package jsf2.sample.beans;
2
3  import javax.faces.bean.ManagedBean;
4
5  import jsf2.sample.model.Cliente;
6  import jsf2.sample.service.CustomerLookupService;
7  import jsf2.sample.service.CustomerSimpleMap;
8
9  @ManagedBean
10 public class BankingBean {
11         private static CustomerLookupService lookupService = new CustomerSimpleMap();
12
13         private String idCliente;
14         private String password;
15
16         private Cliente cliente;
17
18
19         public String mostrarSaldo() {
20                 cliente = lookupService.find(idCliente);
21
22                 if (cliente == null) {
23                         return "desconocido";
24                 }
25
26                 return "infocliente";
27         }
28
29         // AQUÍ VIENEN LOS GETTERS Y SETTERS.
30
31         .......
32 }
```

Con esta clase anotada, unida al lenguaje de expresiones que aparece en el *facelet* (ver por ejemplo la línea 16 del listado 4.12 `#bankingBean.password`), el *framework* es capaz de instanciar un objeto de la clase apropiada[10] y realizar el *binding* con los campos y acciones del *facelet*.

Mapeo automático de resultados a páginas.

Observemos de nuevo el listado 4.14. Vemos en él que el método de acción `mostrarSaldo()` produce dos *outcomes*: `desconocido` e `infocliente` (líneas 23 y 26 respectivamente).

[10] Nótese que la clase del *bean* tiene el mismo nombre que el objeto en el *facelet*, salvo la primera mayúscula

Como ya no definimos las reglas de navegación en el fichero `faces-config.xml` tal y como se hacía en JSF 1.x, el *framework* asume la existencia de dos *facelets* cuyos nombres serán `desconocido.xhtml` e `infocliente.xhtml` a los que se dirigirá en función del resultado del método en cuestión.

Soporte AJAX: `<f:ajax>`

Para dotar a JSF 2.x con soporte para AJAX[11] se introduce la etiqueta `f:ajax`. Para acceder a dicha etiqueta debemos incluir el correspondiente *namespace* en el *facelet*, como se ve en la línea 6 del listado 4.15.

Ahora ya podemos usar la etiqueta (ver línea 25). Al situarla **dentro** del botón (`h:commandButton`), cuando este botón se pulse no se hara un *submit* del formulario sino que, mediante Javascript, se hará una llamada asíncrona al servidor.

Listado 4.15: index.xhtml con AJAX

```
1
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://java.sun.com/jsf/html"
6        xmlns:f="http://java.sun.com/jsf/core">
7  <h:head><title>JSF 2.0: Ajax Support</title>
8  <link href="./css/styles.css"
9        rel="stylesheet" type="text/css"/>
10 <script src="./scripts/utils.js"
11         type="text/javascript"></script>
12 </h:head>
13 <h:body>
14 <div align="center">
15 <table border="5">
16   <tr><th class="title">JSF 2.0: Ajax Support</th></tr>
17 </table>
18 <p/>
19
20 <h:form>
21 <fieldset>
22   <legend>Random Number: Ajax</legend>
23   <h:commandButton value="Show Number"
24                    action="#{numberGenerator.randomize}">
25     <f:ajax render="numField1"/>
26   </h:commandButton>
27   <h2><h:outputText value="#{numberGenerator.number}"
28                     id="numField1"/></h2>
29 </fieldset>
30 </h:form>
31 </h:body>
32 </html>
```

En el servidor se ejecutará el método de acción del *bean* correspondiente y de regreso se actualizará el componente o componentes cuyos ID coincidan con los indicados en el atributo `render` de la etiqueta `f:ajax`. Con respecto al *bean*, podemos ver un ejemplo en el listado 4.16

El método de acción del *bean* tiene la misma firma que en JSF 1.x. La particularidad es que dicho método no retorna ningún *outcome* sino que devuelve un `null`. De esta forma, este *bean* podría utilizarse tanto en una aplicación AJAX como en una tradicional. Si el *bean* se va

[11] Todos los ejemplos se han obtenido de **"the coreservlets.com JSF 2.0 tutorial"** (http://www.coreservlets.com/JSF-Tutorial/jsf2/)

Listado 4.16: NumberGenerator.java

```
1  package coreservlets;
2
3  import javax.faces.bean.*;
4
5  @ManagedBean
6  public class NumberGenerator  {
7    private double number = Math.random();
8    private double range = 1.0;
9
10   public double getRange() {
11     return(range);
12   }
13
14   public void setRange(double range) {
15     this.range = range;
16   }
17
18   public double getNumber() {
19     return(number);
20   }
21
22   public String randomize() {
23     number = range * Math.random();
24     return null;
25   }
26 }
```

a utilizar solamente en una aplicación AJAX el *framework* permite que los métodos de acción tengan un tipo de retorno `void`.

La etiqueta `f:ajax` tiene muchas más opciones y posibilidades cuyo conocimiento detallado excede de los objetivos del presente PFC. Se remite al lector interesado a la especificación de JSF 2.x.

Ciclo de vida de una petición

Al igual que en JSF 1.x, una petición en JSF 2.x tiene un ciclo de vida que atraviesa por las mismas 6 fases que vimos antes (ver 4.1.3). La diferencia aquí viene en las peticiones AJAX, en las que, mediante una estrategia de recorrido parcial de las vistas, se permite a aplicar las fases del ciclo de vida solo a aquellos componentes involucrados en la petición (aquellos cuyos ID figuren en la etiqueta `f:ajax`).

4.2.2. ZK

Con JSF 2.x vimos un *framework* al que se le habían añadido capacidades AJAX. Con ZK[12] no es este el caso. Estamos ante un *framework* construido desde el principio con AJAX en mente. Frente a los marcos de trabajo vistos anteriormente, que son más o menos *page centric*, ZK es un *framework* centrado en componentes y, por tanto, en eventos.

ZK nos permite hacer programación AJAX de forma totalmente transparente de modo que el programador no es consciente de que está trabajando con dicha tecnología. Una aplicación

[12] http://www.zkoss.org/

construida con ZK tiene un aspecto más cercano al de una aplicación de escritorio y su comportamiento se asemeja también al de este tipo de aplicaciones. Se puede decir que un programador acostumbrado a construir aplicaciones de escritorio (con Java Swing por ejemplo) no se sentirá demasiado incómodo con ZK.

ZK nos ofrece, entre otras, las siguientes características:

- Fusión cliente-servidor.
- Aproximación SPI: Las aplicaciones ZK no suelen caracterizarse por navegación entre páginas (aunque nada impide que así se construyan). Tienen más bien una única página principal.
- Un rico juego de componentes que incluye tablas, árboles, listas, menús etc.
- Programación dirigida por eventos (*event driven programming*).
- Un lenguaje de marcas basado en XML, llamado ZUML, para construir las vistas.
- Soporte para los patrones MVC (3.1.1) y MVVM (3.1.2).
- Sencilla integración con otros *frameworks* como Spring, Spring security, Hibernate, etc.

SPI: *Single Page Interface*

En el momento de escribir este documento la última versión publicada es la 6.5, que como principal novedad incluye soporte para dispositivos móviles como tabletas o teléfonos inteligentes.

Fusión cliente-servidor

La arquitectura ZK está basada en lo que se denomina fusión cliente-servidor, que básicamente significa que parte del *framework* reside en el servidor y parte en el cliente y ambas partes actúan como una sola. Vamos a explicar esto de un modo más formal.

En la figura 4.8 podemos ver un esquema del recorrido de una petición AJAX en ZK. Todo empieza con un gesto del usuario sobre el interfaz gráfico, por ejemplo haciendo clic sobre un botón.

La página en el cliente contiene un árbol de objetos javascript (denominados *widgets*) que se corresponden uno a uno con los nodos del árbol DOM del documento. Estos *widgets* capturan los eventos DOM, los convierten en eventos de ZK y se los pasan al *client engine*, una librería javascript que se encarga de la comunicación asíncrona con el servidor.

Este *client engine* hace una petición AJAX al servidor en la que encapsula toda la información necesaria sobre el evento producido. En el servidor esta petición se recoge y procesa mediante el *update engine* que la traslada al objeto componente del servidor asociado con el *widget* cliente que originó el evento. Este componente de servidor crea y encola su propio evento.

Cuando a este evento le toque su turno, se pasará al código manejador del evento correspondiente. En este momento entramos en el código que hemos programado nosotros. El evento

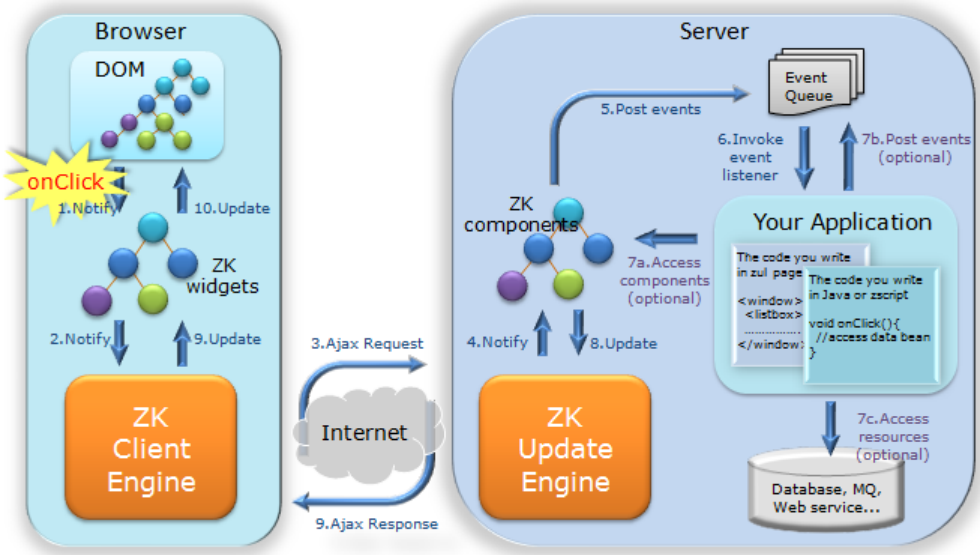

Figura 4.8: ZK. Arquitectura Fuente: http://books.zkoss.org/wiki/ZK%20Essentials

se procesará, probablemente se actuará sobre la capa de persistencia, quizás se encolen nuevos eventos y muy probablemente se modifique el estado del árbol de componentes. A partir de aquí la petición inicia el camino de retorno hasta el árbol DOM del cliente, donde se modificarán únicamente aquellos elementos cuyo estado se haya modificado durante la petición.

Este es el ciclo de vida de una petición una vez construida la página. Pero antes se ha producido el proceso de construcción de la página, que incluye el árbol de componentes del servidor, el árbol de *widgets* y el *client engine*.

La primera vez que se accede a una página de una aplicación ZK se producen la siguiente sucesión de acciones:

1. La página se procesa mediante un *parser* XML. Como resultado se construye el árbol de componentes del servidor.
2. Se crea la página con el árbol de *widgets* que se envía al cliente **junto con el *client engine*** (una sola vez). Ya en el cliente, los *widgets* se "renderizan" a HTML+CSS. La página queda lista para que el usuario interaccione con ella.

Los componentes en ZK pues están formados en realidad por dos objetos: un objeto javascript o *widget* en el cliente y un objeto Java en el servidor. Ambos objetos están conectados mediante los dos *engines* y todo esto de forma transparente para el desarrollador.

MVC en ZK

Para implementar el patrón MVC ZK nos proporciona:

- El lenguaje de marcas ZUML para construir las vistas (ver listado 4.17).

- Objetos *composer* para los controladores (ver listado 4.18).
- El modelo es libre, y podemos implementarlo como queramos.

Listado 4.17: index.zul

```
1  <?page title="ZK Sample"?>
2  <window title="Hello ZK" border="normal" width="400px"
3          apply="zksample.composer.IndexComposer">
4          <vlayout>
5                  <hlayout>
6                          <label value="Introduzca un color" />
7                          <textbox id="txtColor" />
8                  </hlayout>
9                  <label id="lblColor" value="Este texto cambia de color"/>
10                 <button id="btCambiar" label="Cambiar"/>
11         </vlayout>
12 </window>
```

Listado 4.18: IndexComposer.java

```
1  package zksample.composer;
2
3  import org.zkoss.zk.ui.select.SelectorComposer;
4  import org.zkoss.zk.ui.select.annotation.Listen;
5  import org.zkoss.zk.ui.select.annotation.Wire;
6  import org.zkoss.zul.Label;
7  import org.zkoss.zul.Textbox;
8  import org.zkoss.zul.Window;
9
10 public class IndexComposer extends SelectorComposer<Window> {
11
12         /**
13          *
14          */
15         private static final long serialVersionUID = 1L;
16
17         @Wire
18         private Label lblColor;
19         @Wire
20         private Textbox txtColor;
21
22
23         @Listen("onClick = #btCambiar")
24         public void cambiarColor() {
25                 lblColor.setStyle("color: " + txtColor.getValue());
26         }
27
28 }
```

En la página `index.zul` (línea 3) indicamos a la vista quien va a ser su controlador (o *composer*). En el controlador (`IndexComposer.java`) enlazamos los objetos Java de los componentes que vamos a manipular con los *widgets* del cliente mediante la anotación `@Wire` (líneas 17 a 20). Cada componente Java se enlaza con el *widget* cuyo `id` coincida con su nombre y tipo.

Por último, enlazamos el método `cambiarColor()` con el evento `onClick` del botón cuyo `id` es `btCambiar` mediante la anotación `@Listen` (líneas 23 y sucesivas).

MVVM en ZK

Para implementar MVVM ZK nos ofrece:

- El mismo lenguaje de marcas ZUML más unas anotaciones.
- Un *composer* especial: el ZK Binder.
- Los *view model* se implementan con POJOS más unas anotaciones.

Listado 4.19: indexMVVM.zul

```
1  <?page title="ZK Sample MVVM"?>
2  <window title="Hello ZK MVVM" border="normal" width="400px"
3          apply="org.zkoss.bind.BindComposer"
4          viewModel="@id('vm') @init('cursozk.viewmodel.IndexViewModel')">
5          <vlayout>
6                  <hlayout>
7                          <label value="Introduzca un color" />
8                          <textbox id="txtColor" value="@save(vm.color)"/>
9                  </hlayout>
10                 <label id="lblColor" value="Este texto cambia de color"
11                                 style="@load(vm.estilo)"/>
12                 <button id="btCambiar" label="Cambiar"
13                                 onClick="@command('cambiarColor')"/>
14         </vlayout>
15 </window>
```

Listado 4.20: IndexViewModel.java

```
1  package cursozk.viewmodel;
2
3  import org.zkoss.bind.annotation.Command;
4  import org.zkoss.bind.annotation.NotifyChange;
5
6  public class IndexViewModel {
7
8          private String color;
9          private String estilo;
10
11         @Command
12         @NotifyChange("estilo")
13         public void cambiarColor() {
14                 estilo = "color: " + color;
15         }
16
17         public String getColor() {
18                 return color;
19         }
20
21         public void setColor(String color) {
22                 this.color = color;
23         }
24
25         public String getEstilo() {
26                 return estilo;
27         }
28
29         public void setEstilo(String estilo) {
30                 this.estilo = estilo;
31         }
32
33 }
```

En el archivo `indexMVVM.zul` (ver listado 4.19) vemos como se aplica el ZK Binder (línea 3) y se instancia el *view model* (línea 4). En la línea 8 anotamos el atributo `value` para que se salve su valor en la propiedad color del *view model* y en la línea 11 anotamos el atributo `style` para que se cargue su valor desde la propiedad `estilo`. Por último, anotamos el evento `onClick` del botón (línea 13) para que se invoque al método `cambiarColor` del *view model*.

Viendo el *view model* (ver listado 4.20) comprobamos que no es más que un POJO con las propiedades `color` y `estilo` que se enlazan desde la vista. Además tenemos el método `cambiarColor()` con un par de anotaciones que indican que es un método comando (`@Command`) y que modifica una propiedad (`@NotifyChange`) que se recargará en la vista.

Mediante MVVM existe un acoplamiento mucho menor entre la vista y su controlador, ya que todo el enlace de datos se hace a través del ZK Binder dirigido por anotaciones.

4.2.3. ItsNat

El último de los *frameworks* que vamos a tratar es uno de origen español, fundamentalmente desarrollado por Jose María Arranz y la empresa Innowhere. Se trata de un *famework* AJAX puro, con aproximación *server centric* y paradigma SPI SEO .

SEO: *Search Engine Optimization*

La primera gran novedad que presenta ItsNat es en la parte cliente. Nada más que tecnologías web puras: (X)HTML, CSS y nada de lógica. Toda la lógica se ejecuta en el servidor empleando API W3C DOM. En realidad, en el servidor se replica el árbol DOM del cliente; **ItsNat simula un *browser* W3C en el servidor**; los creadores del *framework* lo denominan el principio **TBITS** . Además ItsNat está 100 % libre de ficheros de configuración XML y de programación declarativa.

TBITS: *The Browser Is The Server*

El *framework* nos proporciona dos niveles: core y componentes, construido sobre el core. En realidad, los componentes no son necesarios para construir una aplicación ItsNat. Estos componentes no son más que clases que encapsulan el estado de cualquier elemento visual que tenga un modelo de datos (o de selección). Cualquier elemento HTML puede ser un componente.

El sistema de componentes ItsNat aprovecha gran parte del API de Swing como los modelos de datos y de selección de listas, tablas o árboles[13]. Después de todo, ¿por que no aprovechar algo que ya funciona?

Por contra, ItsNat no es para novatos. Hay que olvidarse de herramientas gráficas *drag & drop*, se exigen conocimientos previos de (X)HTML, W3C DOM o Swing y tiene una curva de aprendizaje apreciable.

ItsNat no nos proporciona un servlet de framework que haga de *front controller*. Nos proporciona un servlet abstracto que tendremos que extender para configurar ItsNat en el método `init`. Una de las tareas que se realizan en la inicialización del *framework* es el registro de las plantillas HTML.

```
docTemplate = itsNatServlet.registerItsNatDocumentTemplate(
"manual.core.example",
"text/html",pathPrefix + "core_example.xhtml");
```

Así, hemos registrado la plantilla `core_examples.xhtml` con el nombre `manual.core.example`. Podremos acceder a ella con una URL de tipo:

```
http://<host>:<puerto>/<app>/<nombre_servlet>?itsnat_doc_name=manual.core.example
```

[13] El autor de este PFC no se resiste a hacer notar que todo el API the Java Swing es uno de los más elegantes ejercicios de programación con los que se ha encontrado en su carrera.

Esto nos mostraría la página registrada como plantilla con algunos añadidos: un poco de código JS insertado al final. Nos quedaría el problema de dotar de funcionalidad a la página. Para ello, tendremos que crear receptores de eventos para las componentes de la página (recordemos que componente puede ser cualquier elemento del árbol DOM).

Como en el servidor tenemos una copia del árbol DOM del cliente (los creadores de ItsNat hablan más bien de que en realidad el navegador es una copia del árbol DOM del servidor), los eventos que se tratarán serán eventos DOM puros.

Aquí dejamos este breve repaso a ItsNat. Se remite al lector interesado al manual de referencia del *framework* (ver [4]).

4.3. Cuadro comparativo de *frameworks*.

En el cuadro 4.1 se muestra un resumen de las principales características de los distintos *frameworks* de capa de presentación para JEE analizados.

	Servlets+JSP	Struts	Struts 2	JSF	JSF 2	ZK	ItsNat
MVC	√	√	√	√	√	√	√
MVVM						√	
AJAX			√ (*)		√	√	√
JCP estándar	√			√	√		
Page Centric	√	√	√	√	√		
SPI						√	√
Open Source	√	√	√	√	√	√(**)	√

(*) Con el apoyo de DOJO Toolkit.

(**) Ediciones EE y PE sujetas a licencia comercial.

Cuadro 4.1: Frameworks JEE. Cuadro comparativo.

Dejando de lado al venerable Struts (parece claro que nadie hoy en día iniciaría un desarrollo nuevo con este *framework*) y a Servlets+JSP (salvo para aplicaciones muy sencillas), tenemos un amplio abanico de opciones para crear nuestra aplicación JEE.

Es indudable que la tendencia hoy en las aplicaciones web es la de proporcionar al usuario una experiencia más rica, lo que nos llevaría a elegir un *framework* con algún tipo de capacidad AJAX. Y no debemos olvidar que al acceso a la red ya se hace más desde dispositivos móviles (*smart phones* y tabletas) que mediante ordenadores de sobremesa o portátiles, con lo que no estaría de más elegir un *framework* que (como ZK en su última versión) tenga esto en cuenta.

Todos los *frameworks* analizados dan soporte a la arquitectura MVC, y todos son gratuitos o de código abierto (aunque ZK tiene un sistema de doble licencia, comercial y GPL, para sus ediciones *Enterprise* y *Profesional*).

Por último, se debe decir que todos ellos imponen algún tipo de lenguaje de marcas para la creación de las vistas, salvo ItsNat, que utiliza únicamente tecnologías web del W3C. Por contra, ItsNat no tiene ningún apoyo de IDE para desarrollar y su curva de aprendizaje es algo más elevada.

Parte II

Formwork: Análisis, diseño e implementación

Capítulo 5

FORMWORK. Análisis y diseño.

"Hay dos maneras de diseñar software: una es hacerlo tan simple que sea obvia su falta de deficiencias, la otra es hacerlo tan complejo que no haya deficiencias obvias"

C.A.R. Hoare.

En este capítulo comenzaremos la construcción del *framework* de capa de presentación para aplicaciones JEE que es el objetivo de este PFC. Nuestro *framework* recibe el original nombre de `Formwork`.

5.1. Descripción general

`Formwork` es un *framework* de capa de presentación para aplicaciones JEE orientado a la creación de aplicaciones de pago de tributos. Estas aplicaciones serán sencillas, de una sola página, con un formulario dividido en apartados que simulará el modelo oficial de presentación en papel.

Este tipo de tributos no suelen capturar un elevado número de datos, apenas los relativos al devengo, al sujeto pasivo y a las cantidades dinerarias que forman la base imponible – si son declaraciones autoliquidables – o la declaración de bienes y derechos – en el caso de que se trate de una declaración informativa –. `Formwork` proporcionará soporte para crear tanto la interfaz gráfica de la aplicación como su lógica de negocio.

5.2. Requisitos.

En `Formwork` se establecen los siguientes requisitos:

1. Se deberá basar en el patrón arquitectónico MVC.
2. Se proporcionará un lenguaje de marcas, basado en XML, para la creación de la vista (página) de la aplicación. Este lenguaje recibe el nombre de FWML .

FWML: *Formwork Markup Language*

3. La página principal de la aplicación se formará con un único formulario. Este formulario estará formado por apartados que a su vez se formará a base de partidas.

4. Las páginas FWML se procesarán en el servidor antes de servirse al cliente.

5. En el servidor se construirá el árbol de componentes de la página. Cada componente se *renderizará* a HTML + CSS + JS antes de ser enviado al cliente.

6. La página de la aplicación se verá respaldada en el servidor por un clase que haga de controlador. Por dicha clase pasarán los eventos de cambio de los componentes y se ejecutará la lógica de negocio correspondiente.

7. Todas las peticiones de cambio de los componentes se harán mediante **AJAX**. `Formwork` será por tanto un *framework* AJAX que se apoyara en el *framework* JavaScript **jQuery**.

8. La lógica de negocio se podrá expresar en la forma de reglas de negocio de tipo "**`cuando`** `condición` **`entonces`** `acciones`". Por ejemplo:

```
regla 1
cuando
    cambie el valor de la partida X
entonces
    acumular el valor de X en la partida Y
fin
```

9. En el inicio de la aplicación se cargarán y compilarán todas las reglas de negocio una única vez.

10. Se usará el motor de reglas de negocio DROOLS de JBoss.

11. `Formwork` no se ocupará de aspectos como seguridad o autenticación. Se delegan estos aspectos en los mecanismos estándar de la especificación JEE o en otros *frameworks* como Spring Security.

5.3. Análisis

RUP: *Rational Unified Process*

Existen muchos procesos de análisis y diseño orientado a objetos (como RUP o los métodos ágiles) pero tratándose este trabajo de un PFC de la UOC seguiremos el método explicado en la asignatura de **Ingeniería del Software Orientada a Objetos** [9], con algunas pinceladas de lo aprendido en la asignatura de **Ingeniería del Software de Componentes y Sistemas Distribuidos** [8].

5.3.1. Casos de uso.

Del análisis de lista de requisitos se obtiene el diagrama de casos de uso de la figura 5.1. Tenemos ocho casos de uso y cuatro actores (ver cuadro 5.1).

Casos de uso	
Init	Inicialización *framework*.
Load	Carga de la página principal.
Load resurces	Carga de recusos de la página principal.
Render	Dibujo de la página principal.
Render response	Dibujo del resultado de una petición.
Service	Petición asíncrona desde el cliente.
Fire business rules	Ejecución de las reglas de negocio.
Submit	Presentación del formulario.
Actores	
User	Usuario de una aplicación `Formwork`.
App	Aplicación `Formwork`.
AppServer	Servidor de aplicaciones donde se despliega una aplicación `Formwork`.
ClientBrowser	Navegador del cliente.

Cuadro 5.1: Resumen de actores y casos de uso

Se han agrupado los casos de uso en tres subsistemas, representados en el diagrama por medio de paquetes: `Infraestructura`, que engloba los casos de uso `Init`, `Load` y `Load resources`; `UI`, que contiene los casos de uso `Render`, `Render response` y `Service` con los casos de uso `Service`, `Submit` y `Fire business rules`. En las siguientes páginas se describen estos casos de uso uno a uno y de manera más formal.

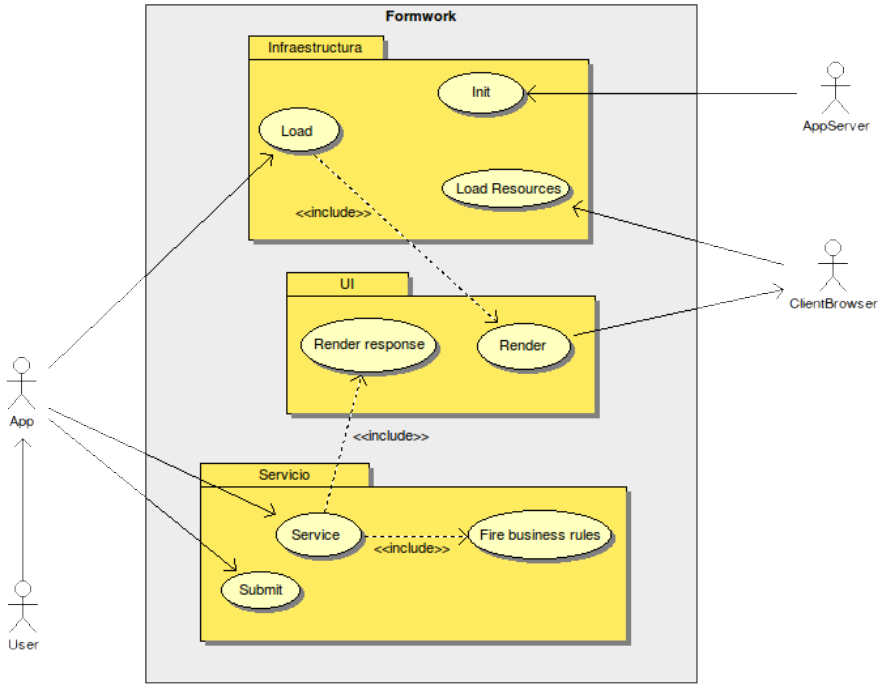

Figura 5.1: Formwork. Diagrama de casos de uso.

Nombre	`Init`
Resumen	Se inicia la aplicación.
Actores	`AppServer`

Flujo principal	1. `AppServer` inicia el despliegue de la aplicación construida con `Formwork` 2. Se invoca al objeto *listener* de contexto: `FormworkListener`. 3. El objeto *listener* instancia un objeto `FormworkContext` 4. El *listener* analiza el fichero de reglas de negocio, lo compila y crea la base de conocimiento. Para todo esto se apoya en el *runtime* de DROOLS. 5. El *listener* almacena `KnowledgeBase` en el objeto `FormworkContext` 6. El *listener* almacena el objeto `FormworkContext` en el `ServletContext` para que quede a disposición de toda la aplicación. 7. El caso de uso finaliza.
Flujos alternativos	
A1	4a1 La aplicación no proporciona ningún fichero de reglas de negocio. 5a1 El flujo continua en el punto 6 del flujo principal.

Nombre	`Load`
Resumen	Se carga la página principal de la aplicación.
Actores	`App`
Flujo principal	1. El usuario solicita la página principal de la aplicación. 2. La petición llega al *servlet* `FormworkServlet`. 3. Se realiza el *parse* de la página mediante la clase `XMLLoader`. 4. El resultado del *parse* es una colección de objetos JAXB. 5. A partir de los objetos JAXB se construyen el árbol de componentes equivalentes: `Formulario` con sus objetos `Apartado` y los objetos `Partida`. 6. Se guarda el árbol de componentes en la sesión. 7. Se ejecuta el caso de uso `Render`. 8. El caso de uso termina.
Flujos alternativos	
A1	4a1 El *parse* de la página principal termina con algún error de XML. 5a1 Se redirige la salida a una página de error. 6a1 El caso de uso termina.

Nombre	`Render`
Resumen	Se crea la página principal de la aplicación y se envía al cliente.
Actores	No intervienen actores directamente.

Flujo principal	1. Se ordena el dibujado de la página principal desde el caso de uso `Load`. 2. Se recorre el árbol de componentes del servidor. 3. Para cada componente, se "renderiza" su valor y estado a HTML + CSS + JS 4. La página creada se envía al cliente. 5. El caso de uso termina.
Flujos alternativos	
No hay flujos alternativos	

Nombre	`Load resources`
Resumen	Se cargan los recursos internos de `Formwork` necesarios para la correcta presentación y funcionamiento de la parte cliente de la aplicación.
Actores	`ClientBrowser`
Prerequisitos	Se ha ejecutado el caso de uso `Render`.
Flujo principal	1. El actor `ClientBrowser` inicia el proceso solicitando los recursos (hojas de estilo y archivos JS) incluidos en la página principal. 2. Llegan a `FormworkServlet` los recursos cuya URL se le hayan mapeado. 3. El *servlet* busca dichos recursos en su *classpath* y los devuelve al cliente. 4. En el cliente se usan los recursos para preparar la página. 5. El caso de uso termina.
Flujos alternativos	
No hay flujos alternativos.	

Nombre	`Service`
Resumen	Se trata una petición AJAX, se desencadena la lógica de negocio de la aplicación y se envía la respuesta al cliente.
Actores	`User` y `App`
Flujo principal	1. El actor `User` inicia el proceso efectuando alguna acción sobre la interfaz gráfica que desencadena un evento. 2. Desde la página se genera una petición AJAX. 3. La petición llega al *servlet* `FormworkServlet`. 4. Se ejecuta el caso de uso `Fire business rules`. 5. Se ejecuta el caso de uso `Render response`. 6. El caso de uso termina.
Flujos alternativos	
No hay flujos alternativos	

Nombre	`Fire business rules`
Resumen	Se ejecutan las reglas de negocio de la aplicación.
Actores	No hay actores directamente implicados.

Flujo principal	1. Desde el caso de uso `Service` se ordena la ejecución de las reglas de negocio. 2. `FormworkServlet` obtiene el controlador de la aplicación desde la sesión. 3. `FormworkServlet` traslada la petición al método correspondiente del controlador. 4. El controlador trata la petición ejecutando las reglas de negocio, con el soporte del motor de reglas DROOLS. 5. El controlador prepara el resultado de la ejecución de las reglas. Esta respuesta puede ser una lista de errores o una lista de partidas afectadas durante la ejecución. 6. El sevlet `FormworkServlet` prepara la respuesta en formato JSON y la devuelve al cliente que modificará el árbol DOM convenientemente. 7. El caso de uso termina.
Flujos alternativos	
A1	4a1 La aplicación no ha proporcionado un fichero de reglas DROOLS y ejecuta las reglas de negocio por algún otro medio, p.e mediante código Java directamente. El caso de uso continúa el flujo principal por el punto 5.

Nombre	`Render response`
Resumen	Se recoge el resultado de una petición AJAX, se prepara la respuesta en formato JSON, y se envía al cliente, donde se modifica el árbol DOM de acuerdo con lo recibido.
Actores	**ClientBrowser** recibe el resultado.
Flujo principal	1. `AppServer FormworkServlet` recupera la lista de partidas afectadas en la ejecución de la petición anterior. 2. Se transforma la lista de partidas en u objeto respuesta en formato JSON. 3. Se envía el objeto JSON al navegador cliente, quien modifica el árbol DOM para reflejar los cambios. 4. El caso de uso finaliza.
Flujos alternativos	
A1	1a1 `FormworkServlet` recoge la lista de errores producidos. 2a1 Se transforma la lista de mensajes de error en un objeto respuesta en formato JSON. 3a1 Se envía el objeto respuesta al cliente, donde se modifica el árbol DOM para presentar al usuario los mensajes de error. 4a1 El caso de uso termina.

Nombre	`Submit`
Resumen	Se realiza la presentación del formulario.
Actores	`User` y `App`.

Flujo principal	1. El actor `User` ordena la presentación del formulario mediante el correspondiente botón. 2. La aplicación traslada la petición al *servlet*. 3. `FormworkServlet` obtiene el controlador de la aplicación y le traslada la petición invocando su método correspondiente. 4. El controlador trata la petición ejecutando las reglas de validación globales, con el apoyo de DROOLS. 5. El resultado es correcto, el controlador ejecuta la presentación del formulario. La forma de hacerlo depende enteramente de la aplicación web construida con `Formwork`. 6. El caso de uso termina.
Flujos alternativos	
A1	4a1 La aplicación no ha proporcionado un fichero de reglas DROOLS y ejecuta las reglas de negocio por algún otro medio, p.e mediante código Java directamente. El caso de uso continúa el flujo principal por el punto 5.
A2	5a2 No se superan las reglas de validación globales. El controlador genera una lista de errores. 6a2 El *servlet* prepara la respuesta de la lista de errores en formato JSON. Se envía la respuesta en cliente. 7a2 El cliente recibe la respuesta y muestra la lista de errores.

5.3.2. Diagrama de clases de análisis.

A lo largo del análisis de los casos de uso hemos ido encontrando una serie de conceptos que deben ser modelados mediante el correspondiente diagrama de clases de análisis. Este diagrama se muestra en la figura 5.2.

En el diagrama se muestra ya un esbozo de la arquitectura de `Formwork`. Aparece el *servlet* que hará de *front controller*, `FormworkServlet`. También podemos ver el *context listener* que se encargará de la inicialización del *framework*.

En el resto del diagrama tenemos la jerarquía de componentes de servidor, formada por la clase abstracta *`Componente`* y los objetos que heredan de ella; la interfaz IRender que tienen que implementar los objetos que dibujen la vista y la interfaz IController, que deberá implementar el objeto que haga de controlador.

Por último cabe destacar la presencia del motor de reglas DROOLS, para dar soporte a la ejecución de las reglas de negocio. Mediante esta funcionalidad el *framework* que diseñamos dará soporte a las tres letras del patrón MVC.

Ahora, para completar el análisis describiremos el comportamiento de cada caso de uso

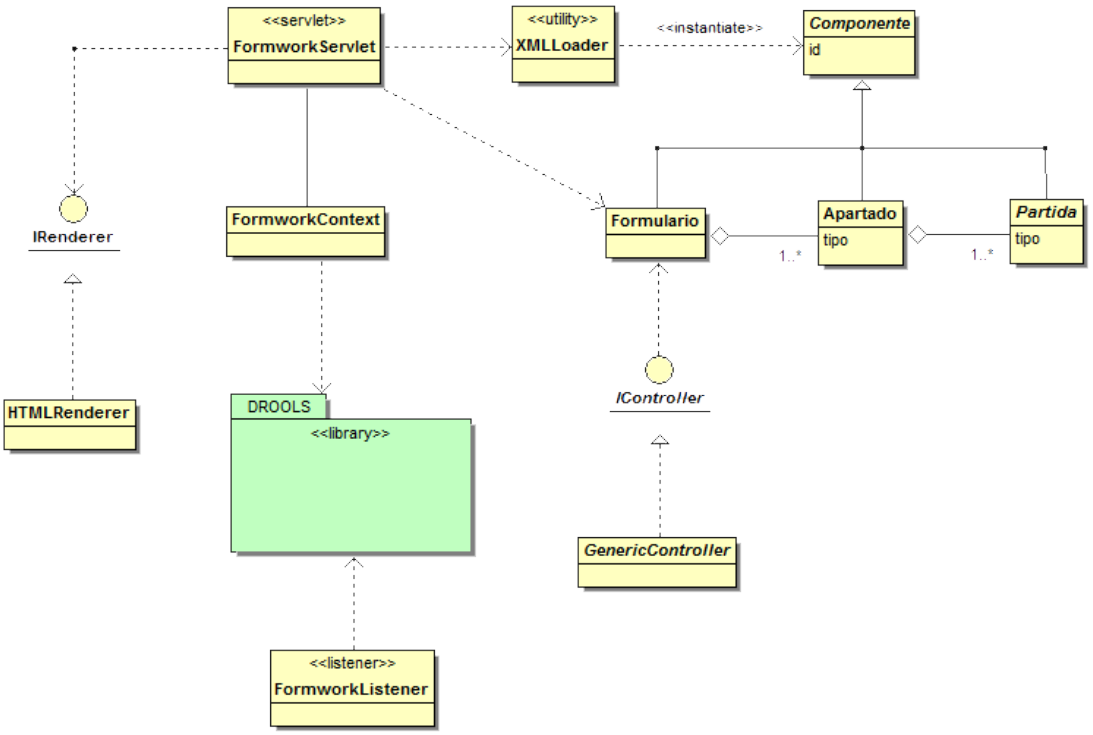

Figura 5.2: Formwork. Diagrama de clases.

mediante diagramas de colaboraciones.

5.3.3. Diagramas de colaboración de los casos de uso.

Vamos a mostrar para cada caso de uso identificado el diagrama de colaboración para su flujo principal.

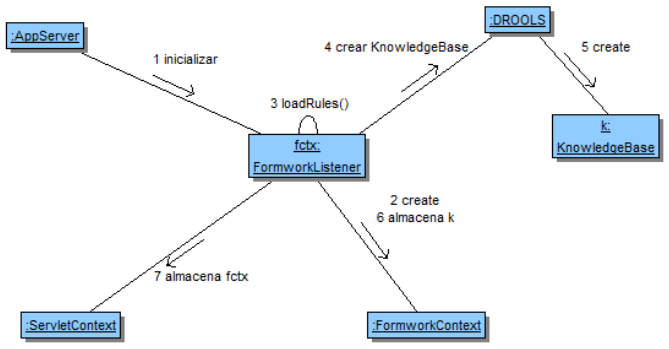

Figura 5.3: Formwork. Diagrama de colaboración Init.

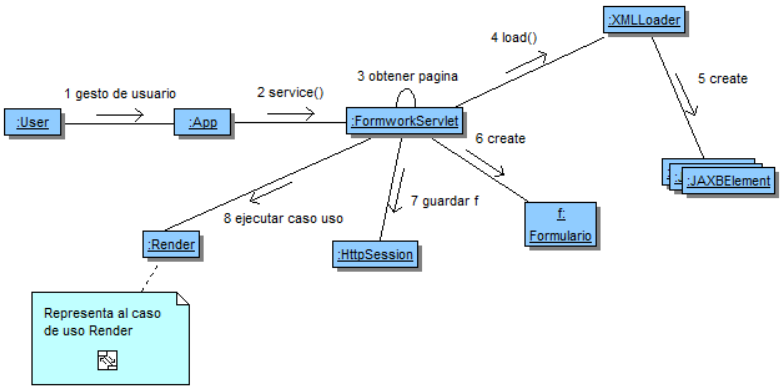

Figura 5.4: Diagrama de colaboración Load.

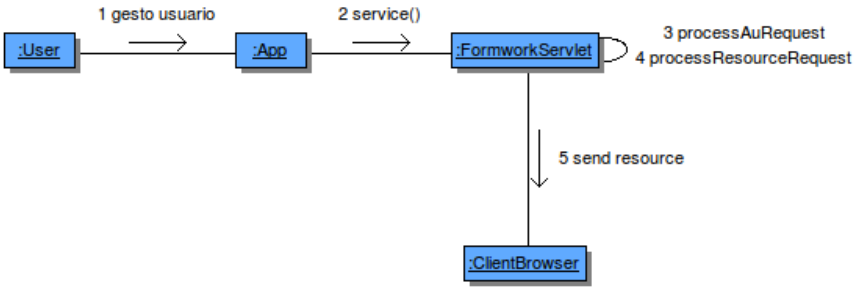

Figura 5.5: Diagrama de colaboración Load resources.

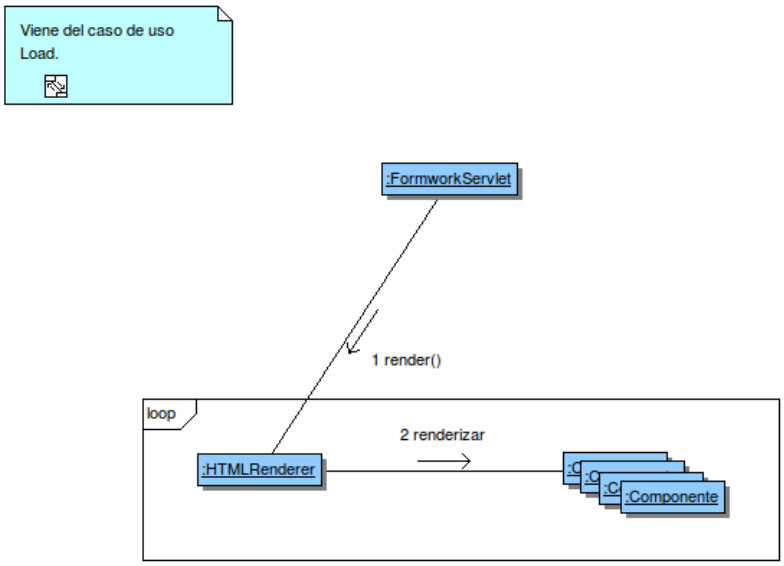

Figura 5.6: Diagrama de colaboración Render.

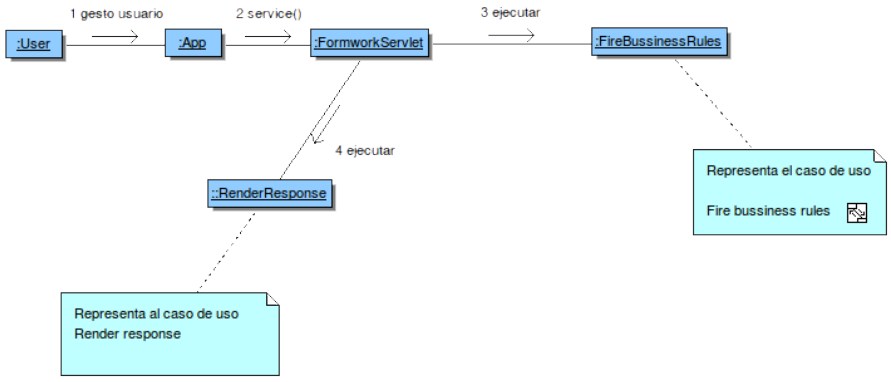

Figura 5.7: Diagrama de colaboración Service.

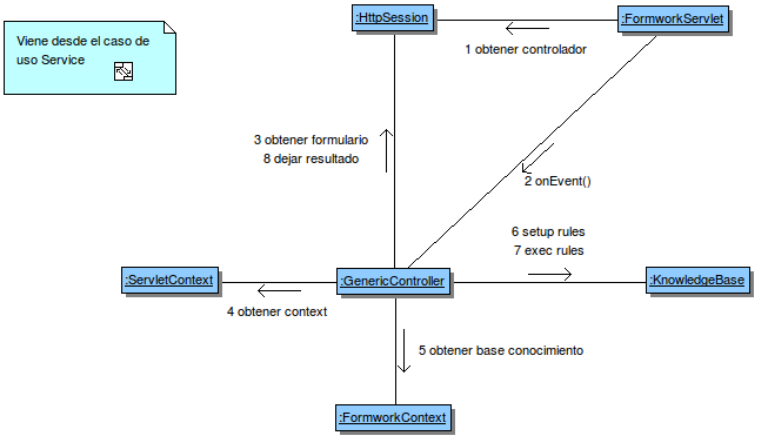

Figura 5.8: Diagrama de colaboración Fire bussiness rules.

Conviene destacar, con respecto al caso de uso **Fire business rules**, que depende casi exclusivamente de la implementación del controlador que se haga en la aplicación que usa `Formwork`. Este diagrama de colaboración de la figura 5.8 debe interpretarse más bien como una receta para el uso del motor de reglas DROOLS.

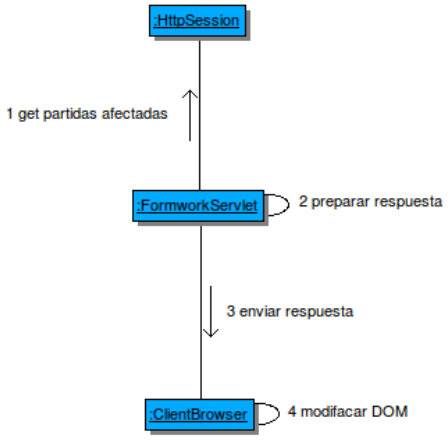

Figura 5.9: Diagrama de colaboración Render response.

Al igual que se indicaba para el caso de uso **Fire business rules**, el caso de uso **Submit** depende en gran medida de la implementación que se haga del controlador en la aplicación que use `Formwork`. En la figura 5.10 se asume la existencia de un servicio externo de presentación del formulario.

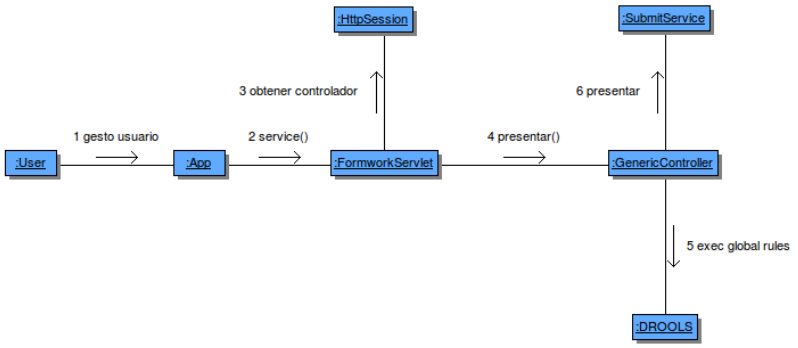

Figura 5.10: Diagrama de colaboración Submit.

5.3.4. Diagrama de clases completo

Al hacer los diagramas de colaboración de caso de uso nos han ido surgiendo nuevas clases que no aparecían el el diagrama de clases original. Tenemos que incorporar estas clases al diagrama para que el análisis quede completo (ver figura 5.11).

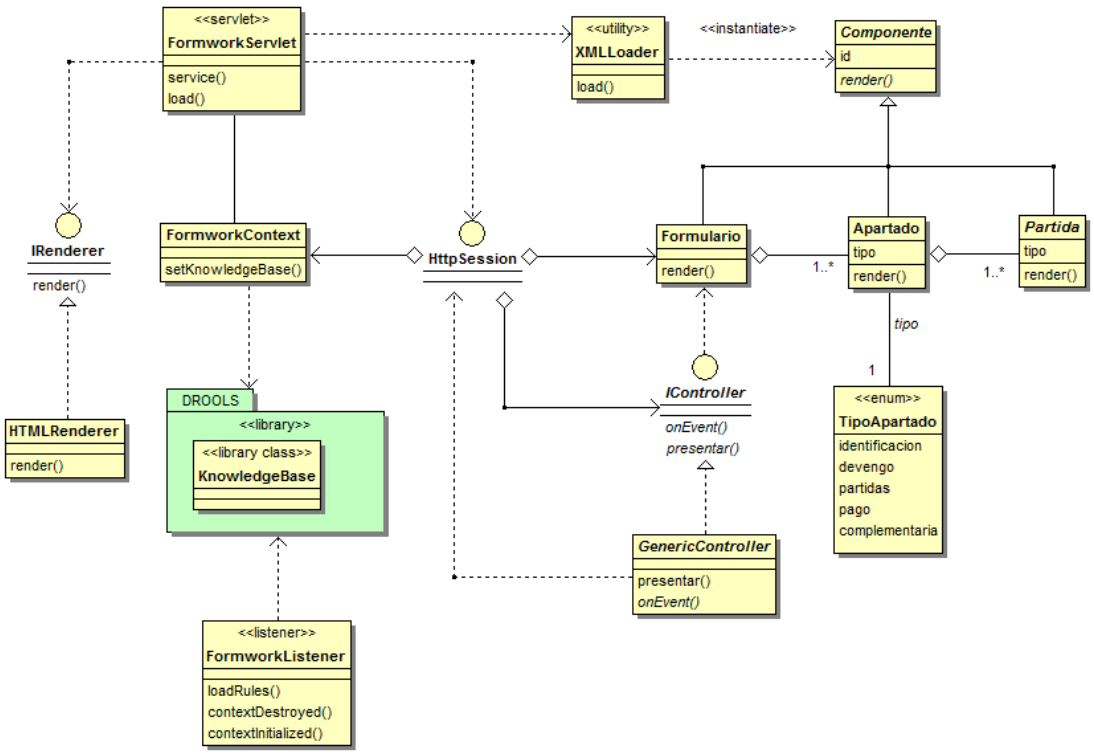

Figura 5.11: Formwork. Diagrama de clases completo.

5.4. Diseño.

5.4.1. Diseño de la capa de presentación

`Formwork` no proporciona las vistas, sino el soporte para crearlas. Ya se ha dicho que las vistas de una aplicación escrita para `Formwork` se codificarán en FWML, un lenguaje de marcas basado en XML. `Formwork` define el lenguaje FWML mediante un esquema W3C XML Schema. Podemos ver un fragmento de este esquema en el listado 5.1

Listado 5.1: FWML W3C XML Schema

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <xs:element name="formulario" type="TipoFormulario" />

        <xs:element name="apartado" type="TipoApartado" />

        <xs:element name="partida" type="TipoPartida" />

        <xs:complexType name="TipoFormulario">
                <xs:sequence maxOccurs="unbounded">
                        <xs:element ref="apartado" />
                </xs:sequence>
        </xs:complexType>
.....
```

Una página FWML sería algo como lo que se muestra en el listado 5.2

Listado 5.2: Una página FWML

```
<?xml version="1.0" encoding="UTF-8"?>
<?page title="Modelo 250"?>
<formulario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                        xsi:noNamespaceSchemaLocation="http://www.uoc.edu/2012/fwp/formwork.xsd">
        <apartado id="ap1" tipo="identificacion" contenido="nif,nombre" />
        <apartado id="ap2" tipo="devengo" contenido="fecha, ejercicio, periodo{0A}" />
        <apartado id="ap3" tipo="partidas">
                <partida etiqueta="P1" tipo="cantidad"/>
                <partida etiqueta="P2" tipo="cantidadNegativa"/>
                <partida etiqueta="P3" tipo="cantidadNegativa"/>
        </apartado>
</formulario>
```

Se aplica el patrón *Composite view* en la generación de la vista a partir de la página FWML.

5.4.2. Diseño paquete Infraestructura

El diseño detallado de las clases Infraestructura se puede ver en la figura 5.12. Se ha aplicado el patrón *Front controller* mediante el *servlet* `FormworkServlet` y el patrón *Observer* con el *listener* `FormworkListener`. Aparecen las clases del API de *servlets* (paquete `javax.servlet`) necesarias para el diseño.

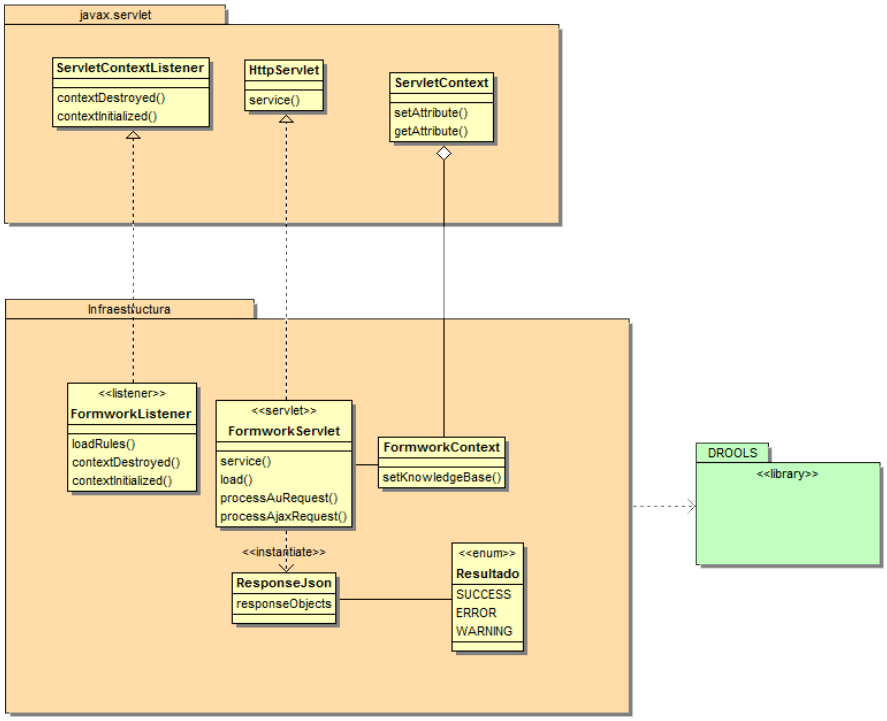

Figura 5.12: Infraestructura. Clases de Diseño.

5.4.3. Diseño paquete UI

El paquete UI incluye todas las clases que tienen que ver con el interfaz gráfico de la aplicación y con su "renderizado". El diagrama de la figura 5.13 muestra las clases de diseño de este paquete.

5.4.4. Diseño paquete Service.

El diseño de este paquete se muestra en el diagrama de la figura 5.14. En este paquete se incluye la clase `GenericController` de la que deben heredar las clases controladoras de las aplicaciones que usan `Formwork`. Las implementaciones concretas de `GenericController` harán posible la realización de los casos de uso **Submit** y **Fire business rules**.

5.4.5. Diagramas de secuencias.

Para finalizar el diseño crearemos algunos diagramas de secuencia para alguno de los casos de uso más importantes del *framework*.

Vamos a representar un escenario de éxito en una petición AJAX. En este escenario están involucrados los casos de uso **Service**, **Fire business rules** y **Render response**, cuyos diagramas de secuencia se mostrarán a continuación.

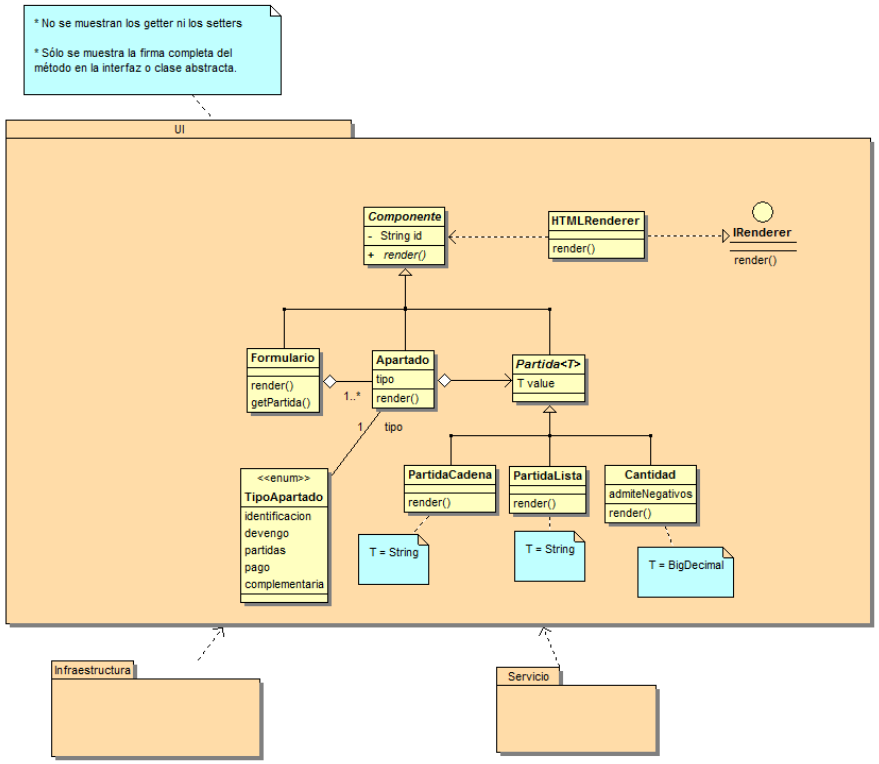

Figura 5.13: UI. Clases de Diseño.

Service

El diagrama de la figura 5.15 muestra la secuencia de mensajes que se dan en el caso de uso Service, el primero de los implicados en la interacción que vamos a describir. En realidad, en este caso de uso todo lo que ocurre es una preparación del entorno – una obtención y creación de los objetos necesarios – para ejecutar con éxito los casos de uso **Fire business rules** y **Render response**

Fire business rules

El caso de uso **Fire business rules** depende en gran medida de la implementación que se haga de la clase controladora. La definición formal del caso de uso nos dice que se invocará al método `onEvent` del controlador para que lance las reglas de negocio de la aplicación.

Para dejar un diagrama de secuencias más completo, vamos a incluir aquí la secuencia de llamadas de la implementación concreta que se ha hecho en la aplicación de ejemplo que se incluye con este PFC, que incluye el uso de DROOLS como motor de reglas. En la figura 5.16 se puede ver este diagrama.

Render response

Después de ejecutar las reglas de negocio, se ha producido como resultado una lista con las partidas que se han modificado. Recordemos que en este escenario estamos tratando el supuesto

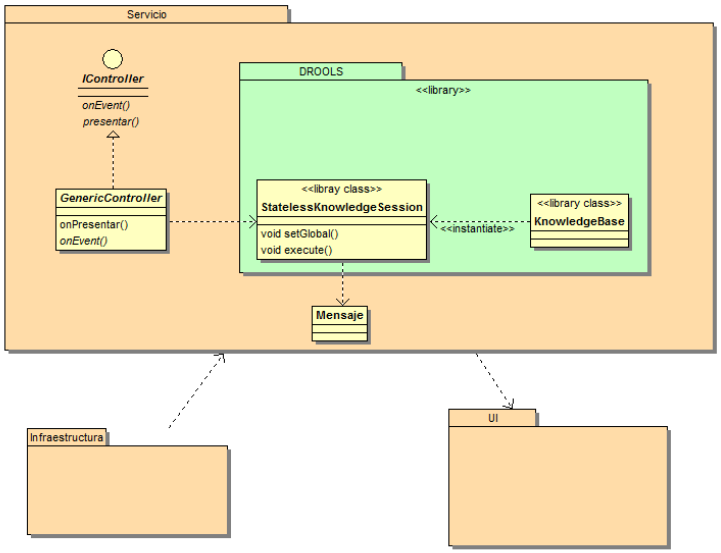

Figura 5.14: Servicio. Clases de Diseño.

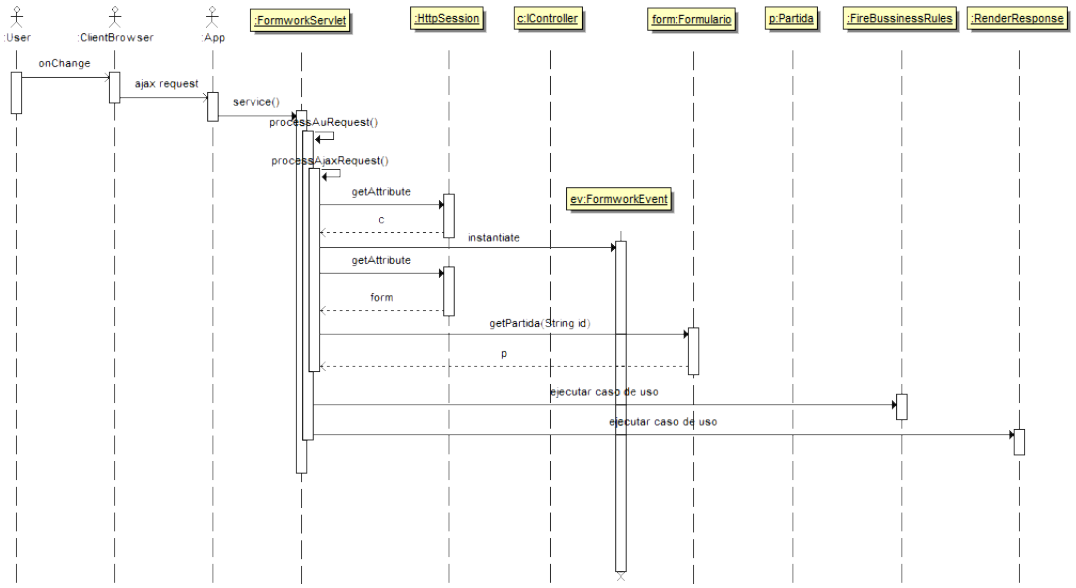

Figura 5.15: Service. Diagrama de secuencia.

de que no se ha producido ningún error de validación durante la ejecución de las reglas de negocio.

En este caso de uso se prepara la respuesta para el cliente, mediante un objeto en formato JSON. Nos ayudaremos de la librería GSON para serializar un objeto Java a formato JSON. En la figura 5.17 tenemos el diagrama de secuencias de este escenario de caso de uso.

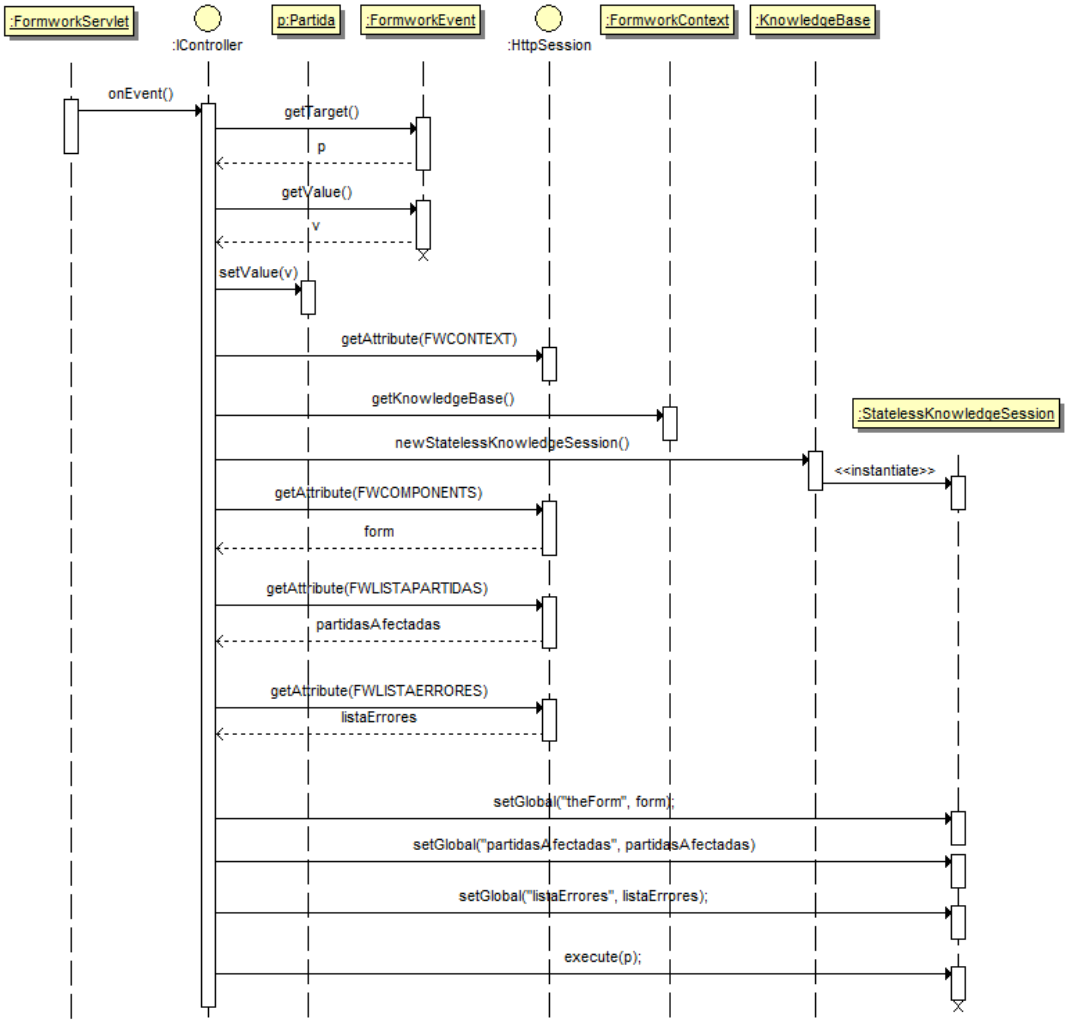

Figura 5.16: Fire business rules. Diagrama de secuencia.

5.5. Resumen final

A lo largo de este capítulo hemos establecido las herramientas necesarias para abordar con garantías la construcción de `Formwork`. De eso tratará el siguiente capítulo, del proceso de implementación del *framework* que hemos ido diseñando durante esta memoria.

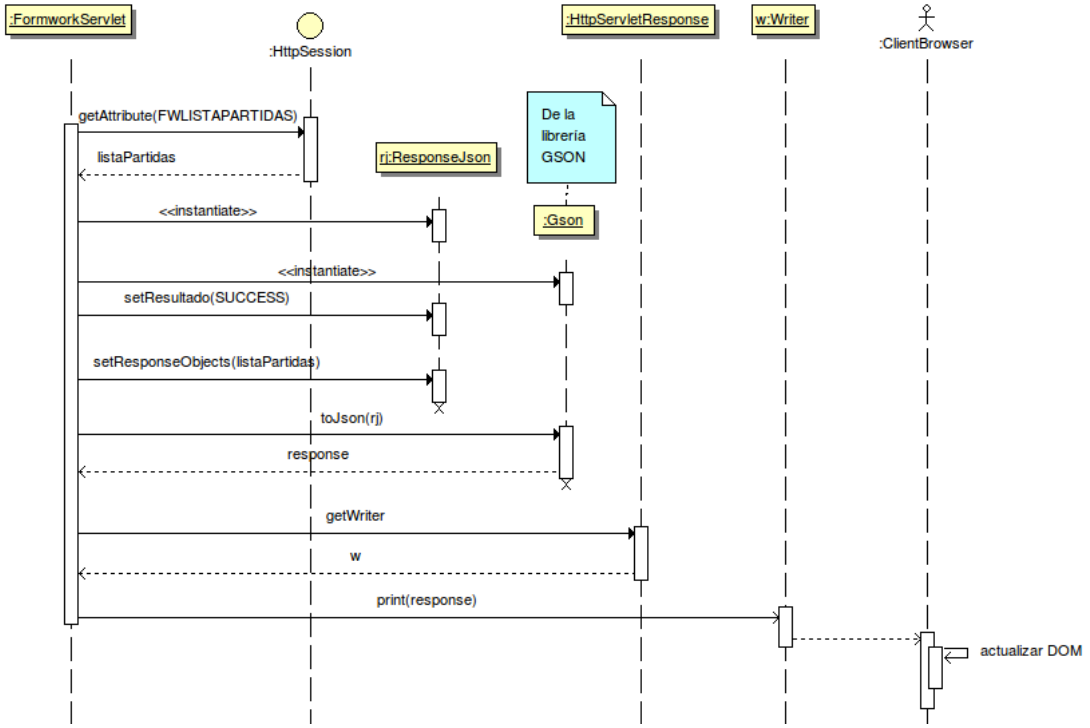

Figura 5.17: Render response. Diagrama de secuencia.

Capítulo 6

FORMWORK. Implementación.

> *"Programa siempre como si supieras que la persona que finalmente mantendrá tu código fuera un asesino psicópata que sabe dónde vives."*
>
> ***Martin Golding.***

En el capítulo anterior completamos el análisis y el diseño de `Formwork`. En este capítulo llevaremos a cabo la codificación del *framework*, que por supuesto se hará mayoritariamente en lenguaje Java aunque también usaremos algo de Javascript (con el apoyo de `jQuery`[1]), el lenguaje `mvel` (utilizado por el motor de reglas DROOLS[2]), HTML y el lenguaje de expresiones del motor de plantillas `Freemarker`[3] que usaremos para la generación de las vistas.

6.1. Descripción general

La implementación del *framework* se ha ido desarrollando en tres máquinas distintas:

- Un portátil con Ubuntu 12.04 Precise Pangolin.
- Un PC con Windows 7.
- Un iMac con Mac OS X 10.8.2 Mountain Lion.

En todos los entornos se ha utilizado Java 7 (tanto el JDK de Oracle[4] como OpenJDK[5]) con **`eclipse`**[6] como IDE (en sus versiones 3.7 Indigo y 4.0.2 Juno) y todo el proyecto se ha gestionado con **Apache Maven**[7] **3.0.4** y el *plugin* **`M2E`**[8] para `eclipse`.

[1] `http://jquery.com`
[2] `http://www.jboss.org/drools`
[3] `http://freemarker.sourceforge.net/`
[4] `http://www.oracle.com/es/technologies/java/index.html`
[5] `http://openjdk.java.net/`
[6] `http://eclipse.org/`
[7] `http://maven.apache.org/`
[8] `http://eclipse.org/m2e/`

Durante la fase de desarrollo se ha utilizado **`Jetty`**[9] como servidor de aplicaciones (que en realidad no es más que un contenedor de *servlets*), por ser un servidor muy ligero, muy rápido en hacer los despliegues (lo que implica un gran ahorro de tiempo durante el desarrollo) y sencillo de integrar con Maven mediante el *Maven Jetty Plugin*[10].

Por último, pero no menos importante, nos queda indicar que se ha usado **`git`**[11] como sistema de control de versiones (con el apoyo del *plugin* **`EGit`**[12] para `eclipse`) manteniendo un repositorio centralizado en **`GitHub`**[13] desde el que se puede descargar el código fuente en su versión más actual, como se explicará más adelante y con mayor detalle.

6.2. El proyecto.

En la figura 6.1 se puede ver la estructura de directorios del proyecto del *framework*. Esta estructura es la típica de un proyecto Maven padre del que dependen dos módulos.

```
PFC-2012/
├── formworkcore/
│   ├── pom.xml
│   ├── src/
│   └── target/
├── formworktest/
│   ├── pom.xml
│   ├── src/
│   └── target/
├── pom.xml
└── README.md
```

Figura 6.1: Formwork. Estructura del proyecto

Estos módulos son:

- `formworkcore`: Que contiene el proyecto del *framework* `Formwork`.
- `formworktest`: Que contiene una aplicación de prueba que usa `formworkcore`.

6.3. Obtener el código fuente de `Formwork`

Antes de proceder a describir como se ha implementado `Formwork` y para poder echar un vistazo rápido a su código debemos obtener y configurar el proyecto en nuestra máquina. Para ello se ofrecen las siguientes alternativas.

[9] `http://jetty.codehaus.org/jetty/`
[10] `http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin`
[11] `http://git-scm.com/`
[12] `http://www.eclipse.org/egit/`
[13] `https://github.com/`

6.3.1. Clonar `Formwork` desde GitHub

Sin duda esta es la forma recomendada por el autor del presente PFC, aunque no sea la manera "oficial". Para ello, basta con tener un cliente de `git` instalado[14] y desde un terminal teclear el siguiente comando:

```
git clone git://github.com/cachocenso/PFC-2012.git
```

Con este comando, `git` descargará el repositorio completo desde GitHub y nos creará una carpeta `PFC-2012` como la que veíamos en la figura 6.1 y con su mismo contenido.

6.3.2. Obtener el código desde la web de GitHub

Si, por la razón que sea, no se dispone de un cliente `git`, podemos descargar el repositorio en un fichero comprimido directamente desde la página web de GitHub.

Para ello, tecleamos en nuestro navegador la dirección:

```
https://github.com/cachocenso/PFC-2012
```

Lo que nos lleva a la página principal del repositorio. Una vez en ella, solamente hay que pulsar en el botón ZIP, tal y como se muestra en la imagen 6.2 para descargarnos el repositorio. Una vez que tengamos el fichero ZIP podemos seguir las instrucciones que se detallan en la sección 6.3.3 para obtener el código fuente.

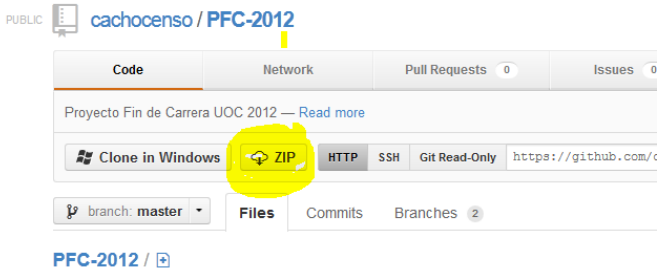

Figura 6.2: Formwork. Descarga ZIP

6.3.3. Obtener `Formwork` desde el fichero adjunto.

Junto con este documento se ha entregado un fichero `pfc-adiazmarti-src.zip` con el contenido del repositorio comprimido. Es suficiente con hacer un `unzip` del archivo para obtener el código del proyecto con la misma estructura de carpetas que obtendríamos clonando el repositorio como se explica en la sección 6.3.1.

```
unzip pfc-adiazmarti-src.zip
```

[14] En sistemas GNU/Linux `git` viene instalado por defecto, en Windows o Mac hay que descargarse e instalar el cliente.

6.4. Compilar y ejecutar.

Una vez que tenemos el código fuente en nuestra máquina podemos proceder a compilar y ejecutar el proyecto. Veremos como hacerlo tanto desde línea de comandos como desde el IDE `eclipse`.

6.4.1. Desde línea de comandos.

Antes que nada, es necesario tener Apache Maven 3[15] correctamente instalado y configurado. En la página web de Maven se puede obtener el producto e instrucciones de instalación para cualquier plataforma.

Teniendo pues Maven 3 instalado en nuestra máquina, la forma más sencilla de obtener los binarios es, desde la carpeta que contiene el fichero `pom.xml` padre (p.e. la carpeta `PFC-2012`), ejecutar el comando:

```
mvn install
```

Esta instrucción compilará y empaquetará los binarios del proyecto. Además, se instalarán estos binarios en el repositorio local de Maven, pero eso es otra cuestión que excede a los objetivos del TFC (se remite al lector interesado a la documentación de Maven).

Si todo acaba con éxito, se obtendrán los siguiente ficheros:

- **`formworkcore-1.0-SNAPSHOT.jar`:** situado dentro de la carpeta `target` del proyecto `formworkcore`. Este archivo contiene el *framework* que hemos desarrollado a lo largo del PFC. Conviene señalar que una aplicación web que quiera utilizar `Formwork` tendrá que incluir esta librería **y todas las librerías de las que `Formwork` depende** que se especifican en el archivo `pom.xml` del proyecto `formworkcore`.
- **`formworktest.war`:** situado dentro de la carpeta `target` del proyecto `formworktest`. Este fichero contiene la aplicación web de prueba de `Formwork` y está listo para ser desplegado en cualquier servidor de aplicaciones JEE 6.

  Como curiosidad, si echamos un vistazo dentro de la carpeta `formworktest/WEB-INF/lib` situada dentro de `target` podremos ver todas las dependencias de `Formwork`.

Pero nada de esto debe preocuparnos. Como se ha dicho, el proyecto está configurado para ejecutarse sobre Jetty con Maven. Para ejecutar la aplicación de ejemplo nos situamos dentro de la carpeta `formworktest` y ejecutamos el siguiente comando:

```
mvn jetty:run
```

Transcurrido un breve tiempo el servidor queda listo para ejecutar la aplicación en la URL:

```
http://localhost:8080/formworktest/index.fwp
```

[15] El proyecto se ha desarrollado con la versión 3.0.4 de Maven

En la figura 6.3 se ve la página principal de la aplicación `formworktest`.

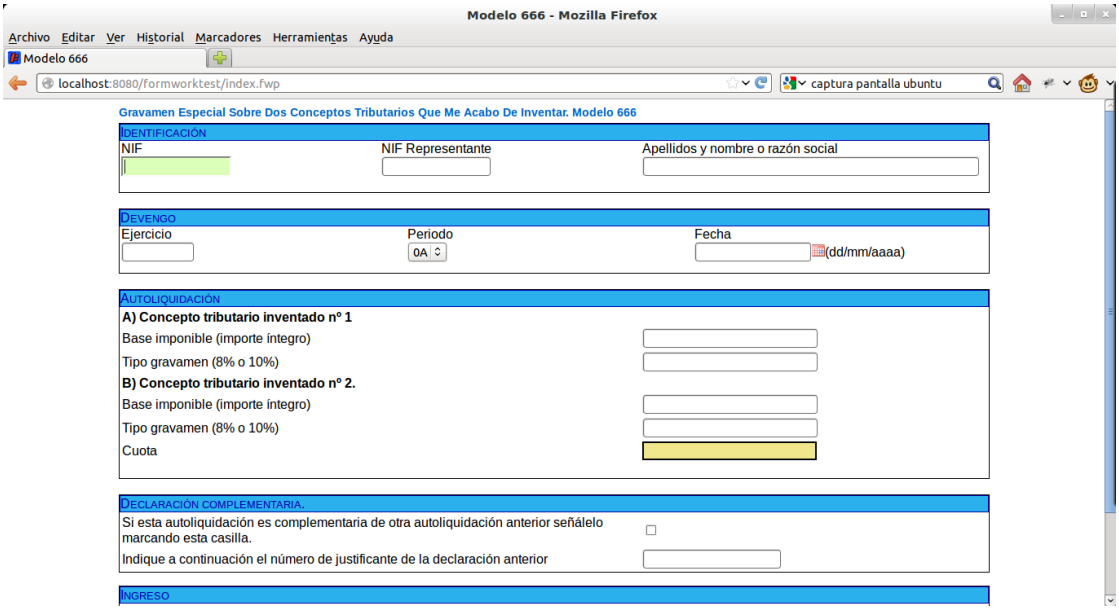

Figura 6.3: Formwork. Página principal.

6.4.2. Desde `eclipse`.

Si se dispone de un IDE `eclipse` con el *plugin* para Maven m2e, podemos hacer todo el proceso explicado en los puntos anteriores desde el propio entorno. Una vez que hayamos obtenido el código fuente por alguno de los métodos explicados anteriormente, procederemos a importar el proyecto dentro del *workspace* de `eclipse`.

Seleccionamos File → Import → Maven → Existing Maven Project

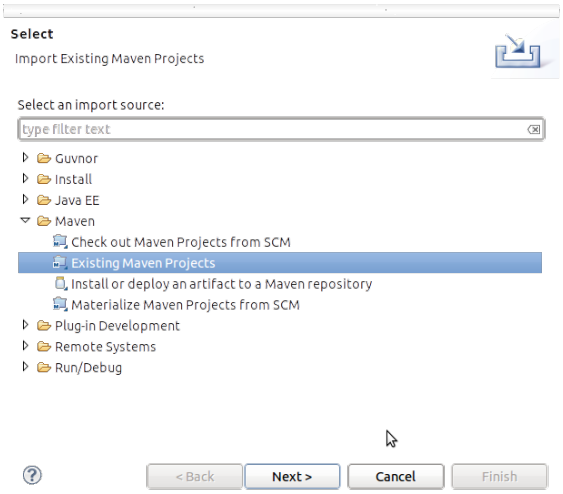

Figura 6.4: Formwork. Importar proyecto Maven.

En la siguiente pantalla seleccionamos la carpeta en la que descomprimimos o clonamos el repositorio del proyecto, en el ejemplo de la imagen 6.5, esta carpeta es `/home/cachocenso/pec3`.

Eclipse detecta el proyecto Maven padre y los módulos hijos. Pulsando en Finish se termina la importación y los proyectos aparecen en el *workspace* de eclipse.

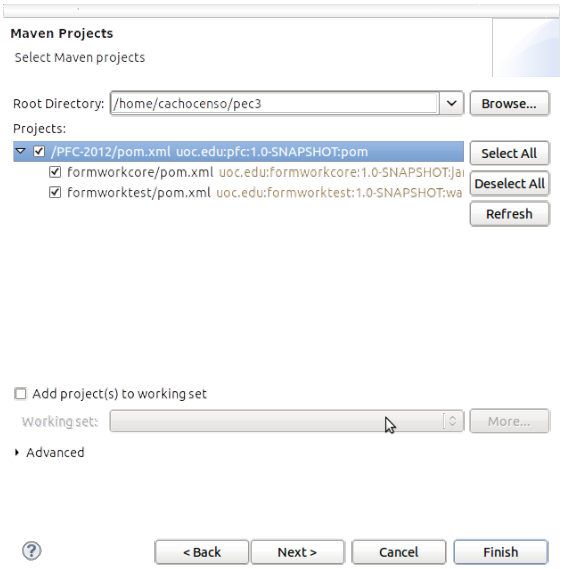

Figura 6.5: Formwork. Importar proyecto Maven carpeta.

Una vez importados los proyectos, ya podemos compilar y ejecutar desde el propio entorno. De forma parecida a como se hacía desde la línea de comandos, seleccionaremos el fichero `pom.xml` y haciendo clic derecho elegimos Run As → Maven install

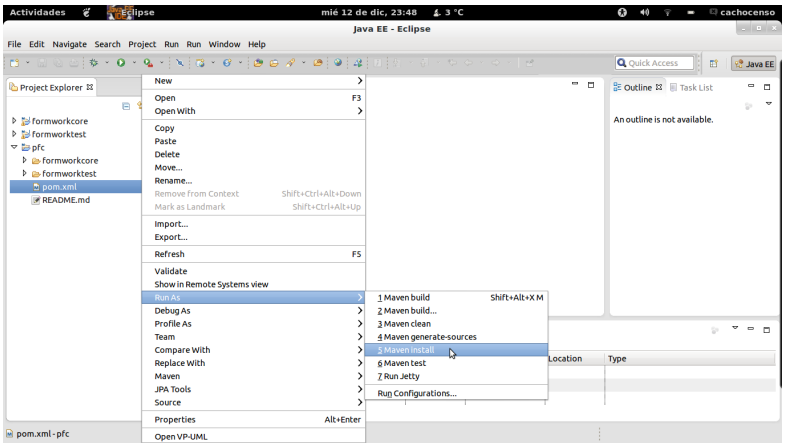

Figura 6.6: Formwork. Compilar desde eclipse.

Para ejecutar, ahora desde el `pom.xml` del proyecto `formworktest` seleccionamos Run As → Maven build ...

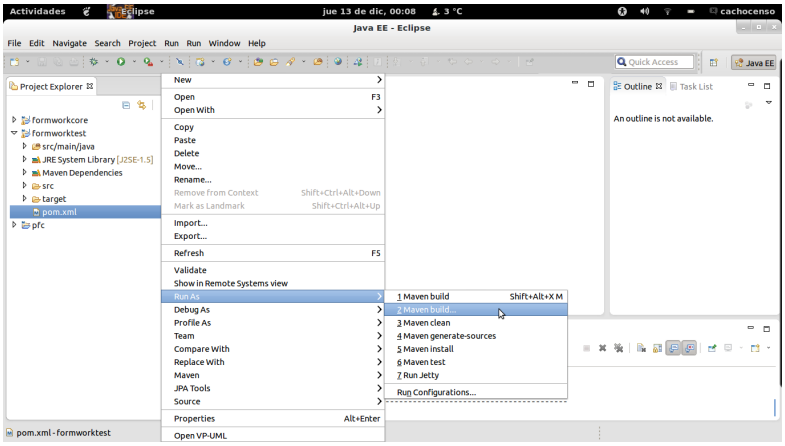

Figura 6.7: Formwork. Ejecutar desde eclipse.

En la ventana que aparece ponemos `jetty:run` en la casilla Goal, pulsamos Run y el servidor Jetty arranca.

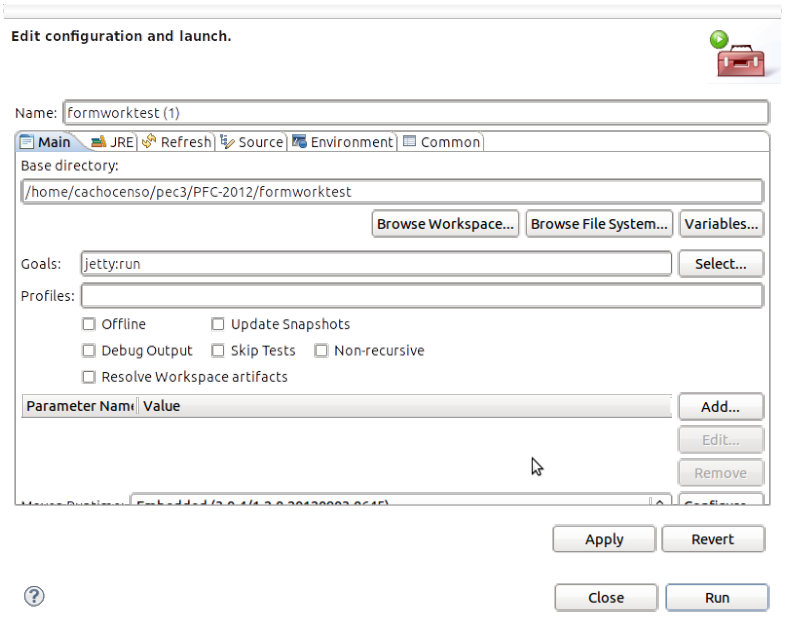

Figura 6.8: Formwork. Ejecutar desde eclipse.

Con el servidor arrancado podemos ejecutar la aplicación en la misma dirección que se indicó en el punto 6.4.1.

6.4.3. Otros medios de instalación.

Si por cualquier motivo no se desea o no se puede emplear Maven para construir el proyecto, en el apéndice A se dan instrucciones para la instalación de Formwork a partir de los binarios entregados en este PFC.

6.5. Implementación

Una vez que ya se tiene descargado y configurado el código fuente, podemos proceder a explicar los aspectos más interesantes, o que han conllevado mayor dificultad, durante la implementación del diseño que obtuvimos en el capítulo 5.

Veremos de una manera muy genérica la forma en que se ha implementado cada uno de los casos de uso identificados y nos detendremos algo más en aquellos que presenten algún interés adicional.

6.5.1. Paquete Infraestructura

Recordemos que el paquete Infraestructura engloba los casos de uso Init y Load.

Init

El caso de uso Init se ha implementado con un `ServletContextListener`, que implementa la clase `FormworkListener` del paquete `edu.uoc.pfc.formwork.infraestructura`. Durante el despliegue de la aplicación, en el método `contextInitialized()` se carga el fichero de configuración de Formwork, el fichero de reglas de negocio de DROOLS (si lo hubiera) a partir del cual se crea la base de conocimientos. Esta base de conocimientos (un objeto de la clase `KnowledgeBase`) se almacenará en un objeto `FormworkContext`, que a su vez se guarda en el `ServletContext`.

El fichero de configuración de Formwork se debe llamar `fw.xml` y debe situarse en la carpeta `WEB-INF` de la aplicación que utilice Formwork. En el fichero de configuración se indica cual es el fichero de reglas de negocio de la aplicación. En el listado 6.1 se muestra el contenido del fichero `fw.xml` de la aplicación de ejemplo.

Listado 6.1: fw.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <fw xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:noNamespaceSchemaLocation="http://www.uoc.edu/2012/fwp/formwork.xsd">
4
5         <rules>
6                 <rules-file>/WEB-INF/reglas.drl</rules-file>
7         </rules>
8 </fw>
```

El *listener* se debe declarar en el fichero `web.xml` de la aplicación:

Listado 6.2: Configurar el listener

```
1 <listener>
2         <description>Inicio de la aplicación</description>
3         <listener-class>
4                 edu.uoc.pfc.formwork.infraestructura.FormworkListener
5         </listener-class>
6 </listener>
```

El procesado del fichero de configuración de Formwork se hace con el API de JAXB con validación de esquema.

JAXB: *Java Architecture for XML Binding*

Load

Recordemos que el caso de uso Load se ejecuta la primera vez que el usuario solicita la página de la aplicación. En este momento entra en escena el *front controller* de Formwork. Como no podía ser de otra forma, se trata del *servlet* `FormworkServlet`.

El *servlet* se debe configurar también en el fichero `web.xml`:

Listado 6.3: Configurar FormworkServlet

```
<servlet>
        <description></description>
        <display-name>FormworkServlet</display-name>
        <servlet-name>FormworkServlet</servlet-name>
        <servlet-class>edu.uoc.pfc.formwork.infraestructura.FormworkServlet</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>FormworkServlet</servlet-name>
        <url-pattern>*.fwp</url-pattern>
</servlet-mapping>
<servlet-mapping>
        <servlet-name>FormworkServlet</servlet-name>
        <url-pattern>/au/*</url-pattern>
</servlet-mapping>
```

Como vemos en el listado 6.3, también se configuran un par de *mappings* para el *servlet*. Para la implementación de este caso de uso nos interesa el *mapping* de las URL `*.fwp`.

`FormworkServlet` no sobreescribe ninguno de los métodos `doXXXX()` para tratar las peticiones. Todo se hace desde el método `service()`. Lo primero es distinguir el tipo de petición en función de la URI recibida. Se van a tratar dos tipos de peticiones distintas, una por cada *mapping* configurado en `web.xml`. Las URI `*.fwp` para cargar la página web de la aplicación y las URI `/au/*` para las peticiones de cambio desde la página ya cargada o para la carga de recursos.

El caso comienza, como se ha dicho, en el método `service` del *servlet*. Como se ha recibido una petición con la URI `index.fwp`. Esta es la página principal de la aplicación y debe cumplir con el esquema W3C (el esquema se puede ver en `formwork.xsd`, en la carpeta META-INF dentro de la rama de recursos del proyecto `formworkcore`). El proceso comienza en el método `load()`. En el listado 6.4 podemos ver el código de `index.fwp`

La página incluye un apartado de identificación, otro de devengo, otro de datos económicos, otro de datos de declaración complementaria y un último apartado de datos del ingreso. Para finalizar, se incluye un botón para realizar la presentación de la declaración.

Lo primero que se hace es el *parse* del fichero `index.fwp` mediante JAXB. Si la página es correcta (como es el caso en la aplicación de ejemplo), obtendremos la colección de objetos JAXB que representan su contenido.

A partir de estos objetos se construye el árbol de componentes del servidor, todos ellos objetos que extienden la clase `Componente` del paquete `edu.uoc.pfc.formwork.ui`. Estos objetos son las clases:

Listado 6.4: index.fwp

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <formulario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:noNamespaceSchemaLocation="http://www.uoc.edu/2012/fwp/formwork.xsd"
5          id="m666"
6          controlador="edu.uoc.pfc.formworktest.controller.TestController">
7
8          <titulo>Modelo 666</titulo>
9          <descripcion> Gravamen Especial Sobre Dos Conceptos Tributarios Que Me Acabo De Inventar. Modelo 666
10         </descripcion>
11         <apartado id="ap1" tipo="identificacion" contenido="nif,representante,nombre" />
12         <apartado id="ap2" tipo="devengo"
13                 contenido="ejercicio, periodo(0A 1T 2T 3T 4T), fecha" />
14         <apartado id="ap3" tipo="partidas" titulo="Autoliquidación">
15                 <etiqueta
16                         valor="A) Concepto tributario inventado nº 1" />
17                 <partida id="p01" etiqueta="Base imponible (importe íntegro)"
18                         tipo="cantidad" />
19                 <partida id="p02" etiqueta="Tipo gravamen (8% o 10%)" tipo="cantidad" />
20                 <etiqueta
21                         valor="B) Concepto tributario inventado nº 2." />
22                 <partida id="p03"
23                         etiqueta="Base imponible (importe íntegro)"
24                         tipo="cantidadNegativa" />
25                 <partida id="p04" etiqueta="Tipo gravamen (8% o 10%)" tipo="cantidad" />
26                 <partida id="p05" etiqueta="Cuota" tipo="cantidad" calculado="si"/>
27         </apartado>
28         <apartado id="ap4" tipo="complementaria"/>
29         <apartado id="ap5" tipo="pago" />
30 </formulario>
```

- `Formulario`
- `Apartado`
- `Etiqueta`
- `Partida` y sus subclases `PartidaCadena`, `PartidaBoolean`, `PartidaCantidad`, `PartidaLista` y `PartidaPeriodo`, una para cada uno de los tipos de datos que se van a pedir al usuario.

Además se instanciará y configurará la clase que hace de controlador de la aplicación, que se define en la página `index.fwp` (ver línea 6 del listado 6.4). Esta clase tiene que implementar la interfaz `IController` del paquete `edu.uoc.pfc.formwork.ui`, aunque lo más normal es que extienda la clase abstracta `GenericController` del mismo paquete.

Posteriormente se verá con más detenimiento esta clase. Por el momento cabe destacar que desde el *framework* se instanciará y se buscará un atributo `HttpSession` que este anotado con la anotación `@Session` como se ve en el listado 6.5

Listado 6.5: GenericController.java - Anotación Session

```
1  public class GenericController implements IController {
2  ...
3          @Session
4          private HttpSession session;
5  ...
6  }
```

Si se encuentra, el *framework* inyecta la sesión HTTP en el controlador mediante el API de reflexión. Una vez configurado, el controlador se guarda en la misma sesión.

Una vez construido este árbol de componentes, se almacena en la sesión y se ejecuta el caso de uso Render para crear la página HTML que finalmente se devolverá al cliente.

6.5.2. Paquete UI

En este paquete teníamos el caso de uso Render.

Render

Llegamos a Render una vez que la página se ha cargado y almacenado en la sesión. Es el momento de transformar ese árbol de componentes en código HTML + JavaScript. La "renderización" se va a realizar con el apoyo del motor de plantilla Freemarker.

El *rendering* se realiza con el apoyo de la clase `HTMLRenderer`. Esta clase recibe el árbol de componentes y el `Writer` de salida en su método `render()`. En este método se carga la plantilla que genera la página principal, se inserta todo el árbol de componentes como modelo de datos de la plantilla y se ejecuta el procesado del modelo, como se puede ver en el listado 6.6

Listado 6.6: Renderizado

```
1
2  public void render(Componente root, Writer out) {
3
4          if (!(root instanceof Formulario)) {
5                  throw new IllegalArgumentException(
6                                  "Se esperaba un objeto Formulario");
7          }
8
9          Configuration configuration = new Configuration();
10         configuration.setClassForTemplateLoading(getClass(), "/");
11
12         try {
13                 Template template = configuration
14                                 .getTemplate("templates/formwork.ftl");
15
16                 Map<String, Componente> dataModel = new HashMap<String, Componente>();
17
18                 dataModel.put("form", root);
19                 template.process(dataModel, out);
20         } catch (IOException e) {
21                 logger.fatal("No se encuentra la plantilla", e);
22         } catch (TemplateException e) {
23                 logger.fatal("Error haciendo rendering", e);
24         }
25 }
```

Vamos a ver un poco más detenidamente el código de la plantilla Freemarker `formwork.ftl`. Tenemos este código en el listado 6.7

Interesa, en el preámbulo de la plantilla, la carga de los recursos que utilizará la parte cliente del `framework`. Por una parte, se utilizan dos *frameworks* JavaScript: jQuery y jQuery UI. El primero nos servirá de apoyo para convertir Formwork en un *framework* AJAX. El segundo nos servirá para crear algunos componentes gráficos del cliente (captura de fechas y mensajes de error). Estos dos *frameworks* se cargan directamente desde su página web (para desarrollar con Formwork es necesario disponer de una conexión a Internet activa).

Pero además se cargan un par de recursos desde el propio *framework*. Una hoja de estilos `formwork.css` y un fichero JavaScript `formwork.js`[16]. Si observamos sus URI de descarga,

[16] Veremos este archivo con más detenimiento a continuación.

Listado 6.7: formwork.ftl

```
1
2  <!DOCTYPE html>
3  <head>
4    <title>${form.titulo}</title>
5    <link rel="stylesheet" type="text/css" href="au/~/css/formwork.css"/>
6    <link rel="stylesheet" type="text/css"
7                href="http://code.jquery.com/ui/1.9.2/themes/base/jquery-ui.css"/>
8    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.js"></script>
9    <script src="http://code.jquery.com/ui/1.9.2/jquery-ui.js"></script>
10   <script src="au/~/js/formwork.js"></script>
11
12 </head>
13
14 <body>
15   <div class="contenido">
16         <div class="titulo">${form.descripcion}</div>
17         <#list form.apartados as ap>
18                 ${ap.render()}
19         </#list>
20
21         <p/>
22         <div id="envio" class="apartado">
23                 <div style ="margin: 5px">
24                         <center><button>Firmar y enviar</button></center>
25                 </div>
26         </div>
27   </div>
28 </body>
```

son de la forma `au/~/css/formwork.css` y `au/~/js/formwork.js`.

Estas URI llegan al *servlet* `FormworkServlet`, que gracias a ese carácter `~` distingue que se le está pidiendo un recurso, lo busca en su *classpath* y lo sirve. En el cuerpo de la plantilla, se extraen el título y la descripción del objeto `Formulario` y se recorre su lista de apartados para que cada uno se "dibuje a sí mismo". Finalmente al cliente se le envía la página que ya vimos en la figura 6.3.

Una vez que la página aparece en el navegador del cliente, este comenzará a interactuar con la misma, con lo que estaremos ya en el caso de uso Service.

6.5.3. Paquete Servicio

Service

El caso de uso Service comienza cuando el usuario ejecuta alguna acción sobre el interfaz gráfico de la aplicación, como por ejemplo introducir un NIF en la casilla correspondiente. Antes de describir lo que sucede, vamos a echar un vistazo al contenido del fichero `formwork.js` (listado 6.8).

En este código JavaScript, que ciertamente es duro de leer, estamos haciendo varias cosas. En primer lugar, transformamos los `<input>` que tengan un atributo "fecha" en objetos `datepicker` de jQuery UI, lo que desplegará un calendario para seleccionar la fecha cuando dicho componente gane el foco, tal y como se ve en la figura 6.9

A continuación (línea 13) se establece un manejador del evento `onChange` para todos los

Listado 6.8: formwork.js

```
1
2  // Establezco un manejador para el evento onReady del documento
3  // mediante jQuery
4  $(document).ready(function() {
5
6          // Inicializo los campos fecha con jQuery UI datepicker
7          $(function() {
8                  $("[fecha]").datepicker();
9          });
10
11         // Establezco un manejador para los eventos
12         // onChange de los elementos input
13         $("input").change(function() {
14                 // Cada vez que un input cambie su valor
15                 // se hace una llamada AJAX al servidor.
16                 $.ajax({
17                         url : "au/update",
18                         type : "post",
19                         dataType : "json",
20                         data : {
21                                 "id" : this.id,
22                                 "value" : this.value
23                         },
24                         success : function(result) {
25                                 if (result.resultado == "ERROR") {
26                                         $.each(result.responseObjects, function(j, error) {
27                                                 var partida = "#" + error.idPartida;
28
29                                                 $(partida).tooltip({
30                                                         content : error.mensaje,
31                                                         items : "input",
32                                                         tooltipClass : "ui-state-error",
33                                                         show : true
34                                                 });
35
36                                                 $(partida).tooltip("open");
37
38                                         });
39                                 } else if (result.resultado == "SUCCESS") {
40                                         $.each(result.responseObjects, function(j, partida) {
41                                                 $("#" + partida.id).val(partida.value);
42                                                 try {
43                                                         $("#" + partida.id).tooltip("destroy");
44                                                 } catch (e) {
45                                                         // Si el elemento partida.id
46                                                         // no tenia un tooltip asignado por un erro
47                                                         // un ejecución anterior se produce una excepción
48                                                         // que puedo ignorar.
49                                                 }
50                                         });
51                                 }
52                         }
53                 });
54         });
55 });
```

elementos `<input>` de la página. Este manejador del evento será una función desde la que se hará una llamada AJAX al servidor. De este modo, cada vez que el usuario cambie el valor de algunos de los campos del formulario se originará una petición asíncrona al servidor.

Esta llamada se hará a la URL `au/update` que, recordemos, está mapeada a `Formworkservlet`. En la petición se enviarán dos parámetros: el identificador de la partida cambiada y su nuevo valor.

El servidor procesará la petición y devolverá una respuesta que se recibirá en la función *callback* que se declaró en el parámetro `success` de la llamada (ver línea 24). Esta respuesta se espera en formato JSON y puede consistir en un mensaje de error (ver figura 6.10) o una lista de las partidas cuyo valor hay que modificar como resultado de la petición (ver figura 6.11.

JSON *JavaScript Object Notation*

En función del tipo de resultado, se mostrará un *tooltip* con el mensaje de error producido en la partida donde se produjo el error o se actualizará el valor de las partidas modificadas durante la ejecución de la petición en el servidor manipulando directamente el árbol DOM.

Figura 6.9: Formwork. Date picker

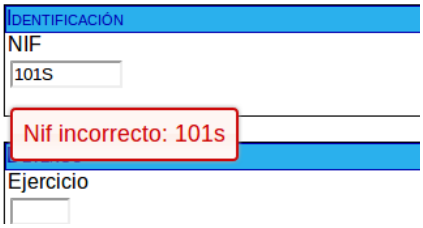

Figura 6.10: Formwork. NIF erróneo

Fire Business Rules

Recordemos que esté caso de uso se ejecutaba desde el controlador de la aplicación. Esta caso de uso comienza en el instante en que el evento de cambio de alguno de los elementos `<input>` de la página se traslada al controlador, llegando al método `onEvent` del mismo que deben implementar todas las clases controladoras.

Antes, en el *servlet* se recibió la petición AJAX. Lo que el *servlet* hace en este caso es:

- Recuperar el controlador de la sesión.
- Crear un evento (clase `FormworkEvent`) y encapsular la información del evento producido (partida cuyo valor ha cambiado y nuevo valor).

En este instante es en el que se invoca el método `onEvent()` del controlador. En este método se invocarán las reglas de negocio cuyo resultado puede ser una lista de errores detectados o una lista de las partidas que se han modificado con la ejecución de las reglas (esta lista contiene siempre al menos la partida cuyo cambio originó el caso de uso).

En el listado 6.9 se puede ver un ejemplo de como podría ser el contenido de un método `onEvent`, en este caso el del controlador de la aplicación de ejemplo, la clase `TestForm`, empleando el motor de reglas de negocio.

Del objeto `FormworkEvent` se extrae la partida y su nuevo valor, se crea una sesión de DROOLS (en este ejemplo se trata de una sesión sin estado, aunque bien podría ser con estado si la aplicación lo requiriese). Se inicializan las variables necesarias para el correcto funcionamiento de las reglas y se ejecutan las reglas asociadas a la partida modificada.

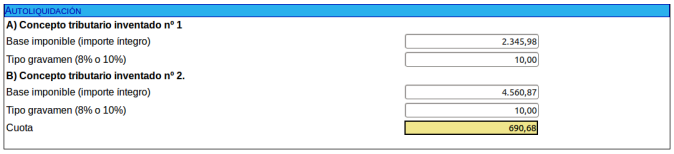

Figura 6.11: Formwork. Partidas modificadas

Listado 6.9: TestForm.onEvent

```
public void onEvent(FormworkEvent evt) {

        // Se extrae la partida y se le asigna el nuevo valoir
        Partida componente = (Partida) evt.getTarget();
        componente.setValue(evt.getValue());

        // Obtenemos y configuramos la base de conocimientos
        KnowledgeBase knowledgeBase = ((FormworkContext) session
                        .getServletContext().getAttribute(Attributes.FWCONTEXT))
                        .getKnowledgeBase();

        // Creamos una sesión de DROOLS y establecemos el árbol de componentes como
        // variable global

        ......

        // Ejecutamos las reglas asociadas al componente cuyo valor ha cambiado.
        // Al regresar de la ejecución de las reglas, una de las dos listas
        // tendrá contenido.
        ksession.execute(componente);
}
```

En el listado 6.10 podemos ver como ejemplo la regla de validación de las casillas NIF.

Listado 6.10: reglas.drl

```
rule "nif-error"
salience 10
when
        $p : Partida((id == "nif"  id == "nifr"), $v:value, !isNifNie(value))
then
        Mensaje $m = new Mensaje(Mensaje.TipoMensaje.ERROR, "Nif incorrecto: " + $v,
                                 $p.getId());
        errores.add($m);
end
```

Recordemos que todas las reglas de negocio están en el fichero `reglas.drl` situado en la carpeta `WEB-INF` de la aplicación de ejemplo `formworktest`. Al acabar la ejecución de las reglas una de las listas, bien la de errores o bien la de partidas modificadas, tendrán contenido. Después, al regresar de este método, se ejecutará el caso de uso Render response.

Render response

Este caso se ejecuta después de el caso de uso Service. Básicamente, se recoge el resultado de la ejecución de las reglas (según qué lista, la de errores o la de partidas, tenga contenido) y se prepara el objeto de respuesta, un objeto de la clase `ResponseJson` con la información correspondiente. Después, se serializa dicho objeto a formato JSON, con el apoyo de la librería Gson, y se envía de regreso al navegador.

En el navegador, se invoca la función *callback* que indicamos en la llamada AJAX (ver

listado 6.8) para que se actualice el estado del cliente manipulando el árbol DOM.

Presentar

Este caso se ejecuta cuando el usuario pulsa el correspondiente botón (`Enviar` o `Firmar y enviar`) en el formulario.

Este evento lo maneja también jQuery, que hace una petición al *servlet* con la url `au/submit`. El *servlet* pasa la llamada al controlador (método `onPresentar`) donde se realizan las validaciones finales (campos obligatorios, etc). Las validaciones puede hacerlas el *framework* si se ha optado por usar DROOLS.

Si las validaciones se superan con éxito, el *servlet* le pide al controlador que construya el registro que se va a presentar. La implementación por defecto está en `GenericController` (método `getRegistro()`) y consiste en transformar la partidas en una lista objetos `Campo` de tipo clave-valor, que se transmiten al cliente en formato JSON.

En el cliente de nuevo se muestra al usuario el contenido del registro que se va a presentar. Se firma con el certificado de cliente (si la aplicación está así configurada) y se presenta. Tanto la opción de firma como la url de presentación final[17] se configuran en el fichero `fw.xml`, tal y como se muestra en el listado 6.11.

Listado 6.11: fw.xml. Configuración de la presentación

```
1 <presentation>
2         <signed>true</signed>
3         <url>dummypres</url>
4 </presentation>
```

Se ha habilitado un sencillo sistema de firma mediante Javascript usando el objeto `window.crypto`. que solo se soporta completamente por Firefox, por lo que la solución propuesta sirve para este navegador exclusivamente. Se ofrece más información sobre las posibilidades de la firma digital en al apéndice B.

6.6. Conclusiones y mejoras

En este PFC hemos aprendido que JEE se trata de una tecnología ya muy madura para el desarrollo de aplicaciones empresariales, que cuenta con una gran cantidad de *frameworks* de apoyo para cada uno de los aspectos que cubre.

Con el desarrollo de un nuevo *framework* para la capa de presentación nos hemos enfrentado a los problemas que plantea un proyecto de estas características: conocimiento profundo de las tecnologías utilizadas, diseño de una buena API, aprovechamiento del trabajo ya existente etc. En definitiva, lo que supone el paso de desarrollador a arquitecto de software.

Con el *framework* aquí desarrollado se podrían llegar a poner aplicaciones en producción con la introducción de algunas mejoras, entre las que destacarían:

[17] En la aplicación de ejemplo se ha configurado un *servlet* que simula la presentación con y sin errores.

- Soporte de internacionalización y localización. Tanto para múltiples idiomas como formato de números y fechas.
- Ampliar el juego de componentes necesario.
- Solución definitiva del problema de la firma electrónica. En este momento, si se quisiera optar por una alternativa multiplataforma y multinavegador solamente nos quedaría el cliente de `@firma`.

Bibliografía

[1] ALEXANDER, CHRISTOPHER: The Timeless Way of Building. OXFORD UNIVERSITY PRESS, 1979.

[2] ALUR, DEEPAK; CRUPI, JOHN y MALKS, DAN: Core J2EE Patterns: Best Practices and Design Strategies (2nd Ed.). Prentice Hall / Sun Microsystems Press, 2003.

[3] AMADOR, LUCAS: Drools Developer's Cookbook. Packt Publishing., 2012.

[4] ARRANZ SANTAMARÍA, JOSE MARÍA: «ItsNat Reference manual», 2011. http://itsnat.sourceforge.net/php/support/docs/manual.htm

[5] BALI, MICHAL: Drools JBoss Rules 5.0 Developer's Guide. Packt Publishing., 2009.

[6] BIEN, ADAM: Real World Java EE Patterns. Rethinking Best Practices. press.adam-bien.com, 2009.

[7] BURNS, ED y KITAIN, ROGER: «JavaServer™ Faces Specification», 2008. http://download.oracle.com/otndocs/jcp/jsf-1.2-REVB-mrel-eval-oth-JSpec/

[8] CABOT SAGRERA, JORDI; CAMPS RIBA, JOSEP MARIA; CEBALLOS VILLACH, JORDI; DURAN MUÑOZ, FRANCISCO JAVIER; MORENO VERGARA, NATHALIE; ROMERO SALGUERO, JOSÉ RAÚL y VALLECILLO MORENO, ANTONIO: Ingeniería del software de componentes y sistemas distribuidos. Fundació per a la Unversitat Oberta de Catalunya, 2006.

[9] FERNÁNDEZ GONZÁLEZ, JORDI; PRADEL I MIQUEL, JORDI y RAYA MARTOS, JOSE ANTONIO: Ingeniería del software orientada a objetos. Fundació per a la Unversitat Oberta de Catalunya, 2005.

[10] GAMMA, ERICH; HELM, RICHARD; JOHNSON, RALPH y VLISSIDES, JOHN: Design Patterns: Elements of Reusable Object-Oriented Software. ADDISON-WESLEY, 1994.

[11] LAMPORT, LESLIE: LaTeX, A Document Preparation System (2nd Ed.). ADDISON-WESLEY, 1994.

[12] SCHMULLER, JOSEPH: Aprendiendo UML en 24 horas. Prentice Hall., 2004.

Apéndice A

Instalación sin Maven ni dependencias

Por restricciones de tamaño, no es posible entregar el binario de la aplicación de prueba del *framework* construido en este PFC con todas las librerías de las que depende.

Pese a que la mejor forma de construir, desplegar y ejecutar la aplicación de ejemplo es mediante Maven, tal y como se explica en la sección 6.3, en este apéndice se ofrecen instrucciones para la instalación de la aplicación a partir de los binarios entregados.

Lo primero que necesitamos es el binario de la aplicación de ejemplo, `formworktest.war` entregado junto con esta memoria.

En segundo lugar necesitaremos el binario del *framework*, el archivo `formworkcore-1.0-SNAPSHOT.jar`, también entregado en este PFC. Después necesitamos todas las dependencias del *framework*. En el cuadro A.1 se ofrecen una lista con enlaces a todas esas librerías para facilitar su descarga:

Una vez que ya disponemos de todo lo necesario (binarios del *framework*, de la aplicación de test y de todas las dependencias), podemos proceder a instalar la aplicación.

Crear un war directamente instalable en un servidor JEE

Para conseguir un fichero WAR directamente instalable en cualquier servidor de aplicaciones JEE debemos seguir los pasos que a continuación se indican:

1. Copiar el archivo `formworktest.war` en alguna carpeta temporal, p.e. en `/home/cachocenso/formworwtest`.
2. Desempaquetar el fichero `formworktest.war` mediante la orden:

   ```
   jar -xvf formworktest.war
   ```

3. Copiar `formworkcore-1.0-SNAPSHOT.jar` y todas las librerías de las que depende dentro de la carpeta `WEB-INF/lib`.
4. Ahora hay que volver a empaquetar el archivo WAR, mediante la orden:

   ```
   jar -cvf formworktest.war .
   ```

DROOLS
https://dl.dropbox.com/u/69538005/formwork/lib/drools-core-5.4.0.Final.jar https://dl.dropbox.com/u/69538005/formwork/lib/drools-compiler-5.4 https://dl.dropbox.com/u/69538005/formwork/lib/mvel2-2.1.0.drools16.jar https://dl.dropbox.com/u/69538005/formwork/lib/knowledge-api-5.4.0.Final.jar https://dl.dropbox.com/u/69538005/formwork/lib/knowledge-internal-api-5.4.0.Final.jar https://dl.dropbox.com/u/69538005/formwork/lib/antlr-3.3.jar https://dl.dropbox.com/u/69538005/formwork/lib/antlr-runtime-3.3.jar https://dl.dropbox.com/u/69538005/formwork/lib/stringtemplate-3.2.1.jar
https://dl.dropbox.com/u/69538005/formwork/lib/ecj-3.5.1.jar
Freemarker
https://dl.dropbox.com/u/69538005/formwork/lib/freemarker-2.3.19.jar
Apache Commons
https://dl.dropbox.com/u/69538005/formwork/lib/commons-io-1.3.2.jar
Gson
https://dl.dropbox.com/u/69538005/formwork/lib/gson-2.2.2.jar
Log4j
https://dl.dropbox.com/u/69538005/formwork/lib/log4j-1.2.17.jar https://dl.dropbox.com/u/69538005/formwork/lib/slf4j-api-1.7.2.jar https://dl.dropbox.com/u/69538005/formwork/lib/slf4j-log4j12-1.7.2.jar
Todas las dependencias en un único zip
https://dl.dropbox.com/u/69538005/formwork/lib/formwork-1.0-SNAPSHOT-libs.zip

Cuadro A.1: Dependencias Formwork

5. Se vuelve a generar el fichero `formworktes.war` que ya se puede instalar en un servidor de aplicaciones.

Poner el *framework* y sus dependencias a disposición del servidor de aplicaciones.

Otra opción, útil sobre todo si se van a desplegar varias aplicaciones que usen nuestro *framework* en el mismo servidor, es poner el binario de Formwork y todas sus dependencias a disposición del servidor de aplicaciones.

Cada servidor de aplicaciones JEE tiene su sistema, aquí vamos a explicarlo para el servidor Glassfish, por ser la implementación de referencia de JEE.

Bastaría con copiar `formworkcore-1.0-SNAPSHOT.jar` y sus dependencias dentro de la carpeta `GLASSFISH-HOME/glassfish/lib`, siendo GLASSFISH-HOME la carpeta donde se ha instalado Glassfish. Con esto se pondría Formwork a disposición de todas las aplicaciones que en dicho servidor se desplieguen.

Desplegar la aplicación en Glassfish

Una vez que se tiene el fichero WAR listo, se procede a su despliegue en Glassfish. La forma más sencilla, sin necesidad de usar consola administrativa, es arrancar el servidor mediante la orden:

```
GLASSFISH-HOME/bin/asadmin start-domain domain1
```

Una vez que el servidor haya arrancado, simplemente copiamos `formworktest.war` en la carpeta `autodeploy` del dominio:

```
cp formworktest.war GLASSFISH-HOME/glassfish/domains/domain1/autodeploy
```

Transcurrido unos segundos el servidor desplegará la aplicación y podremos ejecutarla en la dirección habitual.[1]

```
http://localhost:8080/formworktest/index.fwp
```

[1] Se asume que no se ha modificado el puerto por defecto de Glassfish, el 8080.

Apéndice B

Firma digital

En este PFC se ha propuesto una sencilla solución de firma digital. Lamentablemente, debido a las diferencias entre los navegadores del mercado la solución propuesta solamente funciona con Firefox, que es el único navegador que actualmente soporta por completo el objeto `window.crypto`.

Actualmente existe una discusión viva entre los expertos en seguridad informática sobre la conveniencia o no de confiar algo tan importante como es la criptografía en manos del lenguaje Javascript. Algunos rechazan de plano esta posibilidad, como se puede leer en el artículo de Matasano Security[1], otros consideran que se le debe dar una oportunidad con la inmediata publicación del estándar HTML 5 y su progresiva actualización por parte de los principales navegadores.

Porque la otra opción es la adoptada por la mayoría de las administraciones públicas españolas, la firma mediante un *applet* Java, el cliente de `@firma`[2], bien mediante un integrador propio, bien a través de la red SARA.

Pero esta solución también presenta problemas similares a los que se comentan en el artículo de Matasano. Java en los navegadores se ha convertido en el primer vector de entrada para los ataques de *hackers*, hasta el punto de que los navegadores aconsejan desinstalar o al menos deshabilitar Java de las máquinas cliente.

Por lo tanto, la firma digital en los navegadores es un asunto que no está todavía claro; se elige la solución que se elija presentará problemas de seguridad y de portabilidad entre los diferentes navegadores.

[1] `http://hellais.wordpress.com/2011/12/27/how-to-improve-javascript-cryptography/`
[2] `http://forja-ctt.administracionelectronica.gob.es/web/clienteafirma`

Apéndice C

Planificación del proyecto

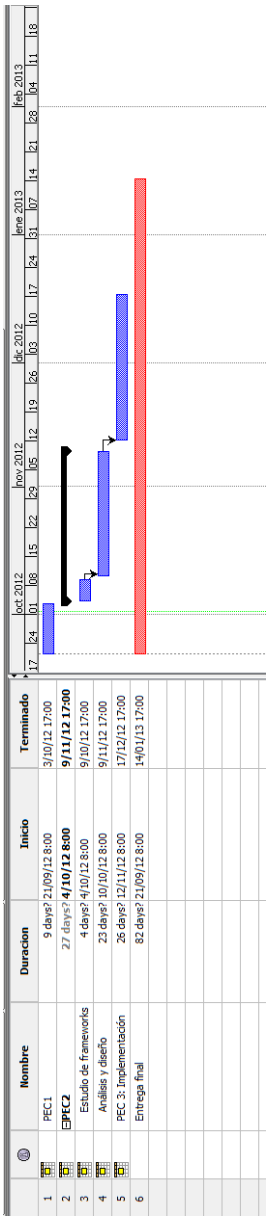

Figura C.1: Planificación del proyecto