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

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

Diseño e implementación de un framework de presentación para J2EE

Jesús Ciriano Sierra
Ingeniería Informática

Óscar Escudero
Universitat Oberta de Catalunya
17 de enero de 2011

A mi abuela Aurita

Agradecimientos

Quiero agradecer el apoyo y la paciencia que ha demostrado mi esposa Laura durante la finalización de mis estudios. También agradecer a mis padres Jesús e Isabel por inculcarme desde pequeño la importancia de la educación y estudios en la sociedad en la que vivimos. Por último agradecer a mi consultor Oscar Escudero por el soporte ofrecido a lo largo del proyecto para la buena finalización del mismo.

Resumen ejecutivo

En este proyecto se diseña e implementa un framework para la capa de presentación de aplicaciones J2EE. Primero se ha realizado un estudio de algunos frameworks de presentación existentes atendiendo a la funcionalidad que proporcionan y estudiando su arquitectura con el objetivo de obtener unos requisitos de partida. Una vez finalizado este estudio, se ha optado por crear un framework basado en acciones.

En la elaboración del diseño se ha tenido en cuenta patrones de diseño, así como buenas prácticas y recomendaciones publicadas en diferentes artículos y libros especializados del sector. La implementación se ha realizado utilizando tecnologías estándar. El framework desarrollado, jNeko, proporciona una arquitectura base bien diseñada construida mediante componentes software reutilizables que trabajan para solventar problemas comunes de la capa de presentación. Los desarrolladores pueden adoptar, extender y configurar el framework para así crear la capa de presentación de aplicaciones web de una forma rápida, flexible y sencilla de mantener.

Índice

Índice de figuras

1. Introducción

1.1. Justificación y contexto del presente proyecto

El avance espectacular que ha sufrido Internet y en concreto la web desde los años 90, ha provocado que sea más importante que nunca definir metodologías y tecnologías estándares que permitan construir aplicaciones web eficientes, robustas y fáciles de mantener. Una de estas tecnologías es la plataforma Java Enterprise Edition (J2EE) que permite desplegar aplicaciones y servicios web dentro de una arquitectura estándar de n-capas siendo las más representantes las capas de presentación, negocio e integración.

Una de las capas más importantes es la capa de presentación donde se encapsula toda la lógica necesaria para dar servicio a los clientes que acceden al sistema. Esta capa intercepta las peticiones de cliente, provee capacidades de login, mantiene la sesión del cliente, controla el acceso a los servicios de negocio y construye respuestas que se devuelven al cliente. J2EE no incluye ninguna utilidad, librería, API o mecanismo que proporcione esta funcionalidad, sino que permite la libertad y flexibilidad de diseñar e implementar cualquier solución mediante los mecanismos que proporciona: Servlets, JSPs, custom tags, etc.

Si no se realiza un buen diseño en esta capa, una aplicación web puede llegar a volverse compleja y difícil de mantener. Por este motivo, el diseño de la capa de presentación ha sido objeto de diversos estudios, dando lugar a la publicación de recomendaciones, buenas prácticas y diversos patrones de diseño con el objetivo de facilitar la creación de diseños exitosos así como reducir la complejidad.

El presente proyecto consiste en el diseño e implementación de un framework para la capa de presentación J2EE, posibilitando la reutilización del mismo de forma que simplificará el desarrollo de nuevas aplicaciones web. Asimismo, se construirá una aplicación de ejemplo que hará uso del framework y que permitirá mostrar claramente el funcionamiento del mismo además de servir de ejemplo práctico. El mayor beneficio que proporciona este framework es que los desarrolladores podrán obtenerlo, ensamblarlo, configurarlo y adaptarlo a sus necesidades sin tener que emplear recursos en solucionar y escribir de nuevo la infraestructura de la capa de presentación. De esta manera se acortarán los desarrollos al aislarles de mucha problemática común que surge al implementar la capa de presentación y permitiéndoles concentrarse en las particularidades de la nueva aplicación a construir.

Con el objetivo de diseñar un buen framework e identificar las necesidades básicas que va a cubrir, se realizará antes un estudio de patrones de diseño y buenas prácticas susceptibles de utilizar en esta capa así como un estudio y evaluación de diferentes frameworks existentes en el mercado, analizando los diferentes enfoques y características que tienen en común.

1.2. Objetivos generales y específicos

El objetivo principal es el desarrollo de un framework para la capa de presentación bien diseñado, siguiendo los patrones de diseño, recomendaciones y buenas prácticas de la capa de presentación. Para lograr este objetivo general, se fijan tres objetivos específicos que se detallan a continuación:

1. Realizar un estudio del dominio

Se pretende identificar y detectar la funcionalidad a cubrir por el framework. Para realizar este análisis de requisitos se realizará el estudio del dominio realizando las siguientes actividades:

- Estudiar la evolución de los diseños de la capa de presentación J2EE (Model I y Model 2)
- Estudiar los patrones de diseño orientados a la capa de presentación: Intercepting Filter, Front Controller, View Helper, Composite View y Service to Worker
- Identificar diferentes frameworks existentes en el mercado y realizar una comparativa de los mismos
- Evaluar un subconjunto representativo de estos frameworks, en concreto Struts, Struts 2 y JSF atendiendo a la funcionalidad que proporcionan y cómo la implementan.
- Definir los requisitos funcionales del framework a construir

2. Diseñar e implementar un framework para la capa de presentación J2EE

Una vez se haya realizado el análisis de requisitos, se diseñará e implementará el framework.

Para el diseño se intentará aplicar patrones de diseño siempre que tengan sentido aplicarlos. Además el diseño separará de forma clara el modelo (lógica de aplicación que interactúa con los datos) de la vista (html presentado al cliente) y el controlador (instancia que pasa la información entre la vista y el modelo). Respecto a la implementación, el framework proveerá clases y componentes reutilizables, basados en tecnologías estándar como Servlets, JSP, xml, etc., y su desarrollo se realizará de forma iterativa en incrementos y refactorizando el código si fuese necesario.

El desarrollador será el responsable de extender o utilizar los componentes del framework para ajustarlos a su aplicación, así como crear un fichero de configuración central en formato xml que enlazará el modelo, la vista y el controlador.

3. Crear una aplicación de ejemplo

Finalmente se realizará una aplicación que mostrará de forma clara el uso del framework y la interacción de los diversos componentes para demostrar su funcionamiento.

1.3. Estudio de posibles alternativas, metodología y solución adoptada

En este apartado vamos a hablar de la metodología que se ha seguido en el proyecto, cuáles han sido las alternativas y qué solución ha sido adoptada.

1.3.1. Metodología seguida a lo largo del proyecto

Sin contar con la redacción de la presente memoria, durante el proyecto se han podido diferenciar tres etapas de trabajo. Debido al pequeño margen disponible de tiempo y al ser una única persona la encargada de todo el proyecto, se ha utilizado una metodología de desarrollo en cascada. Es decir: investigación, análisis de requisitos, diseño, implementación y test en este orden de manera que se ha intentado tener cada etapa finalizada antes de pasar a la siguiente.

Cabe decir que en la fase de implementación se ha utilizado una construcción iterativa, implementando primero las características más importantes y añadiendo de forma incremental las menos relevantes. También se han ido realizando pruebas unitarias y de integración a medida que se ha ido construyendo el framework.

A continuación se exponen las actividades y metodología empleada en cada etapa:

Etapa de documentación / investigación: estudio del dominio

La fase de investigación ha consistido en, por un parte, documentarse todo lo posible en saber a qué tipo de problemas se enfrenta un desarrollador al diseñar e implementar una capa de presentación J2EE, qué diseños y soluciones se han mostrado satisfactorios y qué patrones de diseño han demostrado ser de utilidad.

Sin duda las fuentes de información más importantes empleadas para documentarse sobre los problemas a resolver, consejos de refactorización y patrones de diseño recomendados han sido (Alur, 2003) y (Fowler, 2003). Todo el conocimiento adquirido ha servido para una mejor comprensión del estudio sobre los frameworks y se han empleado para el diseño del framework jNeko y la aplicación de ejemplo.

Una vez entendida la problemática -y armado con un buen conjunto de buenas prácticas y patrones de diseño- se ha realizado un estudio completo de diversos frameworks disponibles en la actualidad. Para cada framework se ha estudiado: su arquitectura, el ciclo de vida de una petición y los mecanismos y facilidades que proporcionan. Se han estudiado los componentes software que componen el framework identificando los puntos fijos y los puntos de extensión del framework e

identificando patrones de diseño y soluciones utilizadas. Dado que la solución adoptada se basa en Struts1 se ha hecho un estudio más profundo de este framework.

Etapa de definición de requisitos / diseño del framework

Una vez estudiado el dominio, se han definido los requisitos del framework a implementar a partir de la información recopilada en la fase de investigación. Debido al estudio minucioso realizado, se ha podido establecer y fijar los requisitos más importantes que debe proporcionar el framework a construir y, un punto muy importante, es que se han podido mantener estables. De esta manera se ha podido realizar un diseño más detallado que si se hubiesen modificado durante el transcurso del proyecto, lo que ha provocado mayor velocidad en la fase de implementación.

El diseño del framework se ha realizado partiendo de cero, basándose en la solución adoptada por Struts1. La arquitectura se ha construido aplicando los patrones de diseño identificados en la etapa de investigación, asignando responsabilidades a las diferentes clases, además se han seguido las técnicas aprendidas en Ingeniería del Software con el objetivo de disminuir el acoplamiento, aumentar la cohesión y potenciar la reutilización del código.

Se comenzó con el diseño de las clases que proporcionaban las características y requisitos más importantes y se finalizó diseñando las clases y operaciones de menor peso o relevancia. Dado que los requisitos han sido estables, se han podido definir operaciones en la fase de diseño con bastante detalle así como diagramas de colaboración que han servido para acelerar la implementación.

Cabe decir que algunas características, como la librería de etiquetas que había que proporcionar, no se diseñaron en detalle debido a la falta de conocimientos y experiencia en la capa de presentación. Al existir incertidumbre por mi parte, quedaron plasmados en el diagrama de clases a un nivel conceptual.

Etapa de implementación del framework

Para implementar el framework se ha utilizado el entorno de desarrollo integrado Eclipse. El método de trabajo ha consistido en implementar las clases centrales y requisitos más importantes e ir realizando pruebas de las diferentes características mientras se implementaban. Gracias a la encapsulación que proporciona el diseño orientado a objetos y dado que se diseñó el framework con interfaces y factorías, se pudieron simular diferentes comportamientos mediante *stubs* antes de implementarlos.

A medida que se iba implementando el framework se comenzó a diseñar e implementar la aplicación de prueba. Esta aplicación de prueba es un prototipo de pequeña complejidad pero que ha intentado utilizar el máximo de características proporcionadas por el framework para mostrar su funcionamiento. La funcionalidad proporcionada se puede encontrar en una aplicación real de mayor envergadura, como la seguridad de acceso, registro, internacionalización, etc.

La decisión de implementar la aplicación de prueba del framework mientras se construía el mismo ha permitido por una parte ir probando la funcionalidad del framework y por otra terminar con la incertidumbre sobre características que no habían sido comprendidas del todo en el diseño como por ejemplo los custom tags que debía proporcionar el framework y cómo debían funcionar.

Además durante la implementación aparecieron oportunidades de añadir características que consideré que iban a ser útiles para la aplicación de prueba como por ejemplo permitir extender los objetos que representan mapeos con la posibilidad de definirles propiedades y que finalmente fué utilizado como mecanismo para implementar la seguridad de la aplicación.

1.3.2. Estudio de alternativas

El objetivo del proyecto era claro: diseñar e implementar un framework. Las alternativas que se han presentado en el proyecto ha sido decidir qué frameworks estudiar dentro de la gama existente y qué enfoque o modelo adoptar para el diseño.

Tal como muestra el estudio y comparativa de los frameworks, dentro de los frameworks predominantes que se utilizan en la actualidad se distinguieron dos grandes familias: los frameworks basados en acciones y los frameworks basados en componentes, por lo que se decidió escoger representantes de cada familia para el estudio.

1.3.3. Solución adoptada

La solución adoptada ha sido el de diseñar e implementar un framework basado en acciones. El motivo de escoger diseñar un framework basado en acciones sobre un framework basado en componentes u otro modelo ha sido que disponía de más información sobre los primeros. Al disponer de tiempos muy ajustados para el diseño e implementación y dado que no había ninguna restricción que lo impidiera decidí seleccionar un modelo basado en acciones minimizando de esta manera el riesgo a no cumplir los hitos.

Para el diseño del framework, que se ha denominado jNeko[1], se han empleado distintos patrones de diseño recomendados para la capa de presentación J2EE: Front Controller, Application Controller, Service To Worker, View Helper,... JNeko ofrece una amplia gama de facilidades y mecanismos para simplificar la capa de presentación, las más relevantes son: control declarativo del flujo de navegación, resultados extensibles, formularios, facilidades para validar e informar de errores de validación, colecciones de etiquetas especializadas, internacionalización, plugins, gestión de excepciones, mapeos extensibles así como la posibilidad insertar código, alterar el flujo de navegación o cancelar una petición antes de ejecutar las acciones.

Ademas de otras características, para facilitar el trabajo a los desarrolladores, jNeko define convenciones para la configuración del framework (de forma que se

1 Neko significa *gato* en japonés.

reduce el esfuerzo en escribir el archivo de configuración), proporciona información detallada en caso de errores en la configuración, incluye un XML Schema para validar la configuración externamente, ofrece información rica y detallada de los errores ya sean de configuración o los que se puedan producir durante la ejecución del framework , por último ofrece la posibilidad de tracear la ejecución del framework. Se puede leer una lista de toda la funcionalidad proporcionada en el apartado de implementación del framework.

1.4. Planificación del proyecto

A continuación se presentan los hitos establecidos, sus fechas, las actividades planificadas en el periodo y los entregables a presentar en cada hito.

1.4.1. Hitos del proyecto

De acorde al calendario establecido existen tres hitos a alcanzar:

H1) PEC2: 11 de noviembre de 2010 - Análisis de requisitos

Actividades:

- Estudiar la capa de presentación J2EE
- Estudiar patrones de diseño de la capa de presentación J2EE
- Estudio de frameworks alternativos
- Evaluación de los frameworks Struts, Struts2 y JSF
- Identificación de funcionalidad a implementar
- Especificación de requisitos del framework a construir

Entregables:

- Especificación de requisitos

H2) PEC3: 20 de diciembre de 2010 - Diseño e implementación del framework

Actividades:

- Diseño del framework
- Implementación iterativa incremental
- Documentar el diseño y la implementación

Entregables:

- Diseño e implementación del framework

H3) Entrega final: 17 de enero de 2011 - Entrega final del proyecto.

Actividades:

- Construir la aplicación de ejemplo
- Redacción de la memoria
- Redacción de la presentación

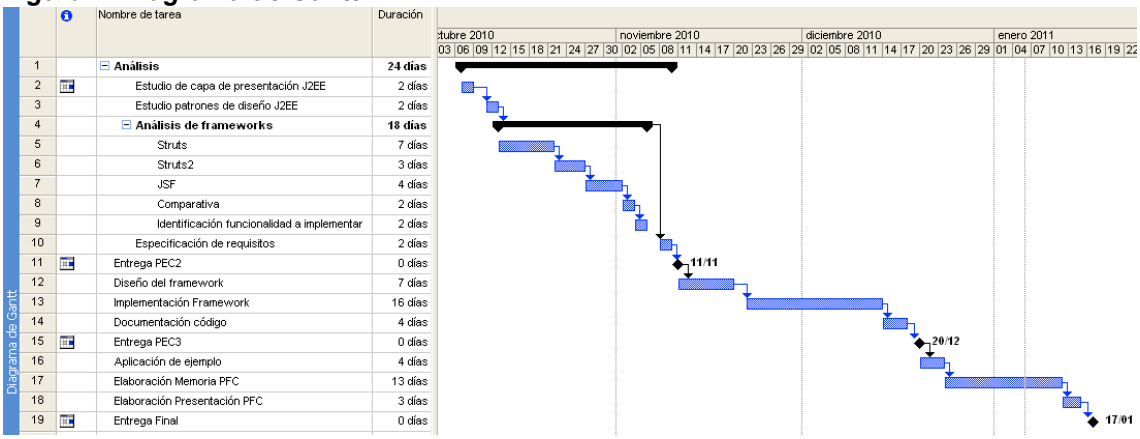
Entregables:

- Código del framework y su distribución
- Aplicación de prueba
- Memoria del proyecto
- Presentación del proyecto

1.4.2. Diagrama de Gantt

El siguiente diagrama muestra la planificación inicial del proyecto.

Figura 1. Diagrama de Gantt



1.5. Productos obtenidos

Los productos obtenidos como resultado de la ejecución del proyecto son:

- **Memoria del proyecto**: *jciriano_memoria.pdf*
- **Presentación / resumen del proyecto**: *jciriano_presentación.ppt*
- **Producto construido**: *jciriano_producto.zip que contiene:*
  - **Framework jNeko**: código fuente, documentación completa javadoc y librería compilada jneko.jar
  - **Aplicación de prueba**: código fuente y el ejecutable apptest.war

1.5.1. Organización del contenido del producto

La organización de los directorios dentro del producto y su contenido es el siguiente:

- **Directorio jneko**: framework jNeko implementado
  - **source**: contiene el código fuente del framework implementado.
  - **javadoc**: contiene la documentación javadoc correspondiente al framework. Se ha generado la documentación para todos los niveles de acceso (público, protegido, privado y privado a nivel de paquete) para que se pueda evaluar mejor, si bien son detalles de la implementación, que queda encapsulada y oculta a un usuario del framework. Por este motivo la documentación tiene un peso considerable (3MB). Si se generase la documentación realmente expuesta al usuario (publica y protected) el espacio ocupado sería mucho menor.
  - **dist**: contiene el archivo compilado del framework *jneko.jar* y una copia del XML Schema utilizado internamente para validar el archivo de configuración

- **Directorio apptest**: aplicación de demostración del framework
    - **source**: contiene el código fuente de la aplicación de prueba. Dentro de este directorio está por una parte el subdirectorio src con las clases java que implementan la demostración y el directorio WebContent que contiene las vistas , descriptores de despliegue y archivos de configuración. Las librerías utilizadas por la aplicación deberían ir en WEB-INF/lib pero se han quitado expresamente para que no ocupen espacio. Dentro del directorio src se ha organizado las clases de la capa de presentación en el subdirectorio com.test.presentation, y las de la capa la lógica de negocio sencilla en el subdirectorio com.test.business
    - **dist**: contiene la aplicación de prueba *apptest.war*

2. Comparativa entre Frameworks J2EE

Actualmente existe una gran cantidad de frameworks disponibles para facilitar el desarrollo de aplicaciones web. Algunos de estos frameworks son productos comerciales desarrollados por empresas, otros son gratuitos y desarrollados por comunidades Open Source y seguramente existan frameworks construidos a medida para uso interno en aplicaciones empresariales.

Si consultamos la enciclopedia libre Wikipedia podemos encontrar un artículo[2] donde se muestra que, actualmente, existen alrededor de cuarenta frameworks de código abierto.

Si bien el abanico de opciones es muy amplio, en un intento de clasificación se ha observado que desde el punto de vista de la arquitectura existen dos grandes familias de frameworks : los frameworks basados en acciones y los frameworks basados en componentes.

Frameworks basados en acciones

Los frameworks basados en acciones aparecieron con la idea de separar de forma clara las peticiones de un cliente en una parte de lógica de proceso y otra parte de presentación. La implementación que se suele usar es el patrón de diseño Modelo-Vista-Controlador denominado Model 2.

En este patrón se utiliza un controlador que centraliza las peticiones y se establecen correspondencias entre las URL de la peticiones con unidades de trabajo llamadas acciones. El trabajo que realiza una acción es realizar funcionalidad específica para una URL determinada a partir de los datos de la petición, de la sesión o de formularios; llamar a los componentes de lógica de negocio y finalmente proporcionar parte de datos del modelo para la generación de la respuesta, normalmente a través de un POJO. Finalmente la acción retorna un resultado que se mapea mediante ficheros de configuración a una JSP que se renderiza como vista.

Algunos frameworks basados en acciones son Struts1, Struts2, Spring MVC y Stripes.

Frameworks basados en componentes

Los frameworks basados en componentes se asemejan más a aplicaciones de escritorio. Establecen una estrecha relación entre los componentes de las vistas mediante clases Java que representan esos componentes, además realizan gestión de eventos y están más orientados a objetos que los frameworks basados en acciones. Un componente puede ser un campo de texto, un formulario HTML o controles a medida ya sea proporcionados por el framework o de creación propia. Los eventos como envíos de formulario, modificación de valores en un campo de formulario o clicks sobre enlaces se mapean a los métodos de la clase que representa al componente o a

2 http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks

clases especiales que escuchan diversos eventos. Uno de los mayores beneficios de este tipo de frameworks es que abstraen más al desarrollador del contenedor J2EE y facilita la reutilización de componentes a través de múltiples aplicaciones.

Algunos frameworks basados en componentes son Wicket, JSF y Tapestry.

2.1. Elección de frameworks a analizar

Dada la cantidad de frameworks existentes, se ha decidido estudiar un representante de cada familia: Struts en el caso de framework basado en acciones y JSF en el caso de framework orientado a componentes.

Cabe decir que en el caso de Struts se va a realizar el estudio tanto de su versión 1 Struts y su versión 2 Struts2. El motivo de seleccionar Struts1 es que es el primer representante de framework basado en acciones que apareció y fue diseñado siguiendo las recomendaciones y patrones de diseño de las Java Blueprints. Además , puesto que se va a realizar el diseño del framework nuevo basado en Struts, se va a profundizar más en su estudio.

Por otra parte Struts2, la última versión de Struts, nace de un framework diferente llamado WebWork y aporta muchísimas características novedosas respecto al primero, más de acorde a la funcionalidad requerida en las aplicaciones actuales. Como veremos, tiene un diseño interno muy configurable y desacoplado que merece la pena estudiar.

Por último como representante de framework por componentes he escogido JSF. Este framework fue definido por un JCP (Java Community Process) y es el estándar de las aplicaciones J2EE.

2.1.1. Enfoque del análisis

El enfoque que se ha realizado no es explicar cómo se utilizan estos frameworks - para ello ya existen diversos tutoriales de gran calidad - sino el obtener una buena visión de sus características y arquitectura.

Para cada framework se realiza una breve introducción, seguidamente se exponen las características más notables que proporciona, a continuación se detalla la arquitectura, componentes y relaciones más importantes y, para finalizar, se expone el ciclo de vida que siguen las peticiones.

Una vez expuesto cada framework, se realiza una evaluación conjunta de los frameworks estudiados explicando la funcionalidad común que proporcionan y cuáles son los mecanismos o solución adoptada para proveerla en cada caso.

2.2 Jakarta Struts 1

Struts fué diseñado con la intención de crear un framework de código abierto para la creación de aplicaciones web, separando de forma sencilla la capa de presentación de la capa de negocio e integración. Fué originalmente creado por Craig McCalahan en Mayo del 2000 y a partir de entonces fué adoptado y mantenido por la comunidad Apache. El proyecto apareció en pleno auge de las puntocom por lo que tuvo gran acogida y se convirtió en el estándar de facto para la capa de presentación de J2EE durante varios años.

La razón principal de su popularidad es la sencillez de su arquitectura y lo bien que se integra dentro de la plataforma J2EE. Cabe decir que este framework se puede ver como la implementación de referencia de la capa de presentación J2EE ya que se implementó siguiendo las recomendaciones, patrones de diseño y mejores prácticas de las *Java Blueprints* de Sun Microsystems.

2.2.1. Características principales

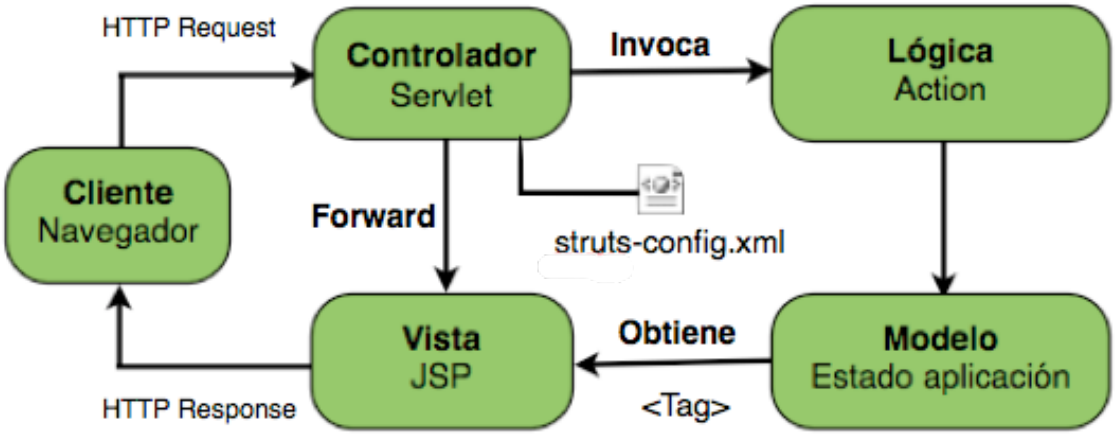
- Es un framework orientado a acciones que implementa el Modelo 2 de MVC
- Control declarativo. Permite realizar correspondencias entre peticiones, objetos que validan los datos de las peticiones, objetos que invocan al modelo y vistas que calculan la presentación. La configuración se realiza mediante un archivo XML que el controlador carga en memoria al iniciar la aplicación y utiliza para tratar cada petición.
- Tiene soporte para la internacionalización de las aplicaciones
- Proporciona una amplia colección de JSP Tags
- Proporciona validación de datos a través de métodos, así como la posibilidad de realizarla de forma declarativa
- Gestión de excepciones. Proporciona un mecanismo de control de excepciones que se especifica de forma declarativa.
- Extensibilidad. Permite insertar plugins para extender el funcionamiento o utilizar plugins de terceros. También permite extender los objetos que representan mapeos de acciones (ActionMapping) con nuevas propiedades y utilizarlas en el archivo de configuración XML.
- Soporta diferentes modelos (JavaBeans, EJB, ...)
- Permite gestionar DataSources

2.2.2. Arquitectura

Struts está formado por un conjunto de clases, servlets y etiquetas JSP que cooperan entre sí para proporcionar un diseño reutilizable del patrón de diseño MVC Model 2. Además contiene una amplia librería de etiquetas y clases de utilidad que

pueden utilizarse independientemente del framework. La figura 2 muestra una visión general de Struts.

Figura 2. Visión general de Struts

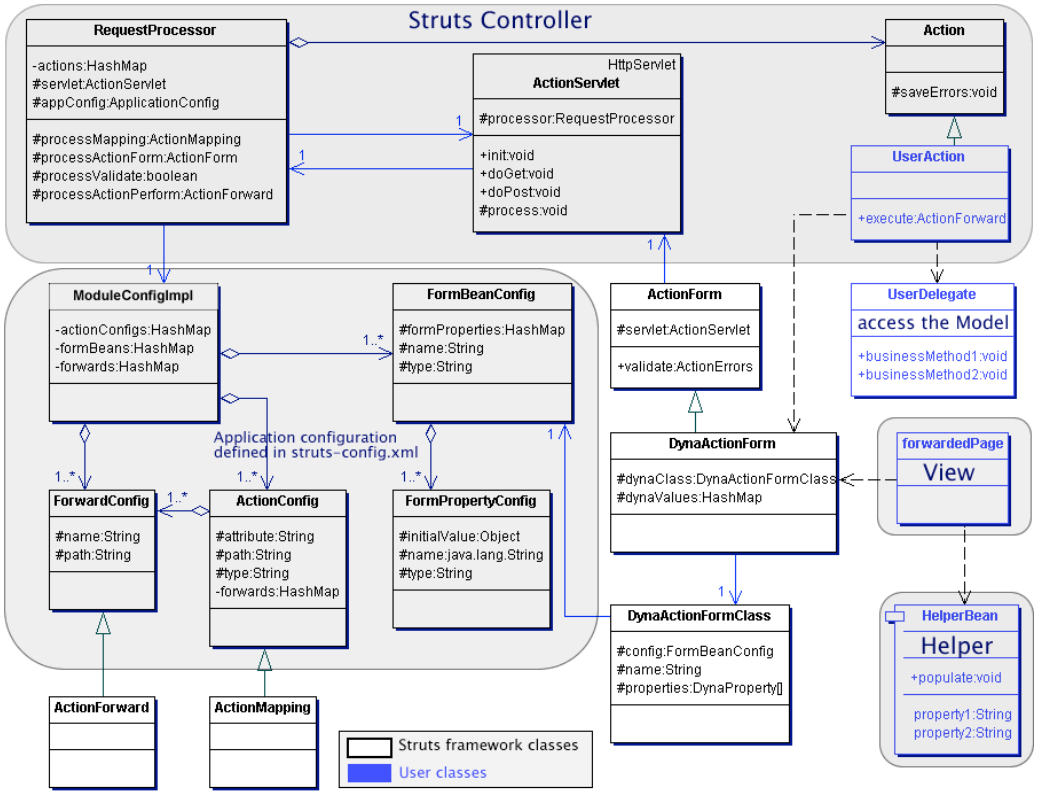


En esta arquitectura se puede observar los siguientes elementos:

- **Navegador:** es un cliente que genera una petición HTTP *HttpRequest*. El contenedor web responderá con una respuesta *HttpResponse*.
- **Controlador:** recibe la petición del navegador y toma la decisión de hacia dónde hay que enviarla para procesarla. En Struts, el controlador sigue un patrón de diseño *Front Controller* implementado como un servlet que trabaja junto a una clase que sigue el patrón *Application Controller*. La configuración del controlador se realiza mediante un fichero de configuración xml *struts-config.xml*
- **Lógica de negocio:** la lógica de negocio modifica el estado del modelo y ayuda a controlar el flujo de la aplicación. En Struts se lleva a cabo mediante una clase *Action* donde se llama al código de la lógica de negocio, normalmente a través de un patrón *Business Delegate*. El controlador puede pasar al *Action* si éste lo necesita un objeto *ActionForm* que representa los datos de un formulario HTML.
- **Modelo:** representa el estado de la aplicación. Los objetos de negocio actualizan el estado de la aplicación.
- **Vista:** suele ser una página JSP que no contiene lógica de navegación ni de negocio ni información del modelo. Para acceder a los datos del modelo que el objeto Action ha puesto a disposición de la vista se utilizan las etiquetas que provee Struts. Estas etiquetas siguen el patrón ***View Helper*** que permiten una separación clara entre presentación y la lógica de acceso y procesado de la información.

En la figura 3 podemos ver el diagrama de clases UML de struts. Donde se pueden observar las relaciones entre las diferentes clases que lo componen.

Figura 3. Diagrama de clases de Struts



En este diagrama podemos ver qué clases proporciona Struts y qué clases tiene que proporcionar el desarrollador (en color azul) para construir una aplicación. Un desarrollador que utilice Struts, debe proporcionar nuevas clases que extiendan de Action y ActionForm para solucionar el problema específico de su aplicación así como proporcionar las vistas. El fichero de configuración struts-config.xml define y relaciona las URL de las peticiones con las acciones y formularios a utilizar, así como las vistas que hay que presentar según el resultado de la ejecución de las acciones.

2.3.3. Ciclo de vida de una petición en Struts1

A continuación vamos a ver los componentes principales, sus responsabilidades y cómo colaboran entre sí para servir una petición.

ActionServlet

La clase ActionServlet es un servlet que se encarga de recibir todas las peticiones de clientes al framework en un único punto de acceso. Se configura de forma estándar mediante el archivo web.xml del contenedor web de forma que todas las URLs que terminen en un sufijo determinado (generalmente la extensión *.do) se consideren peticiones al framework y las sirva este servlet.

Este controlador pasa la petición a un objeto RequestProcessor donde se trata realmente la petición. El RequestProcessor le devuelve la información de la próxima vista que debe mostrar mediante un objeto ActionForward. El controlador despacha la vista y finaliza el tratamiento de la petición.

RequestProcessor

Esta clase es el corazón de Struts. Su misión es la de servir las peticiones que realizan los clientes mediante la invocación de los objetos Action que correspondan así como crear , poblar y validar formularios ActionForm cuando se necesite. También se encarga de gestionar los errores que se puedan producir.

Tal como se ha comentado anteriormente el flujo completo de la aplicación y definición de las clases que proporciona el desarrollador se definen en el fichero de configuración struts-config.xml. El RequestProcessor se encarga del control y ejecución de las peticiones según ésta configuración.

ActionForm

ActionForm es una clase abstracta, que el desarrollador puede extender, cuya utilidad es contener información de datos de entrada del cliente para que se traten en el Action. Normalmente corresponde a un formulario HTML. En el fichero de configuración se establece la relación entre formularios y acciones.

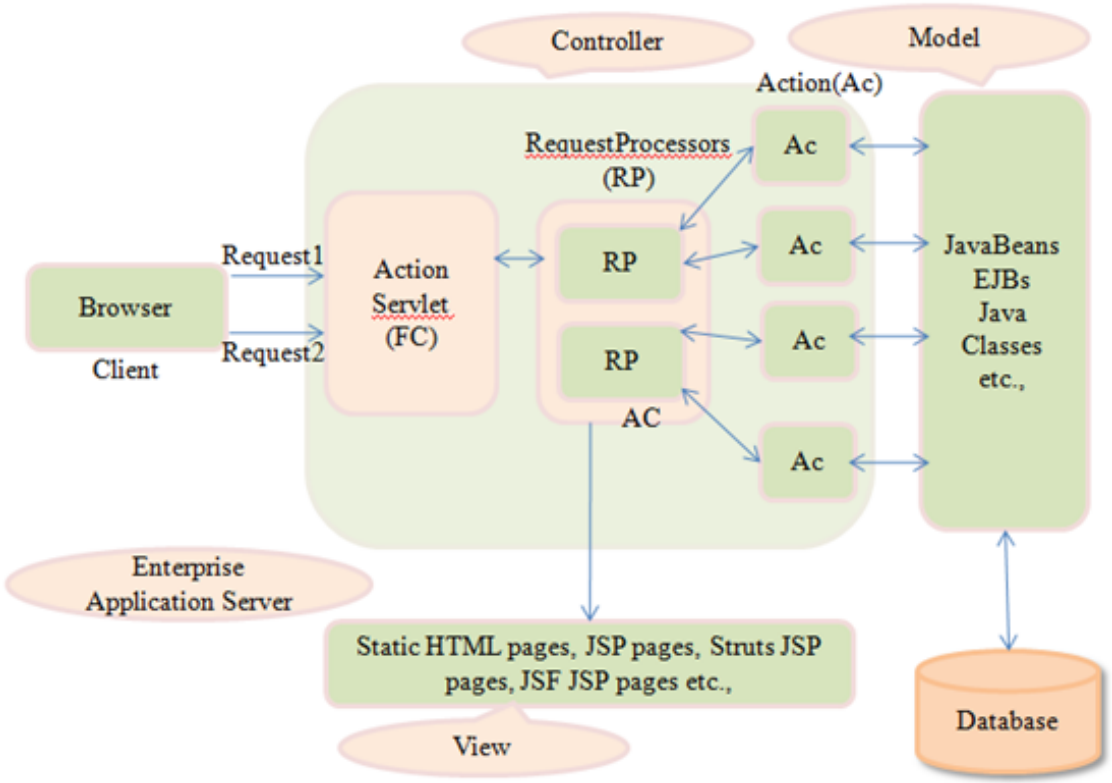
Cuando hay una petición, Struts comprueba si el desarrollador ha definido un formulario, si es así, instancia el formulario correspondiente y lo rellena a partir de los datos del request, lo valida y si no hay errores lo pone a disposición del Action. Si hay errores, el desarrollador puede determinar la siguiente vista a la que hay que ir (normalmente a la que contiene el formulario). Struts proporciona etiquetas para utilizar en las vistas para poder leer y escribir datos al ActionForm así como para poder mostrar errores. Por último cabe decir que se puede configurar que el ActionForm se mantenga durante la sesión.

Action

La clase Action se puede ver como un adaptador (patrón Adapter (Gamma,1995)) entre el contenido de una petición HTTP y la lógica de negocio que debe ejecutarse para esa petición. Extendiendo un Action se consigue que las interfaces de la lógica de negocio sean compatibles con la interfaz de Struts.

En la figura 4 se puede observar cómo una petición se recibe en el Action Servlet, se procesa en el RequestProcessor, que selecciona un Action y lo ejecuta llamando al método execute().

Figura 4. Ciclo de vida de una petición en Struts 1



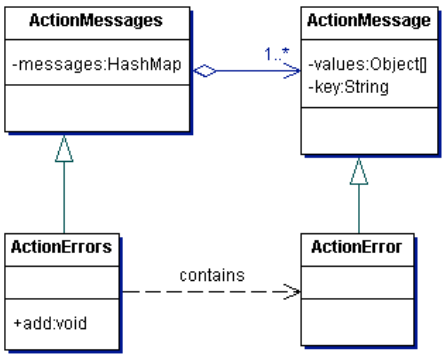
Las acciones que define un desarrollador extienden esta clase Action e implementan el método execute() donde se debe realizar la acción específica. Además cuando Struts llama al método execute() pasa la información del formulario de entrada -siempre que se haya asignado uno a la acción- como parámetro mediante un objeto de la clase ActionForm.

La misión de una acción consiste en interactuar con el modelo. Cabe decir que, la lógica a implementar en un Action debería ser llamar a una interfaz de negocio y no realizar la lógica de negocio directamente, esto se consigue por ejemplo con un BusinessDelegate. También es responsabilidad de la acción retornar mediante un ActionForward la vista lógica a mostrar así como poner a disposición de la próxima vista la información que debe mostrar (normalmente a través de asignar POJOs con la información a la petición HTTP en forma de atributos).

Clases de error

Struts incluye las clases ActionError y ActionErrors (ver figura 5). ActionError encapsula un mensaje de error individual. ActionErrors por su parte es un contenedor de ActionError. Struts proporciona etiquetas para que las vistas puedan tener acceso a los errores además de facilitar la internacionalización de los mismos utilizando una clave que posteriormente será traducida mediante el mecanismo estándar de internacionalización de Java al mensaje de error en el idioma que corresponda.

Figura 5. UML de las clases de error



ActionMapping

Esta clase contiene la relación de los objetos *Action* con los *ActionForm* y los *ActionForward*. La definición de la configuración del control del flujo de la aplicación se lee desde el fichero *struts-config.xml* y se traduce a un conjunto de *ActionMapping* en memoria para que puedan utilizarlas las clases. Además cuando Struts invoca al método execute() de la acción , pasa también el *ActionMapping* relacionado para que la acción pueda realizar consultas del flujo de control.

2.2.4. Patrones de diseño identificados

Los componentes de Struts están diseñados siguiendo los diversos patrones recomendados en la guía *Java BluePrints* de mejores prácticas para el desarrollo J2EE. En particular, los patrones de diseño identificados en Struts son : Front Controller, Application Controller, Service to Worker, View Helper y Composite View.

Front Controller

ActionServlet es un servlet que implementa el patrón Front Controller y cuya misión es el de atender en un único punto central todas las peticiones que solicitan los clientes de la capa de presentación para tratarlas con el resto de componentes del framework.

Action Controller

El ActionServlet delega las peticiones que recibe a un RequestProcessor que se ocupa a partir de un mapeo de seleccionar los comandos a ejecutar y las vistas a servir siguiendo el patrón de diseño Action Controller consiguiendo modularidad y permitiendo extender la aplicación de forma sencilla.

Service to Worker

El RequestProcessor mediante la url de la petición y según la configuración del archivo struts-config.xml determina la acción que hay que invocar.

Estas acciones son un punto de extensión del framework y las debe proporcionar el desarrollador implementando clases que extiendan la clase

Action e implementando el *template method* execute(). Las acciones permiten poner la información del modelo accesible a las vistas.

El conjunto de ActionServlet, RequestProcessor junto con la clase Action, la lógica para servir vistas y las etiquetas proporcionadas implementan la estrategia Command and Controller Strategy del patrón de diseño Service to Worker. La clase Action utiliza el patrón Command (Gamma,1995) de forma que se define una interfaz genérica común para las diferentes acciones a realizar.

View Helper

Tanto las etiquetas que proporciona Struts para utilizar en las vistas, como los ActionForm son una aplicación del patrón de diseño View Helper que evita que exista lógica en la vista con los problemas que eso puede acarrear.

Composite View

Otro patrón de diseño que implementa Struts es Composite View. Con el fin de definir vistas complejas y dado que el acceso a las vistas directamente se considera una mala práctica en este contexto, Struts proporciona una alternativa a la inclusión directa de JSP's para componer las vistas. Esta alternativa es el framework Tiles que proporciona un mecanismo de plantillas que permite realizar la composición de elementos estáticos y dinámicos, proporcionando un look & feel uniforme entre las vistas y separando el layout de su composición.

2.3. Apache Struts2

Struts2 nació de la necesidad de evolucionar el código de Struts con el objetivo de simplificar todavía mas el desarrollo de la capa de presentación de las aplicaciones web. Durante el diseño de la siguiente versión de Struts, que iba a denominarse Struts Ti, se observaron objetivos y puntos en común con el framework WebWork2, por lo que las comunidades decidieron unir esfuerzos y fusionar estos dos proyectos dando lugar al proyecto Struts2.

Este framework ofrece una mayor productividad en la construcción, desarrollo y mantenimiento de aplicaciones. Permite utilizar menos configuración XML mediante anotaciones, define comportamientos por defecto y se rige por convenios inteligentes. Además posee una arquitectura modular con un grado muy bajo de acoplamiento que lo hace muy extensible.

2.3.1. Características Principales

Las características principales de Struts2 y mejoras respecto a la versión anterior son:

- Framework MVC orientado a acciones.
- Arquitectura flexible de bajo acoplamiento mediante plugins e interceptores que permiten personalizar el tratamiento de las peticiones hasta llegar a cada acción de forma individual o para un conjunto de acciones.
- Proporciona un framework de validación flexible que permite desacoplar las reglas de validación del código de las acciones.
- Aproximación jerárquica a la internacionalización que simplifica la traducción de aplicaciones.
- Permite utilizar cualquier clase Java normal (POJO) como acción.
- Conversión de tipos automática que mapea de forma transparente los valores HTTP a las acciones mediante setters, solucionando y simplificando uno de los mayores esfuerzos que se necesitan al crear aplicaciones web.
- Define un lenguaje OGNL compatible con JSTL que expone las propiedades de múltiples objetos como si fueran un único JavaBean.
- Motor integrado de inyección de dependencias que permite inyectar componentes en otros. El motor por defecto utiliza Spring.
- Ficheros de configuración modulares que permiten descomponer la configuración en varios archivos para mejorar la gestión de los mismos en proyectos grandes.
- Proporciona anotaciones Java que reducen la configuración necesaria.
- Se integra fácilmente con otros productos como Hibernate, Spring, SiteMesh, JSTL...

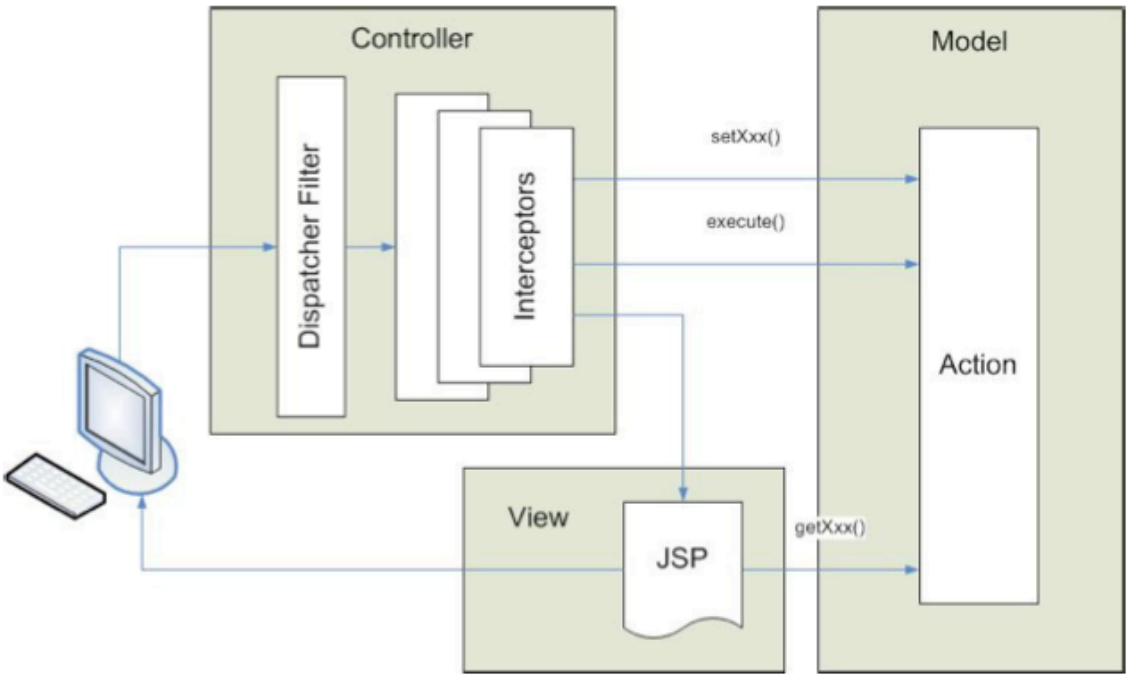
- Proporciona etiquetas reutilizables para las vistas que utilizan temas y plantillas para personalizarse.
- Gran cantidad de etiquetas, desde simples campos de texto hasta calendarios para seleccionar fechas y vistas en árbol.
- Permite plugins para definir nuevos tipos de resultado de forma que se soportan múltiples tecnologías de presentación como JSP, FreeMarker, Velocity, PDF, JasperReports, etc.
- Proporciona una colección de plugins opcionales que proporcionan funcionalidad como por ejemplo el upload de ficheros, prevenir el problema de repetición de posts del mismo formulario o gestionar la seguridad.
- Gratuito y código abierto mediante la licencia Apache License 2.0

2.3.2. Arquitectura

Struts2 es un framework MVC basado en el modelo MVC model 2 pero, a diferencia de éste último, las acciones pueden toman el rol del modelo. Esto significa que las vistas tienen la capacidad de obtener datos directamente de la acción sin necesidad de tener un objeto intermedio con datos del modelo disponible para traspasar a la vista. Por este motivo se le ha llamado a este modelo *pull-MVC*.

Struts2 está compuesto de cinco componentes principales: acciones, interceptores, pila de valores / OGNL, tipos Result y las vistas.

Figura 6. Modelo pull-MVC de Struts2



En la figura 6 se puede observar la relación de el modelo, la vista y el controlador con los componentes principales de Struts2.

Veamos la composición de las diferentes partes de la arquitectura:

- **Controlador:** el controlador está implementado con un filtro DispatcherFilter (siguiendo el patrón de diseño Front Controller mediante la estrategia Filter Controller Strategy) además de interceptores que se usan durante el ciclo de vida de una petición y que se pueden configurar a nivel de acción. Estos interceptores se pueden configurar independientemente para cada acción y proporcionan a una petición diferentes capas de funcionalidad como por ejemplo control de flujo, validación, seguridad, etc. Estos interceptores siguen el patrón de diseño Intercepting Filter estableciendo una cadena de interceptores que se aplican en una petición HTTP hasta llegar a la acción.
- **Modelo:** el modelo está implementado con acciones que pueden trabajar con cualquier tecnología como JDBC, EJB, Hibernate, etc.
- **Vista:** es una combinación de objetos Result. En el diagrama se muestran JSPs pero se puede integrar con cualquier tecnología de presentación como Velocity, XSLT, JSF, JSTL, Tiles, Sitemesh, etc.

El diagrama de la figura 7 muestra distintos componentes de la arquitectura de Struts2. Los componentes principales son:

FilterDispatcher

Es un Front Controller implementado con un filtro estándar J2EE. Su misión es atender de forma centralizada todas las peticiones a la aplicación. Equivale a la funcionalidad de la clase ActionServlet en Struts 1.0

Action

Action es una interfaz que el desarrollador implementa para llamar a la lógica de negocio. El FilterDispatcher se encarga de pasar la petición al Action para que ésta la procese. En Struts2 se puede considerar la acción como una combinación de los objetos ActionForm y Action de Struts1 así como un ValueBean del modelo.

Interceptores (Interceptors)

La inclusión de interceptores es una de las características más potentes e importantes de Struts 2. Los interceptores son como los filtros estándar del contenedor de servlets solo que se configuran a nivel de acción. Permiten ejecutar código antes y después de la ejecución de una acción por lo que se suelen utilizar para agrupar funcionalidad común a diferentes acciones.

Result

Este mecanismo permite que las acciones puedan retornar distintos tipos de resultados. Struts2 incluye una gran colección de resultados predefinidos que proporcionan por ejemplo el renderizado de una vista con JSP, FreeMarker o Velocity , encadenamiento de acciones, retornar streams, generar XML, etc. Los tipos de resultado permiten la integración de diferentes herramientas de presentación. Es extensible y existen diferentes plugins que proporcionan diferentes tipos de resultado.

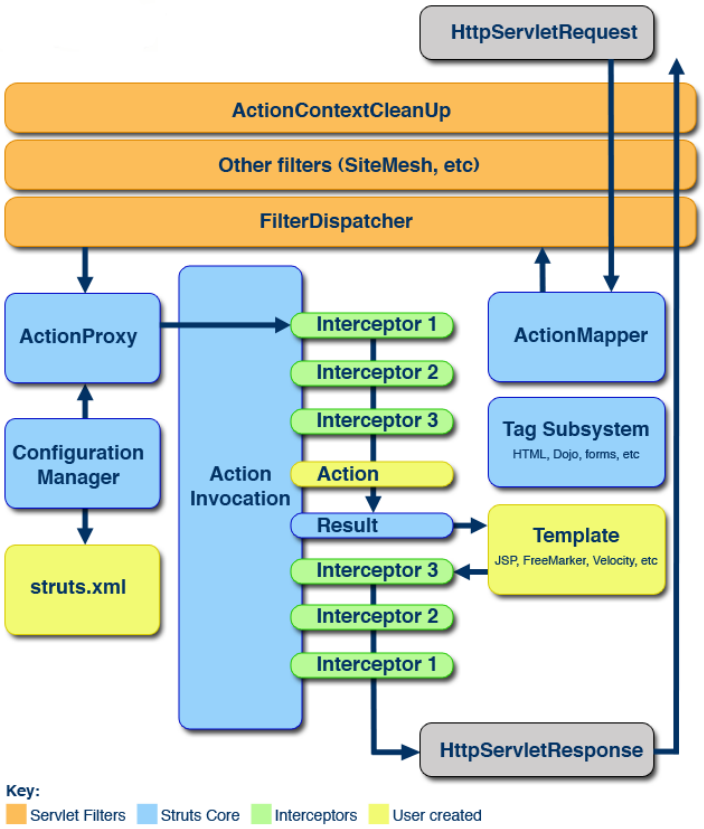
Pila de valores (Value Stack)

La pila de valores es un conjunto de objetos que se utiliza para transportar los datos desde la acción hasta la vista. Las vistas siempre tienen acceso a la pila. Por defecto la instancia de Action que se ejecuta, se coloca en lo alto de la pila de forma que la vista tiene acceso directamente a los datos de la acción, que puede contener datos del modelo, así como el acceso a los diferentes objetos del contenedor web (HttpSession, HttpServletRequest, ServletContext, etc.)

Etiquetas (Tags)

Las vistas acceden al modelo mediante etiquetas, además Struts2 ha implementado un MVC a nivel de etiquetas de forma que separa el renderizado de las etiquetas de su modelo, permitiendo aplicar diferentes temas a la hora de renderizarlas, así como crear nuevos temas. Struts2 incluye de serie temas para renderizar componentes en html, xhtml y css.

Figura 7. Componentes y ciclo de vida de petición en Struts 2



2.3.3. Ciclo de vida de una petición en Struts2

En la figura 7, además de los componentes principales, se puede observar el flujo o recorrido que se realiza para procesar una petición: desde que se recibe hasta que se retorna una respuesta.

Veamos a continuación los componentes por los que se va tratando la petición así como el tratamiento que se realiza según el orden de ejecución:

El flujo comienza en el contenedor de servlets donde se recibe una petición HttpServletRequest y se pasa a través de la cadena de filtros standard del contenedor web para tratarla. A continuación pasa por los siguientes componentes:

- **Filtro ActionContextCleanup**: éste es un filtro opcional que es útil para algunas ocasiones como la integración con otras tecnologías como puede ser SiteMesh
- **Otros filtros**: diversos filtros estándar del contenedor web que se pueden añadir y configurar para integrar diferentes tecnologías
- **Filtro FilterDispatcher**: este filtro hace de *Front Controller* en Struts. Lo primero que hace este filtro es utilizar el *ActionMapper* para determinar si hay que invocar una acción o no. En caso de que haya que invocar una, el *FilterDispatcher* delega el control al *ActionProxy*

- **ActionProxy:** El ActionProxy se ayuda de un ConfigurationManager (inicializado a partir del fichero de configuración struts.xml) para crear un ActionInvocation que implementa el patrón de diseño Chain of Responsability (Gamma,1995) y Command (Gamma,1995). El ActionInvocation se crea con una cadena de interceptores que se ejecutan envolviendo el método de ejecución de la acción. Esta cadena de interceptores se configura en el fichero struts.xml. El desarrollador puede crear nuevos interceptores y configurar diferentes cadenas de interceptores a ejecutar (denominadas *stacks*) tanto a nivel global, como a grupos o paquetes de acciones así como asignarlas a acciones individualmente.
- **ActionInvocation**: El ActionInvocation ejecuta los interceptores (si se ha configurado alguno) y después invoca a la acción. La acción se ejecuta y responde con una cadena indicando un nombre lógico de resultado. A continuación el ActionInvocation busca un resultado Result correspondiente a ese nombre lógico y lo ejecuta, provocando el renderizado de la JSP, plantillas o cualquier tipo de resultado que se haya integrado en el HttpServletResponse.

Para finalizar la cadena de interceptores sigue con su ejecución en orden inverso y la respuesta HTTPServletResponse pasa de nuevo por los filtros estándar finalizando así el tratamiento de la petición.

Características de las acciones en Struts2 respecto a Struts1

En Struts2 las acciones pueden ser cualquier objeto sin necesidad de que implemente ninguna interfaz. Ahora bien, debe definir un método llamado "execute". Este nombre de método se utiliza por convención y se puede modificar en el archivo de configuración. El método retorna un String indicando un resultado lógico. En el fichero de configuración se mapea dentro de una acción los resultados lógicos con los Result reales que se van a renderizar al usuario.

Las acciones no reciben ni formularios ActionForm, ni objetos HttpServlet Request , ni HttpServletResponse. El mecanismo empleado para obtener los datos del request, es definir métodos setters setXxx(). Struts2 mediante reflection llama a los setters con los parámetros del request realizando una conversión de tipos automática.

Para devolver la información que debe mostrar la vista, tampoco es necesario definir HelperBeans y asignarlos al request o a la sesión. En su lugar se pueden definir getters (métodos getXxx()) que retornen la información necesaria para la vista. Struts2 tras ejecutar la acción, expone estas propiedades como si fuera un único Javabean y las pone a disposición de la vista a través de un mecanismo Value Stack. Esta característica es muy potente ya que las acciones quedan aisladas completamente del contenedor web de forma que se simplifican los tests y facilita utilizar objetos mock para probarlas.

De todas formas, en caso de necesitar acceso a los objetos del contenedor web, Struts2 proporciona el acceso mediante inyección de dependencias. Para lograrlo

define diferentes interfaces (como por ejemplo ServletRequestAware o ServletResponseAware) que la acción puede implementar para obtener acceso a los diferentes objetos del contenedor web.

Validación de datos

Struts 2 permite realizar validaciones a través de código en las acciones, así como declarativamente mediante el framework de validación XWork, el cual proporciona reglas de validación que se pueden encadenar. También proporciona anotaciones para validar los diferentes setters de las acciones.

2.4. JavaServerFaces

JavaServerFaces (JSF) es la especificación de un framework MVC basado en un modelo de componentes de interfaz de usuario para el desarrollo de la capa de presentación J2EE. Está declarado como el estándar de presentación dentro de J2EE y fué desarrollado a través de una JCP (Java Comunity Process) compuesta por diferentes expertos en desarrollo web de diferentes organizaciones como Jakarta Struts, Oracle, Sun, IBM, ATG, etc.

Actualmente las implementaciones más importantes de JSF son:

- **JSF Reference Implementation** de Sun Microsystems
- **MyFaces**, proyecto de Apache Software Foundation
- **RichFaces**
- **ICEFaces**. Implementación que contiene diversos componentes para desarrollar interfaces de usuarios más enriquecidas, como editores de texto enriquecidos y reproductores de multimedia entre otros.
- **jQuery4jsf**. Implementación que contiene diversos componentes desarrollados con el framework de javascript jQuery

La versión actual de JSF es la 2.0 que apareció en junio de 2009 como estándar a utilizar con la versión 6 de Java EE.

2.4.1. Características principales

- Framework MVC basado en modelo de componentes de interfaz de usuario
- JSF incluye controles GUI, APIs y custom tags con los que se pueden crear interfaces personalizables y complejas.
- El framework proporciona eventos. El código puede gestionar y responder a diversos eventos como si fuera una aplicación de escritorio, por ejemplo se puede responder al cambio de un valor en un campo de texto, un submit de formulario, ciertas selecciones del usuario, etc.
- Gestión de beans que permiten simplificar el el tratamiento y almacenamiento de los parámetros de las peticiones.
- Validación de formularios y conversión de tipos automáticas. En caso de errores de validación o de conversión de tipos, el formulario se puede volver a visualizar automáticamente mostrando los errores y con los valores previos de los campos insertados.
- Soporte AJAX integrado: se proporcionan etiquetas que implementan funcionalidad mediante AJAX sin tener que manejar JavaScript y que permiten llamar a la lógica de negocio.
- En la versión 1.x se utilizan JSP como vistas por defecto (si bien se puede acomodar a otras tecnologías). En la versión 2.0 se utiliza un sistema de vistas mediante plantillas llamado Facelets, más potentes y flexibles que las JSPs, que

se configuran mediante archivos XML y un lenguaje de descripción de vistas (VDL)

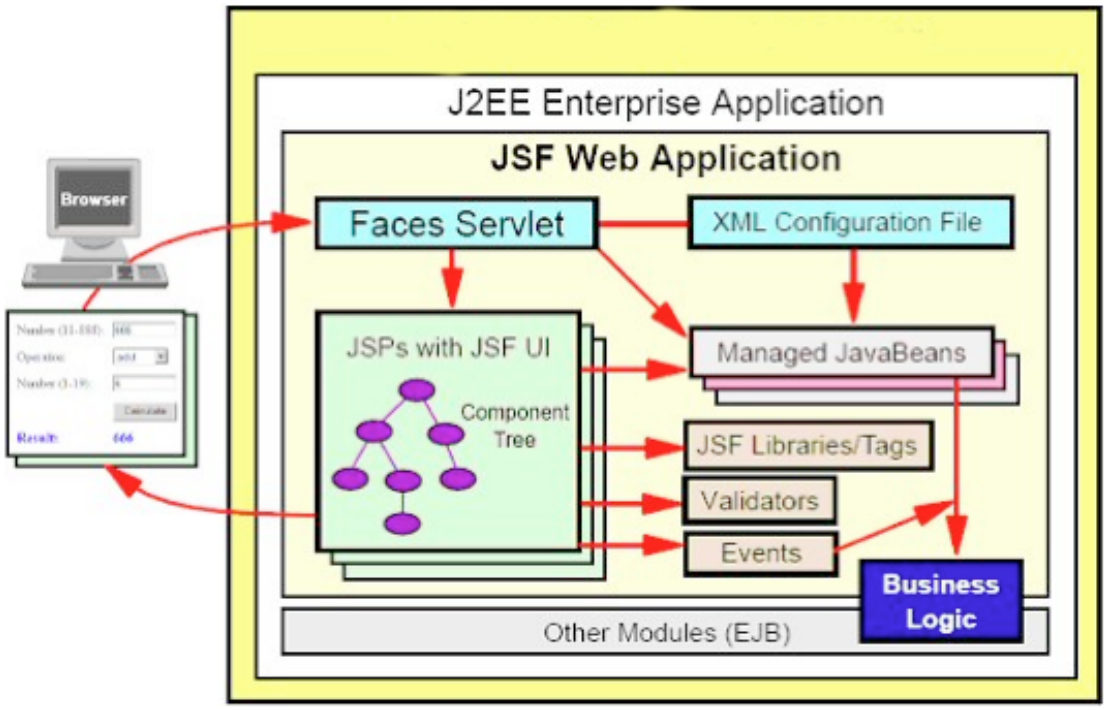
- Configuración centralizada y declarativa mediante archivos
- Al ser un estándar se dispone de una gran variedad de librerías de componentes suministradas por diferentes proveedores

2.4.2. Arquitectura

El modelo de una aplicación JSF es una variante de *MVC Model 2* que, en lugar de proveer una capa fina con un controlador y algunos componentes para facilitar la validación de formularios y gestión de errores, proporciona una capa gruesa con un framework de elementos de interfaz de usuario con el que se intenta ocultar al desarrollador de muchas de las complejidades que surgen al diseñar una presentación web. De esta manera, el modelo que se sigue es más parecido al de un framework de desarrollo de aplicaciones escritorio *GUI* como por ejemplo Swing. En JSF se hace un uso intensivo del patrón de diseño *Composite View*, el patrón *Observer* (Gamma,1995) y el patrón de diseño *Pluggable Look & Feel* conocido como *PLAF*.

En la figura 8 se muestra una visión general de la relación de componentes que forman una aplicación realizada con JSF.

Figura 8. Visión general de componentes de una aplicación JSF



Los componentes principales de JSF son:

- **Faces servlet**: Un servlet (FacesServlet) que actúa de Front Controller centralizando las peticiones y controlando el flujo de ejecución.
- **Faces**: las páginas JSP se construyen mediante componentes JSF, donde cada componente corresponde a una clase Java en el servidor. La estructura de los componentes se realiza mediante el patrón Composite (Gamma,1995) por lo que la estructura de una vista tiene forma de árbol. Por este motivo se le denomina árbol de componentes o Component Tree.
- **Fichero de configuración**: es un fichero de configuración XML (faces-config.xml) que contiene las reglas de navegación entre JSPs, validadores y beans gestionados (Managed JavaBeans).
- **Beans gestionados (Managed beans)**: Son JavaBeans que se definen en el fichero de configuración. Sirven para almacenar los datos de los componentes JSF así como para pasar datos del modelo entre la lógica de negocio y los componentes JSF. Es equivalente a un ActionForm de Struts 1.x
- **Librerías de etiquetas (Tag Libraries)**: JSF proporciona librerías de etiquetas para poder manejar los componentes desde las vistas.
- **Validadores**: Son clases Java que validan el contenido de los componentes JSF, por ejemplo para validar los datos de entrada del usuario.
- **Eventos**: mecanismo que permite que se ejecute código Java ante determinados sucesos como por ejemplo la aceptación de un formulario. En la gestión de eventos se suelen pasar los beans gestionados a la lógica de negocio.

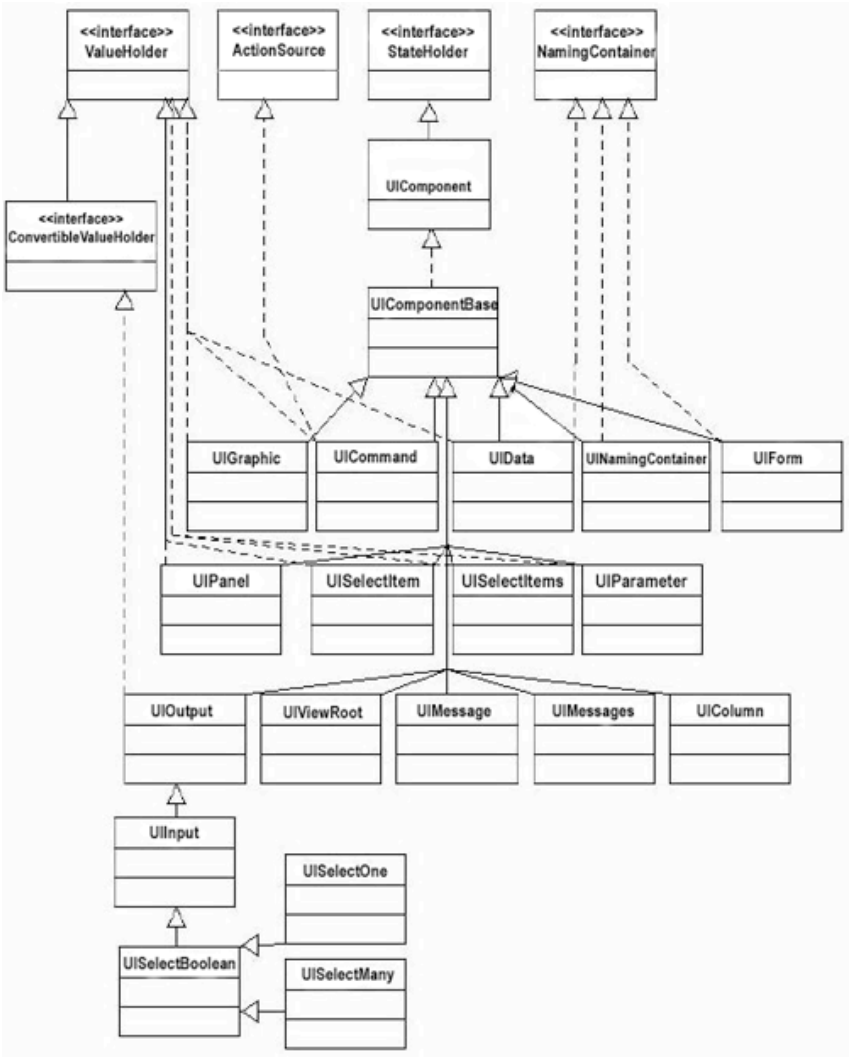
2.4.3. Modelo de componentes

En JSF, se ha realizado una arquitectura mediante componentes y modelos arquitectónicos que cubren diversos aspectos o problemática de la capa de presentación. Está formada por las clases que definen la interfaz de usuario y cuatro modelos que gestionan los eventos, la conversión de datos, la validación de datos y el renderizado de componentes respectivamente y que se exponen a continuación:

Clases de interfaz de usuario

Son clases Java que representan componentes de la interfaz de usuario y representan tanto a elementos simples (como por ejemplo un campo de texto de formulario, un radiobutton o un combo de selección) como a elementos compuestos siguiendo el patrón de diseño Composite (Gamma,1995). (Ver figura 9)

Figura 9. Diagrama de clases de componentes UI



Estas clases son extensibles de forma que los desarrolladores pueden crear sus propios componentes y reutilizarlos en otras aplicaciones. Todas las clases de componentes de Interfaces de Usuario extienden a UIComponentBase.

Modelo de gestión de eventos

El modelo de gestión de eventos está basado en el patrón de diseño *Observer* (Gamma,1995): los componentes emiten eventos a *listeners* internos de JSF para notificar cambios de estado. Estos eventos se encolan y en ciertos momentos del ciclo de vida de la petición se distribuyen a listeners implementados por el desarrollador para tratarlos.

Modelo de conversión

JSF permite asociar un componente con un objeto JavaBean del lado del servidor (Managed Beans) de forma que se tienen dos vistas de los datos del componente: una que pertenece a la vista, que es con la que interacciona el usuario, y otra interna que

se alacena en el JavaBean para utilizar con la lógica de negocio, similar al papel que adoptan los formularios de Struts.

Durante la fase de conversión de datos, dentro de el ciclo de vida de la petición, JSF crea los JavaBeans asociados y pasa los valores de la vista al JavaBean convirtiendo automáticamente los tipos. EL desarrollador puede definir conversores (Converters) y asociarlos a componentes para el caso de conversiones que no vienen de serie en JSF.

Modelo de validación

JSF proporciona un mecanismo para asociar clases java (validadores) a los componentes de la interfaz, de forma que cada componente puede tener de cero a varios validadores asignados. Además los validadores pueden definirse sobre un conjunto de componentes. Durante la fase de validación, dentro del ciclo de vida de una petición, se ejecutan los validadores y, en caso de fallo de las validaciones, se reconstruye la interfaz de nuevo mostrando los errores que provocan el fallo.

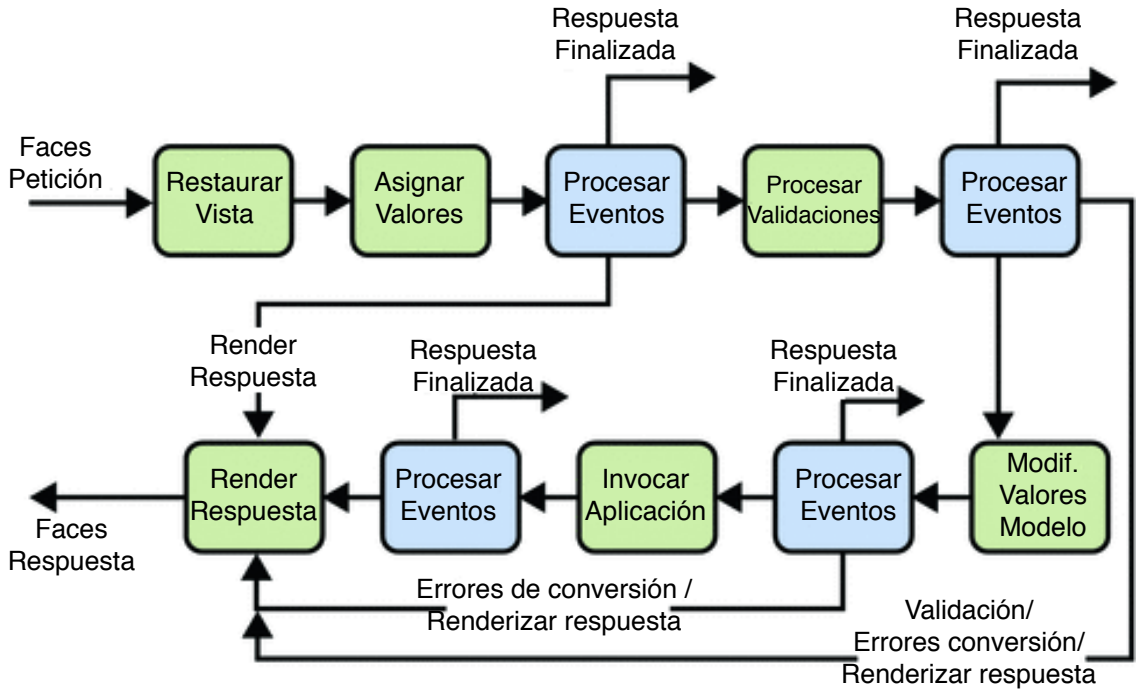
Modelo de renderizado de componentes

JSF separa la lógica y los datos de los componentes de su renderizado. Existe un proceso para decodificar los valores de las peticiones en componentes e inversamente codificar la visualización de los componentes en las respuestas, de forma similar a como se realiza en Swing. Algunos componentes se renderizan asimismo (renderizado directo) mientras que otros utilizan una clase externa o renderer para que les renderize su aspecto (renderizado indirecto). El renderizado es configurable de forma que se permite aplicar temas y cambiar el aspecto de cualquier componente así como adaptarlos para utilizarlos con otros clientes.

2.4.4. Ciclo de vida de una petición JSF

En la figura 10 podemos ver el ciclo de vida de una petición en JSF. Las peticiones se procesan mediante un servlet, que carga la vista apropiada, genera un árbol de componentes, procesa eventos y renderiza la respuesta al cliente. El estado de los componentes gráficos se guarda al final de cada petición y se restauran cuando se crea la siguiente vista.

Figura 10. Ciclo de vida de una petición en JSF



Este proceso se realiza en seis fases. En el diagrama podemos diferenciar las fases principales del ciclo de vida (en color verde) y en qué momento entre estas fases se procesan eventos (en color azul).

Las fases por las que pasa una petición son las siguientes:

Fase de restauración de vista

Esta fase se activa cuando el usuario pide una página JSF ya sea porque ha clickado sobre un enlace o ha aceptado un formulario. En esta fase se construye el árbol de componentes de la vista en el servidor y se le asignan los validadores y oyentes de eventos declarados. En JSF cada vista tiene un identificador. Si el

controlador detecta que ya tenía generado el árbol con el mismo identificador, lo reconstruye, sino crea nuevo.

Fase de asignación de valores

Esta fase se encarga de asignar los valores de la petición (normalmente a partir de los parámetros) a cada uno de los componentes dentro del árbol, realizando las conversiones necesarias. Si se producen errores de conversión, éstos se guardan en una cola en el contexto JSF y se mostrarán cuando se renderize la vista. Al final de esta fase los componentes terminan con los valores asignados y con los mensajes de error y los eventos almacenados en una cola.

Fase de validación

En esta fase se ejecutan todas las reglas de validación de los componentes para comprobar que los valores asignados son válidos. Por cada validación que falla, se genera un error que se inserta en una cola dentro del contexto JSF y se marca el componente como no válido.

En caso de que se hayan producido errores de conversión en la fase anterior o errores de validación en esta fase, se pasa directamente a la fase de renderizado, donde se mostrará la vista actual con los valores aplicados además de informar al usuario de los errores de validación y conversión. Si no ha habido errores se pasa a la siguiente fase.

Fase de modificación de valores

Una vez que los componentes tienen asignados valores correctos, éstos se deben pasar al modelo. Para ello se utilizan los Managed Beans que, tal como se ha comentado anteriormente, son objetos Java que se definen y se utilizan como intermediarios para pasar datos entre los componentes y el modelo. Los valores se asignan a los Managed Beans realizando automáticamente la conversión de tipos que se necesite. Si no se puede realizar alguna conversión, se pasa al renderizado de la vista del árbol actual informando de los errores. Si todo va bien, se procede a la siguiente fase.

Fase de invocación a la lógica de negocio

En esta fase se llama al código de la aplicación, es decir se invoca por fin a la lógica de negocio empleando la información de los Managed Beans que ya contienen los datos perfectamente informados y validados. Una vez ejecutada la lógica, se calcula la próxima vista a renderizar a partir de la navegación declarada en el archivo faces-config.xml y se pasa a la fase final.

Fase de renderizado de la respuesta

En esta fase se renderiza la vista a mostrar (por ejemplo mediante JSPs si es el motor de renderizado utilizado). A esta fase se puede llegar o bien directamente debido a errores de conversión y validación, (en cuyo caso se utiliza el árbol de componentes de la vista que ha fallado para renderizar) ó en caso de que la vista a renderizar sea nueva, se construye un nuevo árbol de componentes. Una vez

renderizado, el servidor almacena el estado de forma que el árbol de componentes estará disponible para utilizarse en la fase de restauración de vista.

2.5. Funcionalidad proporcionada por los frameworks. Comparativa.

En este apartado se va a explicar la funcionalidad común que proporcionan los frameworks comparando la forma en que se proporciona en cada uno:

Control de flujo de navegación declarativo

Se permite construir el flujo de navegación de forma declarativa, esto es, la sucesión de vistas que debe seguir la aplicación dependiendo de la vista actual, los resultados de los procesos que se ejecutan y el estado del servidor. Esta declaración se realiza de forma externa al código, facilitando su creación y mantenimiento.

El mecanismo común en los frameworks estudiados es definir la navegación mediante reglas específicas en un archivo de configuración xml. Además en el caso de Struts2 se permite realizar esta declaración mediante anotaciones Java y, en el caso de JSF, existe la posibilidad de realizar la navegación directamente desde las vistas, si bien no se aconseja utilizar ninguno de estos mecanismos debido a su falta de centralización.

Validación de datos

Los tres frameworks facilitan mecanismos para facilitar el proceso de validación y conversión de tipos de los datos de las peticiones (por ejemplo los datos introducidos en un formulario) así como informar de los errores de validación y conversión al usuario. Los tres frameworks permiten realizar las validaciones a través de código directamente o declarando las validaciones mediante reglas.

Struts1 proporciona:

- un método en el ActionForm que el desarrollador puede sobreescribir para realizar validaciones
- la posibilidad de realizar la validación directamente en los objeto Action
- un plugin llamado Struts Validator que proporciona objetos ValidationForm que permiten la configuración de su validación de forma declarativa

Struts2 proporciona:

- una interfaz Java que pueden implementar las acciones para realizar validaciones
- una colección extensa de anotaciones con reglas de validación que se pueden asignar a los setters y métodos de las acciones para que se validen los valores de forma automática así como la posibilidad de declarar anotaciones nuevas
- posibilidad de declarar reglas de validación en ficheros externos

En el caso de JSF el mecanismo empleado es el de definir y declarar cero ó varios objetos Validator que se asignan a los componentes de interfaz de usuario mediante etiquetas, y que se ejecutan en la fase de validación.

Internacionalización de las aplicaciones

Los tres frameworks facilitan desarrollar aplicaciones en diferentes idiomas y con representaciones de valores adaptados a diferentes países o regiones del mundo. En los tres casos, la internacionalización de las vistas, mensajes de error y representaciones de datos se realiza utilizando los mecanismos standard de internacionalización que vienen incluidos en el lenguaje Java. La traducción y configuración de los diferentes idiomas se realiza en archivos externos.

Gestión de excepciones

Struts1 y Struts2 proporcionan un mecanismo muy sencillo que permite crear clases que manejen las diferentes excepciones que pueden lanzan las acciones al ejecutar la lógica de negocio. En el archivo de configuración se declaran clases ExceptionHandlers que define el usuario y se relacionan con las excepciones que se quiere gestionar.

JSF por su parte no proporciona de base ningún mecanismo para gestionar este tipo de excepciones, de forma que el desarrollador suele tratar directamente las excepciones para redireccionar a páginas de errores utilizando reglas de navegación.

Colecciones de etiquetas

Los tres frameworks disponen de colecciones de etiquetas que proporcionan diversa funcionalidad a las vista tal como generación de html, encapsulación de código para manejar datos del modelo, generar validación en la parte del cliente con Javascript, control de flujo, etc.

Composición de páginas

El mecanismo empleado para proporcionar la composición de páginas es la de integrarse con frameworks externos especializados en esta tarea.

Los frameworks para composición de páginas más empleados actualmente son:

- **Tiles**: un framework de código libre con varios años de experiencia basado en el patrón Composite View que permite componer páginas HTML a partir de plantillas y definiciones. Necesita configuración para cada página. Funciona en los tres frameworks.
- **SiteMesh**: framework de código libre basado en el patrón Decorator. Necesita muy poca configuración. Si bien se integra con gran cantidad de frameworks no se debe utilizar con frameworks basados en componentes como JSF, Tapestry o Wicket

El mecanismo de composición por defecto de Struts1 es Tiles, mientras que Struts2 y JSF permiten integrarse con diferentes frameworks.

3. Especificación de requisitos

Una vez estudiados los diferentes frameworks, se dispone ya de una buena visión de la funcionalidad que se espera de un framework para la capa de presentación J2EE así como diferentes aproximaciones que se han seguido para proporcionarla. En este apartado se van a explicar las características y funcionalidad que debe proporcionar el framework a diseñar que se va a denominar jNeko (Neko: gato en idioma japonés).

La primera característica o requisito que debe cumplir jNeko es que sea un framework basado en el patrón de diseño Modelo-Vista-Controlador. En el estudio de los frameworks se han visto dos aproximaciones distintas que implementan este modelo: Struts 1 y Struts 2, que siguen un modelo basado en acciones, y la solución de JSF que se basa en un modelo basado en componentes. El modelo en el que se va a basar jNeko es el del patrón de diseño MVC Model 2 **basado en acciones**.

Tal como se explicó, los frameworks basados en acciones utilizan un controlador que centraliza las peticiones y establece correspondencias entre las URLs de las peticiones con unidades de trabajo llamadas acciones. El trabajo que realiza una acción es realizar funcionalidad específica para una URL determinada a partir de los datos de la petición, de la sesión o de formularios; llamar a los componentes de lógica de negocio y finalmente proporcionar parte de datos del modelo para la generación de la respuesta. Finalmente se calcula la vista que hay que mostrar y se renderiza.

Las ventajas de utilizar un framework basado en acciones son:

- La aplicación que extiende el framework adopta automáticamente la separación de la presentación y la lógica de control debido a que esos son los puntos de extensión.
- Proporciona la capacidad de crear distintas representaciones de los mismos datos
- Permite reutilizar componentes
- Permite simplificar la construcción y el mantenimiento de la aplicación
- Permite la separación de roles en la construcción: por ejemplo los diseñadores gráficos se pueden centrar en trabajar en las vistas y los programadores en las acciones, formularios, custom tags y código para acceder y tratar el modelo
- Proporciona un punto de control centralizado y mecanismos para la configuración del framework
- Facilita para la realización de pruebas unitarias de los componentes
- Facilita el desarrollo de prototipos rápidos

Tal como se ha visto en el estudio de los diferentes frameworks, existe un conjunto de funcionalidad y características mínimas que se espera de un framework de

desarrollo de la capa de presentación. jNeko va a proporcionar la funcionalidad esperada de forma que debe satisfacer como mínimo los siguientes requisitos:

- **Control declarativo**. jNeko debe permitir construir el flujo de navegación de forma declarativa, esto es, debe permitir declarar la sucesión de vistas que debe seguir la aplicación, definir las acciones a ejecutar y relacionarlos con objetos de validación mediante reglas de forma externa al código.
- **Validación de datos**. Debe proporcionar mecanismos para facilitar el proceso de validación y conversión de tipos de los datos de las peticiones (por ejemplo los datos introducidos en un formulario) así como informar de los errores de validación y conversión al usuario automáticamente.
- **Colecciones de etiquetas especializados.** Se debe proporcionar librerías de etiquetas que faciliten el desarrollo de las vistas y evite el uso de código en las mismas.
- **Internacionalización de las aplicaciones.** jNeko proporcionará mecanismos para facilitar el desarrollo de aplicaciones traducidos a diferentes idiomas.

Además de la funcionalidad mínima necesaria para la construcción de una aplicación web. jNeko proporcionará las siguientes facilidades:

- **Plugins**. Posibilidad de mejorar la aplicación añadiendo funcionalidad de forma declarativa.
- **Resultados extensibles.** Posibilidad de añadir nuevos tipos de resultados de manera similar a Struts2 de forma que no se limite el framework a redireccionar a JSP's sino que se puedan utilizar otras tecnologías de vistas o crear nuevas de forma declarativa o mediante plugins.
- **Gestión de excepciones.** jNeko proporcionará un mecanismo que permita de forma declarativa tratar las diferentes excepciones que pueden lanzan las acciones al ejecutar la lógica de negocio.
- **Composición de vistas**. jNeko contemplará la posibilidad de que el desarrollador pueda realizar vistas compuestas ya sea directamente o proporcionando mecanismos para poder utilizar alguna herramienta externa.

4. Diseño del framework

A continuación se muestra el desarrollo del diseño del framework jNeko. Primero se muestra el diseño de las clases que se encargan de obtener y albergar la información de la configuración del framework. Después se presenta el diseño del *core* del framework, encargado de inicializar el framework y procesar las peticiones. Después se muestran diagramas de los diferentes componentes encargados de implementar diferentes requisitos del framework. Por último se muestra el diagrama de clases que muestra el diseño final.

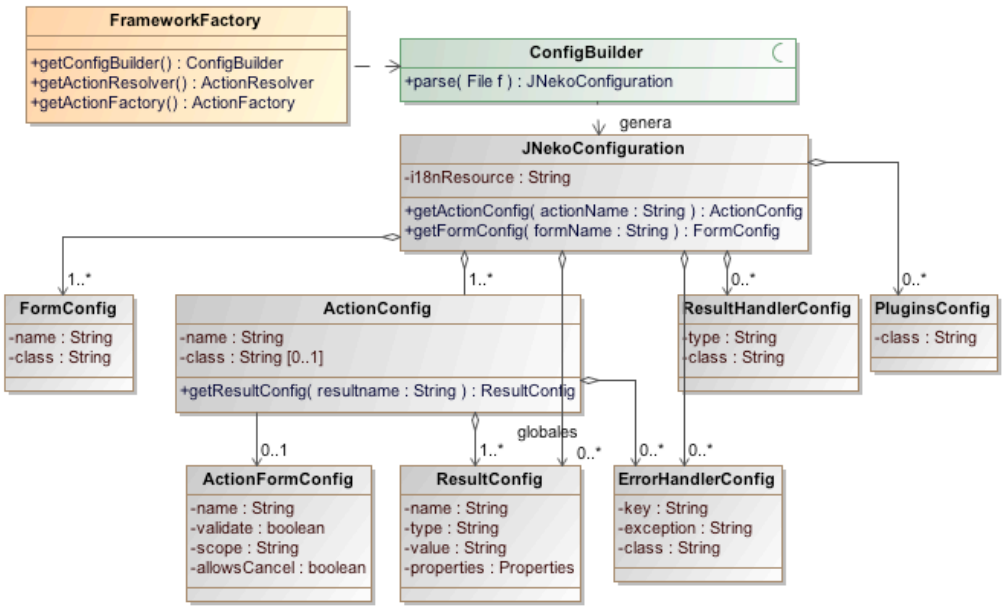
4.1. Diseño de configuración del framework

Antes de empezar a servir peticiones, lo primero que debe hacer el framework al inicializarse es leer la configuración del funcionamiento. La configuración del framework es declarativa y se va a realizar mediante un archivo xml específico que por defecto se encontrará en el directorio WEB-INF con nombre *jneko-config.xml*

En este archivo se definen los aspectos de configuración del framework como los mapeos de acciones, formularios, etc... La información contenida en el archivo va a ser necesaria a lo largo del ciclo de vida de ejecución del framework por lo que para optimizar el rendimiento al inicializarse el framework, se accederá al fichero y se mantendrá una copia en memoria mediante la clase jNekoConfiguration, que servirá de almacén y organización de la configuración además de proporcionar métodos para su consulta.

En la siguiente figura vemos el diagrama de clases responsables de leer la configuración y las clases que forman la configuración.

Figura 11. Diagrama de clases: configuración de jNeko

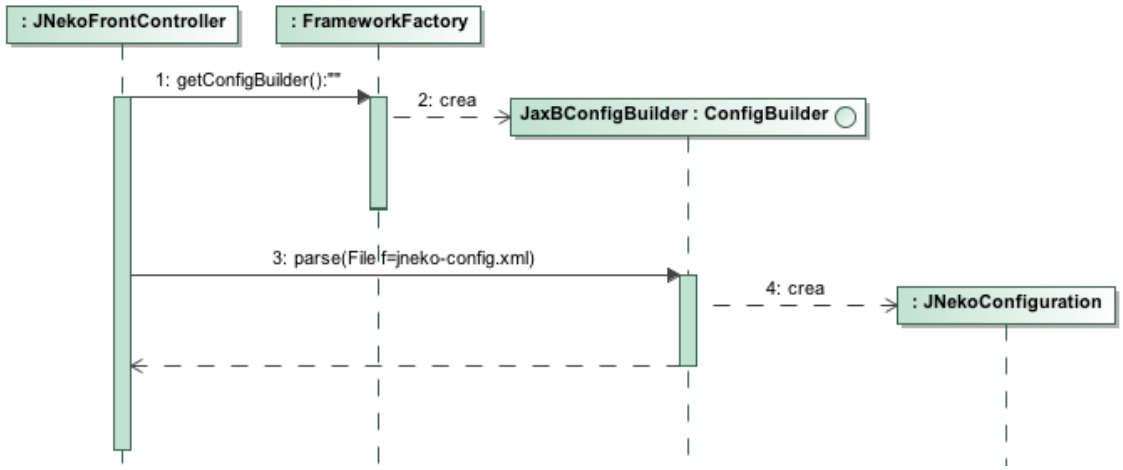


La clase JNekoConfiguration contiene toda la información de la configuración del framework. Para obtener la configuración se utiliza un objeto que implemente la interfaz ConfigBuilder. La implementación de ConfigBuilder deberá leer el archivo xml de configuración y devolver un objeto JNekoConfiguration que contendrá todos los datos necesarios para el funcionamiento del framework.

Internamente se utiliza una factoría FrameworkFactory que contiene un método que nos proporcionará una implementación de ConfigBuilder. La clase FrameworkFactory se basa en el patrón Abstract Factory para instanciar diversos objetos necesarios por el framework. Estos métodos retornan interfaces, de forma que hemos encapsulado código en clases siguiendo el patrón Strategy así que podemos cambiar la implementación o probar diversas implementaciones sin que el resto de componentes se vean afectados.

En el siguiente diagrama de secuencia se puede observar las interacciones y objetos que intervienen para obtener la configuración a partir del archivo.

Figura 12. Diagrama de secuencia: obtener configuración



En el diagrama de secuencia muestra el parseo del archivo xml con un supuesto parseador que utiliza JAXB. En la fase de implementación decidiremos si utilizar DOM, StAX o la implementación que queramos. Esta facilidad es para nosotros como constructores del framework y no para quien lo extiende.

4.1.1. Descripción de las clases de la configuración

La clase JNekoConfiguration se relaciona con siete clases distintas que contienen la configuración del framework. Cada clase encapsula un elemento de configuración distinto definido en el archivo de configuración. El nombre de estas clases acaban con

el sufijo Config y se muestran en el diagrama de clases en color gris para diferenciarlas.

A continuación se exponen las clases de las que se compone y una breve descripción:

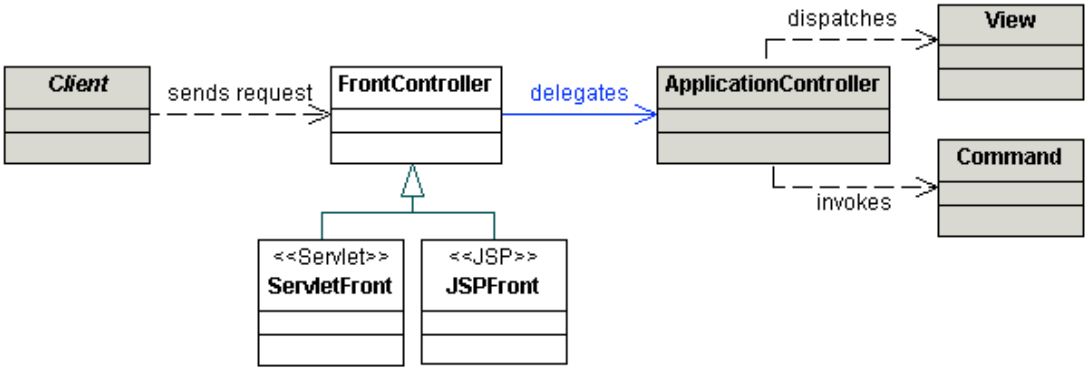
- **FormConfig**: permiten la definición formularios definidos por el usuario (ActionForm). Contienen el nombre lógico del formulario y el nombre de la clase que lo implementa.
- **ActionConfig**: contiene información de un mapeo a una acción definida por el usuario. Un mapeo contiene:
  - un nombre de mapeo que es una cadena que debe ser única, corresponde con el path de la petición con el que se asocia este mapeo.
  - opcionalmente el nombre de la clase Action que se debe ejecutar.
  - uno o varios definiciones de resultados ResultConfig.
  - cero o una definición de formulario a utilizar ActionFormConfig.
  - cero o varias definiciones de gestores de errores
- **ActionFormConfig**: una acción puede utilizar opcionalmente un formulario (ActionForm). En caso de tener un formulario asignado, se dispone del nombre lógico del formulario, un atributo “validate” que indica si el framework tiene que validar o no el formulario, un atributo “scope” para indicar el ámbito (request o session) donde localizar / establecer el formulario cuando se trata y por último un atributo allowsCancel que indica si el formulario permite cancelarse.
- **ResultConfig**: una acción define uno o varios resultados que se transformarán en vistas. Cada resultado tiene un nombre lógico, un tipo de resultado, un valor y unas propiedades. Las acciones retornan cadenas con estos nombres lógicos para que el framework sepa el resultado que tiene que servir. También existen declaraciones de resultados a nivel global aunque tienen preferencia a nivel acción.
- **ErrorHandlerConfig**: contiene información para registrar gestores de errores. Es una clave de error, el nombre de la excepción que debe tratar el gestor de errores y la clase que implementa el gestor de errores.
- **PluginsConfig**: información necesaria para instanciar un Plugin. Se dispone del nombre de la clase de cada plugin a registrar.
- **ResultHandlerConfig**: jNeko permite a semejanza de Struts2, definir nuevos tipos de resultado. Esta clase contiene el nombre lógico del tipo de resultado y una cadena con la clase que permite renderizar el resultado. Cabe decir que jNeko incorpora de serie dos tipos:
  - “dispatcher” - tipo por defecto - se utilizará para servir una página jsp
  - “redirect” - para redireccionar a una URL interna o externa mediante una respuesta HTTP 302

4.2. Diseño del procesamiento de peticiones

4.2.1. Recepción y control centralizado de peticiones

Para recibir las peticiones en jNeko se ha empleado el patrón de diseño ***Front Controller***. Este patrón permite centralizar y procesar todas las peticiones que envía el cliente a través de un único controlador, de forma que se puede centralizar la lógica en un único punto en vez de tenerla duplicada o distribuida entre las vistas o entre otros componentes.

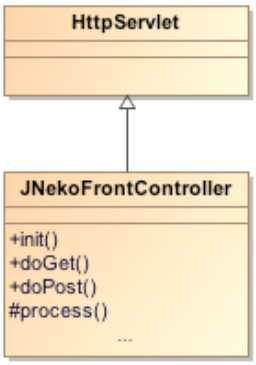
Figura 13. Patrón de diseño Front Controller



El control centralizado facilita además la monitorización, auditoria y seguridad de las aplicaciones al pasar todas las peticiones por un único punto de forma que es más sencillo por ejemplo detectar peticiones no autorizadas y denegar el acceso que si la lógica estuviese desplegada por diferentes puntos de la aplicación.

En jNeko, el papel de controlador lo realiza la clase JNekoFrontController siguiendo la estrategia *Servlet Front Strategy,* de forma que se implementa con un servlet que capturará las URL's dirigidas hacia el framework para procesarlas.

Figura 14. Front Controller mediante Servlet Front Strategy



Como jNekoFrontController es un servlet, se aprovecha la característica de su ciclo de vida para inicializar el framework. En su método init() se leerá el fichero de configuración del framework tal como se ha descrito en el punto anterior, además de crear y ensamblar las clases que se necesiten para su funcionamiento e inicializar los plugins.

Un aspecto a considerar es cómo dirigimos las peticiones para que las sirva el controlador. La forma recomendada de las Java BluePrints es utilizar el mecanismo de mapeo de URLs que proporciona el contenedor servlet. En jNeko mapearemos mediante el descriptor de configuración estándar del contenedor servlet *WEB-INF/web.xml* que todas las URLs cuyo path acaben con el sufijo **.action** se envíen al servlet JNekoFrontController.

Una vez inicializado el framework, las peticiones que JNekoFrontController recibe mediante sus operaciones *doGet* o *doPost*, las redirige a su método *process* para procesarlas.

4.2.1. Procesamiento de peticiones

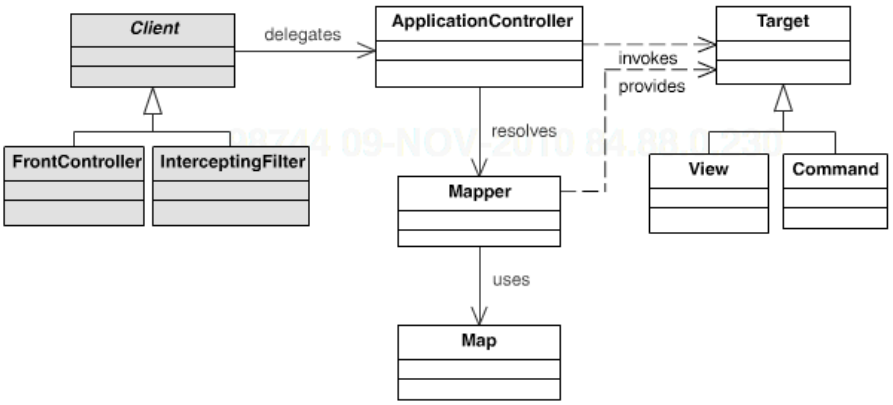
Cuando el controlador recibe la petición, se deben realizar dos procesos:

- Determinar la acción que hay que ejecutar a partir de la URL y ejecutarla
- Determinar cuál es la siguiente vista a mostrar y servirla

El código que hay que ejecutar al recibir una petición y la vista siguiente que hay que servir depende de cada aplicación específica. Nos interesa tener estos aspectos encapsulados e independientes del controlador frontal además de poderlos configurar de forma declarativa.

El patrón de diseño ***ApplicationController*** nos proporciona esta facilidad.

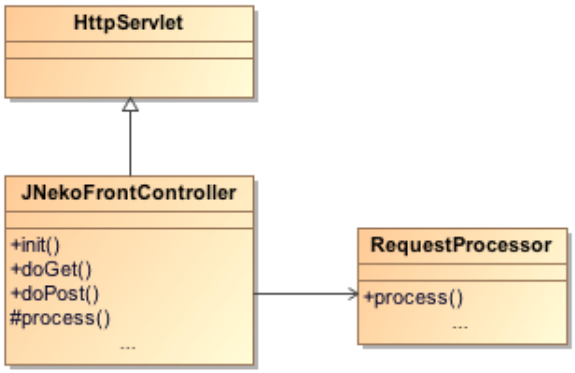
Figura 15. Patrón de diseño ApplicationController



Este patrón separa el control del flujo de navegación y la lógica a ejecutar en componentes separados. Propone que el Front Controller que recibe la petición la delegue a un Application Controller que gestiona las acciones. El Application Controller resuelve la acción que hay que ejecutar a partir de la petición mediante un Mapper e invoca la acción mediante el uso del patrón Command.

En jNeko, se aplica este patrón de diseño introduciendo la clase RequestProcessor que adopta el papel de Application Controller. Cuando jNekoFrontController recibe una petición, la pasa al RequestProcessor mediante el método process() donde se realizará el tratamiento.

Figura 16. RequestProcessor según ApplicationController

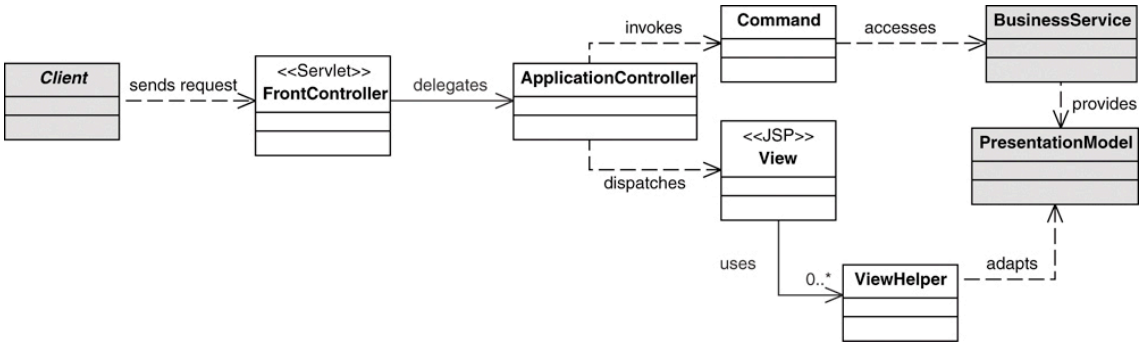


El papel del *Mapper* se realiza entre la clase RequestProcessor y la clase *JNekoConfiguration* vista anteriormente que contiene entre otra información la configuración de los mapeos de acciones y resultados que defina el usuario del framework. El papel de *Map* del patrón de diseño corresponde a un objeto de mapeo *ActionConfig* que asocia una acción *Action* a ejecutar a una petición ademas de aportar la definición de los diversos resultados *ResultConfig* que se pueden despachar según el resultado de la ejecución.

El papel de los Command se implementa mediante la clase Action. El usuario debe proporcionar una clase que extiende de Action e implementar el método abstracto execute() con el código particular que se deba ejecutar.

Cuando se ejecuta una acción, normalmente se desea mostrar en las vistas información perteneciente al modelo de datos. Se ha decidido utilizar el patrón de diseño Service To Worker (ver figura 17) que proporciona una solución de diseño que permite invocar lógica de negocio específica para una petición donde se obtienen los datos del modelo y se ponen a disposición de las vistas dinámicas mediante ViewHekpers.

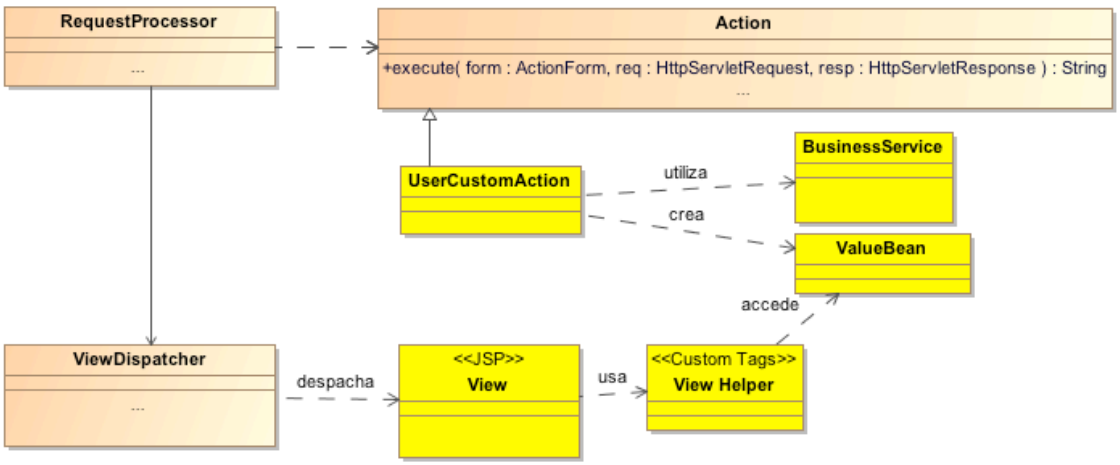
Figura 17. Patrón de diseño Service To Worker



Service To Worker está compuesto por diversos patrones de diseño que trabajan entre sí. Propone un Front Controller que recibe la petición y la delega a un Application Controller que gestiona las acciones. El Application Controller resuelve la acción que hay que ejecutar a partir de la petición y la invoca siguiendo el patrón Command. La acción se comunica con servicios que proporcionan la lógica de negocio y éste retorna parte del modelo Presentation Model. En este punto el Application Controller se encarga de determinar la vista a mostrar y la sirve. Por último mediante View Helpers se localizan, adaptan y transforman los datos del modelo para utilizar en la vista, que puede ser perfectamente una Composite View ya que no depende del patrón.

En la figura 18 se puede observar la aplicación del patrón de diseño Service To Worker en jNeko.

Figura 18. Service To Worker en jNeko



Como se ha mencionado anteriormente, las acciones se definen mediante la clase Action que sigue el patrón *Command.* En la figura 18 se puede observar un ejemplo donde el usuario del framework ha declarado una acción UserCustomAction donde en

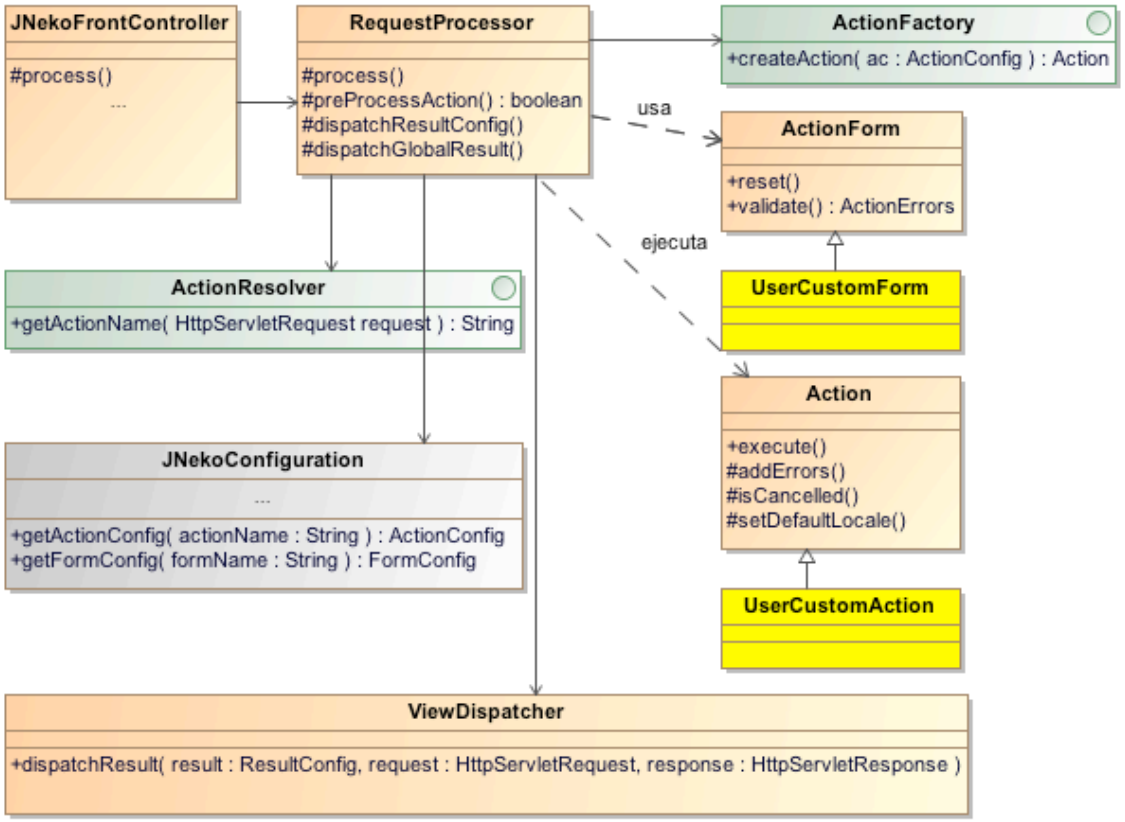
su método execute() invoca a la lógica de negocio y pone los datos para la vista en un ValueBean.

El método execute() retorna una cadena indicando el resultado lógico a ejecutar que determinará la próxima vista. Dado que vamos a poder generar diferentes tipos de resultados, el RequestProcessor va a utilizar un ViewDispatcher que es un componente especializado que se va a encargar de generar las vistas a partir de los resultados definidos.

En jNeko el paso de datos de información del modelo desde las acciones a las vistas se realizará asignándolo los atributos del objeto HTTPServletRequest. También se va a proporcionan formularios como los de Struts1 que se pueden mapear y asignar al request o a la sesión de forma automática mediante la configuración y que serán traspasados desde las vistas a las acciones y viceversa mediante la librería de etiquetas que se proporciona.

El siguiente diagrama de clases muestra el diagrama de clases con las clases encargadas de procesar una petición en jNeko.

Figura 19. Diagrama de clases: tratar una petición en jNeko



Las clases con las que interactúa el RequestProcessor son las siguientes:

- **ActionResolver**: El RequestProcessor utiliza esta interfaz para obtener el nombre lógico de la acción a invocar a partir del request.

  Es una interfaz cuya implementación se obtiene a través del FrameWorkFactory. La implementación por defecto en jNeko es como en Struts1 donde se utiliza el path como correspondencia con la acción que se debe ejecutar. Por ejemplo: la URL http://maquina.com/**usuarios/listar**.action corresponde a la acción declarada en el fichero de configuración con el nombre lógico **/usuarios/listar.** Este es el comportamiento por defecto. Se ha implementado en una interfaz por si en una futuro se permite que el usuario pueda cambiar este comportamiento: (por ejemplo para poder mapear acciones a partir del último componente del path o alguna otra fórmula de correspondencia).

- **JNekoConfiguration**: tal como se ha explicado, es la configuración del framework, contiene métodos para acceder a los diferentes mapeos de acciones y resultados así como otros recursos declarados en la configuración.
- **ActionForm**: clase base que representa un formulario. Es un punto de extensión del framework. El usuario puede extender esta clase para definir sus formularios, declararla en el fichero de configuración y asignarlo a una acción. El RequestProcessor lo utiliza para asignarle valores de la petición, realizar validaciones automáticas y pasar la información a las acciones.
- **Action**: clase abstracta que utiliza el patrón de diseño *Command* para encapsular las invocaciones a la lógica de negocio. Esta clase es un punto de extensión del framework. Action define un método execute(). El usuario debe extender la clase Action e implementar el método para proporcionar la lógica específica de su aplicación.
- **ViewDispatcher**: clase cuya responsabilidad es servir los resultados.

Los pasos para procesar una petición son los siguientes:

- El cliente envía una petición que el controlador recibe con doGet() o doPost()
- El controlador JNekoFrontController llama al método **process**() del RequestProcessor pasándole la petición (*request* y *response* del contenedor servlet) para que la trate.
- El proceso se realiza internamente en varias fases:
  - **resolver mapeo**: en este paso se trata de obtener a partir de la petición y la configuración el objeto ActionConfig que contiene la definición de acción, formulario y vistas relacionadas con la petición.
  - **procesar formulario**: este paso se realiza si la acción configurada tiene definido un formulario. En este paso se instancia el ActionForm(o se obtiene uno de sesión si ya se había creado.) correspondiente a la acción

a ejecutar. Se inicializan sus datos llamando al método reset() y por último se le informan los datos de la petición llamando a los setters del formulario.

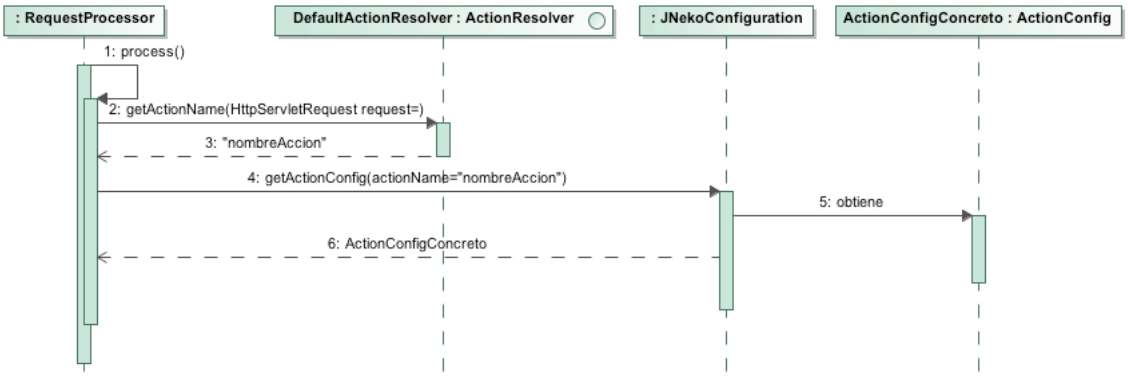
- **procesar validación de formulario**: si hay formulario y hay que validarlo, se valida llamando al método validate(). Si se producen errores de validación, se termina la ejecución de la petición y se sirve el resultado declarado con nombre “input”. Si no hay errores de validación, el proceso continúa.
- **ejecutar acción**: en este paso se invoca a la acción que esté definida en ActionConfig, creándola y llamando a su método execute(), pasándole el formulario ActionForm para que pueda tratar sus datos además del *request* y *response* y el objeto ActionConfig por si necesita consultarlo. Con la cadena que retorna el método se obtiene el ResultConfig de la vista a servir.

- Una vez se ha tratado la petición, se llama a la clase ViewDispatcher para que sirva el resultado mediante su método dispatchResult() y finaliza el procesado.

4.2.3. Resolución de acciones mapeadas a partir de la petición

En la figura 20 se detalla el diagrama de secuencia de la operación processMapping() para resolver el objeto ActionConfig que corresponde a partir de la petición.

Figura 20. Diagrama de secuencia: resolver un mapeo



Ejemplo de configuración de una acción

```
<action name="/security/login" class="com.custom.LoginAction">
       <form validate="true">formLogin</form>
       <result name="success">/jsp/main.jsp</result>
       <result name="error">/jsp/error.jsp</result>
       <result name=”input”>/jsp/formLogin.jsp"</result>
       <result name="gencaptcha" type="captcha">
           <param name="numletras">12</param>
              <param name="estilo">tachado</param>
       </result>
  </action>
```

Se puede observar que se hace uso del ActionResolver para obtener el nombre de la acción y a continuación se accede al jNekoConfiguration que retornará el ActionConfig correspondiente a ese nombre lógico que debe ser único para toda la aplicación.

Por ejemplo: dado el ejemplo de la figura 20, a partir de la URL http://<dominio>/ security/login.action, el ActionResolver retornará “/security/login” y se retornará el ActionConfig que contiene toda al información del mapeo del ejemplo.

4.2.4. Procesar formularios

Este paso se ejecuta si el mapeo tiene configurado un formulario. Se encarga de localizar el formulario ActionForm en el scope que el usuario haya definido en la configuración y si no lo encuentra crear una nueva instancia. A continuación asigna automáticamente los valores del request al formulario para que posteriormente la acción pueda utilizarlo. El usuario registra en el fichero jneko-config.xml los formularios que utiliza la aplicación.

En los siguientes extractos de un fichero de ejemplo, se puede observar un ejemplo donde el usuario define una clase com.custom.FormLogin que extenderá de ActionForm y la registra con nombre “formLogin”.

```
<!-- Formularios de usuario -->
  <forms>
    <form name="busquedaForm" class="com.custom.Form"/>
    <form name="formLogin" class="com.custom.FormLogin"/>
    ...
  </form>
```

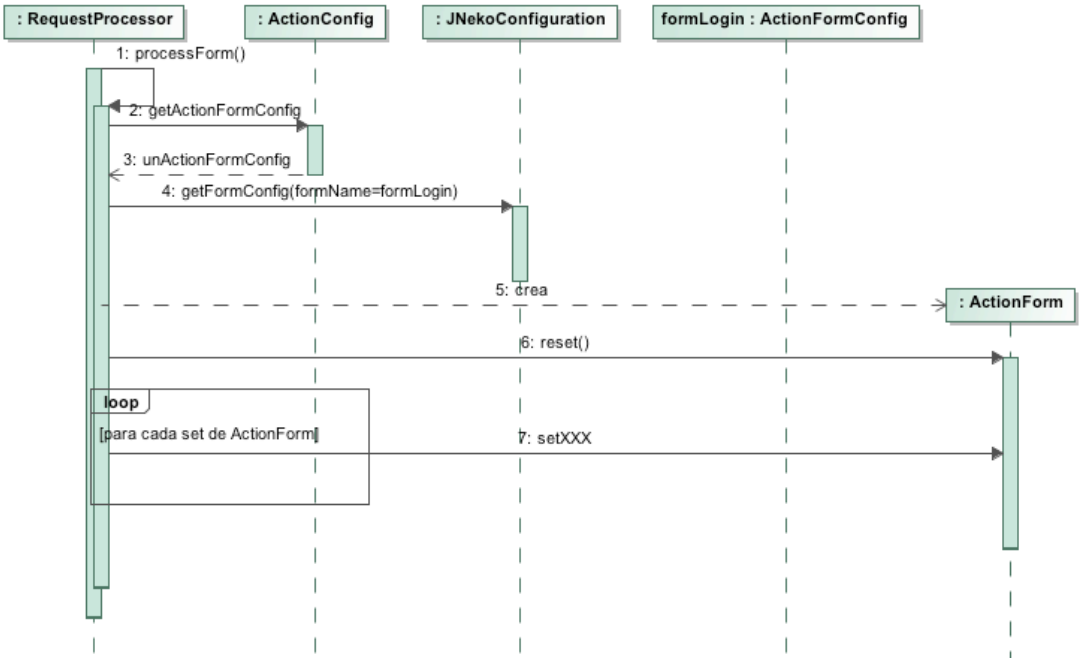
Para que una acción utilice un formulario, debe declararlo en el mapeo de la acción mediante el elemento opcional <form>. En este ejemplo se puede observar un mapeo que quiere utilizar el formulario “formLogin” declarado anteriormente, además de indicar al framework que se quiere validar y que busque la instancia o la establezca en la sesión.

```
  <action ....
    <form validate="true" scope=”session”>formLogin</form>
    ...
    <result name=”success”>/ok.jsp</result>
    <result name=”input”>/jsp/formLogin.jsp"</result>
  </action>
```

Cuando se utiliza un formulario hay que definir obligatoriamente un tipo de resultado con nombre “input”. Este resultado lo servirá automáticamente jNeko en caso de que no se pase la validación de formulario. En este ejemplo siguiente si no se pasa la validación, el framework servirá la vista /jsp/formLogin.jsp ya que es el resultado definido con nombre “input”.

El diagrama de secuencia para procesar el formulario es el siguiente:

Figura 21. Diagrama de secuencia procesar formulario

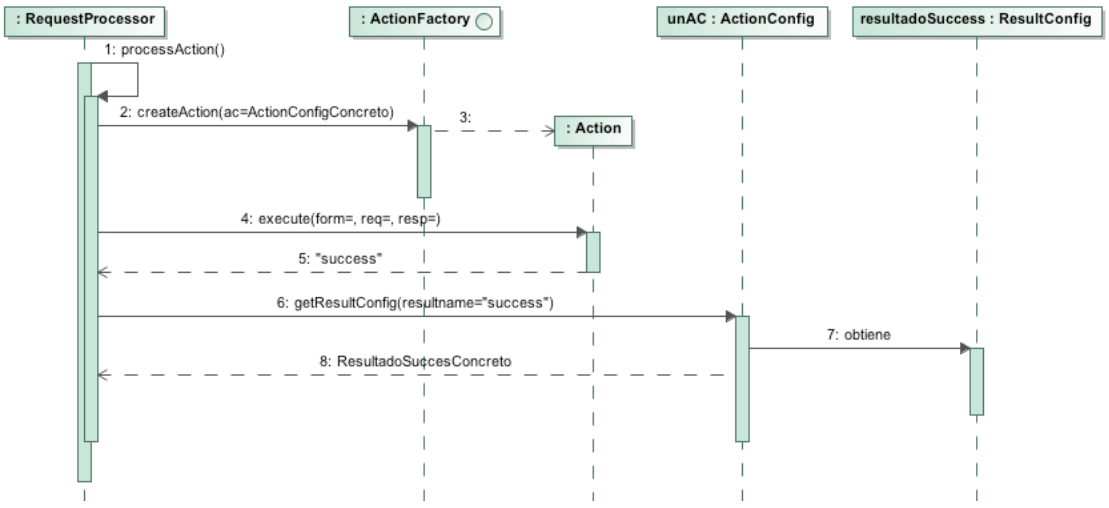


Se puede observar que se acceden a los datos de configuración para obtener el nombre de la clase del formulario. Si el formulario existe en sesión, se utiliza ése, sino se crea uno nuevo. A continuación se llama al método reset() para que se inicialicen sus valores y finalmente se llama a los setters del formulario pasando los datos del request.

4.2.5. Invocación de acciones

El diagrama siguiente muestra los participantes para invocar una acción y sus relaciones.

Figura 22. Diagrama de secuencia invocar una acción



El RequestProcessor crea la clase que implementa la acción a través de un objeto ActionFactory, de esta forma se separa la responsabilidad de creación de las acciones para poder implementarlas como sea más conveniente por ejemplo creando instancias nuevas para cada request o utilizando un pool de Actions. Cabe decir que en la implementación final se ha utilizado como ActionFactory una caché de Actions de forma que se reutilizan las instancias creadas, por lo que el código de las acciones debe ser *thread safe*.

A continuación se invoca a la acción llamando al método *execute*. Este método además de implementar la lógica necesaria, debe retornar una cadena con el resultado de la ejecución. Esta cadena se utilizará para resolver la vista que hay que renderizar.

En el fichero de configuración se declaran resultados. Éstos pueden definirse tato a nivel de acción como a nivel global. Cuando se ejecuta una acción , la cadena que retorna el método *execute* se busca dentro de los resultados declarados en el mapeo de la acción. Si no se encuentra se busca en los resultados declarados a nivel global.

```
<navigation>

<global-results>
               <result name="auth">/mustLoginPage.jsp</result>
               <result name="exception">/errorPage.jsp</result>
</global-results>

 <action ....
   <form>formLogin</form>
  ...
   <result name=”success”>/ok.jsp</result>
   <result name=”input”>/jsp/formLogin.jsp"</result>
   <result name="error">/jsp/error.jsp</result>
   <result name="gencaptcha" type="captcha">
         <param name="numletras">12</param>
         <param name="estilo">tachado</param>
    </result>
  </action>
...
</navigation>
```

La información de un result se almacena en jNeko en un objeto ResultConfig. A partir de la cadena que devuelve el método se accede a la configuración ActionConfig para obtener el ResultConfig correspondiente y se retorna para que se sirva por fin el resultado en el siguiente paso.

4.2.6. Servir resultados y resultados extensibles

El último paso consiste en a partir del resultado *ResultConfig* servir la vista que corresponda. Esta misión se delega en la clase ViewDispatcher que tiene acceso a la definición de los diferentes renderizadores de resultados *ResultHandler* para generar la respuesta.

Un resultado ResultConfig tiene un nombre, un tipo, un valor y opcionalmente parámetros. Por ejemplo:

```
<result name="error">/jsp/versus.jsp</result>
<result name="gencaptcha" type="captcha">
   <param name="numletras">12</param>
   <param name="estilo">tachado</param>
</result>
```

El atributo type indica el tipo de resultado y se utiliza para determinar la clase que debe generar la vista.

Los tipos que vienen ya definidos en jNeko son “dispatcher” y “redirect”:

- “dispatcher” se utiliza para servir la página JSP definida en el cuerpo del elemento <result>. El tipo por defecto si no se especifica es “dispatcher”. Por ejemplo:

```
<result name="success" type="dispatcher">/jsp/ok.jsp</result>
<result name="error">/jsp/error.jsp</result>
```

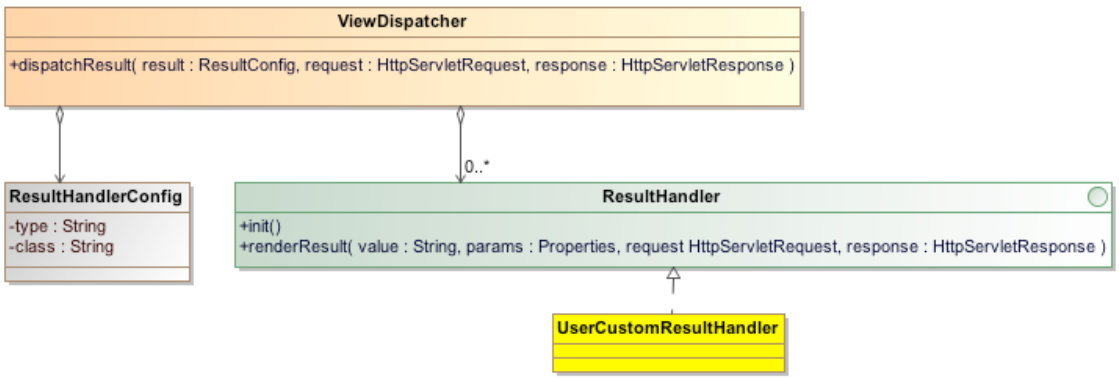
- “redirect” sirve para redireccionar a la URL definida en el cuerpo <result>. Por ejemplo:

```
<result name="gotoExternalBrowser" type="redirect">http://www.google.es</result>
```

Si el tipo de ResultConfig es “dispatcher” o “redirect”, el ViewDispatcher utiliza sus métodos internos para servir la página o redireccionar a la URL y finaliza el procesamiento de la petición.

El diagrama de clases para servir los resultados es el siguiente:

Figura 23: Diagrama de clases: servir resultados



El desarrollador puede crear nuevos tipos de respuesta creando clases que implementen la interfaz ResultHandler. Después debe declararlos en el fichero de configuración definiendo un nuevo tipo con el atributo type.

Por ejemplo el siguiente fragmento declara que los resultados de tipo “captcha” los va a servir la clase Captcha que implementa la interfaz ResultHandler.

```
<!-- Registro de manejadores de resultados o renderers -->
 <reg-result-handlers>
    <result-handler type="captcha" class="com.custom.result.Captcha"/>
    <result-handler .../>
    <result-handler .../>
 </reg-result-handlers>
```

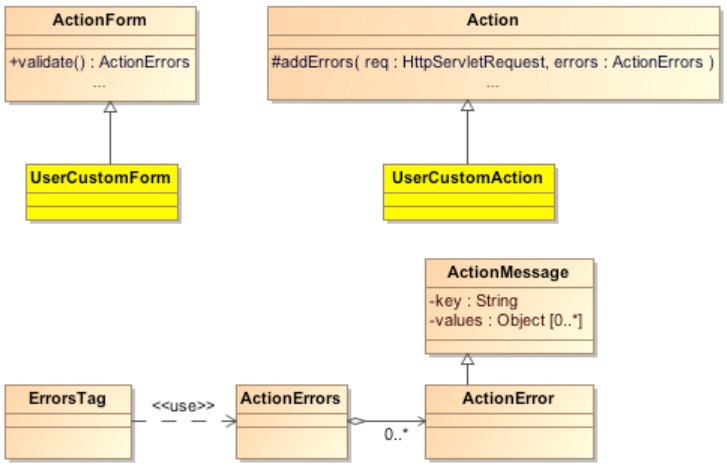
Esta información queda almacenada en los objetos de configuración ResultHandlerConfig. El ViewDispatcher utiliza esta información para determinar la clase que se debe utilizar para renderizar un tipo concreto.

Cuando hay que servir un tipo de resultado definido por el usuario, se busca en la configuración de ResultHandlerConfig la clase que debe renderizar el resultado. El ViewDispatcher utiliza una cache de ResultHandler ya instanciados para renderizar el resultado. Si no se ha instanciado ya la clase resultHandler correspondiente, se instancia una y se le proporciona un ciclo de vida llamando a init() para que pueda inicializar recursos y después almacena la instancia en su caché. A partir de entonces se reutiliza la instancia llamando al método renderResult , pasando por parámetros el valor del contenido del elemento y las propiedades declaradas en el elemento <result> además del request y response para que genere la respuesta necesaria.

4.3. Validación de datos de usuario

Para validar los datos, se define la clase ActionError, que representa un mensaje de error y la clase ActionErrors que es una colección que contiene objetos ActionError.

Figura 24: Diagrama de clases: validación de datos



Los puntos donde se pueden validar datos es en un formulario ActionForm y en una acción Action. En el ActionForm se define un método validate que retorna una colección de errores ActionErrors. Si la colección retornada es null o no contiene elementos se considera que no se han producido errores. En el caso de Action, se proporciona un método protegido *addErrors* que se puede llamar durante la ejecución de la acción para informar de errores.

En el caso de haber errores, éstos se añaden en un atributo del *request*. Las vistas pueden utilizar la etiqueta <html:errors> que implementa la clase ErrorsTag para mostrar los errores que se hayan producido durante la petición en curso.

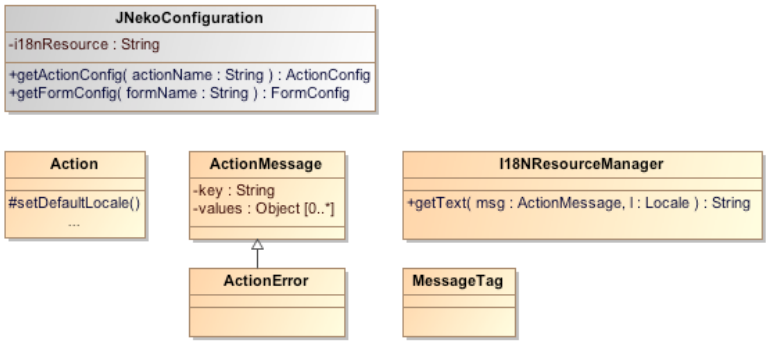
jNeko proporciona un *View Helper* para que la vista pueda mostrar los errores producidos. Este *View Helper* es la etiqueta <html:errors> que implementa la clase ErrorsTag proporcionado en la librería y que se encarga de mostrar los errores que se hayan producido durante la petición en curso. La forma de hacerlo es accediendo al atributo del *request* encargado de almacenar el ActionErrors y mostrarlos en caso de que se encuentren mensajes.

4.4. Internacionalización

Para permitir la internacionalización de las aplicaciones, jNeko va a hacer uso del mecanismo estándar de Java para la internacionalización y localización de aplicaciones de forma que el usuario deberá declarar un archivo para cada región a internacionalizar y que jNeko utilizará como ResourceBoundle para realizar la traducción. El usuario deberá registrar el ResourceBundle en jNeko a través del fichero de configuración mediante el elemento i18n-resource cuyo valor se almacenará en JNekoConfiguration.

```
<!-- Internacionalización -->
<i18n-resource>com.custom.Mensajes</i18n-resource>
```

Figura 25: Diagrama de clases: internacionalización



La internacionalización se realiza a partir de claves que se utilizan para obtener valores en el idioma que sea necesario a través de ficheros .properties que debe proporcionar el usuario del framework. Para realizar las traducciones, el controlador frontal jNekoFrontController crea al inicializar el framework un objeto I18NResourceManager que mantiene en memoria los ResourceBoundles que se van utilizando en la aplicación y se encarga de traducir las claves y argumentos en cadenas de texto mediante el método getText.

El Locale a utilizar se puede fijar en la sesión de un cliente de forma que todas las peticiones del cliente utilicen ese Locale. El Locale se puede establecer por defecto a todos los clientes declarándolo en el archivo de configuración, otra opción es establecerlo desde las acciones llamando al método proporcionado setLocale. Si el Locale no está establecido para un cliente, se utiliza el Locale incluido en el objeto HttpServletRequest.

La clase ActionMessage acepta una clave y argumentos para realizar la traducción. Dado que ActionError hereda de esta clase, significa que los errores también pueden ser traducidos al idioma que se necesite.

Para poder mostrar textos en las vistas, se proporciona una etiqueta <message> implementada en la clase MessageTag que permitirá a las vistas JSP's insertar textos en diferentes idiomas mediante la traducción de la clave al mensaje correspondiente

del idioma. Además algunas de las etiquetas proporcionadas en jNeko (por ejemplo las que generan botones submit o reset en un formulario) declaran un atributo messageKey que permite la internacionalización del control que se renderiza.

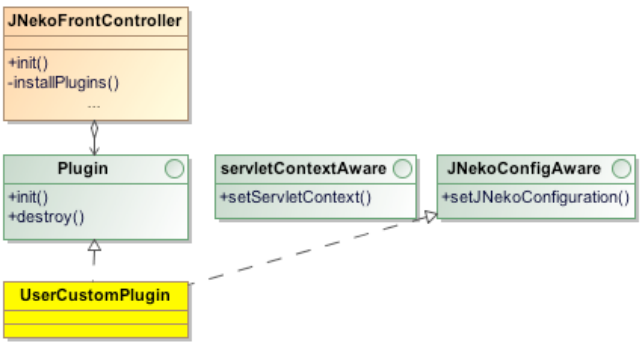
4.5. Plugins

El desarrollador puede declarar Plugins mediante el archivo de configuración.

```
<!-- Registro de plugins -->
<reg-plugins>
  <plugin class="com.custom.Plugin"/>
  <plugin .../>
  <plugin .../>
</reg-plugins>
```

El usuario debe crear una clase que implemente la interfaz Plugin. Esta interfaz declara dos métodos init() y destroy() que serán llamados al inicializarse y al finalizar la ejecución del contenedor respectivamente.

Figura 26. Diagrama de clases: plugins



Se proporcionan también dos interfaces ServletContextAware y JNekoConfigAware que sirven para que el Plugin pueda tener acceso al contexto del Servlet y a la configuración del framework respectivamente. El plugin que necesite acceso a alguno de estos objetos, ya sea para consultarlos o modificarlos, deberá implementar la interfaz para que le sean proporcionados mediante inyección de dependencias. Si en el futuro se quiere dar acceso a otros objetos se pueden añadir más interfaces Aware sin romper a los clientes existentes ni tener que proporcionar nuevos métodos en la interfaz Plugin.

Durante la inicialización de jNeko, el controlador frontal lee la configuración y crea una instancia de cada plugin en el orden declarado. Para cada plugin instanciado le inyecta las dependencias necesarias y llama al método init().

jNeko llamará al método *destroy()* cuando se termine la ejecución del contenedor de servlets en el orden inverso al que fueron inicializados. Por este motivo en la implementación el controlador frontal utiliza una pila donde inserta los plugins a

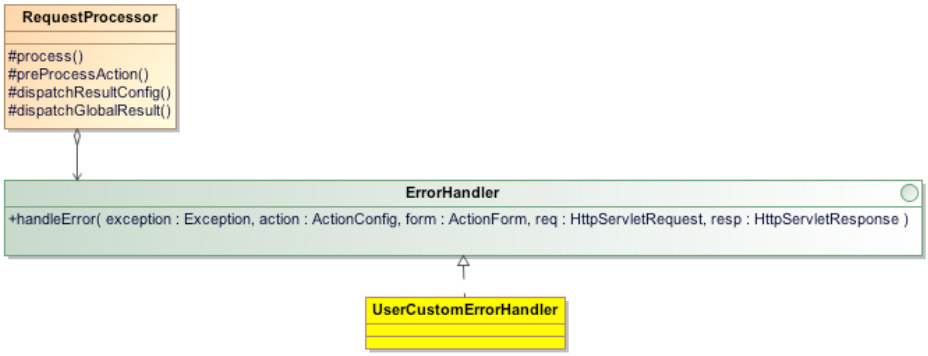
medida que los inicializa para así destruirlos a medida que se extraen de la pila al finalizar la ejecución del contenedor servlet.

Este diseño es una aplicación del patrón de diseño *Adapter* que permite ejecutar código diferente proporcionado por el usuario a través de una única interfaz.

4.6. Gestión de errores

De manera similar al registro de plugins, el desarrollador podrá registrar clases ErrorHandler con el objetivo de poder tratar las excepciones que se puedan producir al invocar las acciones. Para ello el usuario proporcionará clases que implementen la interfaz ErrorHandler y registrará las clases en el fichero de configuración, indicando el tipo de excepción que desea tratar, la clase que realizará el tratamiento y una clave de mensaje internacionalizado para poder internacionalizar el mensaje de error.

Figura 27. Diagrama de clases: gestión de errores



Se pueden declarar gestores de errores de forma global al framework y de forma local a un mapeo de acción.

```
<action name="/register" class="com.test.presentation.actions.RegisterAction">
  ...
  <error-handlers>
      <error-handler key="user.exists"
                exception="com.test.business.exceptions.UserAlreadyExistsException"
                class="com.test.presentation.errorHandlers.MyErrorHandler"/>
       ...
  </error-handlers>
</action>
```

```
<!-- Gestores de errores globales (opcional) -->
    <global-error-handlers>
        <error-handler
          key="default.error.msg"
          exception="java.lang.Exception"
          class="com.custom.UserCustomErrorHandler"/>
          ...
    </global-error-handlers>
```

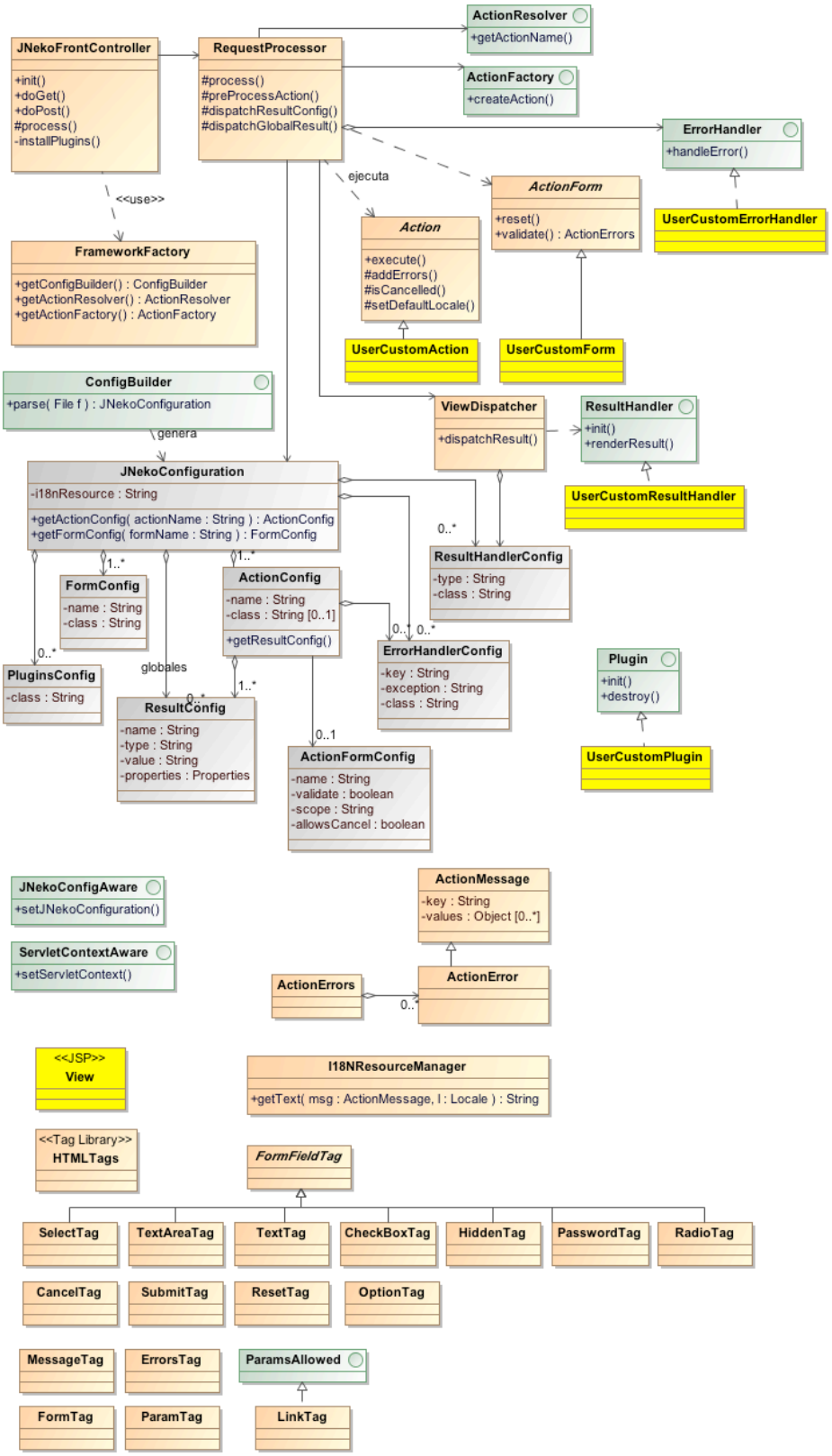
El RequestProcessor es el encargado de capturar las excepciones al ejecutar una acción y localizar mediante el ActionConfig el ErrorHandler que debe manejar la excepción. Se localiza primero en las declaraciones a nivel acción y después a nivel global. Si no se encuentra ningún ErrorHandler para una excepción determinada, se busca alguno que trate la superclase siguiente de la excepción así hasta encontrar alguno que coincida. Si no existe ninguno se lanzará una ServletException. Si lo encuentra llama a la operación handleError y finaliza el tratamiento de la petición. Es responsabilidad del ErrorHandler el generar una respuesta mediante el HttpServletResponse pasado por parámetro.

Si bien el contenedor servlet ya proporciona un mecanismo de gestión de excepciones, se recomienda utilizar el de jNeko ya que recibe información perteneciente al contexto del framework, como el formulario y acción que se estaban ejecutando cuando se produjo la excepción.

4.7. Diseño Final

El siguiente diagrama de clases muestra las clases proporcionadas por jNeko.

Figura 28. Diagrama de clases final



5. Aplicación de prueba

La aplicación de prueba es un pequeño prototipo que pretende mostrar el uso y funcionamiento de diversas características del framework de una forma muy concentrada. En este apartado vamos a ver una descripción general de la aplicación, los casos de uso, el diagrama de navegación de páginas y el diseño de la capa de presentación que se ha realizado.

En el anexo se encuentra una descripción más detallada de la aplicación de prueba además de las instrucciones de instalación.

5.1. Descripción de la aplicación de prueba

Se ha simulado un mini Portal encargado de publicar información de los videojuegos que han salido al mercado, otorgándoles una nota. Los usuarios pueden añadir comentarios a un videojuego con sus opiniones.

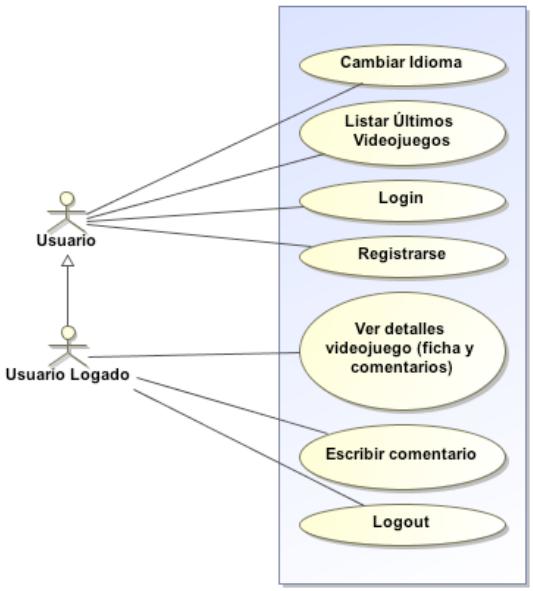
Para poder acceder al detalle de un videojuego, ver los comentarios publicados y añadir los propios se necesita estar registrado. Para permitir el acceso, la aplicación permite que los usuarios se registren y puedan acceder al sistema mediante sus credenciales.

La aplicación está disponible en castellano e inglés.

5.2. Casos de uso

En el siguiente diagrama de casos de uso muestra que todos los usuarios pueden ver la lista de videojuegos, cambiar el idioma, autenticarse o registrarse sin embargo para ver detalles del videojuego, listar los comentarios, escribir un comentario nuevo o deslogarse hay que estar registrado y autenticado en el portal.

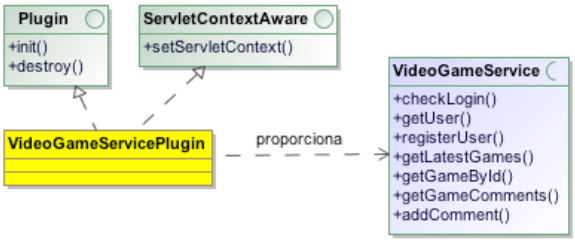
Figura 29. Casos de uso de aplicación de prueba



5.3. Diseño de la capa de presentación

5.3.1. Plugins

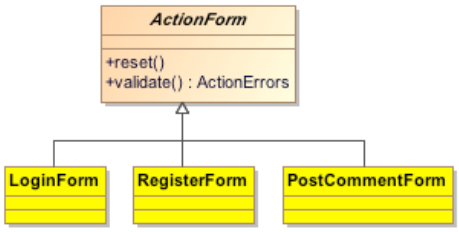
Figura 30. Plugin de la aplicación de prueba



Con el objetivo de demostrar el uso de plugins, se ha diseñado el plugin VideoGameServicePlugin que coloca una implementación de la interfaz de la capa de negocio VideoGameService en el contexto ServletContext de forma que pueda ser accedida por las acciones. En la figura se puede observar que VideoGameServicePlugin implementa la interfaz Plugin, también implementa ServletContextAware dado que necesita acceso al contexto del contenedor web. En su método init() instancia una implementación de VideoGameService y la establece en el contexto.

5.3.2. Formularios

Figura 31. Formularios de la aplicación de prueba



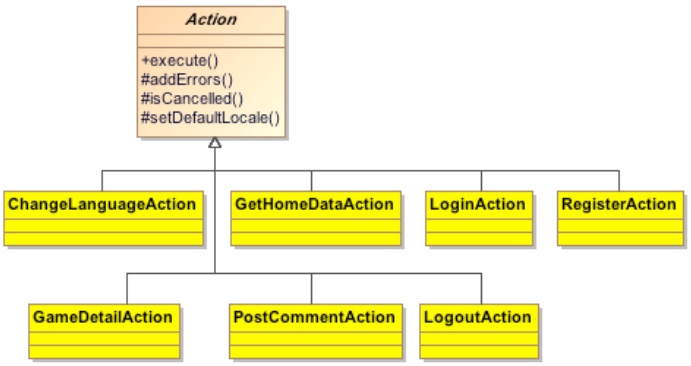
Se definen tres ActionForm para implementar los tres formularios necesarios para la aplicación.

Formulario	Descripción
LoginForm	Contiene los datos para poder acceder al sistema: usuario y contraseña.
RegisterForm	Obtiene los datos a mostrar en la página principal: obtiene el listado de los últimos videojuegos publicados y los pone accesibles en un atributo del request.
PostCommentForm	Realiza el login del usuario. Utiliza el formulario LoginForm.

5.3.3. Acciones

En la siguiente figura se muestran las acciones Action definidas en la aplicación de prueba.

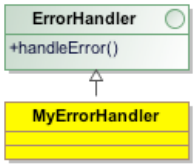
Figura 32. Acciones de la aplicación de prueba



Acción	Descripción
ChangeLanguageAction	Cambia el idioma de la aplicación. Utiliza el método setDefaultLocale heredado.
GetHomeDataAction	Obtiene los datos a mostrar en la página principal: obtiene el listado de los últimos videojuegos publicados y los pone accesibles en un atributo del request.
LoginAction	Realiza el login del usuario. Utiliza el formulario LoginForm.
RegisterAction	Registra los datos de un usuario en el sistema para que pueda acceder. Utiliza el formulario RegisterForm.
GameDetailAction	Obtiene los datos de un videojuego así como los comentarios y los pone accesibles en el request.
PostCommentAction	Añade el comentario de un usuario registrado en un videojuego.
LogoutAction	Finaliza la sesión del usuario logado.

5.3.4. Gestión de errores

Figura 34. Gestor de errores de la aplicación de prueba



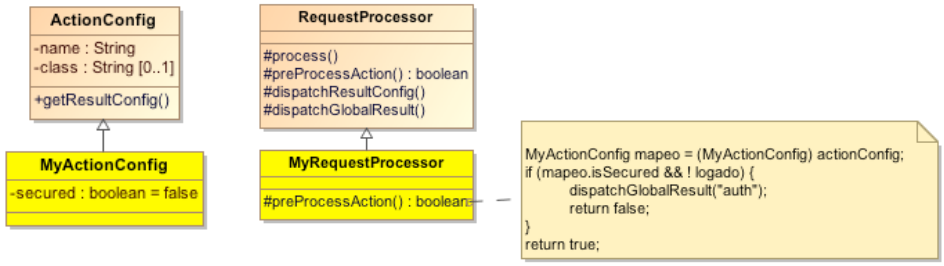
Se crea el gestor de errores MyErrorHandler que captura las excepciones que se puedan producir en las acciones. Esta clase es un ErrorHandler que coloca la información de la excepción en el request y sirve un resultado global mapeado con nombre “exception” que terminará en una vista que informará del error producido.

5.3.5. Seguridad

En la aplicación de prueba, hay ciertas acciones que requieren que el usuario esté logado para poder ejecutarlas y sino se informa que no está autorizado. A estas acciones las llamo acciones protegidas.

Una de las posibles alternativas para controlar el acceso habría sido que cada acción protegida se encargase de comprobar el acceso y permitiese ejecutar la acción o no, pero de esta forma el código habría quedado disperso por las acciones y no sería fácil de mantener ya que habría que insertar o quitar el código cuando se quisiera proteger o desproteger una acción.

Figura 35. Extensión de ActionConfig y RequestProcessor



Para hacerlo de una forma centralizada y baja cohesión, se personalizan los mapeos de las acciones para poder indicar si el mapeo está protegido o no y se realiza la comprobación del acceso en un RequestProcessor personalizado antes de ejecutar las acciones.

Para realizar esto por un lado se marcan las acciones como protegidas o no en el mapeo de acciones extendiendo el ActionConfig por defecto mediante la clase MyActionConfig para añadir un nuevo atributo secured que por defecto será falso. En el archivo de configuración del framework se establecerá a cierto la propiedad en los mapeos protegidos.

```
<action name="/showGame" class="GameDetailAction">
     <set-property property="secured" value="true"/>
     <result>/gameDetailsPage.jsp</result>
</action>
```

Por otra parte se extiende el RequestProcessor y se sobreescribe el método preProcessAction. Este método recibe por parámetro el ActionConfig que se va a tratar y que se sabe que será un MyActionConfig, de forma que se puede determinar si el mapeo es seguro o no para permitir continuar o no dependiendo de si el usuario está logado. Si no hay acceso, se sirve un resultado global que muestra la vista de que no hay permisos y se retorna falso para indicar que no se siga con la ejecución de la petición.

En el archivo de configuración de jNeko se le informa al framework que utilice MyActionConfig en vez de la implementación por defecto de mapeo. También se informa que se utiliza MyRequestProcessor en vez del RequestProcessor por defecto.

De esta manera, el mantenimiento de proteger o desproteger una acción es tan sencillo como establecer a cierto o falso la propiedad secured en el archivo de configuración y si hay que modificar el código se tiene centralizado en un único punto de la aplicación, en MyRequestProcessor.

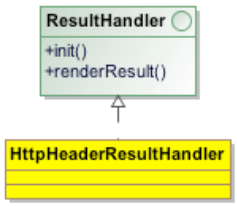
5.3.6. Resultados personalizados

Aunque nada tiene que ver con la aplicación de videojuegos, a modo de ejemplo del uso del framework se ha creado un tipo de resultado nuevo "httpheader" que genera una respuesta HTTP con el status que se indique en un parámetro status dentro del resultado.

```
<!--  Demo de resultado tipo httpheader -->
<action name="/testHttpServiceUnavailable">
     <result type="httpheader">
           <param name="status">503</param>
     </result>
</action>
```

Esta funcionalidad la proporciona HttpHeaderResultHandler que genera la respuesta a partir de los parámetros recibidos en el método renderResult. La demostración de esta funcionalidad se presentará una vista VistaOtrasDemos.

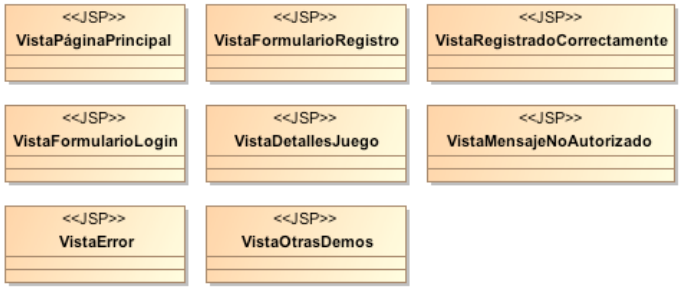
Figura 36. Renderizador de nuevo tipo de resultado "httpheader"



5.3.7. Vistas

Por ultimo, las vistas de la aplicación de prueba son JSP's encargadas de visualizar los datos. Utilizan la librería de etiquetas proporcionada por jNeko para crear enlaces, internacionalizar los textos, definir formularios, informar de errores, etc.

Figura 37. Vistas de la aplicación de prueba



6. Implementación del framework

En este apartado se explica las características del framework que se ha implementado, algunos detalles de la implementación que me han parecido importante destacar, un resumen de los patrones de diseño utilizados y por último las dificultades encontradas durante la construcción.

6.1. Características implementadas

- Mapeo declarativo de path, acciones, resultados
- Utilización de convenciones para evitar tener que escribir demasiada configuración. Esto se consigue con atributos y elementos en el xml que si no están presentes implican valores por defecto. Por ejemplo:
  - Un elemento resultado <result> sin atributo name es equivalente al resultado por defecto <result name="success">
  - Un elemento formulario <form> dentro de una acción sin atributo scope declarado, se considera que es scope = "session". Si no se declara el atributo validate significa que la validación está activada por defecto.
- Las acciones simplemente ejecutan su lógica y retornan un String con el nombre del próximo resultado que hay que servir.
- Resultados declarados a nivel acción o a nivel global. Si en un mapeo no se encuentra un resultado con un nombre determinado, se busca en los resultados declarados a nivel global.
- Los resultados permiten diferentes tipos de vistas, por defecto son páginas jsp (tipo "dispatch"). El framework proporciona también el tipo de resultado "redirect" que permite redireccionar a una URL.
- El desarrollador puede definir y añadir sus propios tipos de resultado y utilizarlos en el archivo de configuración
- Posibilidad de declarar Plugins que permiten ejecutar lógica al inicializarse el framework y al terminarse. Además se proporciona acceso al contexto y a la representación en memoria de la configuración por si se desea consultarla o modificarla.
- Se proporcionan unas interfaces "aware" que permiten a los Plugins y a los renderizadores de resultados ResultHandlers acceso al contexto y a la configuración del framework mediante inyección de dependencias, de forma que en el futuro se pueden ir añadiendo más interfaces de este tipo para proporcionar más datos sin tener que añadir nuevos métodos y romper a los clientes existentes.
- El RequestProcessor se puede extender, reemplazando al que viene por defecto en el framework. De forma que se puede cambiar el funcionamiento interno.
- El RequestProcessor proporciona un método preProcessAction que se puede implementar por una subclase y permite ejecutar código antes de empezar a

ejecutar las peticiones así como la posibilidad de cancelar el procesamiento de la petición.

- Los mapeos ActionConfig pueden extenderse y declarar nuevas propiedades a las que se puede dar valor desde el archivo de configuración
- Proporciona formularios que se informan a partir de los datos de la petición y permiten realizar validaciones de los datos de entrada
- Los formularios permiten la cancelación, evitando que se ejecute la validación y pasando a la acción directamente que podrá determinar si se ha cancelado para poder mostrar un resultado u otro.
- Validaciones a nivel de formulario y a nivel de acción (de forma programática). Los errores de validación pueden ir asociados a las propiedades del formulario o nivel general.
- Se pueden declarar gestores de errores para capturar y tratar excepciones que se producen al ejecutar las acciones. Se pueden declarar a nivel de acción o a nivel global. Se pueden asociar a superclases de excepciones a tratar.
- Las acciones y los gestores de errores retornan una cadena con el nombre de resultado a servir. También pueden generar una respuesta por si mismos y retornar el valor null para indicar que ya no hace falta servir un resultado.
- Las aplicaciones se pueden internacionalizar. Los mensajes de errores de validación, los gestores de excepciones, textos y controles en pantalla permiten la internacionalización. Hay que crear y declarar un archivo de texto tipo ResourceBoundle por cada idioma o locale en que se quiera internacionalizar la aplicación.
- Se proporciona una librería de etiquetas para utilizar en las vistas:
  - link: para poder crear enlaces a acciones. Permiten parámetros para concatenar a la url formada.
  - msg: para poder mostrar texto en diferentes idiomas
  - errors: para poder mostrar los errores de validación que se hayan producido
  - form: para definir un formulario html <form>. Los formularios se acceden desde la vista, mediante las etiquetas proporcionadas, de forma que la vista puede por ejemplo mostrar los valores asignados al mismo.
    - Los formularios permiten etiquetas para definir controles asociados a las propiedades de los formularios: text, password, submit, textarea, hidden, reset, checkbox, radio, select y cancel
- Se puede configurar establecer el Locale en sesión a todas las peticiones. También se proporciona un método en las acciones para poderlo establecer desde las mismas.
- Se puede configurar forzar un contentType a todas las respuestas
- Se puede configurar el poner cabeceras de nocache a todas las respuestas

6.2. Detalles de implementación

- Configuración del framework declarativa mediante archivo xml
- Se ha creado un XML Schema para poder validar el archivo de configuración. Esta decisión ofrece varias ventajas:
  - Evita codificar mucha lógica de validación ya que se encarga el parseador de comprobar la validez. (Aunque se ha empleado bastante tiempo en crear el schema).
  - Proporciona información detallada en caso de errores de validación, informando donde se producen y la causa
  - Permite al desarrollador ver información como valores válidos, orden y cardinalidad de los elementos que se pueden presentar
  - Permite al desarrollador validar el archivo de configuración sin tener que arrancar el framework
- Validación y parseo empleando los estándares DOM y XPath utilizando Java 6 sin depender de librerías externas
- Se han creado múltiples excepciones de los posibles errores producidos en tiempo de ejecución al inicializar el framework o al servir peticiones. El nivel de detalle y granularidad es fino de forma que, aunque finalmente se enmascaren dentro de una ServletException, se puede ver el detalle de los errores en la causa raíz dando información útil al desarrollador del problema. Además permite internamente al framework poder decidir qué hacer al tener más detalle de los errores.
- Posibilidad de tracear el funcionamiento del framework gracias a commons-loggin
- Las clases e interfaces están bien encapsuladas mediante los modificadores de acceso private y package private para evitar exponer detalles de implementación al usuario del framework y permitir la refactorización cuando sea necesario.

6.3. Resumen de patrones de diseño utilizados

Se han utilizado los patrones de diseño **Front Controller** (Alur, 2003), **Application Controller** (Alur, 2003), **Service To Worker** (Alur, 2003) y **ViewHelper** (Alur, 2003)

Además se han utilizado los siguientes patrones:

- **Singleton** (Gamma,1995): Por ejemplo en la clase FrameworkFactory para asegurarnos que sólo hay una instancia
- **Strategy** (Gamma,1995): Se ha utilizado por ejemplo para encapsular la lógica de instanciación de acciones Action. En un principio temprano de desarrollo se utilizó la estrategia NewInstanceActionFactory que instanciaba una nueva acción por

petición y finalmente se cambió por CachedActionFactory que reutiliza las instancias. Si bien no se va a cambiar la implementación de una u otra estrategia durante la ejecución del framework, se han dejado ambas estrategias internamente por si se quisiera en el futuro poder configurar este comportamiento.

- **Template method** (Gamma,1995): El RequestProcessor tiene un algoritmo que llama al método preProcessAction para poder decidir si se sigue procesando la petición o no. Aunque proporciona una implementación por defecto, es un template Method que permite a las subclases implementar este paso.
- **Null Object**: Los forwards en jNeko se han implementado como un mapeo de acción pero sin acción declarada, de forma que siempre hay que retornar el resultado “success”. Esto en el código quedaba implementado con condicionales mirando si la acción es null y escribiendo código para esta casuística. En vez de tener que estar usando casos especiales, se ha implementado el objeto nulo NullAction que evita tener que escribir código para este caso especial.
- En la aplicación de demo, para realizar la lógica de negocio se ha utilizado un **Domain Model** (Fowler, 2003) para representar el modelo del dominio (usuarios, videojuegos y comentarios). También se ha utilizado **Service Layer** (Fowler, 2003) para definir el acceso a la capa de negocio, y se han utilizado **Data Transfer Object** (Fowler, 2003) para transferir datos entre la capa de negocio y la capa de presentación

6.4. Dificultades encontradas

La falta de experiencia con frameworks de este tipo sobre todo en las vistas me han hecho tener que emplear un tiempo considerable a ver cómo se implementa alguna funcionalidad. A modo de ejemplo: nunca he programado un *custom tag*, por lo que me ha costado bastante implementar la etiqueta <form> y controles asociados ya que no tenía muy claro cómo debían comunicarse con el formulario ni tenía bien claro la funcionalidad que debían proporcionar.

Otra dificultad a la que me enfrenté es la construcción de enlaces a acciones desde las vistas <link> que quería que funcionasen independientemente de si se utilizaba o no la extensión *.action* en el enlace y la posibilidad de asociarles parámetros mediante etiquetas <param> que debían comunicarse con la etiqueta padre <link> para ir concatenando parámetros.

He tenido que realizar pequeñas refactorizaciones mientras construía el código. Por ejemplo en la lógica interna del RequestProcessor quería dar la posibilidad de que las acciones sólo tuvieran que retornar una cadena con el nombre de resultado a servir. Pero me di cuenta que quizás una acción quiere generar una respuesta directamente, por ejemplo para generar una imagen dinámica. Aunque se pueden definir diferentes tipos de resultado, decidí que pudiesen de forma sencilla retornar null para indicar que la acción se encarga de generar la respuesta, lo que me cambió buena parte de la lógica del RequestProcessor para adaptarlo a este caso, teniendo en cuenta que está lleno de pequeños métodos que un usuario puede extender y sobreescribir para poder modificar el comportamiento.

Utilicé también un tiempo considerable a crear el XMLSchema de la configuración, pero luego fue de gran ventaja para ir añadiendo características al fichero de configuración, ya que simplificó la programación.

7. Conclusiones

En el proyecto se ha realizado un estudio completo del dominio que ha permitido diseñar e implementar un framework que cubre las necesidades de la capa de presentación de una aplicación de forma satisfactoria. Gracias al uso de patrones de diseño, se ha conseguido una arquitectura bien diseñada con características deseables: el código de control se encuentra localizado y centralizado en un único punto; las etiquetas permiten evitar código en las vistas; las clases e interfaces extensibles proporcionadas, como por ejemplo las acciones y formularios, así como la configuración declarativa proporciona gran flexibilidad y permiten modificar fácilmente el comportamiento.

Además, se ha diseñado e implementado una aplicación de prueba que ha demostrado el buen funcionamiento del framework así como la sencillez y rapidez con la que se puede desarrollar utilizando el mismo, eso sí, a costa de tener que aprender al principio el funcionamiento del framework. Esta aplicación muestra el uso de todas y cada una de las características proporcionadas por el framework.

Por último, comentar que el tiempo proporcionado para implementar y documentar el framework no da mucho de sí para poder investigar e implementar muchas de las características que me gustaría poder haber añadido. Es por eso por lo que me he centrado en proporcionar un framework sólido, que funciona correctamente, que cubre las necesidades de la capa de presentación, que se puede extender fácilmente además de estar diseñado de manera que se permita evolucionar para añadir más características en el futuro sin que sea complejo adaptarlo.

Glosario

- **JavaBean**: modelo de componentes creado por Sun Microsystems para la construcción de aplicaciones en Java.
- **Ciclo de vida**: conjunto de etapas de un objeto o instancia.
- **Command**: patrón de diseño que permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación.
- **Custom tag**: etiquetas a medida que se pueden construir para utilizarse en páginas JSP.
- **Getter**: nombre empleado para referirse a un método accesor de una propiedad en la convención Java Beans, denominado así porque empiezan con el prefijo “get”.
- **Java EE**: nombre que recibe J2EE a partir de la versión 1.4
- **J2EE**: plataforma de programación para desarrollar y ejecutar software de aplicaciones en lenguaje de programación Java con arquitectura de N capas distribuidas.
- **jNeko**: nombre del framework desarrollado en este proyecto.
- **JSP**: tecnología Java que permite generar contenido dinámico para web.
- **Modelo Vista Controlador o MVC**: patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.
- **MVC1 o MVC Model 1**: aproximación de la primera generación de arquitectura en la capa de presentación dentro de J2EE donde la funcionalidad de controlador y vista se realizaban dentro de una JSP con lo que se rompe el paradigma MVC.
- **MVC2 o MVC Model 2**: término inventado por Sun Microystems para describir una arquitectura web basado en peticiones HTTP en el que se pasa la petición de un cliente a un Servlet controlador que actualiza el modelo e invoca a una vista apropiada.
- **Plugin**: conjunto de componentes software que añaden características específicas a un software existente.
- **Request**: petición HttpServletRequest.
- **Response**: respuesta HttpServletResponse.
- **Setter**: nombre empleado para referirse a un método modificador de una propiedad en la convención Java Beans, denominado así porque empiezan con el prefijo “set”.
- **Servlet**: es una clase Java en Java EE que se ajusta a la API Java Servlet, un protocolo por el cual una clase Java puede responder a las peticiones HTTP.
- **Stub**: código utilizado como sustituto de funcionalidad real con el objetivo de realizar pruebas.

Bibliografía

Alur, Deepak; Crupi, John; Malks, Dan (2003). *Core J2EE Patterns: Best Practices and Design Strategies (2.ª ed.).* Palo Alto: Prentice Hall.

Apache Software Foundation. *Struts Framework.* [en línea]. [Fecha de consulta: 28 de diciembre de 2010].
<http://struts.apache.org/1.x/index.html>

Apache Software Foundation. *Struts2 Framework*. [en línea]. [Fecha de consulta: 28 de diciembre de 2010].
<http://struts.apache.org/2.x/index.html>

Bloch, Josua (2008). *Effective Java (2ª ed.).* Boston: Addison - Wesley.

Booch, G; Jacobson, I; Rumbaugh, J (2000). *El lenguaje unificado de modelado. Manual de referencia.* Madrid: Addison - Wesley.

Basham, Bryan; Bates, Bert; Sierra, Kathy (2004). *Head First Servlets & JSP: Passing the Sun Certified Web Component Developer Exam.* Sebastopol: O'Reilly.

Conrad Cunningham, H; Liu, Y; Zhang ,C (2006). "Using Classic Problems to Teach Java Framework Design"*. Science of Computer Programming - Special issue: Principles and practices of programming in Java* (Vol. 59, núm 1-2, págs. 147-169) [en línea]. [Fecha de consulta: 28 de diciembre de 2010].
<http://portal.acm.org/citation.cfm?id=1131905>

Evans, Ian; Gollapudim Devika; Haase, Kim; Jendrock, Eric; Srivathsa, Chinmayee (2009). "Java Server Faces Technology". *The Java EE 6 Tutorial.* [Tutorial en línea]. [Fecha de consulta: 28 de diciembre de 2010].
<http://download.oracle.com/javaee/6/tutorial/doc/>

Fowler, Martin (2003). *Patterns of Enterprise Application Architecture.* Indianápolis: Addison - Wesley.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Indianápolis: Addison - Wesley.

Goodwill, James (2002). *Mastering Jakarta Struts.* Indianápolis: Wiley Publishing Inc.

Johnson, Mark; Singh, Inderjeet; Stearns, Beth (2002). *Designing Enterprise Applications with the J2EE Platform (2ª ed.).* Palo Alto: Addison - Wesley.

Kurniawan, Budi (2008). *Struts2 Design and Programming: A Tutorial (2ª ed.).* Markham: BrainySoftware.

Roughley, Ian (2006). *Starting Struts2.* [Libro en línea]. [Fecha de consulta: 28 de diciembre de 2010].
<http://www.infoq.com/minibooks/starting-struts2>

Wikipedia. "Comparison of web Application Frameworks". Wikipedia, the free encyclopedia. [Artículo en línea].[Fecha de consulta: 28 de diciembre de 2010].
<http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks>

Anexos

A. Instalación / Ejecución del framework jNeko

A.1. Dependencias

El framework jNeko depende de dos librerías externas que se detallan a continuación:

Nombre	Commons BeanUtils
Descripción	Librería de utilidad para poder manejar propiedades de los beans y evitar tener que utilizar reflection directamente
Versión utilizada	1.8.3
Nombre del archivo	commons-beanutils-1.8.3.jar
URL de descarga	http://commons.apache.org/beanutils/

Nombre	Commons Logging
Descripción	Librería que proporciona uniformidad entre diversas implementaciones de herramientas de log
Versión utilizada	1.1.1
Nombre del archivo	commons-logging-1.1.1.jar
URL de descarga	http://commons.apache.org/logging/

Se ha utilizado funcionalidad que sólo se proporciona en la versión de java 6, por lo que es necesario esta versión para su funcionamiento. La versión utilizada para compilar y ejecutar ha sido la **versión 6 update 22**. La versión de update es importante ya que se utilizan librerías internas actualizadas para parsear el xml de configuración.

La versión de especificaciones del contenedor web utilizadas son **Servlet 2.5** y J**SP 2.1**

El contenedor web que se ha utilizado para la compilación y donde se ha probado la ejecución es **Tomcat versión 6.0.29** que se puede descargar de http://tomcat.apache.org/

A.2. Creación y despliegue de una aplicación con jNeko

Para crear / desplegar una aplicación con jNeko hay que realizar estos cuatro pasos:

a) Poner las librerías del framework accesibles a la aplicación a desarrollar / contenedor web. Éstas librerías son jneko.jar , commons-beanutils-1.8.3.jar y commons-logging-1.1.1.jar que se recomienda colocar en el directorio WEB-INF/lib de la aplicación a desarrollar.

b) Extender las clases e interfaces del framework, así como proporcionar un archivo de propiedades por cada idioma en el que se quieran internacionalizar textos.

c) Crear el archivo de configuración del framework y configurarlo adecuadamente.

   El archivo debe llamarse *jneko-config.xml* y debe ponerse dentro del directorio WEB-INF de la aplicación. Se puede partir del archivo *jneko-config.xml* que viene incluido en la aplicación de prueba.

   Para facilitar su edición y validación [3], se ha creado un XML Schema que puede ser utilizado para validar el archivo de configuración. El XML Schema se encuentra en el archivo *jneko_1_0.xsd* dentro del directorio de código fuente jneko/source/org/jneko/schema/ y se proporciona una copia en el directorio de distribución jneko/dist. Este schema se utiliza internamente en el framework para validar y parsear la configuración, pero se puede utilizar por el desarrollador para poder validar en cualquier momento mientras construye su aplicación.

d) Por último configurar el contenedor de servlets para que el Front Controller de jNeko se encargue de inicializar el framework y de centralizar y servir las peticiones.

   Para ello hay que editar el archivo de descripción de despliegue WEB-INF/web.xml y declarar el servlet JNekoFrontController para que se inicialice y configure el framework al iniciarse el contenedor y que sirva todas las url's acabadas en .action tal como muestra el siguiente fragmento:

```
<!--  Configuracion de framework jNeko -->
<servlet>
   <description>Controlador de Framework jNeko</description>
   <servlet-name>jnekoController</servlet-name>
   <servlet-class>org.jneko.action.JNekoFrontController</servlet-class>
   <load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>jnekoController</servlet-name>
   <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

[3] Algunos entornos integrados de desarrollo como Eclipse proporcionan edición asistida de los xml, proporcionando validación y autocompletado mientras se editan si éstos van asociados a un XML Schema (referenciado por el atributo schemaLocation). Si se copia el archivo jneko_1_0.xsd en el directorio WEB-INF junto al archivo jneko-config.xml se dispondrá de edición asistida. La mejor opción y la que se utiliza normalmente sería publicar el schema jneko_1_0.xsd en una URL pública estable, pero esta tarea queda fuera del ámbito del proyecto.

A.3. Configuración y uso del framework jNeko

En este apartado se explica de forma práctica y concisa cómo se configura el framework y qué debe proporcionar el usuario que lo utiliza. La mayor parte de opciones de configuración han sido detalladas durante el cuerpo del proyecto por lo que se comentarán brevemente.

Fichero de configuración jneko-config.xml

Para configurar el framework se utiliza un fichero de configuración que se debe llamar jneko-config.xml y que debe colocarse dentro del directorio de despliegue WEB-INF/. En la siguiente figura se presenta el formato del archivo de configuración y las secciones más importantes del xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<jneko-framework xmlns="http://www.jneko.org/jnekoSchema"
1  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.jneko.org/jnekoSchema jneko_1_0.xsd">

  <config-options>
2   ...
  </config-options>

3 <i18n-resource> ... </i18n-resource>

  <reg-plugins>
4   ...
  </reg-plugins>

  <reg-result-handlers>
5   ...
  </reg-result-handlers>

  <forms>
6 ...
  </forms>

  <navigation>
7 ...
  </navigation>

  <global-error-handlers>
8 ...
  </global-error-handlers>

</jneko-framework>
```

La siguiente tabla enumera los elementos principales que se encuentran en la configuración indicando para qué se utilizan.

Núm.	Elemento	Descripción
1	jneko-framework	Elemento raíz que engloba toda la configuración del framework
2	config-options	Opciones de configuración.

Núm.	Elemento	Descripción
3	i18n-resource	Sirve para indicar el recurso que contiene los textos de internacionalización
4	reg-plugins	Declaración de los plugins a instalar al inicializarse el framework
5	reg-result-handlers	Declaración de clases ErrorHandler para tratar tipos creados por el usuario.
6	forms	Declaración de los formularios ActionForm a utilizar en la aplicación
7	navigation	Para declarar el flujo de navegación. Se definen resultados globales y mapeos de peticiones, acciones, resultados.
8	global-error-handlers	Declaración de gestores de errores a nivel global

Sustituir el RequestProcessor por defecto (opcional)

Si se quiere sustituir el RequestProcessor por defecto por otro, hay que crear una clase que extienda de org.jneko.action.RequestProcessor y declarar el nombre completo de la clase dentro del atributo class en el elemento request-processor de la siguiente manera:

```
<config-options>
  ...
  <request-processor class="com.paquete.MyRequestProcessor"/>
  ...
</config-options>
```

Uno de los métodos proporcionados por el RequestProcessor y que puede ser interesante sobreescribir es el método preProcessAction.

```
protected boolean preProcessAction(ActionConfig actionConfig,
      javax.servlet.http.HttpServletRequest request,
      javax.servlet.http.HttpServletResponse response)
```

Este método se llama antes de ejecutar cualquier mapeo de acción. La acción mapeada a ejecutar se recibe como parámetro. Si se sobreescribe el método y se retorna cierto se continúa con la ejecución, si se retorna falso se termina el procesamiento de la petición.

Sustituir el ActionConfig por defecto (opcional)

Para poder sustituir la clase ActionConfig por defecto se debe crear una clase que extienda de org.jneko.config.ActionConfig y declarar la clase mediante el atributo class dentro del elemento action-config:

```
<config-options>
  ...
  <actionConfig class="com.paquete.MyActionConfig"/>
  ...
</config-options>
```

La nueva clase puede declarar propiedades que pueden ser informadas desde los mapeos en la configuración mediante el elemento set-property, así por ejemplo si se declarasen en MyActionConfig dos atributos “saludo” y “activarDebug”, sus valores pueden ser informados con el siguiente fragmento que provocaría la llamada a los métodos setSaludo y setActivarDebug.

```
<action ...>
  <set-property property="saludos" value="Hola"/>
  <set-property property="activarDebug" value="true"/>
</action>
```

Cabe decir que en los métodos de las diferentes interfaces y objetos que se pueden extender en el framework y que reciban un ActionConfig, podrán realizar un cast a MyActionConfig para acceder a estas propiedades.

Establecer contentType por defecto a las respuestas (opcional)

Se puede establecer un contentType fijo a todas las respuestas cuando se sirve una petición. Por defecto no se establece ningún contentType. Por ejemplo para establecer siempre el contentType text/xml se declararía de la siguiente manera:

```
<config-options>
  ...
  <contentType>text/html</contentType>
  ...
</config-options>
```

Insertar cabeceras en las respuestas para que no se cacheen (opcional)

Si se desea que los resultados no se cacheen, se pueden establecer cabeceras en las respuestas para indicar que el cliente no las cache. Si no se desea cachear, se declara de la siguiente forma:

```
<config-options>
  ...
   <nocache>true</nocache>
  ...
</config-options>
```

Si no se indica nada en el archivo de configuración tiene el mismo efecto que nocache a falso, por lo que los resultados pueden ser cacheados.

Declaración de recursos de internacionalización

Para la traducción de textos hay que proporcionar un archivo con extensión .properties por cada idioma / región en el que se quiera proporcionar traducciones y hacerlo accesible en el classpath. El formato de este archivo debe seguir el formato de los ResourceBoundles de la internacionalización de Java y debe declararse mediante el elemento de la siguiente manera:

```
<i18n-resource>com.test.presentation.AppMessages</i18n-resource>
```

En este ejemplo debería encontrarse en el classpath un archivo AppMessages.properties con la traducción de claves a textos traducidos en el lenguaje por defecto y cero o varios archivos AppMessages_<idioma>_<país>.properties por cada idioma o región en el que se quiera proporcionar traducción.

Plugins

Los plugins deben ser clases que implementen la interfaz org.jneko.plugin.Plugin y deben declararse en el atributo class del elemento plugin dentro de el elemento reg-plugins.

```
<reg-plugins>
   <plugin class="com.plugin.CustomPlugin1" />
   <plugin class="com.plugin.CustomPlugin2" />
   ...
</reg-plugins>
```

Formularios

Un formulario debe extender e implementar la clase abstracta org.jneko.action.ActionForm. El formulario debe seguir las convenciones de JavaBeans, es decir declarar métodos getter y setter que recuperan o establecen valores de las propiedades.

El formulario debe inicializar sus valores en el método reset y realizar validaciones en el método validate. Si se producen errores de validación hay que retornar un objeto ActionErrors con los errores producidos.

Los formularios se declaran en el archivo de configuración mediante el elemento form dentro del elemento reg-forms. Por ejemplo:

```
<forms>
   <form name="loginForm" class="com.forms.LoginForm"/>
   <form name="registerForm" class="com.forms.RegisterForm"/>
   ...
</forms>
```

El nombre de la clase que implementa el formulario se especifica en el atributo class. En el atributo name se debe asignar un nombre lógico único al formulario para utilizarlo en el resto de la configuración.

Acciones

Las acciones deben extender la clase org.jneko.action.Action e implementar el método:

```
String execute(ActionConfig actionConfig, ActionForm form,
javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response)
```

Este método debe retornar el nombre de resultado lógico que se debe servir después de la ejecución. Existe la opción de generar la respuesta en este método, en ese caso se puede retornar null que indica que no hay que servir ningún resultado.

El método recibe el mapeo ActionConfig que contiene la información de mapeo de la acción que se está sirviendo y la instancia de formulario ActionForm que se utiliza en este mapeo. En caso de que no se haya declarado formulario, tiene el valor null.

Resultados

Un resultado tiene un nombre lógico y un tipo. También tiene un valor y opcionalmente parámetros que son pares de <nombre,valor>

Un resultado se declara de la siguiente manera y aparecen en un mapeo de acción o en los resultados globales:

```
<result name="nombre" type="tipo">
   valor
   <param name="nombreparam">valorparam</param>
   <param ...>
</result>
```

El nombre por defecto de un resultado si no se especifica el nombre es "success" y el tipo por defecto de un resultado si no se especifica el tipo es "dispatch". El valor de la etiqueta result es todo el contenido de texto que aparece en su cuerpo sin tener en cuenta los elementos param.

jNeko viene de serie con dos tipos que son *dispatch* y *redirect*. Si se quieren servir jsp's se utiliza dispatch y se pone el nombre de la jsp en el valor. Por ejemplo:

```
<result name="success" type="dispatch">/exito.jsp</result>
<result name="error" type="dispatch">/informeError.jsp</result>
```

Nótese que si se utilizan las convenciones los siguientes resultados son equivalentes a los del ejemplo anterior:

```
<result>/exito.jsp</result>
<result name="error">/informeError.jsp</result>
```

El otro tipo de resultado que proporciona jNeko es redirect que permite redireccionar a una URL. Por ejemplo :

```
<result name="extSearch" type="redirect">http://www.google.com</result>
```

Mapear acciones

Mapear una acción en jNeko consiste en asociar el path de una petición a una acción, declarando un formulario opcional y uno o varios resultados.

Para declarar un mapeo se utiliza el elemento action dentro de la sección navigation del xml de configuración.

```
<navigation>

  <action name="/security/login" class="com.custom.LoginAction">
     <form validate="true" scope=”request”>formLogin</form>
     <result name="success">/jsp/main.jsp</result>
     <result name="error">/jsp/error.jsp</result>
     <result name=”input”>/jsp/formLogin.jsp"</result>
  </action>

</navigation>
```

El mapeo de este ejemplo asocia la clase com.custom.LoginAction a una petición con path /security/login.action.

Además este mapeo declara mediante el elemento form que debe utilizarse un formulario formLogin que debe recuperarse / establecerse en el ámbito del request y que debe validarse. El valor por defecto del atributo validate si no se declara es “true” y el de scope es “session”. En el cuerpo del elemento form se indica el nombre lógico del formulario que se quiere utilizar.

Por último el mapeo declara tres resultados distintos. Dado que se ha declarado utilizar un formulario, es obligatorio que uno de los resultados tenga nombre “input” que es el resultado que se servirá en caso de que el formulario tenga errores de validación.

Por último destacar que se pueden construir mapeos para servir un resultado directamente. La forma de hacerlo es no declarando el atributo class y declarando un resultado “success” que es el que se servirá en el mapeo. Por ejemplo:

```
<action name="/otherDemos">
     <result>/otherDemos.jsp</result>
</action>
```

En este ejemplo no se ejecuta ninguna acción y se servirá el resultado “success” que en este caso es un dispatch de la página otherDemos.jsp.

Este mecanismo permite poder enlazar unas vistas con otras pasando por el controlador en vez de que se enlacen directamente unas vistas a otras. De esta forma se puede mantener los enlaces de forma declarativa y localizada en un punto central.

Crear tipos de resultado

Para crear nuevos tipos de resultados hay que implementar la interfaz ResultHandler y declarar la nueva clase dentro de el elemento <reg-result-handlers> especificando el nombre del tipo al que se asociará el ResultHandler.

El siguiente ejemplo declara un nuevo tipo de resultado “captcha” y declara que para servir este resultado se encargará la clase com.result.CaptchaResultHandler.

```
<reg-result-handlers>
  <result-handler type="captcha" class="com.result.CaptchaResultHandler" />
  ...
</reg-result-handlers>
```

Librería de etiquetas

Para poder utilizar la librería de etiquetas que proporciona jNeko desde una jsp, se debe declarar en la jsp mediante la directiva:

```
<%@ taglib prefix="html" uri="http://www.jneko.org/tags/html"%>
```

Las etiquetas proporcionadas por jNeko son:

Etiqueta	Descripción
cancel	Genera un botón de cancelación de formulario.
checkbox	Genera un control de formulario de tipo checkbox.
errors	Genera una lista con los mensajes de error que se encuentran en la petición.
form	Se encarga de renderizar un formulario Html
hidden	Genera un control de formulario de tipo hidden.
link	Genera un enlace a una acción / mapeo.
msg	Muestra un mensaje de texto internacionalizado.
option	Genera una opción para un control select.
param	Permite añadir un parámetro al enlace generado por la etiqueta link.
password	Genera un control de formulario de tipo password.
radio	Genera un control de formulario de tipo radio.
reset	Genera un boton de reset de formulario.
select	Genera una seleccion <select> donde se puede escoger una opcion de entre multiples.
submit	Genera un boton de submit de formulario.
textarea	Genera un control de formulario de tipo textarea.
text	Genera un control de formulario de tipo text.

B. Manual Aplicación de Prueba: manual instalación y de usuario

B.1. Dependencias

La aplicación de demostración necesita incluir en su directorio WEB-INF/lib las librerías para utilizar el framework:

- jneko.jar
- commons-beanutils-1.8.3.jar
- commons-logging-1.1.1.jar

Además utiliza las siguientes librerías:

Nombre	JSP Standard Tag Libs
Descripción	Librería de etiquetas estándar, (definición e implementación de Jakarta) utilizadas por las vistas de la demo por ejemplo para iterar sobre colecciones
Versión utilizada	1.1.2
Nombre del archivo	jstl.jar y standard.jar
URL de descarga	http://jakarta.apache.org/site/downloads/downloads_taglibs-standard.cgi

Nombre	Log4j
Descripción	Framework para poder generar trazas, muy configurable. La aplicación de demo incluye un archivo de configuración de forma que permite ver información de debug del framework. Esta librería no es necesaria pero si recomendada ya que así se puede tracear la inicialización y ejecución del framework.
Versión utilizada	1.2.16
Nombre del archivo	log4j-1.2.16.jar
URL de descarga	http://logging.apache.org/log4j/1.2/

Cabe decir que si bien las librerías no se han incluido dentro del directorio de código fuente, sí las he incluido dentro del archivo war, ya que no aumentaba apenas el tamaño y así se puede desplegar directamente.

B.2. Instalación

Se detalla a continuación el proceso seguido para instalar la aplicación de demo *apptest.war*

1. Comprobar que el contenedor web utiliza la versión 6 de java update 22 o superior

2. Comprobar que el contenedor web es compatible con Servlet 2.5 y JSP 2.1

3. Desplegar el archivo war según el contenedor específico.

En mi caso el contenedor utilizado ha sido Tomcat 6.0.29. La forma más sencilla de desplegar la aplicación es copiando el archivo *apptest.war* al directorio *webapps* de la instalación de Tomcat. Éste directorio viene configurado de serie para que se desplieguen automáticamente los archivos war que se ponen en ese directorio. A

continuación levantar el servidor si no estuviese levantado y suponiendo que se está en la máquina local y el puerto configurado es el 8080 acceder a la url http://localhost:8080/apptest

Si todo ha ido bien se debería ver una pantalla como la siguiente:

que indica que se ha instalado correctamente.

B.3. Manual de uso / descripción

La aplicación muestra en su página principal los títulos de los últimos 10 videojuegos que han puntuado los editores. Los usuarios pueden acceder a la información de los videojuegos y ver más detalles del mismo, además de poder añadir comentarios sobre el juego y ver los comentarios que han publicado otros usuarios.

Si se pulsa sobre el enlace del título del videojuego, se accede a la ficha del videojuego, pero sólo si se es un usuario registrado y logado. Si no se está logado se informa mediante una vista de que no es posible el acceso y se da la opción de entrar en el sistema utilizando las credenciales o bien crear un nuevo usuario.

En la zona superior izquierda de la cabecera existen enlaces para acceder al sistema o crear un nuevo usuario. Cuando un usuario está logado, se sustituyen estos enlaces por un saludo al usuario y la posibilidad de deslogarse. Si se pulsa sobre la imagen del gato se vuelve a la página de inicio.

Para logarse, se necesita un usuario y un password. Por defecto hay creado un par de usuarios. Se puede acceder con nombre de usuario demo y password demo. Si las credenciales son incorrectas, se muestran los errores encima del formulario.

Una vez logado ya puede acceder a la ficha y publicar comentarios. Si se desloga ya no es posible acceder a la ficha ni publicar los comentarios.

Otra opción es registrarse. Para ello se presenta un formulario donde se piden bastantes datos (la finalidad es mostrar el uso de diferentes etiquetas de controles de formulario). Los errores de validación se muestran en la misma vista.

Si se intenta registrar un usuario ya existente (por ejemplo utilizando un nombre de usuario test) aunque no sería lo correcto, se ha implementado que la acción lance una excepción (para demostrar el uso de gestión de errores).

Si el registro es correcto se muestra una pantalla de informando que se ha realizado correctamente y ya se pueden utilizar las credenciales en el formulario de login.

La aplicación está internacionalizada. En la zona superior derecha de la cabecera de las páginas existen enlaces que permiten escoger el idioma de la aplicación entre castellano e inglés.

En la parte inferior de la página principal hay un enlace a “Otras demos” donde se muestra el funcionamiento del tipo de resultado “redirect” proporcionado por el framework que permite hacer redirecciones a una URL así como el funcionamiento del tipo de resultado “httpheader” que genera respuestas HTTP con status declarado en los result del archivo de configuración. Si bien no es demasiado útil, el objetivo es mostrar cómo definir y utilizar un nuevo tipo de resultado.

Por último cabe decir que no existe persistencia ya que no es relevante a la capa de presentación. Al terminar el contenedor servlet se pierden los usuarios registrados nuevos y los comentarios.

B.4. Características del framework utilizadas

Las características del framework que se muestran en la demo son:

- **Mapeo de peticiones, acciones y resultados**. Flujo de navegación en general.
- **Forwards**: (mapeos sin acción asociada) permiten acceder a una vista pasando por el RequestProcessor en vez de directamente. Por ejemplo en el mapeo / showLoginForm
- **Plugins**: la demo declara un plugin encargado de crear e inicializar un servicio que implementa la capa de negocio y colocarlo en el ServletContext para que se pueda acceder al mismo desde diversos puntos.
- **Inyección de dependencias con interfaces Aware**: el plugin hace uso de ServletContextAware para que le inyecten el ServletContext. En el futuro se podrán ir añadiendo nuevas interfaces Aware para poder pasar nuevos datos sin romper los clientes.
- **Uso de las etiquetas (View Helpers) proporcionados por jNeko**: Se han empleado todas las etiquetas en la demostración. En todas las vistas se utilizan las etiquetas de internacionalización *msg* y la de enlaces *link.* En el formulario de login y

sobretodo en el del registro se utiliza la etiqueta *form* y sus múltiples controles que establecen propiedades como *radio*, *checkbox*, *select*, etc.

- **Internacionalización**: se muestra el uso de la internacionalización de textos y mensajes de error, algunos formateados con argumentos. Se demuestra el uso de los archivos ResourceBoundle con los textos. Se demuestra el uso de establecer el locale en la sesión mediante los métodos proporcionados en Action.
- **Crear nuevos tipos de resultado**: declaración, creación y utilización del resultado tipo httpheader
- **Formularios y validación**: en login y registro se realizan validaciones que informan de errores y provocan que se sirva el resultado con nombre “input”. Si se producen errores, éstos se informan sobre los formularios.
- **Formularios en diferentes ámbitos**: por defecto están en sesión. El formulario de login se ha declarado en scope “request”. Se muestra también cómo los View Helpers (las etiquetas) recogen los datos que existen en el formulario mostrándolos en el form de la jsp.
- **Validación en las acciones**: validación a nivel de acción en el LoginAction para comprobar si las credenciales son válidas.
- **Preprocesar peticiones**: se extiende el RequestProcessor para preprocesar las peticiones, si no se esta logado y el mapeo esta declarado como seguro, rompe el flujo normal y sirve el resultado que pide autorización.
- **Definir nuevas propiedades en los mapeos de acciones**: Se extiende ActionConfig y se declara la propiedad *secured* para poder indicar que se necesita logarse o no para ejecutar la acción. En el preproceso se utiliza esta nueva propiedad.
- **Automatización de servir resultado “input” cuando falla la validación**
- **Uso de gestor de excepciones a nivel global y a nivel acción**: se han declarado dos gestores de errores, uno a nivel de acción y otro global. El framework se encarga de buscar localmente o globalmente el gestor adecuado y si no se encuentra buscar gestores asociados a las superclases de la excepción lanzada.
- **Poder debugar el framework con log4j**: la demo declara un archivo de propiedades log4j.properties donde se h configurado el traceo a nivel de debug del framework
- **Validación con XMLSchema**: si se crea un archivo de configuración mal formado o si no se ponen ciertos valores o atributos, al inicializar el framework se muestra información detallada de donde ocurre el error. Se ha realizado un esfuerzo considerable en declarar el schema. En un futuro se podría hacer más restrictivo para ser menos tolerante a fallos.
- **Result type tipo redirect para redireccionar a url’s**: se muestra en la página de otras demos los enlaces a Google y StackOverflow.com. Éste último combinado con la seguridad de los mapeos.
- **Visualizar errores**: la etiqueta *errors* visualiza los errores que se han producido

- **Visualizar errores de una propiedad**: no se muestra en la demo, es un argumento que se puede pasar a la etiqueta errors con el nombre de propiedad para mostrar sólo los errores asociados a la propiedad. Se podría poner debajo de cada control de formulario para informar nivel de propiedad.
- **Establecer cabeceras nocache en la respuesta**: La demo está configurada para poner cabeceras de nocache en la respuesta
- **Fijar locale (fixLocale)**: No se ha utilizado en esta demo, se realiza con código a nivel Action.
- **Fijar contentType**: No se ha utilizado en esta demo, pero se muestra su apartado en la configuración.
- **Opción de validar o no formulario**: por defecto los formularios se validan, se podría poner a falso mientras se desarrolla.
- **Botón de cancelación de formulario**: El formulario de registro muestra como cancelar la petición de forma que no se ejecuta la acción y se puede ir a otro resultado.