Disseny i implementació d'un marc de treball (framework) de presentació per aplicacions J2EE

Guillermo Szeliga Cabezuelo

Enginyeria en Informàtica

Josep Maria Camps Riba

14/04/2011

Agradecimientos

Sonará a cliché, pero realmente agradezco a mi familia por el soporte y paciencia que me han prestado durante todos estos años, a Marta por ser comprensiva en esos días en los que tengo que dedicar un esfuerzo extra en mis estudios, además de su buena onda y felicidad, a mi Xbox 360 por darme dosis de relax entre el trabajo y aquellos momentos en los que necesito concentrarme en cosas totalmente diferentes e igualmente exigentes, y mi gata Pampa por hacerme compañía siempre por las noches.

Este trabajo se lo dedico también a mi Abuelo que se fue este año y que me dejo una frase muy buena entre tantas una vez que me vio con el libro de especificaciones de las API de Windows preparando la tesina en mi casa de Buenos Aires, haces varios años ya, me dijo “Yo no sé como hacés para leerte todo eso, pero con hacerlo, yo ya te daría el título directamente”. Un fenómeno.

Resumen

Este documento cubrirá diferentes aspectos relacionados con el diseño de un *framework* de presentación orientado a aplicaciones web. Comenzaremos analizando cuales serían los aspectos o características esperadas de una solución de este tipo, siempre desde la subjetividad y la experiencia, como puede ser el concepto de *zero configuration*, el cual intenta reducir el número de ficheros de configuración necesarios para la puesta en marcha, o el de *convention over configuration*, el cual intenta reducir el número de decisiones que los desarrolladores deban tomar, ganando en flexibilidad pero sin perder flexibilidad, y sobre todo permitiendo al desarrollador concentrarse en aquellos aspectos que son particulares a su solución.

Posteriormente, veremos cuales han sido las decisiones de diseño de algunos de los frameworks mas conocidos actualmente y así poder identificar cuales han sido sus aciertos y errores y, de alguna manera, poder determinar si nuestras intenciones van por buen camino o si deberíamos reconsiderar algunas de ellas para evitar posteriores errores.

A partir de estos puntos, esperemos que, al final del proceso, podamos contar con un producto inicial bien pensado, útil y si la suerte nos sonríe, con innovaciones o ideas que permitan posteriores soluciones que nos faciliten un poco más el trabajo de desarrollador.

Índice de contenido

1 Introducción

1.1 Punto de partida y aportación

Realmente el punto de partida de mi propuesta viene dada un poco por la experiencia de haber trabajo con varios frameworks de este tipo y conocer más o menos las ventajas y limitaciones que poseen cada uno de ellos. Es por ello que, en cuanto se proceda con la lectura de mis objetivos, uno podrá tener una idea clara de un concepto que creo muy importante: *configuración cero*. Esto quiere decir que la premisa de este framework, aunque con el poco tiempo con el que contamos para plasmar un diseño competitivo con muchas otras soluciones existentes en el mercado, será la de eliminar ficheros de configuración que, aunque claros al principio, suelen complicar con el tiempo el mantenimiento de los sistemas, sobre todo haciendo referencia a versiones de Spring anteriores a la 3 o Struts 1.2.x, donde los ficheros que contenían la configuración del sistema se hacían tan grandes que era muy difícil seguir la lógica establecida. Para ello, explotaremos en primera instancia recursos como *annotations*, gran aportación de la JDK 1.5, que nos permite embeber información en las propias clases, lo cual es grandioso, y junto con librerías tremendamente potentes como son *javassist, cglib y asm,* tenemos un conjunto de herramientas que nos proporcionan una flexibilidad impresionante a la hora de gestionar la configuración del framework, además de proporcionar maneras claras y concisas a los desarrolladores de como conseguir el comportamiento esperado con los recursos proporcionados. En segundo plano, aunque no sé si dispondré del tiempo deseado, me gustaría poder implementar solucionar dinámicas utilizado *aspects* que realmente reduzcan el total de código asociado con *loggers* que muestren información sobre el flujo de ejecución de la aplicación y dejar al desarrollador concentrarse en trazas más específicas de su solución. A partir de estas ideas iniciales, también me gustaría comentar que, para formalizar un poco más el proyecto, he creado un repositorio *Mercurial* en *bitbucket.org*, donde mi intensión es publicar el código fuente de mi propuesta, gestionar errores, mejoras y nuevas funcionalidades mediante un sistema de creación de *issues* y, sobre todo, la disponibilidad de la definición de una *Wiki*, sobre la cual me centraré luego de la entrega de la PAC3, donde intentaré documentar el *framework*, plasmar una *wishlist*, y mostrar ejemplos de implementación de las diferentes características con las que cuenta el *framework.*

Sinceramente, no sé a que lugares me podrá llevar este proyecto, pero una cosa que si se es que aprenderé cosas que hasta el momento no he tenido la oportunidad de indagar, y sobre todas las cosas, conocer la complejidad que puede llegar a representar un *framework* de presentación bien definido y funcional al cien por ciento. Es un desafío pero gratificante en muchos sentidos.

1.2 Objetivos

Entre los objetivos que intentaremos conseguir podemos mencionar

1. *Configuración sencilla e intuitiva*: La configuración inicial del *framework* deberá ser sencilla, de fácil comprensión y sin demasiadas complicaciones para conseguir un sistema funcional con poco esfuerzo. Para ello haremos uso de dos tecnologías muy interesantes y potentes para tal fin:

    1. *Annotations*: Es una forma sencilla de introducir *metadata* en el código fuente, de tal manera que, posteriormente, en tiempo de ejecución o compilación, podemos obtener información relacionada a nuestro framework y utilizarla para definir el comportamiento deseado.
    2. *Aspects*: Nos permitirá introducir funcionalidades adicionales en la aplicación huésped de forma transparente y sin necesidad de tener relación o conciencia de las funciones realizadas por esta última. De esta manera, seríamos capaces de introducir, por ejemplo, trazas de logs adicionales que muestren que es lo que está ocurriendo detrás de escena y ser capaces de identificar un comportamiento no esperado.

2. *Claridad de conceptos y funcionalidades:* Cuando me refiero a claridad de conceptos, esto puede venir dado por diferentes razones, ya sea por la nomenclatura utilizada en el diseño del *framework*, sobre todo haciendo hincapié en nombres de componentes o módulos que sean explicativos por si mismos, que proporcionen una idea inicial sobre la utilidad que obtendremos al hacer uso de los mismos, o también por la documentación proporcionada, la cual estará compuesta tanto por un manual de tipo técnico junto con una wiki, siempre proporcionando ejemplos de código funcionales y de fácil ejecución, utilizando herramientas como puede ser *Maven*, la cual nos permite definir la configuración necesaria para la

ejecución de nuestro proyecto.

3. *Validación de contenidos genérico:* Bajo la misma premisa de sencillez, la validación de contenidos no debería ser demasiado compleja ni necesitar demasiada configuración adicional, dado que cada valor de entrada debe ser alojado en una propiedad, y como dicha propiedad tendrá un tipo asociado o meta-información que nos indique como debemos tratar el dato entrante, podemos decir que simplemente tenemos que centrarnos en utilizar reglas de validación acordes a la información proporcionada y a retornar un resultado esperado.

4. *Gestión organizada y Feedback*: Desde mi punto de vista, resulta realmente importante poder obtener información de la comunidad que pueda generarse a partir de un proyecto, así como de los diferentes participantes del mismo, por lo que he decidido crear un proyecto en *bitbucket.org,* el cual me permitirá, por un lado, gestionar la evolución del proyecto de forma ordenada, ya sea mediante la utilización de ***Issues***, así como de la construcción y actualización de una ***Wiki***, además de permitirme controlar y disponer de mi código fuente mediante ***Mercurial*** (http://mercurial.selenic.com/*)*, el cual me permitirá distribuir mi código fácilmente, sin perder rastros de los colaboradores (que no se si será mi caso), además de poder trabajar de forma continuada y controlada sin necesidad de contar con conectividad hacia un repositorio centralizado. De esta manera, podre gestionar las diferentes partes que conformarán el proyecto y verificar su evolución de forma continuada.

Sumado a estos conceptos más bien genéricos, me gustaría mencionar como objetivos concretos del *framework* a diseñar los siguientes puntos:

- Utilización de URLs de tal forma que las mismas formen parte de la propia interfaz de la aplicación, es decir, que sean explicativas y claras en sus intensiones.
- Posibilidad de definición de múltiples controladores que posibiliten la relación de métodos en un único controlador, agrupados bajo un criterio funcional o conceptual, que sean visibles en las URL de forma prácticamente automática y transparente, sin necesidad de configuraciones adicionales.
- Definición clara y sencilla de diferentes roles, como pueden ser controladores, objetos de modelo, validadores, etc.
- Permitir un sencillo acceso y manipulación del modelo, además de la validación del mismo.
- Definición de un conjunto de *taglibs* de soporte, que ayuden al desarrollador en el acceso a características específicas de nuestro *framework*, como pueden ser a gestión de mensajes de error o recursos propios de la aplicación.

Para conseguir todos estos objetivos, se planteará el diseño y desarrollo de los siguientes componentes:

1. *Dispatcher*: Componente principal encargada de gestionar el flujo de la aplicación y determinar quien será el encargado de procesar una solicitud entrante.
2. *Scanner*: Componente encargado del rastreo de clases que contengan meta-información relacionada al framework e indexar dicha información para un acceso posterior más cómodo.
3. *PhaseManager*: Componente encargado de determinar cual es el siguiente paso a seguir a partir del resultado obtenido de una fase anterior, lo que puede significar la ejecución de un *request handler* o la resolución de una vista.
4. *ContextJunkDepot*: Componente que permite el almacenamiento de información generada durante el procesamiento de una solicitud y poner la misma a disponibilidad de todos los componentes de forma transversal.
5. *Controller*: Componente encargado de la manipulación de una solicitud entrante y de proporcionar un resultado que indique hacia donde debe continuar el flujo de ejecución.
6. *Validator*: Componente encargado de aplicar un conjunto de reglas de validación sobre un determinad valor e informar si el mismo cumple con las restricciones establecidas.
7. *Converter*: Componente encargado de convertir información de un tipo a otro, generalmente se encargarán de transformar una cadena en un tipo acorde a su contenido.
8. *Taglibs*: Conjunto de librerías que permitirán la interacción entre un fichero JSP y componentes a nivel de servidor relacionados a nuestro *framework*, como pueden ser gestión de errores, mensajes, etc.

Estos componentes son más bien un borrador de una idea que puede evolucionar o cambiar cuando entremos en la fase de diseño del *framework*, pero reflejan más que nada cual será la intensión de nuestro *framework* a modo global.

1.3 Enfoque y método seguido

Nos centraremos principalmente en conseguir un *framework* funcional en aspectos esperados y conocidos, como puede ser validaciones, conversión de datos, control de flujo, etc. Pero también no intentaremos "reinventar la rueda", lo que quiero decir es que hace mucho tiempo que existen soluciones contrastadas, de gran utilidad, y que en nuestro caso nos permitirán concentrarnos en nuestra solución. Estoy hablando de librerías como *common-beanutils*, la cual me permitirá la utilización de patrones para asignar y recuperar valores de propiedades asociadas a *beans* del modelo, concepto relacionado también con *Expresion Language (EL)* a nivel JSP, junto con la posibilidad de implementar *Converters*, los cuales son componentes encargados de convertir información entrante en un tipo dado. Además, tenemos la librería *reflections*, basada en *javassist* y *asm,* las cuales, entre muchas otras cosas, me permitirán crear interfaces o métodos dinámicos que aligeren la definición de los controladores a la hora de definir *getters* y *setters*, y por ende, permitir al desarrollador codificar solo el código de negocio sin preocuparse de nada más, o localizar definiciones de controladores o validadores en el *classpath* de forma automática utilizando simplemente *annotations* específicas, etc.

Por otro lado, la configuración del proyecto estará gestionada por *Maven*, definiendo tanto las dependencias del *framework* junto con el proceso de empaquetado mediante ficheros POM. También intentaré montar una suite, aunque básica, de JUnits que comprueben, por lo menos, la integridad del código a nivel funcional, dado que pruebas de aceptación o comportamiento es un poco más complicado y el factor tiempo no estará de mi parte. Finalmente, como se ha mencionado antes, como SCM utilizaré *Mercurial*, dado que, por un lado, me permite trabajar en diferentes lugares sin necesidad de contar con una conexión de red hacia el repositorio centralizado, y por otro, porque estoy habituado a este tipo de repositorio dado que soy usuario frecuente de Linux.

La forma de diseñar y codificar la propuesta será básicamente ir creando los diseños en papel, luego procederemos a su codificación, realizar una prueba conceptual mediante un *JUnit* y, finalmente, si se comprueba que los diseños producen el resultado esperado, lo oficializamos mediante un gráfico UML que represente el diseño de cada uno de los componentes. Supongo que las partes principales del *framework* serán diseñadas partiendo del *dispatcher* y de la forma en la que identificaremos cada uno de los componentes configurados a partir de las *annotations* asociadas y parámetros de arranque del *framework*, para luego definir de que forma se identificará y delegará el procesamiento de una solicitud hacia un controlador o a una vista, y de esta manera completar un flujo básico. Luego, procederemos a definir la utilidad de validación y conversión de datos junto con la gestión de errores durante el flujo de ejecución. Finalmente, habiendo conseguido un flujo completo en cuestiones de funcionalidades y comportamiento, procederemos con el desarrollo de *taglibs* que proporcionen ayuda al desarrollador a la hora de necesitar acceder a información sobre el flujo ejecutado, concretamente, acceder a los errores capturados o mensajes configurados en ficheros de propiedades. Finalmente, formalizaremos un poco más el conjunto de pruebas sobre el *framework* y actualizaremos toda documentación que nos quede pendiente, especialmente hablando de la *Wiki.*

1.4 Planificación

Tarea	Hito	Entregable	Fecha inicio	Fecha Entrega
PAC1	Elaboración documento	Plan de Trabajo	03/03/11	16/03/11
	Comenzar capítulo 1 memoria		17/03/11	
PAC2	Análisis Spring MVC		17/03/11	
	Análisis Struts 2		23/03/11	
	Análisis JSF		30/03/11	
	Finalización capítulo 1 y 2	Memoria	06/04/11	14/04/11
PAC3	Diseño arquitectura (UML)		16/04/11	
	Desarrollo		24/04/11	
	Aplicación ejemplo		15/05/11	
	Finalización capítulo 3	Memoria	17/05/11	23/05/11
Presentación virtual	Diseño documento	Presentación	24/05/11	01/06/11
Lliurament final	Terminar temas pendientes	Memoria + Presentación virtual	02/06/11	13/06/11

1.5 Productos obtenidos

Finalizadas las tareas mencionadas anteriormente y alcanzados los objetivos establecidos, los productos que se esperan obtener son los siguientes:

- *framework:* Librería principal en la cual se encontrará prácticamente todo el trabajo realizado a lo largo del PFC. Esta contendrá tanto el *core* de nuestro proyecto junto con clases e interfaces bien definidas que permitan la personalización de funcionalidades y la comunicación con el *framework*.
- *framework-taglibs:* Consistirán en un conjunto de JSTL que permitan un acceso más cómodo y abstracto a información relacionada con el ciclo de procesamiento de una solicitud HTTP, concretamente en nuestro caso, la funcionalidad proporcionada será la de acceder a posibles errores que hayan tenido lugar durante el procesamiento así como a mensajes estáticos o parametrizados cuyos contenidos tengan que ser mostrados en la interfaz web cliente.
- *framework-sample-application:* A modo de ejemplo, procederemos a definir una aplicación sencilla en funcionalidad pero que permita visualizar las capacidades del *framework* definido de una forma clara y funcional.
- *Wiki:* Me interesa mucho poder crear una Wiki a modo de manual de uso mostrando ejemplos y casos de uso del entorno desarrollado. Esto lo haremos gracias a herramientas proporcionadas por *bitbucket.org*.

1.6 Descripción de contenidos

A continuación, el documento se estructurará en dos capítulos principales. El capítulo número 2 nos presentará una introducción a nivel arquitectónica de 3 *frameworks* para aplicaciones web que considero reflejan muchas tendencias de la actualidad y errores de diseño que se han cometido en versiones anteriores; tal es el caso de Spring MVC, gran *framework* que nos presenta puntos innovadores como la utilización de IoC o sintáxis de tipo RESTful en mapeo de URLs entre otras cosas, Struts 2, versión posterior del exitoso y difundido Struts 1.2.x que nos muestra como aprender de nuestros errores y a como escuchar a la comunidad de desarrolladores que te rodea, y finalmente JavaServer Faces, el cual, con la introducción de las nuevas especificaciones JEE6 y el auge de contenedores de aplicaciones como Glassfish, estamos frente a un *framework* que está creciendo en utilización y comunidad, además de presentarnos una forma diferente de pensar el diseño de una aplicación web, dado que el mismo es un *framework* web orientando a componentes.

Por otro lado, en el capítulo 3, nos introduciremos en el diseño del *framework* que nos ocupa, intentando abarcar cada uno de sus aspectos particulares, justificar las decisiones de diseño tomadas y evaluar un poco, a partir de lo conseguido, hacia donde podríamos continuar su evolución si hubiera un futuro claro para su utilización.

Finalmente, plasmaremos nuestras sensaciones y conclusiones sobre el proyecto abordado e intentaremos aportar algo en este mundo tan interesante como es el de las aplicaciones web.

2 Alternativas existentes en la actualidad

2.1 Spring Web MVC

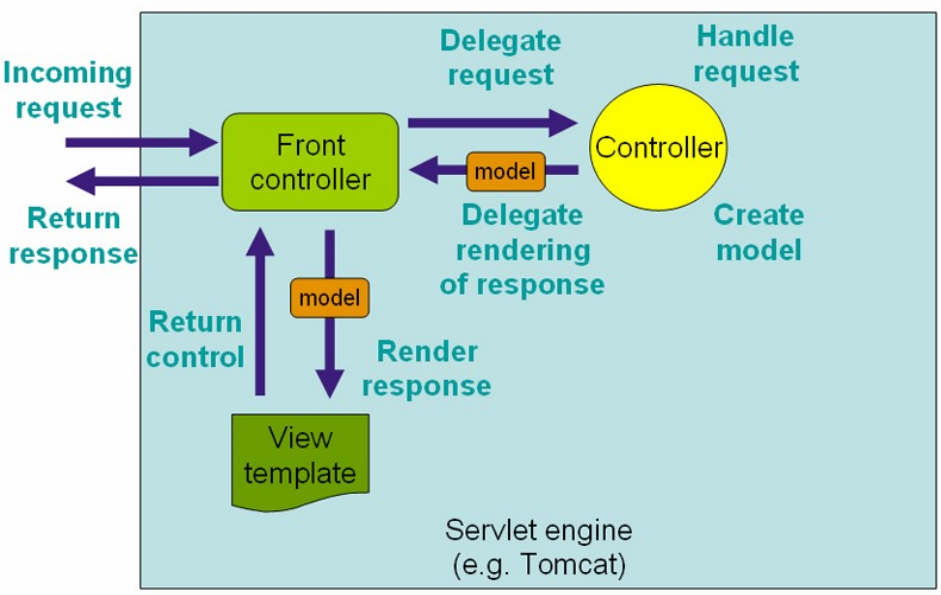
Me gustaría comenzar por un *framework* con el cual me siento identificado en cuanto a filosofía y forma de hacer las cosas: Spring Wev MVC. Este, como la gran mayoría de *frameworks* de presentación, ha sido diseñado a partir de un componente principal llamado *DispatcherServlet*, el cual, como su nombre puede sugerir, se encarga de despachar o entregar solicitudes a *handlers*, los cuales cuentan con configuración relacionada a vistas, *locale* o incluso temas de presentación. Cada *handler* se configura principalmente con dos anotaciones propietarias: *@Controller* y *@RequestMapping.* Adicionalmente, en la última versión disponible del framework, se permite también la definición de aplicaciones y sitios web mediante lo que se conoce como RESTful haciendo uso de una anotación llamada *@PathVariable.*

Las ventajas de este *framework* residen principalmente en la flexibilidad de poder utilizar cualquier objeto como soporte de un objeto formulario o *command,* evitando así la necesidad de implementar un objeto o interfaz específica del *framework* para tal fin. Por otro lado, la relación de la información remitida desde el cliente con un objeto de soporte es realmente sencilla, dado que trata tipos inesperados de información como errores de validación evaluados por la propia aplicación, y en caso que los tipos sean correctos, la relación y asignación de información se lleva a cabo de forma directa con los objetos de negocio definidos.

Lo mismo ocurre con la resolución de vistas, un controlador puede escribir directamente en el *stream* de respuesta, pero usualmente existe una instancia de un objeto de tipo *ModelAndView*, el cual se compone de un nombre de vista y un *Map* con el modelo, el cual contendrá nombres de *beans* y sus correspondientes objetos como pueden ser un objeto *command* o un formulario.

Para conseguir la resolución de vistas, existen múltiples caminos a seguir, ya sea mediante nombres de *beans*, un fichero de propiedades o incluso la implementación de un objeto de tipo *View Resolver.* De esta manera, el modelo se implementa mediante el *Map,* lo que permite la abstracción de la tecnología utilizada para las vistas. De esta manera, podemos utilizar tanto JSP como Velocity o cualquier otra tecnología similar.

Uno de los puntos fuertes del *framework* en cuestión es que esta completamente integrado con el contenedor *Spring IoC*, permitiéndole utilizar cualquiera de las características que *Spring* proporciona. De esta manera, podemos plasmar el flujo de ejecución de Spring MVC de la siguiente manera:



El componente *DispatcherServlet* es en realidad un *Servlet*, el cual se declara en el fichero *web.xml* de la aplicación, especificando un patrón de mapeo que indica que solicitudes deberán ser gestionadas por dicho controlador. Además de esto, cabe considerar que cada *DispatcherServlet* posee su propio *WebApplicationContext*, el cual hereda todos los *beans* definidos en el *WebApplicationContext* raíz. Principalmente, mediante la utilización de este último, tenemos la

posibilidad de configurar y utilizar *beans* especiales para el procesamiento de solicitudes y resolución de vistas, entre los cuales podemos mencionar:

- *Controllers*: Justamente compone la parte C del MVC
- *Handler Mappings*: Gestionan la ejecución de una lista de pre-procesadores y post-procesadores junto con controladores que serán ejecutados si coinciden con ciertos criterios especificados.
- *View Resolvers*: Encargados de la resolución de vista a partir de sus nombres.
- *Locale Resolvers*: Componente capaz de determinar el locale que el cliente este utilizando, con el propósito de proporcionar la capacidad de internacionalización.
- *Theme Resolvers*: Componente encargado de determinar el tema que la aplicación web puede utilizar, ofreciendo de esta manera diseños personalizados.
- *Multipart File Resolvers*: Funcionalidad necesaria para el procesamiento de subida de ficheros desde formularios HTML.
- *Handle Exception Resolvers*: Funcionalidad que permite el mapeo de excepciones hacia vistas u otras implementaciones de control de errores más complejas.

Generalmente, el flujo de procesamiento de una solicitud suele ser de la siguiente manera:

1. Un *WebApplicationContext* es buscado y relacionado a la solicitud en cuestión mediante un atributo de la misma, el cual podrá ser utilizado tanto por el controlador como por otros elementos que participen en el proceso.
2. Un *locale resolver* es relacionado a la solicitud para permitir a los elementos en el proceso resolver el *locale* a utilizar al procesar una solicitud, ya sea generando una vista, preparando información, etc.
3. Un *theme resolver* es relacionado a la solicitud para permitir a los elementos, como pueden ser las vistas, determinar que tema de estilos deben utilizar.
4. En el caso que sea especificado un *Multipart File Resolver*, la solicitud es inspeccionada en búsqueda de contenidos *multipart.* En el caso que fuese encontrado, se construirá un objeto *MultipartHttpServletRequest* para un correcto procesamiento posterior por otros elementos participantes en el proceso.
5. Se procese con la búsqueda de un *handler* adecuado para el procesamiento de la solicitud entrante. En caso que fuese encontrado, la cadena de ejecución asociada al *handler* será ejecutada (es decir, preprocessors, postprocessors y controllers) con el propósito de preparar el modelo o procesamiento.
6. En el caso que un modelo sea devuelto, la vista es procesada y devuelta. Si ningún modelo es devuelto, ninguna vista es generada, dado que la solicitud podría ya haber sido tratada.

En en caso que se produjera alguna excepción durante el flujo antes mencionado, existen un conjunto de *Handler Exception Resolvers* declarados en el *WebApplicationContext* encargados de recoger las excepciones que son lanzadas durante el procesamiento, permitiendo de esta manera la definición de un comportamiento personalizado para la gestión de excepciones.

En definitiva, habiendo recorrido un poco las diferentes características del *framework*, podemos mencionar como principales ventajas:

1. La división entre controladores, JavaBean que componen el modelo y vistas esta realmente bien definida.
2. Dado que el mismo esta basado en múltiples interfaces, el *framework* resulta realmente flexible en cuestiones de configuración, teniendo incluso la posibilidad de ser configurado vía *plugging* en interfaces propias, además de contar con clases propietarias que puedan ser tenidas en cuenta a la hora de la definición de implementación.
3. La existencia de *interceptors* junto con controladores hacen realmente sencillo la reducción del comportamiento común a la hora de procesar múltiples solicitudes.
4. Realmente es un *framework* que no depende de una cierta tecnología de vistas. De esta manera,

no es necesario utilizar JSP si no se quiere así, pudiendo utilizar otras tecnologías como *Velocity, XLST* o alguna otra tecnología. Incluso sería posible la utilización de un mecanismo propio de vistas que puede ser fácilmente implementado mediante las interfaces adecuadas.

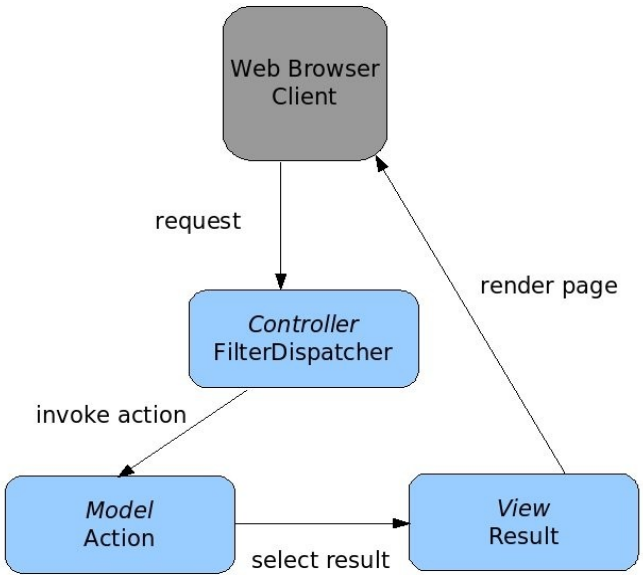
5. Los controladores pueden ser configurados mediante IoC como muchos otros objetos, haciéndolos de esta manera fáciles de probar y elegantemente integrados con otros objetos gestionar por Spring.
6. Dado que no es obligatoria una herencia concreta para la definición de la capa de presentación así como una dependencia explícita de los controladores con el *dispatcher servet*, la ejecución de suites de pruebas sobre la misma es realmente sencilla.
7. La arquitectura que compone el *framework* permite la definición de una capa web *thin client* por sobre nuestra capa de negocios, incentivando la utilización de buenas prácticas. No solo esto, sino que también *Spring* proporciona un *framework* integrado para todas las capas de la aplicación.

2.2 Struts 2.0

Se trata de la segunda generación de un *framework* para aplicaciones web que implementa el patrón de diseño de Modelo-Vista-Controlador (MVC). Dicho *framework* ha sido construido en base a buenas prácticas y a patrones de diseño aceptados por la comunidad de desarrolladores, al igual como fue el caso de su predecesor, cuya principal ventaja en su momento fue justamente la incorporación del patrón MVC en un *framework* web.

Ahora bien, dadas a las ya conocidas limitaciones de la primera versión, *Struts 2* ha intentado aprovechar lo aprendido de errores pasados y desarrollar una implementación mucho más limpia del modelo MVC. Al mismo tiempo, ha introducido nuevas características a nivel arquitectónico que han hecho del *framework* una herramienta mucho más flexible y limpia con la cual desarrollar. Entre las características nuevas podemos encontrar cosas como *interceptors* para establecer capas de *cross-cutting concerns* independientes de la lógica de una acción, configuración basada en *annotations* para reducir una posible polución del *classpath*, un potente lenguaje llamado *OGNL* totalmente transversal al *framework*, y una API basada en *tags* que soporta componentes de interfaz modificables y reutilizables.

Como sabemos, el patrón MVC proporciona *separation of concerns* en tres tipos diferentes: *model, view y controller*, permitiendo de esta manera la gestión de la complejidad de un software de tamaño considerable al dividirlo en diferentes componentes de bajo nivel. En el caso de Struts 2, estos componentes son *Action, Result y FilterDispatcher*, cuya interacción sería de la siguiente manera:

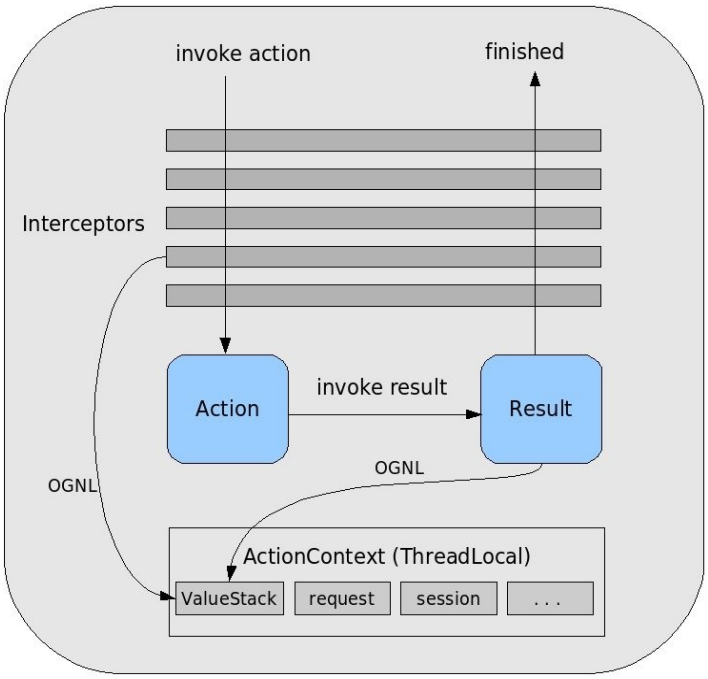


En el caso de *Struts*, se implementa una variante del patrón MVC conocida como *front controller MVC*, lo que significa que el controlador se encuentra en el inicio y es el primer componente que actúa en el procesamiento. Principalmente, el objetivo de un *controller* es el procesamiento de mapear una solicitud entrante hacia una *action* concreta, dado que, en una aplicación web, las solicitudes HTTP entrantes pueden considerarse como *commands* que el usuario dirige a la aplicación.

En nuestro caso, el rol del *controller* esta a cargo del *FilterDispatcher*. Este objeto se trata de un *servlet filter* que inspecciona cada solicitud entrante para determinar que *action* debería gestionar dicha solicitud, teniendo como única tarea la de informar al *framework* que URL debe ser mapeada con que *action*. Esta información puede ser proporcionada mediante la utilización de ficheros XML o mediante *annotations*, las cuales permiten llevar al *framework* a lo que se conoce como *zero-configuration*, intentando derivar toda la meta-información de la aplicación, como puede ser que URL mapea contra que *action*, desde la convención más que desde la configuración.

En el caso de una *action*, estas podemos considerarlas una encapsulación de las llamadas hacia la lógica de negocio en una única unidad de trabajo además de servir como punto de transferencia de información. De esta manera, podemos considerar que una aplicación contiene un cierto número de acciones para gestionar cualquiera sea el número de *commands* que estos expongan al cliente, donde, cuando el *controller* reciba una solicitud, este debe consultar su información de mapeo y determinar cual de las *action* existentes debería gestionar la solicitud, y una vez encontrada, el *controller* entrega el control del procesamiento de la solicitud a la correspondiente *action* mediante su invocación. El proceso de invocación, conducido por el *framework*, preparará tanto la información necesaria y ejecutará la lógica de negocio del *action*. Cuando este finalice, el mismo ejecutará un *forward* hacia un componente *view.*

Como hemos mencionado antes, el *view* es el componente de presentación del patrón MVC. Mientras que existen múltiples alternativas para una vista, el rol de la misma esta bastante claro; el traducir el estado de la aplicación en una presentación visual con la cual el usuario pueda interactuar. Es este flujo de ejecución, el *action* es el encargado de seleccionar que resultado deberá ser procesado en la respuesta, teniendo la posibilidad de seleccionar un cierto número de resultados con el cual el mismo estará asociado. De esta manera, podemos decir que un posible flujo en Struts 2 sería de la siguiente manera:



Asumiendo que el *FilterDispatcher* ya ha ejecutado su trabajo de control, que el *framework* ya ha seleccionado un *action* y a iniciado el proceso de invocación, este será ejecutado para luego seleccionar el resultado adecuado. Ahora bien, en el diagrama también podemos ver nuevos componentes: *interceptors*, el *ValueStack* y *OGNL.*

En el caso de los *interceptors*, estos serán invocados tanto antes como después de cada *action*, pero no necesariamente deben hacer algo todas las veces que son invocados; muchas veces el control simplemente pasa a través de ellos. Algunos interceptores solo hacen trabajo antes que el *action* sea ejecutada, y otros hacen ciertos trabajos luego, pero el punto fuerte e importante en todo esto es que los *interceptors* permiten tareas comunes y transversales que son definidas en componentes limpios y reutilizables que pueden mantenerse independientes del código del *action.* Estos casos pueden asociarse, por ejemplo, al *logging*, *data validation, type conversion* y *file uploads*, dado que se trata de funcionalidades que no forman parte de la unidad de trabajo de un *action*, sino que mas bien cuestiones mas administrativas.

En el caso del *ValueStack* y *OGNL*, podemos decir que el *ValueStack* es simplemente un área de almacenamiento que contiene toda la información de dominio de la aplicación relacionada con el

procesamiento de la solicitud. Se puede considerar como si fuera una hoja borrador donde el *framework* realiza su trabajo con la información mientras va resolviendo problemas relacionados con el procesamiento de la solicitud. Por otro lado, tenemos *OGNL*, el cual es un lenguaje que permite la referencia y manipulación de información en la *ValueStack.* El truco y la utilidad de estos dos componente es que no pertenecen a ningún componente del *framework*, lo que permite que, por ejemplo, tanto los *interceptor* como los resultados pueden utilizar *OGNL* para la obtención de valores desde la *ValueStack.* La información de la *ValueStack* sigue el procesamiento de la solicitud a través de todas las fases dado que la misma se almacena en un contexto *ThreadLocal* llamado *ActionContext.* Este último contiene toda la información que crea el contexto en el cual una acción ocurre, incluyendo la V*alueStack*, pero también otras cosas que el *framework* necesita utilizar de forma interna, como puede ser la *request*, la *session* y *application maps* provenientes de la API del Servlet.

De esta manera, podemos tener una visión general de la arquitectura del *framework* que nos ocupa y de que forma cada uno de sus componentes interactúan entre si, el cual sigue manteniendo la misma filosofía o puntos comunes con otros *frameworks* existentes en el mercado, sobre todo la introducción de *annotations* para la configuración del mismo, aunque no tan potente como podríamos decir de Spring, pero realmente es un prinicipio muy prometedor, sobre todo teniendo en cuenta que este ha introducido muchas funcionalidades de otro *framework* que realmente era interesante en su momento llamado *WebWorks.*

2.3 Java Server Faces

En el caso que de JavaServer Faces (JSF), este simplifica el desarrollo de interfaces sofisticadas para aplicaciones web principalmente definiendo componentes de un modelo relacionados a un ciclo de procesamiento de solicitudes bien definido, permitiendo de esta manera:

- El desarrollo de aplicaciones de soporte (*backend)* sin necesidad de preocuparse acerca de los detalles del protocolo HTTP e integrarlas con la interfaz de usuario a través de un modelo gestionado por eventos (*event-driven)* con interfaces *type-safe.*
- Trabajar en el *look & feel* de las páginas sin necesidad de conocimientos de programación mediante el ensamblaje de componentes que encapsulan toda la lógica de interacción de usuario, minimizando de esta manera la necesidad de embeber lógica directamente en las páginas que componen la interfaz de usuario.
- El desarrollo de herramientas muy potentes tanto para el desarrollo del *backend* como del *frontend.*

Concretamente, JSF es una especificación con implementaciones ofrecidas por múltiples proveedores. Este define un conjunto de componentes para la interfaz de usuario, básicamente un mapeo *one-to-one* hacia el formulario HTML además de algunos extras que pueden ser utilizados *out-of-the-box* y de una API que permite la extensión de los componentes estándar o el desarrollo de nuevos. Por otro lado, los validadores que se adjuntan a los componentes se encargan de validar la información introducida por el usuario, la cual es posteriormente propagada hacia los objetos que componen la aplicación. En el caso de los *event handlers*, estos suelen ser activados por acciones del usuario, como puede ser el pulsar un botón o un enlace, cambiando de esta manera el estado de un uno de los componentes o invocando el código residente en el *backend.* El resultado del procesamiento de evento controla que página es mostrada a continuación, con ayuda de un *handler* de navegación configurable.

Mientras que el lenguaje HTML es el lenguaje de marcas seleccionado por la mayoría de las aplicaciones web, JSF se esta limitado a este o a ningún otro lenguaje. *Renderers* que estan totalmente separados de los componentes UI controlar el verdadero contenido que se envía al cliente, permitiendo de esta manera que el mismo componente UI pueda ser emparejado con *renderers* diferentes y producir diferentes resultados, pudiendo denominarlo de similar manera que en *Swing: "pluggable look & feel".*

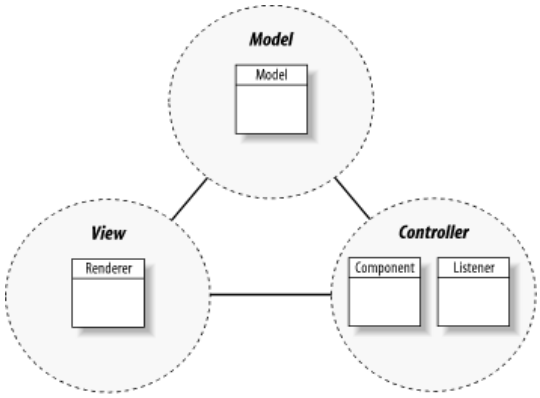
Realmente existe una gran flexibilidad en la forma en que se decida desarrollar la interfaz de usuario. Eso si, todas las implementaciones de JSF deberán soportar JavaServer Pages (JSP) como tecnología en la capa de presentación, con componentes JSF representados por elementos propios de tipo *action o* comunmente conocidos como *custom tags.* Además de esto, la API de JSF es lo suficientemente flexible para soportar otras tecnologías de presentación además de JSP. Por ejemplo, es posible utilizar solo código Java para la creación de componentes JSF, lo cual es similar a como una interfaz *Swing* es desarrollada. Alternativamente, es posible relacionar componentes JSF a nodos

en plantillas descriptas por HTML plano, propuesta explorada por proyectos como Barracuda/XMLC y Tapestry.

Ahora bien, ahondando un poco en en cuestiones de arquitectura web, hasta el momento la recomendación de muchos desarrolladores experimentados recomendaban soluciones no tan rigurosas basadas en el patrón de diseño MVC. Este patrón fue descripto por Xerox en un número de publicaciones a finales de los años 80, en conjunto con el lenguaje *Smalltalk*. Este modelo ha sido utilizado desde entonces por aplicaciones *GUI* desarrolladas en un cierto conjunto de lenguajes de programación.

Mientras que muchos *frameworks* de aplicaciones web para Java soportan dicho patrón de diseño en un alto nivel, no esta del todo soportado de la misma forma estricta y precisa como en un framework de presentación. Por ejemplo, Struts, la vista es representada mediante un conjunto de páginas JSP, el *controller* por medio de un servlet que delega el trabajo real hacia una clase de tipo *Action*, y el modelo por clases de la aplicación que usualmente interactúan con una base de datos o cualquier otro tipo de medio de almacenamiento. Las interfaces entre las diferentes partes, sin embargo, no se encuentran definidas como un conjunto específico de métodos con parámetros declarados. Por otro lado, el control se pasa entre las diferentes partes usando métodos genéricos, y la información que la siguiente parte necesita se encuentra disponible mediante colecciones normales como pueden ser colecciones con todos los parámetros HTTP, cabeceras, atributos, etc. Otra diferencia importante entre una GUI y un *framework* para aplicaciones web es que este último reconoce un solo tipo de evento: la solicitud HTTP. El código que gestiona este macro evento tiene que indagar en la información que la conforman para determinar si el usuario ha introducido nueva información en algún campo, solicitar por un tipo diferente de presentación seleccionando una opción en un menú, ordenar que una transacción finalice, o alguna otra cosa. Un *GUI framework*, por otro lado, esta basado en eventos bien definidos, como puede ser el cambio de un valor, la selección de un ítem de un menú o el pulsado de un botón, con *handlers* que gestionan cada evento.

Concretamente, el modelo MVC de JSF esta diseñado de la siguiente manera:

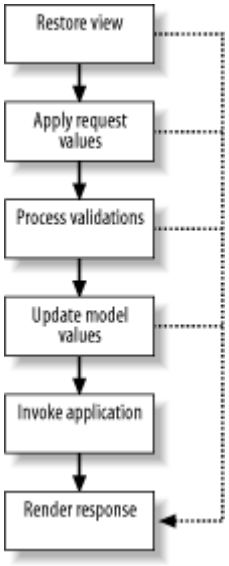


El modelo esta representado por propiedades de los objetos de la aplicación, como por ejemplo la propiedad *username* en un objeto que mantenga información relacionada a información del usuario. Los componentes JSF UI declaran que eventos pueden disparar, como por ejemplo un evento de tipo "cambio de valor" o "botón pulsado", y *event listeners* externos (los cuales representan el *controller)* adjuntos junto a los componentes que gestionar dichos eventos. Un *event listener* podría asignar valores a propiedades de un componente JSF o invocar código de soporte (*backend)* que procese la información remitida. Una clase procesadora independiente (que representa la vista) procesa cada componente JSF UI, haciendo posible procesar instancias del mismo componente en diferentes maneras (por ejemplo, ya sea como un botón o un enlace, o utilizando otros lenguajes de marcas diferentes). Además de los componentes UI, JSF también define *artifacts* como validadores o convertidores, cada uno con un propósito bien definido. De esta manera, el resultado final es un código modular de una interfaz fácil de mantener, familiar para desarrolladores de aplicaciones GUI, y que se presta a la utilización de herramientas de soporte.

Por otro lado, también es cierto que JSF podría no ser adecuada para todas las aplicaciones web. Si se desarrolla un sitio web donde los aspectos dinámicos son limitados a cosas como recuperación de información desde la base de datos, generar contenido dinámico de menús, agregar información dinámica mediante cookies u otras características que hagan fácil el mantenimiento del contenido de la web y el acceso a diferentes partes del sitio, JSF podría ser un suicidio. La mejor opción para utilizar JSF es en una aplicación web real, es decir, un sitio con mucha interacción de los usuarios, en vez de un sitio web con algunos contenidos dinámicos. Ahora bien, también es cierto que JSF no

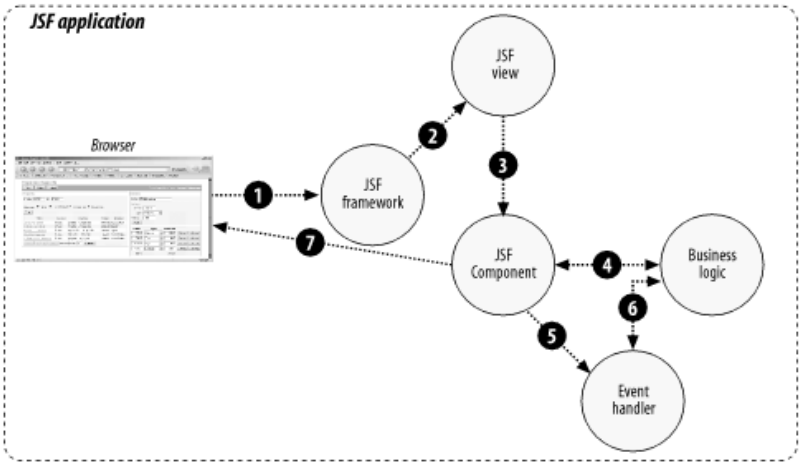
necesariamente reemplace las tecnologías actuales, sino que más bien es un complemento que proporciona una estructura y mantenibilidad a la interfaz de la aplicación.

Habiendo abarcado un amplio número de conceptos relacionados sobre JSF, como otros *framework*, este cuenta con un ciclo de procesamiento en fases, el cual podemos plasmar de la siguiente manera:



- En la fase *Restore View*, los componentes que conforman la vista son restaurados tanto desde información alojada en la solicitud o desde información guardada en el servidor.
- Luego, en la fase *Apply Request Values*, cada componente en la vista busca su propio valore en la solicitud y lo guarda.
- Los componentes podrían validar sus nuevos valores en la fase denominada *Process Validation*.
- Las propiedades del modelo son actualizadas con los nuevos valores en la fase *Update Model Value.*
- Cuando las propiedades del modelo han sido actualizadas con los últimos valores, los *event listeners* pueden llamar código en el *backend* para procesar la nueva información en la fase *Invoke Application.*
- Finalmente, una respuesta a la solicitud es enviada, utilizando la misma vista o una nueva. Todo esto ocurre en la fase *Render Response.*

Consiguiendo un flujo entre los componentes antes mencionado mas o menos de la siguiente manera:



2.4 Conclusiones

Habiendo revisado de forma general cada uno de los frameworks propuestos, realmente las conclusiones que podemos sacar es que cada uno se adapta a diferentes necesidades y características de los proyectos. Podemos decir que la diferencia principal entre *Struts* y *JSF* es que el primero es orientado a *actions*, lo que quiere decir que nos proporciona la habilidad de mapear URLs hacia operaciones, y por ende, a código en el *backend*, teniendo un *layout* y un *workflow* que tiende a estar más orientado a páginas e intentando simplemente aislar al menos la relación de la información remitida en una solicitud con las clases que implementan las diferentes *actions.* mientras que el segundo esta orientado hacia *components*, que como hemos visto antes, estos se desarrollan como componentes individuales, de forma similar a un GUI perteneciente a librerías que componen un *fat client.* A la vez, estos componentes tienen eventos, y el código asociado a estos esta escrito para trabajar con aquellos componentes relacionados.

Podemos decir que, en general, un framework orientado a *actions* tiende a ser más *thinner* en cuestiones de como interactúa el código desarrollado y una solicitud HTTP. Por otro lado, un framework orientado a *components* puede ser mas manejables, dado que las herramientas que puedan haber disponibles tienden a ocultar la parte pesada de cualquier componente, pero que también puede contar con desventajas relacionadas a su posible tamaño y complejidad a la hora de, por ejemplo, convertir solicitudes HTTP en un equivalente a un click de ratón, pudiendo significar más complejidad de comprensión cuando el *framework* presenta comportamientos inesperados o dificulta el avance de proyecto.

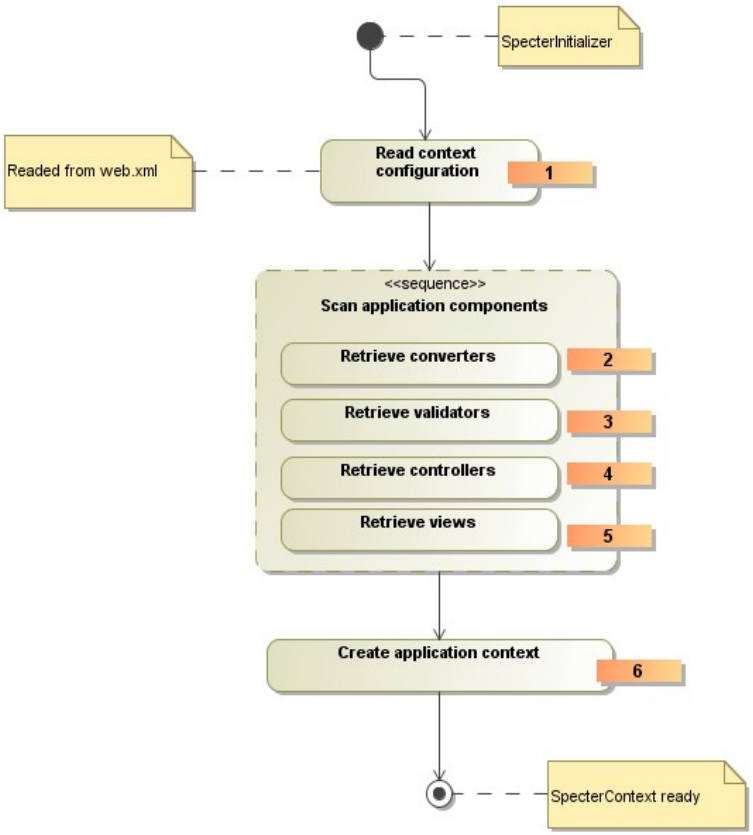
Junto al conjunto de framework orientados a *actions*, podemos mencionar también Spring MVC, sobre el cual ya se han enumerado las ventajas que proporciona con respecto al resto de frameworks, pero podemos mencionar como una de las principales la posibilidad de utilizar un *framework* con *dependency injection*, además de ser capaz de integrarse con JSF sin ningún tipo de problemas, permitiéndonos así obtener lo mejor de ambos mundos.

Ahora bien, hay un factor que, según mi experiencia, también es muy importante a la hora de seleccionar el framework a utilizar, y es la comunidad de desarrolladores/usuarios de cada uno de ellos junto con una documentación clara de las diferentes características que nos proporciona el framework de turno, sobre todo porque cuando nos encontramos con un problema de implementación, nuestras armas para poder solventarla y salir airosos de la misma es disponer de información orientativa a nivel funcional, que nos permita en cierta manera conocer más a fondo el comportamiento de la aplicación, y también poder contar con la suerte de que alguien se haya topado con el mismo problema y, gracias a la intervención de otros usuarios, se haya encontrado una solución a la misma, incluso junto a la intervención de los propios desarrolladores del *framework*. Con esto quiero decir que, por más potente que sea un *framework* o utilidad, muchas veces la poca ayuda que pueda encontrarse o información de uso de la misma suele disuadir a su utilización. Justamente, un problema actual con Struts2 es que, cuando se intenta encontrar información especifica sobre el comportamiento de ciertos componentes, muchas veces nos encontramos con información relacionada a la versión 1, lo cual nos indica que, por un lado, la versión 2 del *framework* no es utilizada por mucha gente, y por otro lado, la falta de información muchas veces nos ha llevado a tener que descargar el propio código fuente del *framework* para ver en concreto que es lo que estaba ocurriendo internamente en un punto de ejecución, haciéndonos perder tiempo en algo sobre lo cual hubiera sido más sencillo contar con una simple explicación de su comportamiento asociado.

3 Diseño e implementación

3.1 Introducción

El *framework* sobre el cual hemos estado trabajando he decidido denominarlo *specter*. Este *framework* realmente no fue concebido con el propósito de inventar cosas nuevas, debido más que nada a la escasez de tiempo con la que contamos para tal objetivo, pero si intenta aprovechar muchas de las ideas desarrolladas en otros *frameworks* estudiados que me han parecido interesantes y que en su conjunto creo nos permitiría ser capaces de potenciar un desarrollo rápido, de fácil implementación e intuitivo en su concepción y utilización. A modo de introducción, podemos decir que este *framework* no requiere de ficheros de configuración para su puesta en marcha, sino que todo lo contrario, el propio *framework* es el encargado de recolectar información relacionada a los diferentes componentes que pueden llegar a conformar nuestra aplicación, gracias a la utilización de *annotations*, cada una con fines específicos. Todo este proceso de recolección de información se desarrolla gracias a una clase llamada *SpecterInitializer*, el cual, como su propio nombre indica, es la encargada de recolectar información referente a componentes que conformarán posteriormente lo que será el propio contexto de la aplicación, entre otras cosas. A continuación podemos ver un pequeño diagrama de flujo con los diferentes pasos de inicialización del contexto:



Como comentábamos anteriormente, una aplicación que haga uso de *specter* como *framework*, lo primero que deberá hacer es configurar la clase *SpecterInitializer*, la cual se trata simplemente de un *servlet* cuya función será la de preparar el contexto de ejecución de la aplicación y, posteriormente, gestionar las diferentes *urls* entrantes.

Iniciada la carga de la aplicación, el primer paso en el proceso de creación del contexto de la aplicación *(Paso 1*) es la recuperación de los diferentes parámetros que permite el proceso de inicialización. Entre ellos podemos encontrar:

- *viewLocationRootPath:* Parámetro que contendrá la ruta a partir de la cual el framework podrá localizar referencias a vistas (de momento solo acepta formato jsp) devueltas como resultado de la ejecución de un flujo en nuestra aplicación.

- *ApplicationResources:* Uno o más nombres de ficheros de recursos, separados por coma, disponibles en el *classpath* del sistema. A modo de referencia, la nomenclatura de los nombres proporcionados deberá ser la misma que la utilizada en la clase ResourceBundle
- *defaultLocale*: Idioma por defecto con el cual la aplicación será iniciada. La nomenclatura de la misma será del tipo: ***language_country_variant***

Una vez procesados dichos parámetros, y si fueran todos correctos, obtendríamos una instancia de la clase *SpecterContextConfig*, la cual nos permitirá continuar con los siguientes pasos que conforman el proceso de escaneado de componentes. Este proceso en cuestión será realizado por la clase *SpecterContextScanner*, gracias al uso de una librería llamada reflections*,* la cual nos permitirá llevar a cabo un análisis de meta-información en tiempo de ejecución sobre cada una de las clases que se encuentren disponibles en el *classpath* de la aplicación y así poder determinar la existencia de componentes que conformarán el contexto de ejecución.

Lo primero que se busca son los denominados *converters* (Paso 2), los cuales consisten en componentes encargados de transformar información entrante y proporcionar un resultado de un cierto tipo. Estos componentes son localizados gracias a la utilización de una *annotation* específica llamada *ConvertionRule* (analizaremos con más detalle su funcionamiento posteriormente). En el siguiente paso (Paso 3), el proceso es similar a los *converters*, con la diferencia que en este caso lo que estamos intentando localizar son aquellas clases marcados como *validators*, gracias a la utilización de una *annotation* específica llamada *ValidationRule, que, c*omo bien su nombre indica, se tratan de componentes encargados de validar información entrante y determinar si la misma es correcta y no según un conjunto de reglas establecidas, además de proporcionar algún mensaje descriptivo que ayude a entender el porque de dicha invalidez. Continuando con el reconocimiento del contexto de la aplicación, tenemos la recuperación de *controllers* (Paso 4), los cuales son los encargados de controlar el flujo de ejecución en caso de recibir una *request* entrante para ser gestionada. Adicionalmente, podemos decir que este tipo de componente se encuentran relacionados con los anteriormente localizados, dado que estos últimos serán los encargados de transformar y validar la información entrante, proveniente de una *request,* antes de ser gestionada por un *controller.* Finalmente, el último paso del proceso de escaneado es el de reconocimiento de las vistas disponibles (*Paso 5)*, el cual consiste en buscar en el punto de partida señalado por el parámetro *viewLocationRootPath* todas aquellas vistas disponibles e indexarlas para futuras referencias.

Una vez finalizado el proceso de escaneado del contexto, el último paso antes de dar por finalizado el proceso de reconocimiento es la creación del propio contexto (*Paso 6*), el cual consiste en la creación de una instancia de la clase *SpecterContext*, la cual contendrá toda la información recaudada hasta el momento para su futura disponibilidad y acceso en otros puntos de nuestro *framework*.

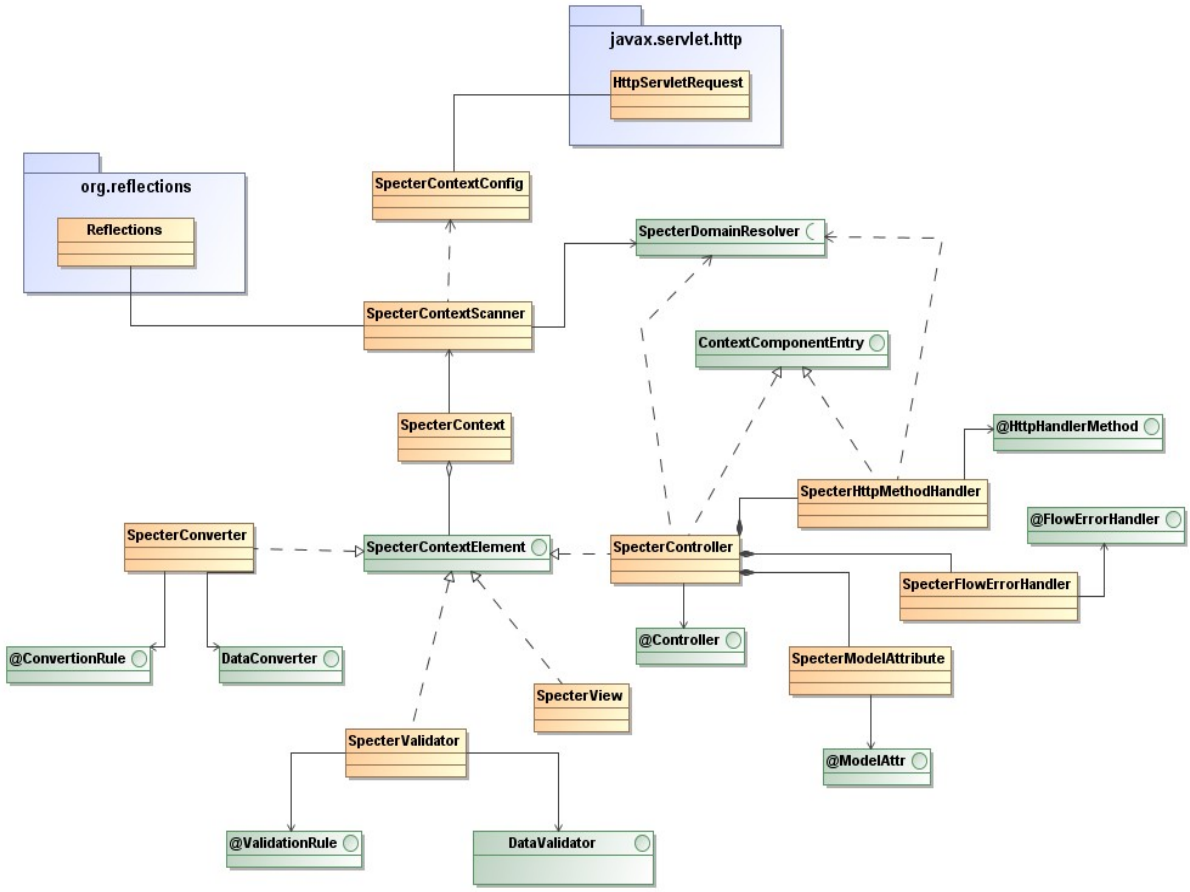
3.2 Dominio

Teniendo una visión general de que es lo que ocurre en el proceso de inicialización del contexto, podemos adentrarnos un poco más en la estructura interna del mismo. Como podemos observar, el resultado del proceso de inicialización del contexto consiste en una instancia de tipo *SpecterContext*. Este objeto se compondrá de elementos de tipo *SpecterContextElement*, entre los cuales podemos encontrar *SpecterConverter*, *SpecterValidator*, *SpecterView* y *SpecterController*, es decir, todos aquellos elementos que hemos mencionado en el apartado anterior. A excepción de componentes de tipo *SpecterView*, el resto de elementos se encuentran asociados a *annotations* que permitan su localización y detección de su naturaleza dentro del *classpath* la aplicación desarrollada. Eso no quita que algunos de estos elementos se apoyen en interfaces que dejen más clara las firmas de métodos que pueden ser implementados; tal es el caso de la interfaz *DataConverter*, la cual define que funcionalidades se espera que sean implementadas en el caso de *converters*, y tenemos también la interfaz *DataValidator*, la cual tiene el mismo fin que la anterior pero en este caso para los *validators*. Otras de las razones por las cuales he decidido definir dichas interfaces es que muchas veces me he encontrado en situaciones donde no tengo del todo claro que tipo de información puede recibir un método o que firma debería tener (concretamente con Spring 3), por lo que creo que, utilizando interfaces, estoy informado a los desarrolladores de cuales son las posibilidades de dichos elementos y que pueden esperar de ellos sin necesidad de tener que adivinar muchas veces cuales son las capacidades del *framework*.

Por otro lado, otra característica importante es la utilización de la interfaz *ContextComponentEntry* tanto por parte de *SpecterView* como de *SpecterController*. Su utilidad se verá posteriormente con mayor detalle, pero principalmente lo que nos permite dicha interfaz es poder utilizar los elementos que la implementen como entradas en un índice sobre el propio contexto definido, el cual será consultado a la hora de determinar como proceder en la gestión de una *url* entrante.

Durante el proceso de configuración de un *controller*, entre los pasos implicados, uno de ellos consiste en el reconocimiento de *converters* y *validators* que deben ser utilizados durante el

procedimiento de preparación de un *controller* para su uso. Es por ello que, a medida que es leída la meta-información contenida en la clase relacionada al *controller*, es importante poder contar con acceso a la información recaudada en relación a *converters y validators* en pasos anteriores y así de esta manera poder validar la existencia de los elementos indicados como necesarios mediante *annotations* y verificar su correcta utilización. Para tal fin, se ha definido una interfaz de tipo *SpecterDomainResolver*, la cual permitirá principalmente consultar la información recogida por el *SpecterContextScanner* por el propio método de creación de un *controller* y así poder verificar la existencia de los componentes referenciados según las *annotations* utilizadas, sin necesidad de conocer la estructura o implementación de como se almacena la información recogida en el proceso de reconocimiento de *converters* y *validators*.

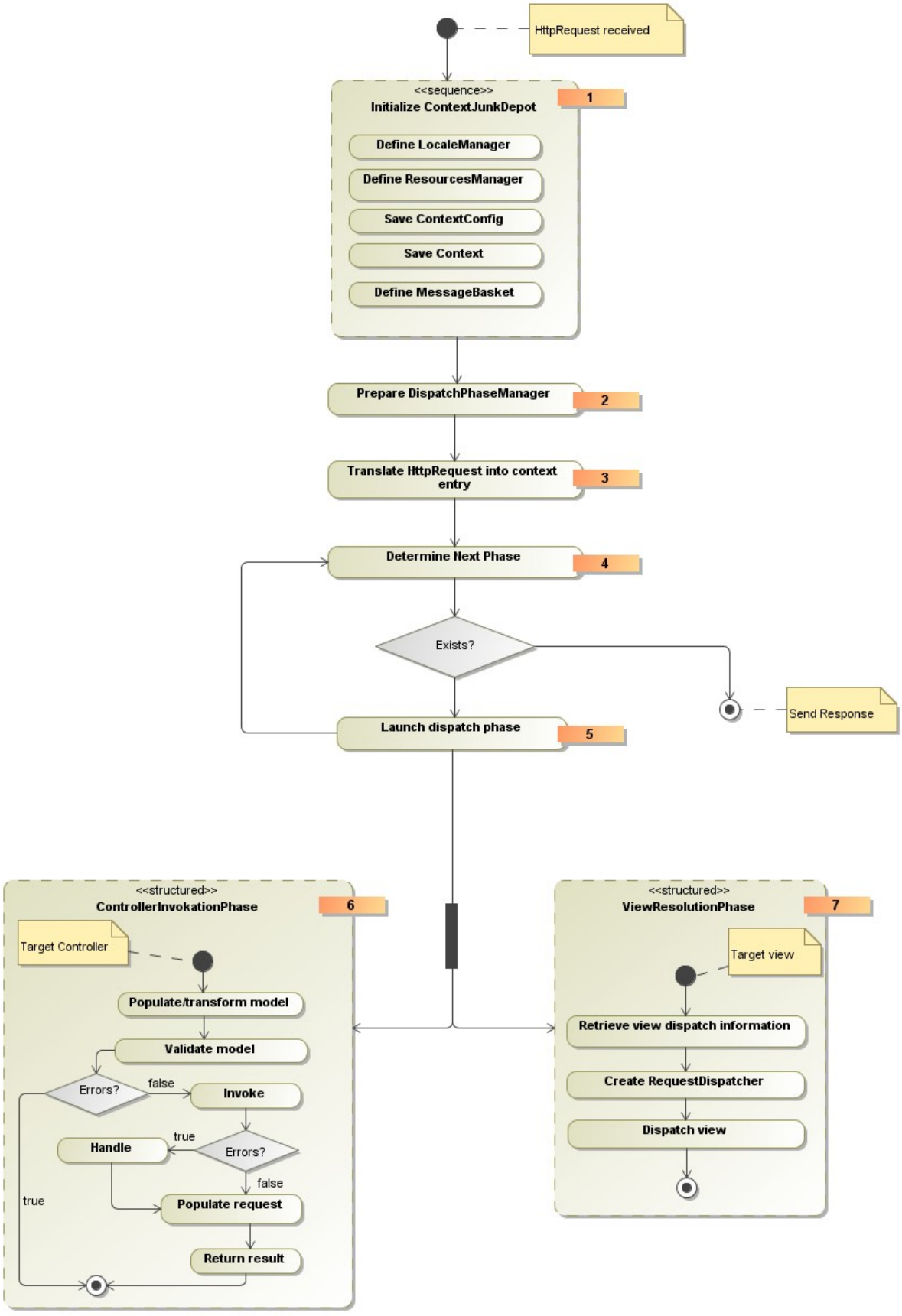


De esta manera, a medida que vamos definiendo un *controller*, se va recuperando información relacionada a métodos encargados de gestionar *urls* entrantes (*SpecterHttpMethodHandler)*, atributos que conforman el modelo de la aplicación (*SpecterModelAttribute)*, provenientes tanto de la propia *request o session* en forma de atributos o que simplemente deberán ser inicializados o definidos por el propio método de gestión, o métodos encargados de gestionar aquellas incidencias que puedan llegar a ocurrir durante el procesamiento de una *request* entrante (*SpecterFlowErrorHandler)*. También podemos ver en es caso que cada uno de los elementos que componen un *controller* también tiene asociados una *annotation* específica que define la naturaleza y finalidad del elemento indicado.

Como se puede ver luego de esta explicación, cada uno de los componentes que conforman el contexto están anotados de forma específica, además de poseer relaciones y roles bien definidos, que nos permiten movernos y utilizarlos de forma clara y sencilla. Ahora bien, una vez conocido el contenido y relaciones del contexto, podemos pasar a ver que ocurre cuando recibimos una *request* entrante a procesar.

3.3 Dispatching process

Una vez el contexto de la aplicación inicializado, el *framework* ya esta listo para el tratamiento de *request* entrantes. Ente proceso consiste en una serie de pasos que procederemos a detallar a continuación a partir del siguiente gráfico:



Cuando el objeto *SpecterInicializer* recibe una *request* entrante, el primer paso en el proceso de tratamiento (*Paso 1*) consiste en la creación e inicialización de un objeto de tipo *ContextJunkDepot.* Este objeto tiene como finalidad proporcionar diferentes tipos de objetos de forma *transversal* al flujo de ejecución, con esto queremos decir que, por ejemplo, si quisiéramos acceder a la instancia del *SpecterContext* existente, no será necesario pasar dicha instancia como parámetro a cada objeto que lo requiera, sino que con solo utilizar el *ContextJunkDepot* este se encargará de recuperarlo por él. Ahora bien, este objeto solo será posible recuperarlo siempre y cuando se cuente con acceso al objeto *request* gestionado, dado que, principalmente, su implementación se basa en el uso de una clase llamada ThreadLocal, la cual, como se argumenta en su especificación, nos permitirá almacenar copias de objetos que vivirán tanto como el *thread* sobre el cual esta siendo ejecutado. De esta manera, lo que conseguimos es guardar recursos en el *thread* activo que serán accesibles por todas los objetos intervinientes y sin preocuparnos de lo que puede llegar a ocurrir en otros *threads* paralelos, almacenando nuestra instancia de *ContextJunkDepot* en la *request* procesada. Los recursos proporcionados a partir de este objeto son los siguientes:

- *SpecterLocaleManager*: Objeto encargado de la gestión del idioma utilizado en la recuperación y manipulación de recursos de la aplicación. En caso que no sea especificado un *locale* inicial se recogerá el *locale* del sistema.
- *SpecterResourceManager*: Objeto encargado de recuperar recursos almacenados en ficheros de propiedades, configurados mediante la utilización del parámetro de inicio *ApplicationResources*, según el valor actual del *locale* configurado en la aplicación. Cualquier cambio aplicado en el idioma del sistema será automáticamente notificado a dicha instancia.
- *SpecterContextConfig:* Objeto contenedor de configuración de inicio del sistema.
- *SpecterContext*: Objeto contenedor de información relacionada a componentes escaneados durante el inicio del sistema: *controllers, converters, validators y views*. Este a la vez proporciona funcionalidades de búsqueda (a modo de índice) de componentes según sintaxis predefinida.
- *SpecterMessageBasket*: Objeto utilizado para el registro y gestión de errores ocurridos durante el flujo de tratamiento de *request*.

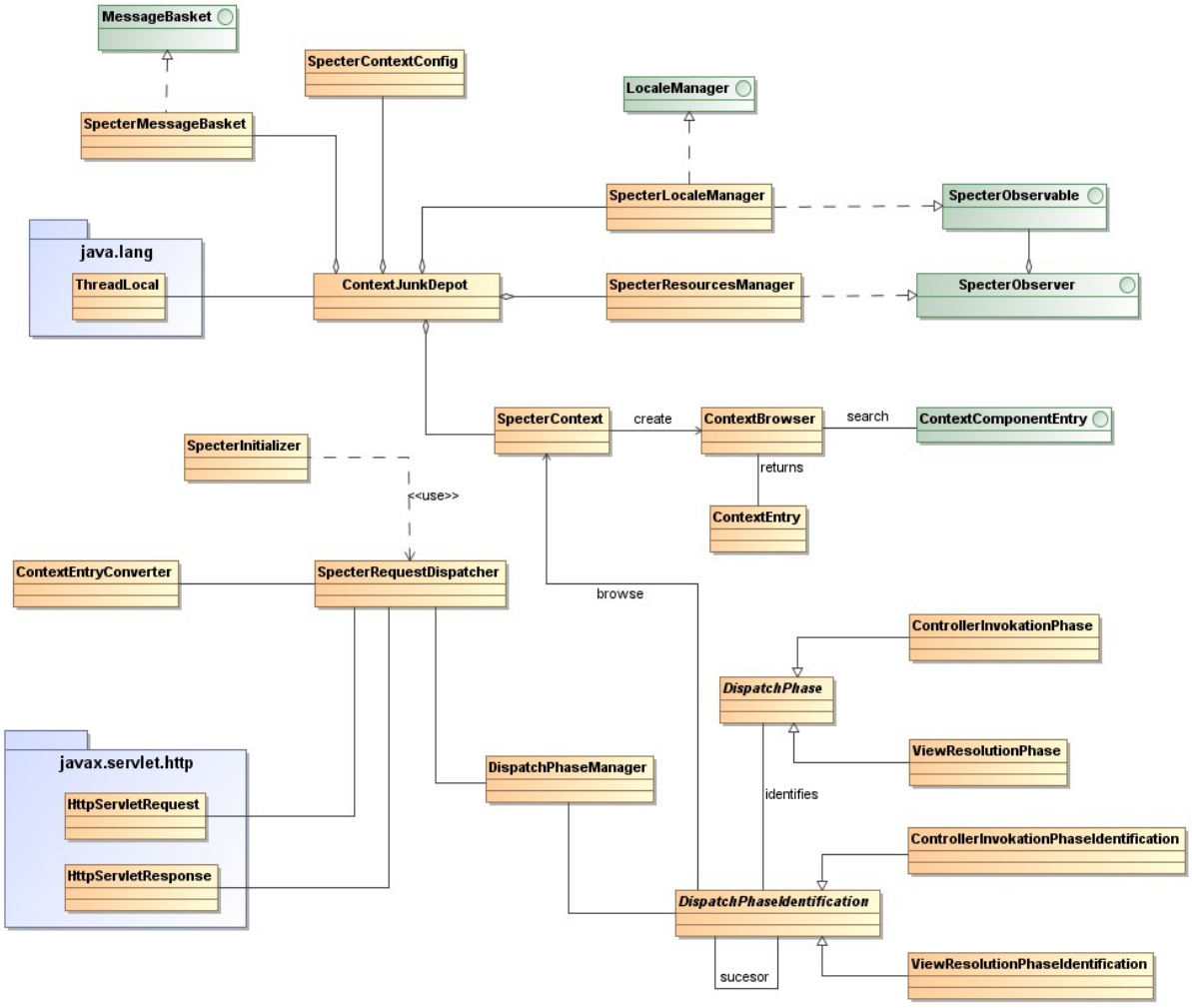
Como podemos ver, el conjunto de objetos disponibles durante el flujo de ejecución es amplio y cubre los diferentes aspectos del *framework*, lo que nos aporta una cierta flexibilidad en la definición de las diferentes partes del mismo y nos permite focalizarnos en las funcionalidades concretas. Una vez preparado el *ContextJunkDepot*, ya podemos continuar con el proceso de gestión de *request,* para lo que procedemos con la creación el objeto diseñado para tal fin: *SpecterRequestDispatcher* (Paso 2).

Este objeto tiene como objetivo principal la gestión del proceso de traducción entre la *request* entrante y las diferentes fases de gestión involucradas. Es por ello que comenzamos por la preparación del *dispatcher* mediante la utilización del *SpecterContext*, que como hemos mencionado anteriormente, es el objeto que contiene toda la información relacionada a los componentes detectado durante el inicio del sistema. Una vez preparados, lo siguiente a hacer consiste en la traducción de la *request* entrante en un valor o *entry* que el *framework* sea capaz de comprender y con la cual trabajar (*Paso 2)*. Para ello, previamente a cualquier proceso de detección de pasos a seguir, lo primero que se hace es procesar la *request* por medio de un objeto llamado *ContextEntryConverter,* cuya función principal es la de transformar el valor de la *url* en una cadena que nos sirva como *entry* de búsqueda dentro del índice que contiene el contexto de la aplicación, analizando y traduciendo sus diferentes partes en fases de ejecucución. Es por ello que, en el paso 3, podemos observar como entramos en un bucle de búsqueda de fases, la cual consiste en ir extrayendo información de la *entry* generada anteriormente e ir disgregando partes de la misma, las cuales se irán traduciendo en una o mas *DispathPhase* a ejecutar hasta finalizar el proceso de *dispatch* y devolver una respuesta al cliente. Por ejemplo, suponiendo que tengamos una *url* del tipo http://context/users/create.html, los pasos podrían ser los siguientes:

1. Utilizamos *ContextEntryConverter*, teniendo como resultado: *entry:* users/create;
2. Detectamos siguiente fase:
    1. *entry*: users/create; *match:* users; *type*: C*ontroller*; *remaining:* create
    2. *entry:* create; *match*: create; *type*: *HttpMethoHandler;* remaining*:* -
3. Devolvemos fase: *ControllerInvokationPhase*

4. Ejecutamos fase y obtenemos como resultado '*users/createFom*'
5. Detectamos siguiente fase:
    1. entry: users/createFom; match: users/createForm; type: View; remaining: -
6. Devolvemos fase: ViewInvokationPhase.
7. Ejecutamos fase y obtenemos como resultado ''.
8. Finalizamos proceso dispatch.

Durante el proceso de ejecución de fases podemos ver que este no finaliza sino hasta que no quede más nada que hacer, lo que es indicado por medio de una cadena vacía, punto en el cual el proceso de traducción finalizaría con la devolución de una respuesta al cliente. Ahora bien, entrando un poco más en detalle sobre el proceso que nos ocupa, este es llevado a cabo por un conjunto de clases de tipo *DispatchPhaseIdentification*, relacionadas mediante el patrón *chain of responsability,* lo que nos proporciona una cierta flexibilidad si en un futuro necesitáramos incorporar nuevas fases en el proceso. Cada objeto encargado de la identificación de una fase esta relacionada con la fase en si, permitiéndonos especializar la lógica de identificación de cada fases pero a la vez contando con la suficiente generalización en el proceso utilizado en la identificación/ejecución. La capacidad de generalización es proporcionada por las clases *DispatchPhase y DispathPhaseIdentification*, mencionadas anteriormente, y sobre todo por las clases *ContextBrowser y ContextEntry*, las cuales posibilitan la navegación del contexto de forma flexible en búsqueda de componentes que coincidan con las entradas proporcionadas. Para mayor claridad en las relaciones podemos observar el siguiente gráfico:



Si revisamos el gráfico del apartado 3.2, veremos que existen dos elementos que implementan la interfaz *ContextComponentEntry.* Esta interfaz permite la búsqueda de componentes registrados en el contexto de la aplicación siguiendo las reglas de búsqueda antes mencionadas, es decir, utilizando una jerarquía de componentes en base a una sintaxis típica de *url*. De esta manera, por ejemplo, la clase *ControllerInvokationPhaseIdentification* es capaz de buscar *controllers* y *handlers* aplicando el mismo método, lo mismo podríamos haber aplicado en búsquedas de *views*, pero dada la simplicidad con la que indexan en el sistema, considero que para esta versión no se justificaba la utilización de la interfaz antes mencionada.

Una vez detectada cual es la fase que debemos ejecutar, procedemos con su ejecución. Actualmente, contamos con dos tipos de fases: una fase relacionada con la manipulación de una *request* entrante mediante un método de gestión definido en un *controller (Paso 6)* o simplemente la localización y redireccionamiento del control hacia una vista *(Paso 7).*

En posteriores secciones trataremos con más detalle el funcionamiento de cada una de ellas, pero a modo de introducción, cuando se trata de una fase de ejecución de invocación de un *controller,* lo primero que hacemos es recuperar el modelo persistido, ya sea en forma de atributos de *request* o en la propia sesión del usuario. Durante el proceso de restauración del modelo, cada valor es sometido por un proceso de transformación según el tipo de dato definido en el atributo destino, punto en el cual intervienen componentes mencionados anteriormente: *converters*. El sistema cuenta con un conjunto de *converteres* específico para cada tipo de datos existente en Java, pero también existe la posibilidad de registrar nuestros propios *converters*, en caso que necesitemos. Una vez restaurado el modelo, el siguiente paso es su validación, la cual difiere de la validación implícita que implica su transformación al tipo de dato correcto en que aquí tenemos la posibilidad de aplicar reglas de validación según reglas de negocios u otros aspectos no cubiertos por el mero proceso de transformación. Estas tareas son llevadas a cabo por componentes de tipo *validators*, los cuales tienen la responsabilidad de validar los contenidos para los que fueron asignados y devolver mensajes en caso que encuentren algún problema durante su ejecución, condicionando el flujo de ejecución en relación a si continuamos o no con la invocación del *controller*. Una vez superadas todas las reglas de validación definidas, procedemos finalmente con la invocación del método de gestión de la *request* entrante. Ahora bien, este tipo de métodos siempre deben retornar algún valor, el cual determinará el camino que debemos seguir en el tratamiento de la *request* entrante, a no ser que durante su ejecución hayan ocurrido problemas o errores que deberían ser tratados. Justamente para este fin se definen un conjunto de componentes de tipo *FlowErrorHandler*, los cuales determinarán que debe hacer el *framework* cuando se produce un error de ejecución. Independientemente de lo ocurrido durante el procesamiento de la *request*, una vez finalizada su ejecución, el siguiente paso es repoblar la *request* y la sesión con el estado actual del modelo permitiendo que posteriores fases de ejecución cuenten con toda la información del mismo para su propia ejecución. Finalmente, la fase retorna un resultado final a partir del cual se continuará con la resolución de fases de ejecución.

En el caso de una fase de invocación de una vista, esta resulta mas sencilla, dado que simplemente el flujo de la misma consiste en detectar que vista es la que debemos procesar a partir del valor o *entry* proporcionado. Una vez encontrada, debemos preparar el entorno de *dispatch* para entregar el control a la vista en cuestión y así devolver una respuesta al cliente.

Hasta aquí una visión general de que es lo que ocurre una una *request* del cliente es recibida por el *framework*, hemos vista que, en resumen, la misma puede ser traducida en la invocación de un *controller* o de una vista y es por ello que es momento de indagar un poco más en cada una de estas fases.

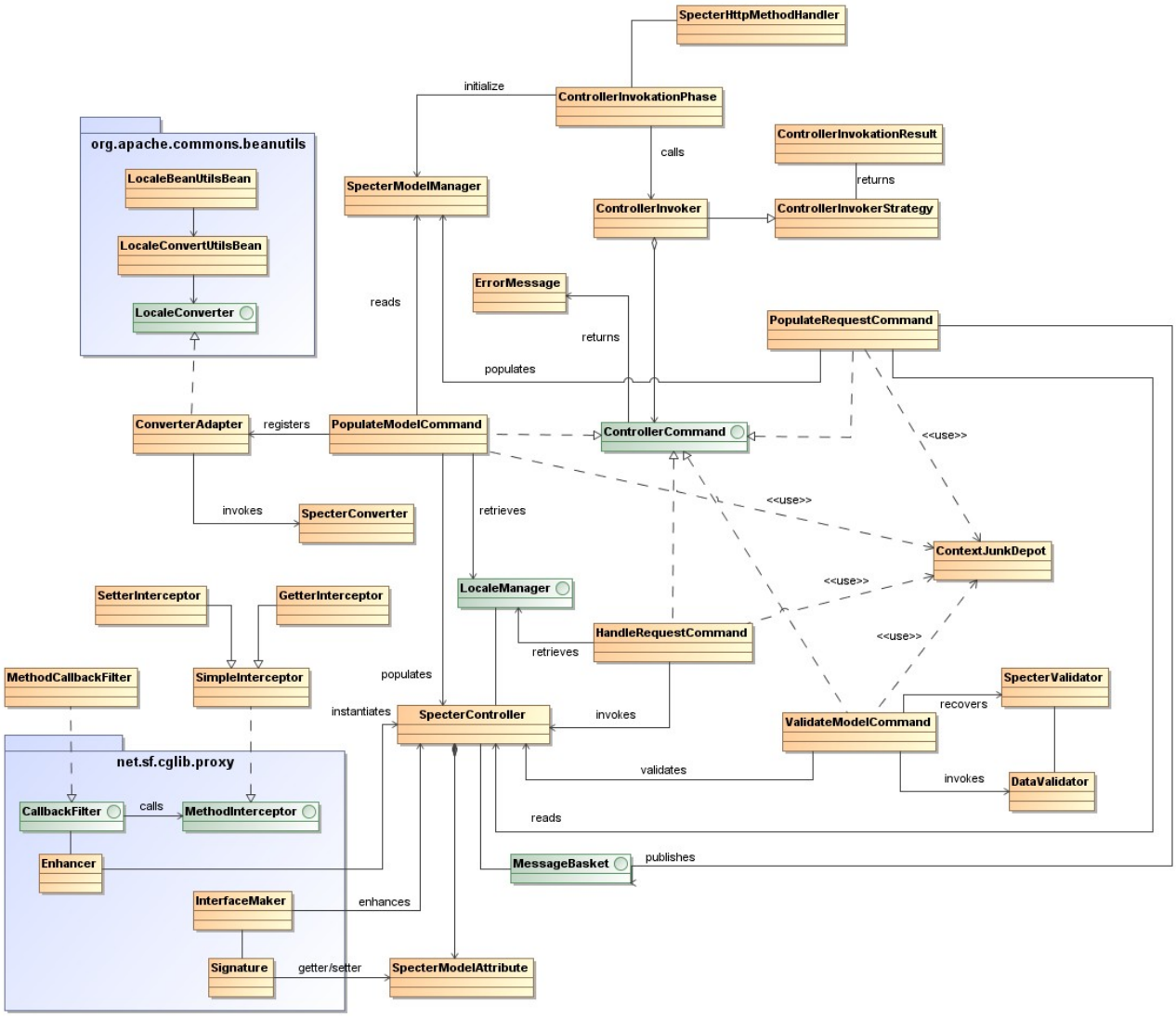
3.4 Controller invokation process

Como hemos mencionado en el apartado anterior, una de las posibles fases puede implicar la ejecución de un método de gestión de *urls* (*handler)* perteneciente a un *controller* concreto. Para ello, se ha definido una clase llamada *ControllerInvokationPhase*, la cual se encargará de preparar y validar el contexto necesario para la ejecución del *handler* correspondiente, además de llevar a cabo la invocación misma del *controller*. Es por ello que, para poder concretar tal objetivo, en el momento de su creación, esta clase recibirá toda la información del *handler* a invocar mediante una instancia de tipo *SpecterHttpMethodHandler*, la cual proporcionará información tanto del método que debe ser invocado así como del contexto que el mismo necesitará para concretar su ejecución.

Antes de dar comienzo a la invocación, se crea una instancia de tipo *SpecterModelManager*, la cual permitirá acceder o manipular el modelo de forma transparente y transversal al flujo de ejecución, dado que la misma será almacenada dentro del *ContextJunkDepot*. Una vez preparado el modelo, creamos un objeto de tipo *ControllerInvoker* e iniciamos la invocación. Esta invocación esta conformada por un conjunto de pasos que definen una estrategia clara a seguir por este o cualquier otro objeto que quiera cambiar el comportamiento del proceso de invocación. La misma se puede observar en la clase *ControllerInvokerStrategy*, donde los pasos seguidos son los siguientes:

1. *populate*: Consiste en la recuperación de todos los atributos existentes tanto en la *request* como en *sesión* y asignar los necesarios por el *controller* para llevar a cabo la invocación. En el proceso de asignación, se procede con el primer paso de validación a nivel de tipos, dado que los valores entrantes, en forma de cadenas, deberán ser asignadas a atributos en el controlados que puede ser de cualquier tipo, lo que puede llevar a la ocurrencia de incidencias durante el proceso de transformación.
2. *validate*: Una vez cargado el modelo, el siguiente paso es la validación del mismo según reglas definidas a nivel de *controller*, donde, en caso de incumplimiento de algunas de esas reglas, el flujo de invocación se verá interrumpido y de devolverán mensajes descriptivos de los problemas encontrados.
3. *Invoke*: En caso que todos los pasos anteriores hayan sido superados correctamente, el siguiente paso consiste en la invocación del *handler* en cuestión, a la espera de un resultado que indique cual es el paso a seguir en la ejecución.
4. *handle errors*: Si ocurriera algún problema en la ejecución de cualquiera de los pasos anteriores, y si el *controller* contara con métodos anotados como *error handlers*, este paso intentaría invocar dicho método para permitir la gestión de los problemas encontrados y esperar instrucciones sobre cual es el paso a seguir en dicha situación.
5. *populate request*: Tanto en caso de error como en una ejecución correcta, el último paso consiste en repoblar o sincronizar es estado actual del modelo con el objeto *SpecterModelManager*, para de esta manera permitir a procesos posteriores contar con el último estado del mismo.
6. *return result*: El último paso simplemente consiste en informar si han ocurrido o no problemas durante la ejecución y cual es el paso siguiente a seguir.

Cada uno de estos pasos se encuentran especificados y materializados en un conjunto de objetos de tipo *command,* tal como se detalla a continuación:



Cada uno de estos objetos *command* definen la lógica a seguir en cada uno de estos pasos, teniendo como "director de orquesta" al objeto *ControllerInvoker*, el cual extiende de la clase *ControlerInvokerStrategy*. La utilización de objetos *command* en este caso nos aporta flexibilidad en la implementación de cada uno de los pasos, dado que estamos abriendo la posibilidad de definir otros tipos de procesos de invocación con variantes en su funcionamiento interno, pero sin necesidad de tener que salirse del esquema principal de invocación. Incluso sería posible ampliar la estrategia a seguir reutilizando la implementación de muchos pasos ya definidos. Entre los pasos de tratamiento de una *request* que han sido materializados tenemos:

1 PopulateModelCommand

El objetivo principal de esta etapa es la de recuperar valores guardados como atributos, ya sea en sesión o en la propia *request,* y asignarlos a una instancia del *controller* en cuestión para que el mismo cuente con toda la información requerida en el tratamiento de la *request* entrante. Este procedimiento tendrá en cuenta aquellos atributos del *controller* que hayan sido marcados como atributos del propio modelo mediante la utilización de la *annotation* @*ModelAttr*, donde también es posible definir el alcance de dicho atributo, lo que quiere decir que el mismo puede estar alojado en la *request* o en la sesión activa. El proceso de asignación/recuperación de valores de atributos es llevado a cabo por una librería llamada *Commons BeanUtils,* la cual nos permite el acceso dinámico a propiedades, aprovechando el patrón preestablecido en los nombres utilizados para propiedades de tipo *getter* y *setter*, junto con la potencia de *Reflection* e *Introspection*. Ahora bien, para poder hacer uso correcto de esta librería, y dado que nos interesaba establecer un proceso totalmente dinámico de lectura y asignación de atributos simplemente mediante la utilización de la *annotation @ModelAttr*, nos encontramos con el dilema de como crear métodos dinámicos de tipo *getXXX() y setXXX()* que permitan a *BeanUtils* hacer su trabajo y a la vez reducir el número de líneas a escribir en la clase *controller* con métodos tan "tontos" como los mencionados anteriormente. La solución fue utilizar una librería que nos permita la manipulación y creación de clases de forma dinámica que tengan la capacidad de manipular *bytecode*, y aquí es donde entra en juego *Cglib*. Se trata de una librería muy potente que posibilita la generación de código dinámico en tiempo de ejecución, entre muchas otras cosas, pero que nos ha permitido concretamente crear una interfaz dinámica en tiempo de ejecución que contendrá todos los métodos *getter* y *setter* necesarios, según se hayan anotados como atributos del modelo, para luego hacer que la clase del *controller* utilizada implemente dicha interfaz de forma dinámica antes de crear una instancia de la misma y así, de esta manera, permitir a *BeanUtils* acceder a los atributos señalados mediante la *annotation @ModelAttr* de forma transparente. De esta forma, habiendo integrado completamente *BeanUtils* en nuestra lógica, el siguiente paso consiste en informar a *BeanUtils* de aquellos *converters* que hayan sido especificados por el desarrollador mediante la *annotation @ConvertionRule*. Originalmente, *BeanUtils* proporciona una considerable cantidad de conversores que tiene la capacidad de transformar cadenas entrantes en tipos conocidos de Java, pero, como hemos mencionado en puntos anteriores, el desarrollador tiene la posibilidad de registrar sus propios *converters*. Este registro de conversores se realiza previamente al proceso de *populate* de valores mediante la clase *LocaleConverterUtilsBean*, donde, como se puede observar en el nombre, el método de conversión será siempre sensible al *locale* establecido (concretamente al recuperado mediante el *SpecterLocaleManager*), lo que puede ser útil en caso de tener que convertir fechas o números, entre otras cosas. A modo de observación, y dado a una limitación de diseño, los *converters* se registran solo a nivel de clase y por tipo destino, lo que quiere decir que solo podemos tener un *converter* de un cierto tipo registrado por clase, lo que imposibilita la aplicación de diferentes *converters* del mismo tipo sobre diferentes atributos del *controller* y explotar así posibilidades de personalización de ejecución mediante el uso de parámetros en las propias *annotations.* Entonces, para registrar los *converters* definidos por el desarrollador, haremos uso de una clase llamada ConverterAdapter, la cual, como su nombre indica, nos permitirá adaptar nuestra implementación de *converter* a la interfaz esperada por la propia librería, logrando ser totalmente transparente la integración de nuestros conversores por los esperados. Una vez registrados todos los *converters* personalizados, es lanzado el proceso de *populate* de atributos de la *request* y luego de la sesión, recuperados mediante *SpecterModelManager,* para finalmente entregar el control al siguiente *command* en el proceso de tratamiento.

2 ValidateModelCommand

Una vez el modelo asignado en cada uno de los atributos marcados como tal en la instancia del *controller*, el siguiente paso consiste en la validación del contenido recuperado según un conjunto de reglas adicionales que comprobarán los valores de forma semántica al contexto de la aplicación. Este tipo de validaciones están relacionadas con los elementos de tipo *validator*, los cuales, como se ha explicado anteriormente, deben ser definidos por el desarrollador y registrados mediante la *annotation @ValidationRule*, leída en tiempo de ejecución. Cada *validator* definido deberá estar asociado a una *annotation* personalizada y a un tipo, donde la *annotation* será utilizada en tiempo de ejecución para indicar al *framework* cuando deberá aplicar un *validator* concreto, teniendo

obviamente en cuenta el tipo de dato asociado. A diferencia de los *converters*, los *validators* sí pueden ser aplicados a nivel de atributo, lo que permite la personalización de la validación en cada atributo a validar gracias a la potencia de nos proporcionan las *annotations* con sus parámetros, aunque esta opción queda abierta al desarrollador de turno. Adicionalmente, cada clase *validator* deberá implementar la interfaz *DataValidator*, principalmente por mi interés de dejar clara cual es la interacción e información proporcionada por el *framework* cuando el mismo es ejecutado. Posteriormente veremos con mayor detalle como definir nuestros propios *validators*, pero a modo general, con estos requisitos básicos, tendremos suficiente para comenzar a definir *validators* totalmente personalizados.

Entonces, una vez iniciado el *command*, lo primero que se hace es recorrer los atributos del modelo (*controller)* y verificar cuales de ellos necesitan ser validados y quien será el responsable de llevarlo a cabo. Para ello, cuando se detecta un atributo con validación asignada, utilizamos el *SpecterContext* para buscar si el *validator* se encuentra registrado. Si fuera el caso, procedemos a crear una instancia del mismo y procedemos con su ejecución según la firma definida en *DataValidator*. En caso que la validación fuera correcta, el proceso continuará con la siguiente regla de validación indicada o con el siguiente atributo si fuera el caso, pero en caso que el resultado fuera erróneo, procedemos a recoger los mensajes de error registrados por el *validador,* para posteriormente ser recogidos y determinar si debemos o no continuar con el proceso de tratamiento de la *request*, y continuamos con la ejecución de la siguiente regla, hasta finalizar con el proceso de validación. Una vez validados todos los atributos del modelo, finalizamos nuestra tarea devolviendo el control al siguiente *command* en el proceso de tratamiento.

3 HandleRequestCommand

En caso que tanto los proceso de conversión y validación hayan resultado correctos, finalmente llegamos a la etapa de invocación del método encargado de la gestión de la *request* entrante contenido en el *controller* inicial. Antes de entregar el control al *handler*, verificamos si el *controller* contiene un atributo marcado como *@ApplicationLocale*, el cual indicará al *framework* la necesidad de asignar el *locale* activo actualmente en el sistema, ya sea para modificar el idioma actual, lo que lanzaría de forma totalmente transparente una recarga de los *resources* contenidos en el *SpecterResourceManager*, o si se quisiera utilizar dicho valor para formato de fechas o números o recuperación de recursos no gestionados por la aplicación. Posteriormente, teniendo todo el contexto establecido, inclusive una instancia de *MessageBasket* personalizada para el *controller* mediante la *annotation @Messages*, se procede a entregar el control al método *handler* y a esperar una respuesta del mismo. Finalizada su ejecución, lo primero que se hace es verificar si han ocurrido problemas durante su ejecución mediante la existencia de mensajes de error. Si fuera el caso o hubiera tenido lugar alguna excepción inesperada durante la ejecución, se recuperarían todos mensajes de error generados y activaríamos el *flag* específico para indicar que "las cosas no han terminado como se esperaba". En cualquier caso, una vez finalizada la ejecución del *handler*, procedemos a entregar el control al siguiente *command* en el proceso de tratamiento de la *request* entrante. Cabe destacar que, cuando desde este *command* se indica que han ocurrido problemas durante su ejecución, la clase *ControllerInvoker* intentará entregar ejecutar algún método marcado como *@FlowErrorHandler* para determinar cual debe ser el paso a seguir dada la existencia de errores de ejecución, ya que, en caso que todo haya salido correctamente, se espera que el método *handler* retorne un valor indicativo que muestre cual será el siguiente paso en la ejecución del flujo de tratamiento. Veremos más sobre *controllers* en secciones posteriores.

4 PopulateRequestCommand

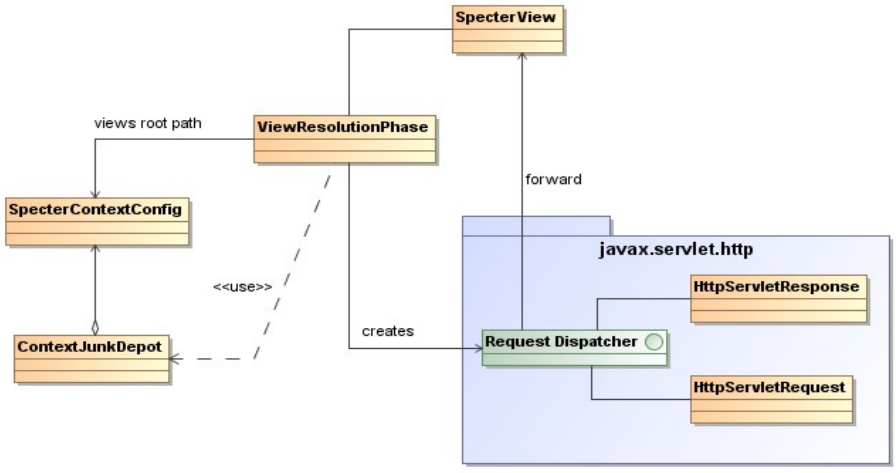
Finalmente, ya sea en caso de error o de éxito en el tratamiento de la *request*, el último *command* se encargará simplemente de repoblar la *request* o sesión según corresponda con los actuales valores del modelo, para que los mismos estén disponibles en futuros procesos de gestión de *request* o para la elaboración de la respuesta final al cliente. A diferencia de *PopulateModelCommand*, aquí tenemos un paso adicional que es la publicación también de aquellos mensajes de error recogidos durante la ejecución, para que, si existieran procesos de tratamiento de errores posteriores, ya sea en forma de mensajes o en la elaboración de una vista de cliente, estos puedan contar y gestionar los resultados obtenidos. Justamente para este fin se han creado un conjunto de *taglibs* de ayuda o asistencia al desarrollador que permitan acceder a estar características de forma transparente sin necesidad de conocer detalles de implementación internos. Veremos más detalles en secciones posteriores.

Como se puede observar, cada uno de los *commands* antes especificados tienen una tarea muy concreta definida, además de estar colocados según un orden específico pero sin relación directa entre ellos, lo que nos proporciona capacidad para alterar ordenes de ejecución o incorporar nuevos pasos sin en el proceso sin verse afectados estructuralmente de ninguna forma concreta. Ahora bien,

como se puede observar en el último gráfico presentado, el único punto común con el que cuenta todos ellos es la utilización del objeto *ContextJunkDepot*, el cual, como se ha presentado anteriormente, posee información sobre el contexto tanto a nivel de *framework* como de la ejecución que tiene lugar, lo que significa que todos los objetos *command* dispondrán de toda esta información en todo momento, además de contar con la posibilidad de poder incorporar nuevas características en el futuro y así y todo evitar modificar la definición de cualquiera de ellos, teniendo un campo de acción totalmente *transveral* a la arquitectura, lo que nos aporta muchas ventajas y seguramente algunos inconvenientes.

3.5 View Invokation process

Anteriormente hemos visto que unas de las posibles fases en el flujo de tratamiento de una *request* puede estar relacionada con la invocación de un método de gestión de *request* contenido en un *controller.* Otra de las posibles fases que pueden ser ejecutada en este flujo esta relacionada con *views*: *ViewResolutionPhase*. Dentro del propio *framework,* como es posible deducir a partir del nombre, *s*u principal cometido es la de resolver y redireccionar el flujo de ejecución hacia una vista definida por el desarrollador, la cual, por cuestiones de tiempo, ha sido limitada sólo a aquellas en formato JSP. La vista solicitada, como hemos mencionado en anteriores secciones, es localizada a partir del valor especificado mediante el parámetro de inicio *viewLocationRootPath*, punto de partida a partir del cual el *framework* genera su propio índice con todas aquellas vista definidas. Ahora bien, una vez iniciada esta fase, lo primero que se hace es recuperar la instancia del objeto *SpecterContextConfig* (haciendo uso de *ContextJunkDepot)* para luego, en conjunto con la instancia de la clase *SpecterView,* ser capaz de resolver, de forma relativa a la ubicación de la aplicación, la localización del fichero físico y así proceder a la ejecución de un *forward* mediante el uso de una instancia de tipo *RequestDispatcher.* Una vez finalizado el proceso de *forward,* retornamos el control al proceso de *dispatch* con un valor de retorno vacío, lo cual indicará al *framework* que el tratamiento a finalizado. A modo de aclaración, aquí tenemos un pequeño gráfico de las clases implicadas durante la ejecución de esta fase:



3.6 Custom Converters

Anteriormente hemos hablado que unas de las características del *framework* es la posibilidad de definir clases encargadas de transformar una cadena entrante en un tipo de dato destino. A este tipo de componentes se denominan *converters*, los cuales entrarán en acción principalmente en el proceso de *populate* de atributos provenientes de la *request* entrantes o de la sesión activa hacia un componente de tipo *controller*. Cada atributo del modelo, anotado adecuadamente mediante la *annotation @ModelAttr*, estará forzosamente relacionado a un tipo de dato o clase específica. El proceso de *populate*, el cual tiene lugar durante la ejecución del objeto *command* *PopulateModelCommand*, será el encargado, entre otras cosas, de comprobar cuales son los *converters* registrados en el *controller* de turno para luego informar a la herramienta de *populate* (*BeanUtils)* que los mismos deberán ser invocados durante el proceso de transformación de tipos.

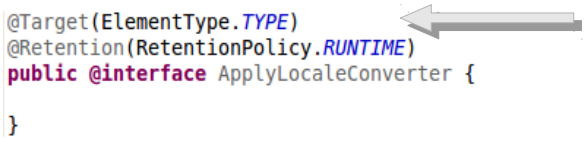
Actualmente *BeanUtils* proporciona implementaciones de *converters* para los siguientes tipos destino:

- java.lang.BigDecimal
- java.lang.BigInteger
- byte and java.lang.Byte
- double and java.lang.Double
- float and java.lang.Float
- int and java.lang.Integer
- long and java.lang.Long
- short and java.lang.Short
- java.lang.String
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

Como se puede observar, la librería cubre una gran parte de tipos utilizados por Java, y teniendo en cuenta que las clases utilizadas de *BeanUtils* son sensibles al *locale* especificado (el cual siempre será el idioma activo en el *framework)* tenemos un conjunto de *converters* bastante interesantes para comenzar a trabajar. Ahora bien, seguramente nos encontraremos con el caso en el que necesitaremos ampliar el número de *converters* disponibles, ya sea por conseguir un comportamiento distinto al proporcionado originalmente o simplemente para contar con *converters* hacia tipos de dato personalizados, podemos hacerlo, pero para ello, hay que tener en cuenta dos restricciones:

1. Puede existir un único *converter* por tipo de dato, donde si registráramos un *converter* cuyo tipo destino coincide con otro *converter* ya registrado, el primero reemplazará al segundo.
2. Cada *converter* es registrado a nivel de *controller*, lo que quiere decir que si definimos una *annotation* a nivel de atributo, el *framework* la detectará. La *annotation* asociada deberá ser definida para ser utilizada a nivel de clase (@Target=ElementType.TYPE).

Para definir un *converter* personalizado, lo primero que debemos hacer es diseñar y crear la *annotation* que utilizaremos para referirnos a nuestro *converter.* A modo de ejemplo, supongamos que tenemos la siguiente definición:



```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ApplyLocaleConverter {

}
```

Importante ElementType.TYPE

Como se puede ver en la gráfica, es importante recordar (más por una cuestión de ahorrar futuras confusiones) que las *annotations* que sean creadas deberán ser definidas para ser utilizar a nivel de clase, correspondiéndose con la restricciones de uso antes mencionada. Por otro lado, aunque no se muestre en el ejemplo, aquí tenemos la posibilidad de especificar todos los atributos que queramos, dado que la instancia de la *annotation* especificada en el *controller* será pasada como parámetro al método de conversión definido, abriendo la posibilidad de proporcionar algo de información adicional que pueda ser de ayuda en su ejecución, así como definir la política retención de la *annotation* a gusto del desarrollador, siempre y cuando la misma sea accesible en tiempo de ejecución, momento en el cual se comprueba los *controllers* definidos en búsqueda de este tipo de *annotation.*

Una vez definida la *annotation*, el siguiente paso es la creación del propio *converter*. Para ello, lo primero que debemos hacer es crear una clase que implemente la interfaz *DataConverter*, la cual podemos ver a continuación:

```
public interface DataConverter<E> {
    public E convert(Annotation p_SourceAnnotation, Object p_oSource,
            Class<E> p_oType);
}
```

Cuando dicha interfaz sea implementada, lo siguiente será especificar el tipo de dato destino que la misma es capaz de convertir, adaptando de esta forma el "contrato" de nuestra clase *converter* al tipo destino especificado. Como se puede observar en su definición, contamos con un único método, el cual devuelve una instancia del tipo especificado, y espera una serie de parámetros:

1. *sourceAnnotation*: Instancia de la *annotation* asociada a nuestro *converter* que ha sido especificada en el *controller* origen.
2. *source*: Objeto entrante que deberá ser transformado al tipo destino. Generalmente será de tipo *java.lang.String*.
3. *type*: Clase del tipo de dato destino.

Implementada la interfaz, lo siguiente es indicar al *framework* de la existencia de nuestro *converter*, para lo que debemos proceder a anotar nuestra clase con la *annotation @ConvertionRule*, la cual, no solo servirá para que el *framework* en tiempo de ejecución sea capaz de encontrar y registrar nuestro *converter*, sino que también nos permitirá asociar la *annotation* que hemos creado en el paso anterior con nuestra clase, especificando el tipo de dato que nuestra clase es capaz de convertir. De esta forma, cuando el *framework* detecte la *annotation @ConvertionRule*, el mismo registrará nuestra clase *converter* (en el ejemplo LocaleConvertionRule) y la asociará a la *annotation* personalizada especificada (en nuestro ejemplo *@ApplyLocaleConverter).* De esta manera, la clase LocaleConvertionRule quedaría como a continuación:

```
@ConvertionRule(associatedTo=ApplyLocaleConverter.class,target = Locale.class)
public class LocaleConvertionRule implements DataConverter<Locale>{

    @Override
    public Locale convert(Annotation p_SourceAnnotation, Object p_oSource, Class<Locale> p_oType) {

        String[] aLocale = ((String)p_oSource).split("_");

        if(aLocale.length == 1)
        {
            return new Locale(aLocale[0]);
        }
        else if(aLocale.length == 2)
        {
            return new Locale(aLocale[0],aLocale[1]);
        }
        else if(aLocale.length == 3)
        {
            return new Locale(aLocale[0], aLocale[1], aLocale[2]);
        }

        return null;
    }

}
```

Refrescando un poco como funcionaba el proceso de invocación de un *controller*, uno de los pasos en los que consiste dicha ejecución es en la recuperación de todos aquellos atributos guardados, ya sea en la *request* o en la propia sesión, para luego asignarlos al *controller* en todos aquellos atributos de modelo marcados como tal mediante la *annotation @ModelAttr.* Esta tarea concretamente se encuentra implementada en el objeto *command* PopulateModelCommand, el cual, para poder llevarla a cabo, uno de los pasos que este *command* debe ejecutar es la localización de aquellos *converters* solicitados a nivel de *controller,* buscando si las *annotation* a nivel de clase están relacionadas con clases de tipo *converters* registradas, y si así fuera el caso, informar de dichos *converters* a la librería encargada del mapeo de atributos, que como ya sabemos ,se trata de *Common BeanUtils*. Ahora bien, ¿Como es posible que podamos notificar a *BeanUtils* de nuestros propios *converters* de forma totalmente transparente y sin aparente acoplamiento de nuestra solución con su especificación?. Pues bien, esto lo conseguimos gracias a una clase llamada *ConverterAdapter*, la cual, como su propio nombre indica, permite adaptar nuestra especificación de *converter* con la propia esperada por *BeanUtils.* Todo *converter* de *BeanUtils* necesita implementar la interfaz *org.apache.commons.beanutils.Converter* o cualquiera de sus subtipos; en nuestro caso, dado que hemos decidido definir *converters* sensibles al *locale* de turno, hemos utilizado la interfaz *org.apache.commons.beanutils.LocaleConverter*, por lo que, cuando llega el momento de registrar un *converter* personalizado, lo que hacemos es crear una instancia de *ConverterAdapter*, la cual implementa la interfaz *LocaleConverter,* y pasamos al constructor una instancia de nuestro *converter*, junto con la *annotation* asociada, el tipo destino y el *locale* del sistema mediante el uso de *SpecterLocaleManager*. De esta forma, cuando *BeanUtils* invoque al *Adapter*, el mismo se encargará de invocar la instancia de nuestro *converter* de forma totalmente transparente para ambas partes,

evitando un acoplamiento entre ambas especificaciones. De esta forma, con unos simples pasos, somos capaces de integrarnos en el propio flujo de *populate* de *BeanUtils* sin realmente atarnos a la solución, lo que nos aporta libertad de cambiar la implementación de la misma sin impactar directamente en las solución que implemente nuestro *framework.*

3.7 Custom Validators

Una vez recuperado el modelo de la aplicación, el siguiente paso natural sería la comprobación de la información recuperada mediante la ejecución de un conjunto de reglas. Para ello, el *framework* proporciona un mecanismo de ejecución de un conjunto de comprobaciones sobre aquellos contenidos recuperados desde la *request* o la sesión agrupadas en componentes llamados *validators*. Estos componentes tendrán como objetivo la validación de un determinado valor entrante, mediante la ejecución de un conjunto de comprobaciones totalmente personalizadas, e informar al *framework* sobre el resultado obtenido, es decir, indicar si la validación a finalizado satisfactoriamente o no e informar las razones de dicho resultado en forma de mensajes.

El componente encargado de la gestión y ejecución de los *validators* dentro del flujo de tratamiento de *request* entrantes es el *ValidateModelCommand*, el cual, como hemos visto anteriormente, analizará cada uno de los atributos del modelo en búsqueda de *validators* y procederá con su ejecución. Ahora bien, ¿Como debemos diseñar nuestros *validator* de forma que *ValidateModelCommand* sea capaz de encontrarlos y ejecutarlos cuando nosotros le indiquemos?. La respuesta es sencilla.

El procedimiento de diseño de un *validator* es semejante a la definición de un *converter.* Lo primero que debemos hacer es diseñar la *annotation* que relacionaremos con nuestro *validator*, y que servirá para que el *command* sea capaz de detectar que debe ejecutar una validación sobre el atributo asociado. Dicha *annotation* puede tener la siguiente forma:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidateUser {

}
```

Importante ElementType.FIELD

Como se puede observar, la estructura de la *annotation* es similar a la de los *converters*, con la particularidad de que estas deberán ser utilizadas a nivel de atributo (*ElementType.FIELD)* para que, cuando se analicen los atributos del modelo, las correspondientes *annotations* de validación sean detectadas adecuadamente. Otra cosa interesante es que, dado que estamos utilizando una *annotation* para indicar cuando una validación es requerida, la misma podrá ser definida de forma que pueda ser asignados atributos a completar con información adicional que luego, en el momento de ejecutar la validación correspondiente, el *validator* pueda extraer esta información adicional y utilizarla para personalizar su ejecución sobre el atributo relacionado.

Un requerimiento de los *validators* es la implementación de la interfaz *DataValidator*, la cual requerirá, en el momento de referenciarla, concretar sobre que tipo de dato será capaz de actuar, de tal forma que el "contrato" del *validator* se adapte al mismo. Dicha interfaz tiene la siguiente forma:

```
public interface DataValidator<E> {
    public boolean validate(Annotation p_oSourceAnnotation, E p_oSource)  throws SpecterValidationException;

    public List<ErrorMessage> getErrors();
}
```

DataValidator nos introduce a dos métodos: *validate,* invocado por *ValidateModelCommand* cuando este se encuentre con una *annotation* de validación asociada en un atributo del modelo, y *getErrors*, el cual será invocado por *ValidateModelCommand* cuando la validación asociada no retorne un resultado satisfactorio y este necesite "dar explicaciones" del porqué de dicha situación al cliente. Indagando un poco más en la definición de la interfaz, podemos comenzar por el método *validate,* el cual cuenta con dos parámetros:

1. *sourceAnnotation:* Instancia de la *annotation* asociada al *validator* de turno, especificada en el *controller* origen.
2. *Source:* Valor origen sobre el cual deberán ser aplicadas las reglas de validación definidas en el *validator*.

En caso que se produzca una situación inesperada durante la ejecución del *validator*, la firma del mismo contempla el lanzamiento de excepciones de tipo *SpecterValidationException*, la cual provocará que automáticamente se cancele el flujo de validación y se retorna la descripción de dicha excepción al cliente. Por otro lado, el método *getErrors*, será accedido solo en caso que el método *validate* retorne *false*, lo que indicaría que la reglas de validación definidas no han sido superadas por el valor comprobado.

Una vez implementada dicha interfaz, el siguiente paso es indicar al *framework* de la existencia de nuestro *validator* y con que *annotation* vamos a relacionarla en nuestra solución, de tal forma que, cuando el mismo encuentra nuestra *annotation,* automáticamente ejecutará el *validator* asociado sobre el atributo anotado. Para conseguir dicho comportamiento, lo que será necesario es utilizar la *annotation @ValidationRule* en nuestra *validator*, la cual requerirá especificar como parámetro la clase de *annotation* asociada, quedando algo como sigue a continuación:

```
@ValidationRule(associatedTo=ValidateUser.class)
public class UserValidationRule implements DataValidator<User>{

    private List<ErrorMessage> _lstErrors = new ArrayList<ErrorMessage>();

    @Override
    public boolean validate(Annotation p_oSourceAnnotation, User p_oSource)
            throws SpecterValidationException {

        boolean blResult = true;

        if(p_oSource.getName() == null || p_oSource.getName().trim().length() <= 0)
        {
            _lstErrors.add(new ErrorMessage("name","user.name.empty"));

            blResult = false;
        }

        if(p_oSource.getLastname() == null || p_oSource.getLastname().trim().length() <= 0)
        {
            _lstErrors.add(new ErrorMessage("lastname","user.lastname.empty"));

            blResult = false;
        }

        return blResult;
    }

    @Override
    public List<ErrorMessage> getErrors() {

        return _lstErrors;
    }

}
```

Como se puede observar en el ejemplo anterior, tenemos la *annotation @ValidationRule* a nivel de tipo, a la cual estamos especificando como parámetro nuestra *annotation* definida anteriormente *@ValidateUser*. Además, como se puede observar, cada campo validado tiene su propio mensaje de error, cada uno de ellos representado por una instancia de tipo *ErrorMessage*, que luego será utilizada por el *framework* para la recuperación del contenido del correspondiente mensaje desde los recursos configurados (más detalles veremos en secciones posteriores). También se puede observar como estamos especializando la interfaz *DataValidator* para que la misma solo acepte valores de tipo *User*, restringiendo de esta forma el dominio sobre el cual dicho *validador* es capaz de actuar.

3.8 Controllers

Superados todas las fases anteriores, el siguiente paso es la ejecución de un método o *handler* encargado de la manipulación de una *request* entrante y de especificar al *framework* cual debe ser el siguiente paso a seguir. Cada *handler* se encuentra definido en una clase anotada como *controller* mediante la utilización de la *annotation @Controller*. Esta *annotation,* como tantas otras que hemos mencionado anteriormente, es la encargada de indicar al *framework* de la existencia de dicho *controller*, de tal forma que, cuando este detecte su presencia, el mismo procederá con el análisis e indexación de todo su contenido. Cada *controller* estará asociado a un nombre único, el cual por defecto será el nombre de la clase definida, pero contando con la posibilidad de especificar un nombre personalizado haciendo uso del parámetro *name.* Además, debemos recordar que en este

punto de la definición, es adecuado especificar aquellos *converters* que nos interesa que el *framework* utilice en el procesamiento de los atributos del modelo, como se muestra a continuación.

```
@Controller(name="language")
@ApplyLocaleConverter
public class LanguageController
```

Se puede observar, por un lado, como estamos personalizando el nombre del *controller* mediante el parámetro *name*. Por otro lado, si recordamos el ejemplo anterior de *converter*, se puede ver como estamos referenciando su utilización en el momento del procesamiento del modelo. Ahora bien, refrescando un poco la memoria, cada *controller* esta conformado principalmente por tres componentes:

- *Model attributes*: Son los atributos que anteriormente han sido recuperados, ya sea desde la *request* o la sesión, convertidos y validados.
- *Http handler*: Métodos encargados de la manipulación de una *request* entrante y de indicar al *framework* el siguiente paso en el flujo de ejecución.
- *Error handlers*: Métodos encargados de la gestión de errores durante el tratamiento de una *request.* También deberán indicar cual será el paso a seguir en dicha circunstancia.

Como es de suponer, cada componente de los mencionados anteriormente esta asociado a una *annotation* concreta. En el caso de atributos del modelo, la *annotation* en cuestión es la *@ModelAttr*. Esta tiene como finalidad indicar al *framework* que se espera un valor para este atributo desde la *request* o la sesión. Esta *annotation* proporcionará un nombre por defecto bajo el cual el *framework* registrará su existencia, cuyo valor consiste en el nombre con el que ha sido definido en la clase origen, pero además contamos con la posibilidad de modificar dicho comportamiento mediante el parámetro *qualifier.* Otra característica de esta *annotation* es la posibilidad de especificar desde donde se supone que dicho atributo debe ser recuperado, cuyo lugar por defecto es desde la *request*, pero también es posible la modificación del mismo haciendo uso del parámetro *scope*, el cual acepta dos valores: *REQUEST* o *SESSION*. Un posible ejemplo de uso de dicha *annotation* sería como sigue:

```
@ModelAttr(qualifier="user")
@ValidateUser
private User user;
```

Aquí se puede ver como utilizar la *annotation @ModelAttr*, donde se especifica un nombre diferente del atributo para una mejor compresión de la referencia y además evitar acoplar la capa de presentación con nuestra especificación. Por otro lado, si recordamos el ejemplo de *validator* del apartado anterior, podemos ver como se solicita la validación de dicho atributo mediante la *annotation @ValidateUser*.

En relación a los *http handlers*, estos componentes son indicados mediante la *annotation @HttpHandlerMethod*, la cual, como suele ser usual, por defecto registrará su existencia bajo el nombre del método anotado, pero también es posible modificar dicho valor haciendo uso del parámetro *name.* Otra cosa que debemos tener en cuenta es que, cuando se produzcan problemas durante la ejecución de cualquiera de los *handler* definidos, si queremos notificar al *framework* de dicha situación, tenemos dos posibilidades: podemos registrar su ocurrencia en forma de *ErrorMessage* en una instancia de *SpecterMessageBasket* proporcionada por el propio *framework* en aquel atributo del *controller* anotado con la *annotation @Messages, o* simplemente lanzar una *RuntimeException*, la cual será capturada y registrada automáticamente por el propio *framework* en el momento de su ocurrencia. Ahora bien, cuando se produzca un problema o excepción durante la ejecución de un *handler*, existe la posibilidad de indicar quien será el encargado de gestionar dicha incidencia y de determinar cual debe ser el paso a seguir. Esta tarea puede recaer sobre cualquier método anotado bajo la *annotation @FlowErrorHandler*, con la particularidad de que solo puede existir un *flow error handler* por defecto en cada *controller* (caracterizado por el hecho de que el parámetro *name* de la *annotation @FlowErrorHandler* no ha sido especificado) pero pueden existir tantos como se desee, siempre y cuando su nombre sea único dentro del propio *controller*. De esta forma, si se deseara personalizar el tratamiento de errores para un *handler* concreto, simplemente debe ser especificado el parámetro *errorHandlerName* de la *annotation @HttpHandlerMethod* con el nombre de cualquier de los *flow error handlers* definidos o simplemente dejarlo en blanco dar lugar a que se utilice el *flow error handler* por defecto. Un ejemplo de *controller* podría ser como a continuación:

```java
@Controller(name="users")
public class UserController {

    @ModelAttr(qualifier="user")
    @ValidateUser
    private User _user;

    @HttpHandlerMethod
    public String create()
    {
        return "user/creationSuccess";
    }

    @FlowErrorHandler
    public String handleErrors()
    {
        return "user/creation";
    }

}
```

Aquí se puede observar como el método *create* se encuentra anotado como *handler (*el cual, al no haber sido especificado ningún nombre, el *framework* utilizará el valor 'create' como clave del registro en el sistema) y el método *handlerErrors* como *flow error handler* por defecto, lo que significa que, en caso que ocurran problemas durante la ejecución del método *create,* el tratamiento de errores será proporcionado por el *flow error handler* antes mencionado.

Ahora bien, desde el punto de vista cliente, ¿Como hacemos para invocar dicho *handler* en caso que nos interese?. Pues muy sencillo, tanto la *annotation @Controller* como *@HttpHandlerMethod* construyen una pequeña jerarquía o árbol que podrá ser referenciada siguiendo la misma sintaxis de cualquier *url*. Es decir, siguiendo el ejemplo, si nos interesa gestionar un *submit* por medio del método *create* definido en la clase anterior, lo único que debemos hacer es construir una *url* que se corresponda con la estructura del *controller* de la forma:

http://[application_context]/[controller]/[handler]

Por lo que en nuestro caso nos quedaría

http://www.test-application.com/users/create

De esta forma, cuando la *url* entrante sea analizada por la clase *DispatchPhaseManager*, la misma llegará a la conclusión de debe ejecutar el método anotado como *handler* y esperar una respuesta del mismo para determinar debe ser el siguiente paso a seguir. Como se puede observar, tanto en la respuesta del método *create* como del método *handleErrors* ambos retornan una cadena con una sintaxis específica. En ambos casos tenemos claros ejemplos de como redireccionar hacia una vista o hacia otro *handler* definido ya sea en el mismo *controller* o en cualquier otro definido en el sistema, pero, a diferencia del ejemplo anterior, a ser referencias internas a otros componentes, no será necesario especificar la parte del contexto de la aplicación, sino más bien la jerarquía interna. Por ejemplo, en el caso de 'user/creationSuccess', lo que estamos especificando realmente al *framework* es que, a partir de la raíz marcada por el parámetro de inicio *viewLocationRootPath*, se navegue en cada uno de los subdirectorios separados por '/' hasta llegar a la vista llamada 'creationSucces.jsp', y una vez encontrada, redireccionar el flujo de ejecución hacia la misma.

Para finalizar, una última característica de los *controllers* es la posibilidad que tienen los mismos de manipular el *locale* activo del sistema, el cual será asignado por el *framework* en caso que contenga un atributo anotado con la *annotation @ApplicationLocale*, donde el tipo esperado de dicho atributo deberá ser *LocaleManager*. De esta forma, si quisiéramos cambiar en caliente el lenguaje de la aplicación, simplemente haciendo uso de la instancia del tipo mencionado, podremos cambiar el *locale*, además de lanzar automáticamente la carga de los recursos correspondientes al nuevo idioma seleccionado, lo que afectará directamente al *layout* y comportamiento de la aplicación.

3.9 Message Basket y Taglibs

Anteriormente hemos mencionado una clase llamada *SpecterMessageBasket*. La misma fue concebida inicialmente como un "repositorio" de mensajes donde cualquier componente del *framework* pueda depositar un mensaje de error cuyo origen posiblemente sea la ocurrencia de algún tipo de incidencia, y de esta forma suspender el flujo de tratamiento, independientemente del punto en el que estemos, entregando a continuación información sobre el conjunto de problemas que han ocasionado dicho comportamiento. Finalmente, por cuestiones de diseño o tiempo, el alcance de dicho componente ha estado reducido solamente al contexto de ejecución de cualquiera de las fases de tratamiento de una *request* que anteriormente hemos hablado. Básicamente, tenemos dos tipos de errores: aquellos que pueden ocasionarse durante la ejecución de cualquiera de las fases de tratamiento o aquellos que simplemente tendrán lugar durante la ejecución de la solución implementada a partir del uso de nuestro *framework*, ya sea utilizando la instancia de *MessageBasket* proporcionada a partir de la *annotation @Messages o* aquellos errores señalados a partir de la ejecución de un *validator*. Cualquiera fuera su origen, en caso que el *framework* detectara que existen errores acumulados, inmediatamente se detendrá la ejecución del flujo de tratamiento y se dará lugar a la lógica de tratamiento de errores, trasladando dichos errores en un contexto accesible de tal forma que la lógica de presentación sea capaz de mostrar el nuevo estado adecuadamente. Concretamente, estos mensajes son almacenados en la propia respuesta en forma de atributos, donde existen dos posibilidades de acceder a los mismos: recuperando el valor asociado de forma manual y tener un completo control sobre que hacer con él o utilizar *taglibs* de asistencia proporcionados por el *framework*, de tal forma que podamos despreocuparnos de la implementación interna de la gestión y almacenamiento de errores y solo concentrarnos en como vamos a presentar dicha información al cliente. Para ello contamos principalmente con dos *tag:*

- *SpecterErrorTag:* Permite recuperar el valor de un determinado mensaje de error, si existe.
- *SpecterErrorGroupTag:* Permite la recuperación de uno o más errores clasificados bajo un determinado grupo.

En el momento de instanciar un *ErrorMessage*, tenemos tres posibilidades de hacerlo:

- Podemos especificar una *key* única junto con un *message* asociado (el cual puede ser un texto limpio o el nombre de una *property* definida en los recursos de la aplicación) identificando el mensaje de forma única dentro del conjunto.
- Podemos definir que un *ErrorMessage* pertenezca a un determinado *group*, haciendo posible tener tantos mensajes como queramos bajo un mismo grupo.
- Podemos no especificar ni *key* ni *group* y solo definir el mensaje a mostrar, haciendo que dicho mensaje pase a formar parte de un grupo genérico llamado GLOBAL_ERROR.

Por ejemplo, un caso útil para un *group* podría ser si quisiéramos recuperar todos los errores posibles asociados a un determinado atributo del modelo, pudiendo tener tantos errores como validaciones se ejecuten sobre el mismo, o también podemos identificar un error de forma única mediante una *key* que nos interese ubicar en la presentación de una forma muy concreta y sobre el cual tenemos que tener un control especial, o podemos generar un mensaje de error que no necesite estar asociado a nada en concreto sino que mas bien puede tratarse de un mensaje global de error dentro del contexto, como puede ser una pérdida de conexión a la base de datos o un error en un proceso de guardado de información.

Ahora bien, cuando necesitemos mostrar dichos mensajes en la capa de presentación, podemos hacer uso del *taglib* antes mencionado en cualquiera de los ficheros JSP utilizados en la presentación. Para ello, primer debemos declarar el uso de las mismas colocando en la cabecera de nuestro fichero:

```
<%@ taglib uri="http://specter.uoc.edu/utils" prefix="spr" %>
```

Una vez referenciada, si quisiéramos acceder y visualizar directamente el valor de un mensaje de error con una *key* específica y que puede o no estar asociada a un determinado *group*, podemos utilizar el *tag SpecterErrorTag*. Si recordamos nuestro ejemplo de *validator*, podemos ver como dentro del mismo son generados dos mensajes de error diferentes, ambos con una *key* y *message* específico. Si quisiéramos acceder a los mismos haciendo uso de este *tag,* deberíamos escribir algo como a continuación:

```
<spr:specter-error-message key="name" group="user"/>
```

Aquí se puede ver como es especificado el atributo *key* y también el atributo *group*, pero ¿Porqué el atributo *group* si no ha sido especificado ningún grupo asociado? Aunque el atributo *group* es opcional, cuando se traten de mensajes generados dentro de un *validator*, este tipo de agrupación es realizado automáticamente por el propio *framework* y se realiza a partir del nombre del atributo que esta siendo validado, lo que quiere decir que, todos los errores de validación asociados a un atributo serán automáticamente agrupados bajo un grupo cuyo nombre coincidirá con el nombre del atributo en el modelo. De esta forma, si quisiéramos recuperar todos los errores de validación asociados a un determinado atributo, no tenemos mas que utilizar el *tag SpecterErrorGroupTag* de la forma:

```
<spr:specter-error-group var="currentError" group="user">
    <li>${currentError.message}</li>
</spr:specter-error-group>
```

A diferencia del caso anterior, donde se recuperaba un mensaje en concreto, al utilizar dicho *tag,* lo que estamos haciendo es recuperar todos los mensajes relacionados con el atributo *user* (o lo que es lo mismo con el grupo *user*), donde cada mensaje recuperado será almacenado en una variable llamada *currentError* (o cualquier nombre que se desee), y a partir de este punto podemos hacer con el mensaje recuperado lo que queramos y presentarlo como más nos plazca. Este tipo de comportamiento de agrupación automática es realizado solamente en el caso de ser mensajes generados por *validators*, pero en el resto de la aplicación, si quisiéramos agrupar mensajes, deberá realizarse de forma explícita, dado que el atributo *group* de un mensaje es totalmente opcional.

Adicionalmente, así como a la hora de crear un nuevo *ErrorMessage* tenemos la posibilidad de especificar o un mensaje en limpio o el nombre de una propiedad definida en un recurso de la aplicación (generalmente hablamos de ficheros de properties), también podemos recuperar estos textos almacenados en forma de *property* directamente en el momento de ejecución de una vista. Para ello, se ha definido un tercer *tag* llamado *SpecterMessagetTag*, el cual, como su propio nombre indica, nos permitirá acceder al contenido de una *property* definida en un fichero de recursos directamente de la forma:

```
<spr:specter-message key="user.name"/>
```

De esta forma, estaríamos accediendo a una *property* de tipo *user.name=Name*, mostrando el literal 'Name' en pantalla. Una última cosa a tener en cuenta es que todos los recursos recuperados desde ficheros de *properties*, el proceso de recuperación es sensible al *locale* activo en la aplicación, por lo que, en caso que nos interese recuperar recursos en diferentes idiomas, además de definir los correspondientes ficheros de recursos por cada idioma utilizado, no tenemos más que cambiar el *locale* activo mediante la clase *SpecterLocaleManager* para que dicho cambio tenga efecto inmediato en el proceso de lectura de recursos.

3.10 Draft Drawer

Muchas veces (y me incluyo) a medida que un proyecto crecía en el tiempo, dependiendo del *framework* que estaba siendo utilizado, llegaba un momento en el cual era muy difícil saber que funcionalidades se encontraban disponibles en la aplicación, llevando a situaciones en las que muchas veces volvíamos a repetir un trabajo que ya había sido llevado a cabo y que, si hubiéramos contado con alguna herramienta gráfica que nos permita visualizar las diferentes *urls* declaradas junto con sus parámetros, nos hubiéramos ahorrado muchos dolores de cabeza. Es por ello que, frente a esta experiencia mencionada, y que aún hoy en día sigue siendo un problema (desde mi punto de vista), he decidido definir una funcionalidad adicional en el *framework* que, simplemente solicitando un informe de lo que hasta el momento ha sido definido en materia de componentes de una solución y de funcionalidades o *urls* activas , el mismo se encargará de generarlo por nosotros, y así, de esta forma, visualizar de una forma gráfica "de que esta hecha nuestra solución".

Para acceder a dicha funcionalidad realmente es muy sencillo, simplemente debemos especificar el parámetro *viewAppDraft* en cualquier momento y automáticamente *specter* nos generará una página web con toda la información relacionada a nuestra aplicación de la forma:

Validators

- **ValidateUser**:
  - *Class:* edu.uoc.pfc.specter.sample.validators.UserValidationRule
  - *Key:* interface edu.uoc.pfc.specter.sample.validators.ValidateUser

Converters

- **LocaleConvertionRule**:
  - *Class:* class edu.uoc.pfc.specter.sample.converters.LocaleConvertionRule
  - *Key:* interface edu.uoc.pfc.specter.sample.converters.annotations.ApplyLocaleConverter
  - *Annotation:* interface edu.uoc.pfc.specter.sample.converters.annotations.ApplyLocaleConverter
  - *Target Type:* java.util.Locale

Controllers

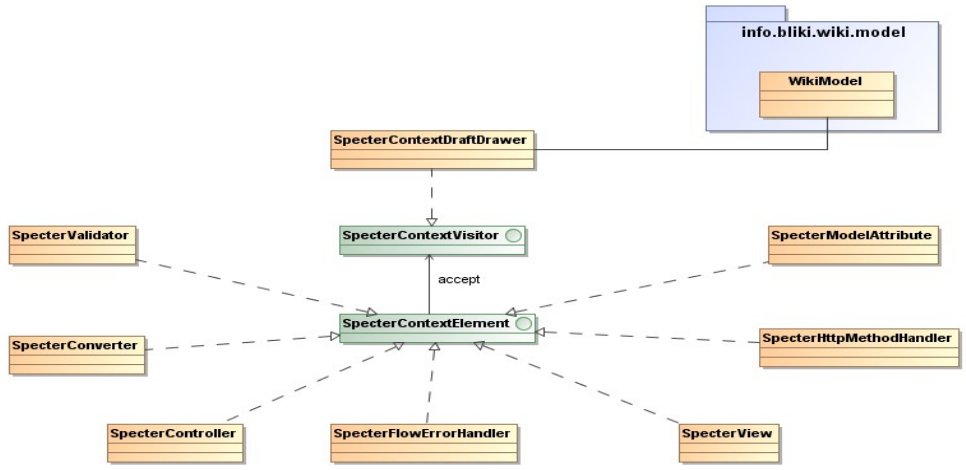
- **users**
  - *Class:* class edu.uoc.pfc.specter.sample.controllers.UserController
  - *Key:* users
  - *Locale field:* Not defined
  - *Messages field:* Not defined
  - ***Attributes***
    - ***user*** [*scope*=REQUEST; *validators*=ValidateUser]
  - ***Http Handlers***
    - ***create*** [errorHandler=DEFAULT]
  - ***Flow Error Handlers***
    - ***DEFAULT*** [method=handleErrors]
- **language**
  - *Class:* class edu.uoc.pfc.specter.sample.controllers.LanguageController
  - *Key:* language
  - *Locale field:* _localeManager
  - *Messages field:* Not defined
  - ***Attributes***
    - ***locale*** [*scope*=REQUEST; *validators*=]
    - ***return*** [*scope*=REQUEST; *validators*=]
  - ***Http Handlers***
    - ***change*** [errorHandler=DEFAULT]

Views

- *user/creation* [file=/home/guillermo/software/workspace/specter/source/specter-sample-webapp/src/main/webapp/WEB-INF/jsp/user/creation.jsp]
- *user/creationSuccess* [file=/home/guillermo/software/workspace/specter/source/specter-sample-webapp/src/main/webapp/WEB-INF/jsp/user/creationSuccess.jsp]

Se puede ver que, cada uno de los elementos que componen nuestra solución y que tienen relación directa con el *framework* tiene su propio apartado, mostrando información relevante de cada uno ellos y que nos permite saber directamente de que forma han sido configurados en el arranque.

Llevar a cabo esta solución ha sido muy sencillo gracias a la naturaleza jerárquica del contexto construido por el *framework* en el arranque, por lo que, utilizando el patrón de diseño *visitor*, somos capaces de recorrer toda la estructura del contexto de forma transparente y plasmar su contenido de una forma gráfica e intuitiva. Por un lado para experimentar y por otro por una cuestión de facilidad he decidido utilizar una librería llamada *gwtwiki*, la cual, entre muchas otras características, es capaz de transformar *wikitext* en *HTML*, consiguiendo de esta forma, una forma sencilla y simplificada de generar código HTML escribiendo muy poco. Aquí podemos ver un gráfico de las clases intervinientes en el proceso:



Se puede ver que la clave de esta funcionalidad reside en dos aspectos: primero, que todos los elementos que componen el contexto de la aplicación implementan la interfaz *SpecterContextElement*, la cual nos permite "visitar" su instancia así como aquellos elementos que lo componen internamente, por ejemplo, recordemos que un S*pecterController* contiene también elementos de tipo *SpecterHttpMethodHandler, SpecterModelAttribute y SpecterFlowErrorHandler*. De esta forma, nuestro objeto de tipo S*pecterContexVisitor* va navegando por la estructura del contexto y generando el contenido del informe. También cabe destacar que, aquí hemos implementado un único tipo de visitante, pero si quisiéramos definir cualquier otro visitante que sea capaz, por ejemplo, de generar un PDF o XML en vez de una página HTML o cualquier otro tipo de formato que se nos ocurra podríamos hacerlo sin problemas simplemente definiendo una clase que implemente la interfaz *SpecterContextVisitor.*

3.11 Logger Aspect

A pesar de que se traten de utilidades internas del *framework*, creo que vale la pena comentar su existencia. Inicialmente, mi intensión era proporcionar un conjunto de aspectos que puedan ser utilizados mediante un *load-time weaver*, proporcionando utilidades como *generic loggers* o alguna utilidad de seguridad, pero creo que, dado que sería necesario configurar un *java agent* adicional para que el mismo lleve a cabo el *weaving* de los aspectos definidos junto con un fichero *aop.xml* adicional, sería complicar demasiado la configuración de la aplicación de ejemplo, además de ser utilidades muy específicas que requerirían un diseño bastante más que importante, y es por ello que me he volcado por definir un conjunto de aspectos encargados de dejar trazas durante la ejecución de un flujo de interpretación de una *request*. Básicamente existen tres aspectos de *logger:*

- *CommandLoggerAspect*: Si revisamos el diseño del *CommandInvoker*, podremos ver que este esta compuesto por un conjunto de objetos *command,* los cuales implementan la interfaz *ControllerCommand*. Este aspecto justamente saca provecho de ello y se encarga de monitorizar el antes y despues de la ejecución del método *execute* de cualquier instancia que implemente dicha interfaz. De esta forma, podremos ver como se van ejecutando cada uno de los *commands*, en que orden y si los mismos han finalizado o no con errores.
- *ConverterLoggerAspect*: En este caso se ha definido un aspecto cuya función será la monitorización de cualquier ejecución de método *converter* de cualquier instancia la clase *ConverterAdapter*, de tal forma podamos registrar cada una de las llamadas por parte del *Common Beans* y dejar registrado cual es el *converter* que procede a invocar junto con el tipo de dato asociado y el valor origen que debe ser transformado. De esta forma seremos capaces de registrar en el sistema todas las llamadas a *converters* definidos en la solución desarrollada a partir de nuestro *framework.*
- *ValidatorLoggerAspect*: Aspecto encargado de monitorizar cualquier llamada al método *validate* de la interfaz *DataValidator*, la cual, como hemos visto anteriormente, será utilizada por aquellos *validators* que hayan sido definidos en la solución que haga uso de nuestro *framework*. De esta forma, cuando se proceda con la validación de un valor, el sistema registrará de que *validator* se trata, junto con el tipo de dato que esta a punto de validar y el valor origen.

El proyecto esta configurado de tal forma que sea capaz de llevar a cabo lo que se conoce como *source weaving*, lo que significa que el *weaver* forma parte del propio proceso de compilación y la entrada del mismo consiste en el código fuente de clases y aspectos, estos últimos haciendo uso sintaxis *AspectJ.* Para ello, aprovechando la flexibilidad de Maven, simplemente hemos tenido que configurar el *plugin aspectj-maven-plugin* de forma que en las fases de *compile y test-compile* sea lanzado el *weaver* automáticamente como podemos ver a continuación:

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.3.1</version>
    <configuration>
      <source>1.6</source>
      <target>1.6</target>
      <outxml>true</outxml>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>compile</goal>       <!-- use this goal to weave all your main classes -->
                <goal>test-compile</goal>  <!-- use this goal to weave all your test classes -->
            </goals>
        </execution>
    </executions>
</plugin>
```

4 Conclusiones

Realmente me he dado cuenta que, con el poco tiempo con el que contamos, conseguir un diseño que cumpla lo que yo interpreto como características esenciales de un *framework* de presentación no es tarea fácil, aunque tengo que reconocer que, aunque dura, me la he pasado muy bien. Es interesante ver que nivel de profundidad y detalle se puede alcanzar con soluciones de este tipo, existen un sin fin de características por mejorar de las implementadas y muchas funcionalidades que me he dejado en el tintero que se me iban ocurriendo a medida que desarrollaba mi propuesta. Sobre todo me ha llamado la atención lo complejo y variado que puede resultar el tratamiento de una *request* entrante y que largo puede llegar a ser el desarrollo de una solución de esta naturaleza sin siquiera poder tener resultados "tangibles" y comprobables en una implementación *end-to-end.*

También tengo que reconocer que he aprovechado esta oportunidad para indagar en tecnologías que hasta el momento no había tenido la oportunidad de tratar con ellas demasiado y que considero responsables de la gran variedad y potencialidad con la que cuentan los *frameworks* de presentación actuales. Me estoy refiriendo de *Cglib*, la cual me parece alucinante lo que podemos llegar a hacer con ella, con el único *handicap* de que la documentación existente es poco clara y suficiente, llevando a muchos desarrolladores (me incluyo) a tener que aprender a partir de la prueba y error; y también a *reflections*, la cual, a pesar de que creo que aún tiene cosas por mejorar, su propuesta me parece muy buena y muy interesante, permitiéndonos proporcionar soluciones personalizadas y prácticamente transparentes para el desarrollador de forma muy sencilla y *no intrusiva.* A modo de mención de honor, también he utilizado una librería que me ha parecido curiosa y útil para la generación de información de configuración de la aplicación: *gwtwiki*, la cual, a pesar del uso que le he dado, creo que estamos ante un proyecto con un cierto atractivo y que abre un conjunto de posibilidades interesantes, como generaciones alternativas de documentación en formato *Wiki* o *HTML* totalmente personalizada, como alternativa de *Javadocs,* por ejemplo.

Por otro lado, tengo que decir que he notado que, a pesar de su aún vigente utilidad, *Common BeanUtils* se ha quedado un poco en el tiempo, sobre todo en las limitaciones del diseño de *converters*, dado que, mi intensión inicial era personalizar el proceso de conversión a nivel de atributo, aprovechando la potencia que nos brinda las *annotations*, pero me he dado cuenta tarde de que solo es posible aplicarlas a nivel de clase, siendo aún peor el hecho de que no existe forma de pasar como parámetro el atributo que esta siendo convertido, principalmente porque la propia API no lo contempla, lo que me lleva a pensar que esto puede ser considerada una limitación importante para los días que corren.

Lamentablemente, por cuestiones de disponibilidad, me he tenido que dejar muchas ideas que rondaban mi cabeza, como puede ser integración con *Spring* o una implementación totalmente personalizada de *converters* que permita un completo acceso y manipulación del atributo manipulado. También estaba interesado en trabajar con aspectos pero de una forma diferente de como lo he hecho, es decir, en vez de generar *bytecode* en el momento de compilación del proyecto, mi intensión inicial era genera aspectos en tiempo de ejecución de forma totalmente transparente, para de esta forma no tener que depender de la estructura de interfaces del *framework* y de esta forma ser capaz de, por ejemplo, definir un conjunto de reglas de *log* del sistema, y de esta forma lograr incrementar la potencialidad de la solución, haciéndola no tan dependiente del código.

En resumen, puedo decir con sinceridad de que, si hubiera contado con más tiempo, habría pensado muchas cosas de otra manera y de una forma mucho más genérica, que me permitan integrarme con otras soluciones y a la vez conseguir un diseño totalmente escalable, pero viendo el tiempo que he podido dedicarlo, creo que finalmente he conseguido algo decente y sobre todo que ha cumplido con los objetivos propuestos.

5 Glosario

- *Annotations*: Técnica con la cual es posible introducir meta-información en el propio código fuente y de esta forma influenciar en el comportamiento de la aplicación en tiempo de ejecución.
- *Aspects:* Consiste en una funcionalidad que se encuentra relacionada con muchas partes de la aplicación pero que a la vez no se encuentra relacionada directamente con la funcionalidad principal de la misma, violando de alguna forma el concepto de *separation of concerts*, pero permitiendo por otro lado la posibilidad de definir *cross-cutting concerns*.
- *Bytecode: C*onjunto de instrucciones diseñadas para una ejecución eficiente mediante un interprete, además de ser adecuado para una posterior compilación a código de máquina o para reducir dependencias de software y hardware y de esta forma permitir su ejecución en múltiples plataformas.
- *Classpath*: Punto de partida a partir del cual se informa a la máquina virtual de Java donde se encuentran todas aquellas clases y paquetes necesarios para la ejecución de aplicaciones.
- *Cross- cutting concerns:* Consisten en aspectos de una aplicación que afectan de alguna forma otros aspectos, los cuales se caracterizan por no ser tan fáciles de separar del resto del sistema, tanto en el diseño como en la implementación, resultando en duplicación de código (*scattering*), en un incremento de dependencias entre sistemas (*tangling*) o ambas.
- *Separation of concerns:* Consisten en el proceso de separación de un programa en diferentes funcionalidades que se solapen lo menos posible. Usualmente se alcanza haciendo uso de la modularidad y encapsulación.
- *Dependency injection:* Técnica a partir de la cual se indica a parte de un programa que otras partes este puede utilizar, ya sea proporcionando referencias a dependencias externas o a componentes de software, separando de esta forma el comportamiento mismo del proceso de resolución de dependencias, consiguiendo desacoplar componentes de software altamente dependientes.
- *Framework*: Abstracción a partir de la cual código común proporciona funcionalidades genéricas, las cuales pueden ser selectivamente sobrecargadas o redefinidas por código de usuario, proporcionando de esta forma funcionalidad específica.
- *Inversion of Control* (IoC): Se trata de un estilo de construcción de software donde el código genérico controla la ejecución del código específico, con la connotación de que tanto el código reutilizable como el código específico son desarrollados de forma independiente, conformando una única aplicación totalmente integrada.
- *JUnit:* Framework de pruebas unitarios desarrollado para el lenguaje Java.
- *Locale:* Conjunto de parámetros que definen el lenguaje, país y variante que determinan el idioma con el cual el usuario visualizará una interfaz.
- *Maven*: Software de gestión de proyectos y automatización de procesos de construcción.
- *Mercurial*: Herramienta de control de versiones distribuida y multi-plataforma para código fuente, usualmente utilizada por desarrolladores.
- Out-of-the-box: Termino que intenta denotar ítems, funcionalidades o características que no requieren ninguna instalación adicional o modificación.
- *RESTful*: Termino que se refiere generalmente a servicios que utilizan el protocolo HTTP y los principios REST, los cuales se pueden que están conformados por 3 tipos de recursos: la URI base, el Media Type soportado por el servicio y un conjunto de operaciones soportadas por el servicio utilizando métodos HTTP.
- *Taglibs:* Un *taglib* o *tag library* es una colección de módulos reutilizables o *custom actions*, los cuales pueden ser invocados mediante el uso de *custom tags* en ficheros JSP.
- *Thin client*: Consiste en un ordenador software que tiene una gran dependencia en otro ordenador (su servidor) para cumplimentar los roles tradicionales de la informática.
- *Zero configuration*: Termino que hace referencia a la no necesidad de tener que escribir ficheros de configuración xml o ficheros de propiedades a la hora de utilizar un *framework*, aprovechando todas las ventajas proporcionadas por las *annotations.*

6 Bibliografía

- *Spring Web MVC Reference*. http://static.springsource.org/spring/docs/3.0.x/reference/mvc.html
- *Software Design Patterns for Information Visualization* http://vis.berkeley.edu/papers/infovis_design_patterns
- *Design Patterns Quick Reference* http://www.mcdonaldland.info/2007/11/28/40/
- *Design Patterns (Computer Science)* http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
- *Struts2 In Action*, Donald Brown and Chad Michael Davis http://www.manning.com/dbrown/
- *JavaServer Faces, Hans Bergsten* http://www.hansbergsten.com/aboutjsfbook.jsp
- GwtWiki http://code.google.com/p/gwtwiki/
- Cglib http://cglib.sourceforge.net/
- Apache Commons BeanUtils http://commons.apache.org/beanutils/
- AspectJ In Action http://www.manning.com/laddad2/

7 Anexos

7.1 Aplicación ejemplo

Tanto el código fuente del *framework* como la propia *wikipedia* se encuentran alojadas en www.bitbucket.org. Para acceder a ellos, en caso del código fuente, el mismo se encuentra alojado en un repositorio Mercurial a partir del cual podemos obtener una copia ejecutando:

```
hg clone https://gszeliga@bitbucket.org/gszeliga/specter
```

Posteriormente, si quisiéramos obtener una copia de la Wiki no tenemos mas que ejecutar:

```
hg clone https://gszeliga@bitbucket.org/gszeliga/specter/wiki
```

Posiblemente se requiera de permisos para su descarga, por lo que, si interesara, no duden en contactar conmigo. Una vez descargado el código fuente, lo siguiente será preparar nuestro entorno de trabajo. Para ello necesitaremos:

- *Apache Maven 2.2.1* (http://maven.apache.org/download.html)
- *Eclipse SDK v3.6.1 Helios* (http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/heliossr2)
- Instalar *plugins* de Eclipse:
  - *Maven* - http://m2eclipse.sonatype.org/sites/m2e
  - *Mercurial plugin* - http://cbes.javaforge.com/update

En el caso de *Maven*, una vez descargado, es necesario verificar que el comando *mvn* se encuentre disponible en cualquier punto de nuestro sistema, para lo que simplemente deberíamos actualizar la variable de entorno *PATH* con la ruta donde *Maven* haya sido descomprimido.

Hecho esto, el siguiente paso consiste en ingresar en el directorio */{YOUR_PATH}/specter/source* y escribir en linea de comandos:

```
mvn clean install
```

Este proceso puede demorarse un poco en caso que fuera la primera vez que ejecutamos *Maven* en nuestro sistema, principalmente porque el mismo comenzará a descargar todas las dependencias del proyecto desde Internet, para posteriormente proceder con la compilación, configuración, testeo y empaquetado del proyecto. Finalizadas todas las tareas implicadas en el comando anterior, nos queda arrancar nuestra aplicación de ejemplo, por lo que procedemos a ingresar en el directorio */{YOUR_PATH}/specter/source/specter-sample-webapp* y escribir en la línea de comandos:

```
mvn jetty:run
```

De esta forma, arrancaremos un servidor de aplicaciones de tipo Jetty que automáticamente desplegará nuestra aplicación web de ejemplo, por lo que, simplemente escribiendo en cualquier navegador la *url*

http://localhost:9090/specter-sample

automáticamente accederemos a la página principal de nuestra aplicación de ejemplo.

A la vez, si quisiéramos utilizar el modo *debug* en la aplicación, podríamos configurar una tarea *Maven Build* en *Eclipse* donde los correspondientes parámetros serían:

- *base directory*: ${workspace_loc:/specter-sample-webapp}
- *goals*: clean jetty:run
- En la pestaña *Environment* creamos una variable llamada MAVEN_OPT cuyo valor será '-Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,address=4000,server=y,suspend=n'.

Finalmente, ejecutando dicha tarea en *debug mode* dentro de Eclipse, automáticamente seremos capaces de debugar por el código fuente del *framework*.

7.2 Dependencias

7.2.1 specter-framework

7.2.1.1 Ficheros

Filename	Size	Entries	Classes	Packages	JDK Rev	Debug
asm-3.1.jar	42.02 kB	24	23	2	1.2	release
cglib-2.2.jar	272.15 kB	240	226	7	1.2	debug
logback-classic-0.9.9.jar	116.70 kB	119	90	17	1.5	debug
logback-core-0.9.9.jar	188.05 kB	187	155	22	1.5	debug
gson-1.4.jar	162.47 kB	143	130	3	1.5	debug
google-collections-1.0.jar	624.60 kB	520	505	4	1.5	debug
commons-beanutils-1.8.3.jar	226.58 kB	155	137	6	1.3	debug
commons-codec-1.2.jar	29.38 kB	30	19	5	1.1	debug
commons-httpclient-3.1.jar	297.85 kB	183	167	8	1.2	debug
commons-io-2.0.1.jar	155.77 kB	122	104	6	1.5	debug
commons-lang-2.4.jar	255.67 kB	148	127	9	1.2	debug
commons-logging-1.1.1.jar	59.26 kB	42	28	2	1.1	debug
dom4j-1.6.jar	306.01 kB	208	190	14	1.3	debug
httpunit-1.7.jar	506.25 kB	383	367	10	1.2	debug
bliki-core-3.0.16.jar	426.38 kB	322	280	26	1.5	debug
javassist-3.8.0.GA.jar	580.04 kB	352	333	17	1.2	debug
servlet-api-2.5.jar	102.65 kB	68	42	2	1.5	debug
jtidy-4aug2000r7-dev.jar	134.60 kB	129	119	3	1.1	release
junit-4.4.jar	157.69 kB	188	154	20	1.5	debug
commons-compress-1.0.jar	140.48 kB	94	70	11	1.4	debug
aspectjrt-1.6.10.jar	113.50 kB	145	128	10	1.5	debug
jettison-1.2.jar	70.78 kB	59	44	5	1.5	debug
jetty-6.1.10.jar	487.65 kB	250	220	14	1.4	debug
jetty-util-6.1.10.jar	156.72 kB	108	93	6	1.4	debug
servlet-api-2.5-6.1.10.jar	129.33 kB	78	42	2	1.4	debug
reflections-0.9.5-RC2.jar	79.68 kB	78	64	6	1.5	debug
slf4j-api-1.5.5.jar	21.29 kB	32	21	3	1.3	debug
js-1.6R6.jar	793.84 kB	323	294	19	1.3	debug
stax-api-1.0.1.jar	25.89 kB	48	40	5	1.2	debug
xml-apis-1.0.b2.jar	106.76 kB	217	184	17	1.2	release
Total	**Size**	**Entries**	**Classes**	**Packages**	**JDK Rev**	**Debug**
30	6.61 MB	4,995	4,396	281	1.5	27
compile: 19	compile: 3.91 MB	compile: 3,123	compile: 2,778	compile: 158	-	compile: 17
test: 9	test: 2.41 MB	test: 1,566	test: 1,373	test: 84	-	test: 8
runtime: 2	runtime: 304.75 kB	runtime: 306	runtime: 245	runtime: 39	-	runtime: 2

7.2.1.2 Repositorios

Repo ID	URL	Release	Snapshot
java.net-repo	http://download.java.net/maven/2/	Yes	Yes
codehaus.org	http://snapshots.repository.codehaus.org	Yes	Yes
apache.snapshots	http://people.apache.org/repo/m2-snapshot-repository	-	Yes
reflections-repo	http://reflections.googlecode.com/svn/repo	Yes	Yes
central	http://repo1.maven.org/maven2	Yes	-
codehaus-release-repo	http://repository.codehaus.org	Yes	Yes
info-bliki-repository	http://gwtwiki.googlecode.com/svn/maven-repository/	Yes	-

7.2.1.3 Coordenadas

Artifact	java.net-repo	codehaus.org	apache.snapshots	reflections-repo	central	codehaus-release-repo	info-bliki-repository
asm:asm:jar:3.1	-	-	-	-		-	-
cglib:cglib:jar:2.2	-	-	-	-		-	-
ch.qos.logback:logback-classic:jar:0.9.9	-	-	-	-		-	-
ch.qos.logback:logback-core:jar:0.9.9	-	-	-	-		-	-
com.google.code.gson:gson:jar:1.4	-	-	-	-		-	-
com.google.collections:google-collections:jar:1.0	-	-	-	-		-	-
commons-beanutils:commons-beanutils:jar:1.8.3	-	-	-	-		-	-
commons-codec:commons-codec:jar:1.2	-	-	-	-		-	-
commons-httpclient:commons-httpclient:jar:3.1	-	-	-	-		-	-
commons-io:commons-io:jar:2.0.1	-	-	-	-		-	-
commons-lang:commons-lang:jar:2.4	-	-	-	-		-	-
commons-logging:commons-logging:jar:1.1.1	-	-	-	-		-	-
dom4j:dom4j:jar:1.6	-	-	-	-		-	-
httpunit:httpunit:jar:1.7	-	-	-	-		-	-
info.bliki.wiki:bliki-core:jar:3.0.16	-	-	-	-	-	-	
javassist:javassist:jar:3.8.0.GA	-	-	-	-		-	-
javax.servlet:servlet-api:jar:2.5	-	-	-	-		-	-
jtidy:jtidy:jar:4aug2000r7-dev	-	-	-	-		-	-
junit:junit:jar:4.4	-	-	-	-		-	-
org.apache.commons:commons-compress:jar:1.0	-	-	-	-		-	-
org.aspectj:aspectjrt:jar:1.6.10	-	-	-	-		-	-
org.codehaus.jettison:jettison:jar:1.2	-	-	-	-		-	-
org.mortbay.jetty:jetty:jar:6.1.10	-	-	-	-			-
org.mortbay.jetty:jetty-util:jar:6.1.10	-	-	-	-			-
org.mortbay.jetty:servlet-api-2.5:jar:6.1.10	-	-	-	-			-
org.reflections:reflections:jar:0.9.5-RC2	-	-	-		-	-	-
org.slf4j:slf4j-api:jar:1.5.5	-	-	-	-		-	-
rhino:js:jar:1.6R6	-	-	-	-		-	-
stax:stax-api:jar:1.0.1	-	-	-	-		-	-
xml-apis:xml-apis:jar:1.0.b2	-	-	-	-		-	-
Total	**java.net-repo**	**codehaus.org**	**apache.snapshots**	**reflections-repo**	**central**	**codehaus-release-repo**	**info-bliki-repository**
30 (compile: 19, test: 9, runtime: 2)	0	0	0	1	28	3	1

7.2.1.4 Árbol de dependencias

- edu.uoc.pfc:specter-framework:jar:1.0.0-SNAPSHOT
  - org.reflections:reflections:jar:0.9.5-RC2 (compile)
    - com.google.collections:google-collections:jar:1.0 (compile)
    - javassist:javassist:jar:3.8.0.GA (compile)
    - ch.qos.logback:logback-classic:jar:0.9.9 (runtime)
      - ch.qos.logback:logback-core:jar:0.9.9 (runtime)
    - dom4j:dom4j:jar:1.6 (compile)
      - xml-apis:xml-apis:jar:1.0.b2 (compile)
    - com.google.code.gson:gson:jar:1.4 (compile)
    - javax.servlet:servlet-api:jar:2.5 (compile)
  - httpunit:httpunit:jar:1.7 (test)
    - jtidy:jtidy:jar:4aug2000r7-dev (test)
  - org.mortbay.jetty:jetty:jar:6.1.10 (test)
    - org.mortbay.jetty:jetty-util:jar:6.1.10 (test)
    - org.mortbay.jetty:servlet-api-2.5:jar:6.1.10 (test)
  - org.codehaus.jettison:jettison:jar:1.2 (test)
    - stax:stax-api:jar:1.0.1 (test)
  - rhino:js:jar:1.6R6 (test)
  - commons-beanutils:commons-beanutils:jar:1.8.3 (compile)
    - commons-logging:commons-logging:jar:1.1.1 (compile)
  - cglib:cglib:jar:2.2 (compile)
    - asm:asm:jar:3.1 (compile)
  - commons-io:commons-io:jar:2.0.1 (compile)
  - info.bliki.wiki:bliki-core:jar:3.0.16 (compile)
    - commons-httpclient:commons-httpclient:jar:3.1 (compile)
      - commons-codec:commons-codec:jar:1.2 (compile)
    - commons-lang:commons-lang:jar:2.4 (compile)
    - org.apache.commons:commons-compress:jar:1.0 (compile)
  - org.aspectj:aspectjrt:jar:1.6.10 (compile)
  - junit:junit:jar:4.4 (test)
  - org.slf4j:slf4j-api:jar:1.5.5 (compile)

7.2.2 specter-framework-taglibs

7.2.2.1 Ficheros

Filename	Size	Entries	Classes	Packages	JDK Rev	Debug
asm-3.1.jar	42.02 kB	24	23	2	1.2	release
cglib-2.2.jar	272.15 kB	240	226	7	1.2	debug
logback-classic-0.9.9.jar	116.70 kB	119	90	17	1.5	debug
logback-core-0.9.9.jar	188.05 kB	187	155	22	1.5	debug
gson-1.4.jar	162.47 kB	143	130	3	1.5	debug
google-collections-1.0.jar	624.60 kB	520	505	4	1.5	debug
commons-beanutils-1.8.3.jar	226.58 kB	155	137	6	1.3	debug
commons-codec-1.2.jar	29.38 kB	30	19	5	1.1	debug
commons-httpclient-3.1.jar	297.85 kB	183	167	8	1.2	debug
commons-io-2.0.1.jar	155.77 kB	122	104	6	1.5	debug
commons-lang-2.4.jar	255.67 kB	148	127	9	1.2	debug
commons-logging-1.1.1.jar	59.26 kB	42	28	2	1.1	debug
dom4j-1.6.jar	306.01 kB	208	190	14	1.3	debug
specter-framework-1.0.0-SNAPSHOT.jar	120.91 kB	117	80	23	1.6	debug
bliki-core-3.0.16.jar	426.38 kB	322	280	26	1.5	debug
javassist-3.8.0.GA.jar	580.04 kB	352	333	17	1.2	debug
servlet-api-2.5.jar	102.65 kB	68	42	2	1.5	debug
jsp-api-2.1.jar	98.28 kB	101	84	4	1.5	debug
junit-4.4.jar	157.69 kB	188	154	20	1.5	debug
commons-compress-1.0.jar	140.48 kB	94	70	11	1.4	debug
aspectjrt-1.6.10.jar	113.50 kB	145	128	10	1.5	debug
reflections-0.9.5-RC2.jar	79.68 kB	78	64	6	1.5	debug
slf4j-api-1.5.5.jar	21.29 kB	32	21	3	1.3	debug
xml-apis-1.0.b2.jar	106.76 kB	217	184	17	1.2	release
Total	**Size**	**Entries**	**Classes**	**Packages**	**JDK Rev**	**Debug**
24	4.57 MB	3,835	3,341	244	1.6	22
compile: 20	compile: 4.03 MB	compile: 3,240	compile: 2,858	compile: 181	-	compile: 18
test: 1	test: 157.69 kB	test: 188	test: 154	test: 20	-	test: 1
runtime: 2	runtime: 304.75 kB	runtime: 306	runtime: 245	runtime: 39	-	runtime: 2
provided: 1	provided: 98.28 kB	provided: 101	provided: 84	provided: 4	-	provided: 1

7.2.2.2 Repositorios

Repo ID	URL	Release	Snapshot
java.net-repo	http://download.java.net/maven/2/	Yes	Yes
apache.snapshots	http://people.apache.org/repo/m2-snapshot-repository	-	Yes
reflections-repo	http://reflections.googlecode.com/svn/repo	Yes	Yes
codehaus-release-repo	http://repository.codehaus.org	Yes	Yes
central	http://repo1.maven.org/maven2	Yes	-
info-bliki-repository	http://gwtwiki.googlecode.com/svn/maven-repository/	Yes	-

7.2.2.3 Coordenadas

Artifact	java.net-repo	apache.snapshots	reflections-repo	codehaus-release-repo	central	info-bliki-repository
asm:asm:jar:3.1	-	-	-	-	✓	-
cglib:cglib:jar:2.2	-	-	-	-	✓	-
ch.qos.logback:logback-classic:jar:0.9.9	-	-	-	-	✓	-
ch.qos.logback:logback-core:jar:0.9.9	-	-	-	-	✓	-
com.google.code.gson:gson:jar:1.4	-	-	-	-	✓	-
com.google.collections:google-collections:jar:1.0	-	-	-	-	✓	-
commons-beanutils:commons-beanutils:jar:1.8.3	-	-	-	-	✓	-
commons-codec:commons-codec:jar:1.2	-	-	-	-	✓	-
commons-httpclient:commons-httpclient:jar:3.1	-	-	-	-	✓	-
commons-io:commons-io:jar:2.0.1	-	-	-	-	✓	-
commons-lang:commons-lang:jar:2.4	-	-	-	-	✓	-
commons-logging:commons-logging:jar:1.1.1	-	-	-	-	✓	-
dom4j:dom4j:jar:1.6	-	-	-	-	✓	-
edu.uoc.pfc:specter-framework:jar:1.0.0-SNAPSHOT	-	-	-	-	-	-
info.bliki.wiki:bliki-core:jar:3.0.16	-	-	-	-	-	✓
javassist:javassist:jar:3.8.0.GA	-	-	-	-	✓	-
javax.servlet:servlet-api:jar:2.5	-	-	-	-	✓	-
javax.servlet.jsp:jsp-api:jar:2.1	-	-	-	-	✓	-
junit:junit:jar:4.4	-	-	-	-	✓	-
org.apache.commons:commons-compress:jar:1.0	-	-	-	-	✓	-
org.aspectj:aspectjrt:jar:1.6.10	-	-	-	-	✓	-
org.reflections:reflections:jar:0.9.5-RC2	-	-	✓	-	-	-
org.slf4j:slf4j-api:jar:1.5.5	-	-	-	-	✓	-
xml-apis:xml-apis:jar:1.0.b2	-	-	-	-	✓	-
Total	**java.net-repo**	**apache.snapshots**	**reflections-repo**	**codehaus-release-repo**	**central**	**info-bliki-repository**
24 (compile: 20, test: 1, runtime: 2, provided: 1)	0	0	1	0	21	1

7.2.2.4 Árbol de dependencias

- edu.uoc.pfc:specter-framework-taglibs:jar:1.0.0-SNAPSHOT
  - javax.servlet.jsp:jsp-api:jar:2.1 (provided)
  - edu.uoc.pfc:specter-framework:jar:1.0.0-SNAPSHOT (compile)
    - org.reflections:reflections:jar:0.9.5-RC2 (compile)
      - com.google.collections:google-collections:jar:1.0 (compile)
      - javassist:javassist:jar:3.8.0.GA (compile)
      - ch.qos.logback:logback-classic:jar:0.9.9 (runtime)
        - ch.qos.logback:logback-core:jar:0.9.9 (runtime)
      - dom4j:dom4j:jar:1.6 (compile)
        - xml-apis:xml-apis:jar:1.0.b2 (compile)
      - com.google.code.gson:gson:jar:1.4 (compile)
      - javax.servlet:servlet-api:jar:2.5 (compile)
    - commons-beanutils:commons-beanutils:jar:1.8.3 (compile)
      - commons-logging:commons-logging:jar:1.1.1 (compile)
    - cglib:cglib:jar:2.2 (compile)
      - asm:asm:jar:3.1 (compile)
    - commons-io:commons-io:jar:2.0.1 (compile)
    - info.bliki.wiki:bliki-core:jar:3.0.16 (compile)
      - commons-httpclient:commons-httpclient:jar:3.1 (compile)
        - commons-codec:commons-codec:jar:1.2 (compile)
      - commons-lang:commons-lang:jar:2.4 (compile)
      - org.apache.commons:commons-compress:jar:1.0 (compile)
    - org.aspectj:aspectjrt:jar:1.6.10 (compile)
    - org.slf4j:slf4j-api:jar:1.5.5 (compile)
  - junit:junit:jar:4.4 (test)
  - org.slf4j:slf4j-api:jar:1.5.5 (compile)