Desarrollo de aplicaciones con J2EE
Proyecto onLineStore

Javier Belzunce Flores
ETIG / ETIS

Verònica Peña Pastor

26 de junio de 2008

El presente documento representa la memoria del proyecto de final de carrera ubicado en el área de J2EE. En él se realiza un estudio detallado del framework JavaServer Faces (de ahora en adelante, JSF's) para el desarrollo de aplicaciones Web, y muestra como integrarla con otros marcos de trabajo, específicamente, Spring y JPA. Para ello, crearemos la aplicación Web onLineStore, un sistema de catálogo de productos online. Con ello, cubriremos todas las fases del diseño de una aplicación Web, incluyendo el descubrimiento de los requisitos del negocio, el análisis, la selección de tecnologías, la arquitectura de alto nivel, y el diseño a nivel de la implementación. Discutiremos las ventajas y desventajas de las tecnologías utilizadas y mostraremos aproximaciones para diseñar algunos de los aspectos clave de la aplicación.

Este trabajo no es simplemente un trabajo de desarrollo, sino también de investigación y de adquisición de los conocimientos teóricos y prácticos necesarios para llevar a término el proyecto.

Este documento incluye pues, una explicación del marco de referencia en el cual se desarrolla el proyecto, la justificación de su necesidad, los objetivos que se pretenden conseguir, la metodología empleada, la descripción de las fases de análisis y diseño, el producto obtenido y las conclusiones del trabajo.

Tabla de Contenidos

Índice de figuras

1. Introducción

Este documento provee una visión global del enfoque de desarrollo propuesto para la ejecución del proyecto de final de carrera onLineStore. El proyecto se lleva a cabo bajo la metodología de Rational Unified Process. Es importante destacar esto puesto que utilizaremos la terminología RUP en este documento. Se incluirá el detalle para las fases de Inicio y Elaboración y adicionalmente se esbozarán las fases posteriores de Construcción y Transición para dar una visión global de todo proceso.

1.1 Propósito alcance y objetivos

Las páginas de servidor activas de Java (JavaServer Faces, de ahora en adelante JSF's) se están convirtiendo en tecnología líder en la publicación dinámica de sitios Web. Este método, que se basa en el uso de un lenguaje que se está imponiendo como estándar para Internet, es un nuevo marco de trabajo para interfaces de usuario en aplicaciones J2EE. Por diseño, es particularmente útil con aplicaciones basadas en la arquitectura MVC (Model-View-Controller) y provee la facilidad de uso que se necesita para hacer disponibles páginas HTML dinámicamente en sitios Web.

La siguiente figura muestra la tendencia a la utilización, por parte de los desarrolladores, de los diferentes frameworks para J2EE en el mercado, durante los últimos 3 años:

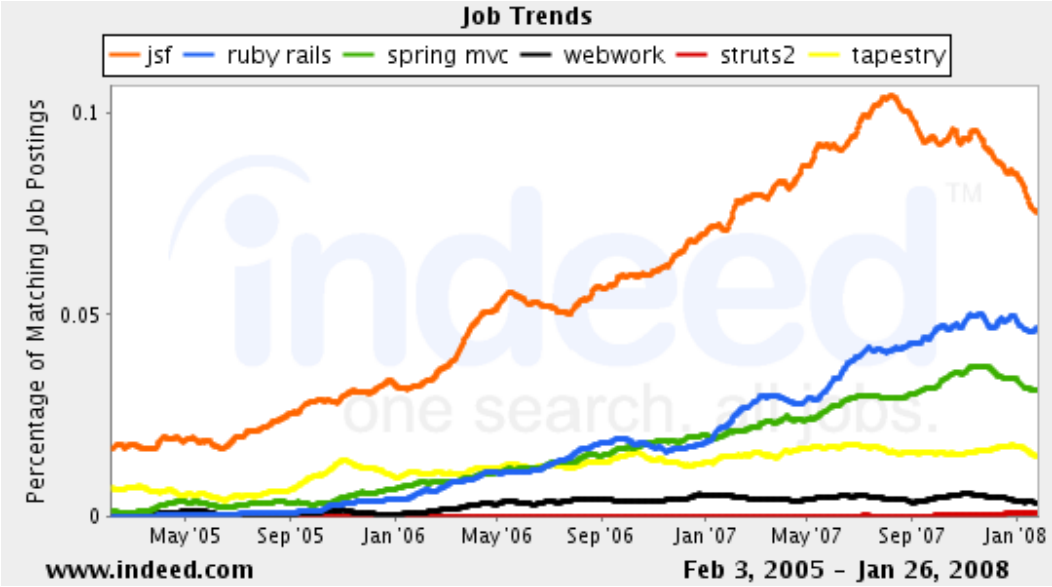

Figura 1. Utilización de los diferentes frameworks para J2EE

A grandes rasgos, esta tendencia a utilizar JSF's como framework preferido de trabajo en el desarrollo de aplicaciones J2EE se puede explicar por los siguientes motivos:

1. El soporte de JSF en IDEs como Eclipse, Netbeans, etc. es mucho mejor
2. Constantemente se crean nuevos componentes JSF
3. Gran soporte de JSF en la industria
4. JSF es parte de Java EE (Struts, por ejemplo, no)

5. Todos los servidores de aplicaciones, por tanto, incluyen JSF
6. Tendencias o modas

A nivel técnico, una de las grandes ventajas de la tecnología JSF's es que ofrece una clara separación entre el comportamiento y la presentación. Las aplicaciones Web construidas con tecnología JSP conseguían parcialmente esta separación. Sin embargo, una aplicación JSP no puede mapear peticiones HTTP al manejo de eventos específicos de componentes o manejar elementos UI como objetos con estado en el servidor. La tecnología JSF's nos permite construir aplicaciones Web que implementan una separación entre el comportamiento y la presentación tradicionalmente ofrecidas por arquitectura UI del lado del cliente.

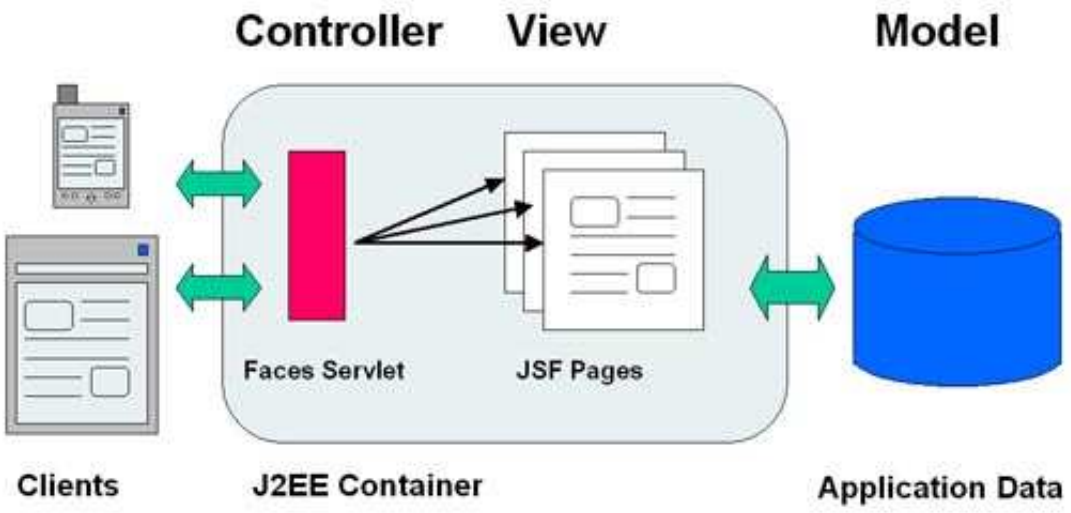

Figura 2. Modelo vista controlador

La separación de la lógica de la presentación también le permite a cada miembro del equipo de desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo, y proporciona un sencillo modelo de programación para enlazar todas las piezas. Por ejemplo, los Autores de páginas sin experiencia en programación pueden usar las etiquetas de componentes UI de la tecnología JSF's para enlazar código de la aplicación desde dentro de la página Web sin escribir ningún script.

Otro objetivo importante de la tecnología JSF's es mejorar los conceptos familiares de componente-UI y capa-Web sin limitarnos a una tecnología de script particular o un lenguaje de marcas. Aunque la tecnología JSF's incluye una librería de etiquetas JSP personalizadas para representar componentes en una página JSP, los APIs de la tecnología JSF's se han creado directamente sobre el API JavaServlet. Esto nos permite hacer algunas cosas: usar otra tecnología de presentación junto a JSP, crear nuestros propios componentes personalizados directamente desde las clases de componentes, y generar salida para diferentes dispositivos cliente.

Pero lo más importante, la tecnología JSF's proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario, y manejar eventos.

El presente proyecto es un estudio detallado de esta tecnología y muestra como integrarla con otros marcos de trabajo, específicamente, Spring y JPA. Para ello, crearemos la aplicación Web onLineStore, un sistema de catálogo de productos online. Usando el ejemplo onLineStore, cubriremos todas las fases del diseño de una aplicación Web, incluyendo el descubrimiento de los requisitos del negocio, el análisis, la selección de tecnologías, la arquitectura de alto nivel, y el diseño a nivel de la implementación. Discutiremos las ventajas y desventajas de las tecnologías utilizadas y mostraremos aproximaciones para diseñar algunos de los aspectos clave de la aplicación.

La aplicación onLineStore, es una aplicación Web del mundo real, suficientemente realista para proporcionar una importante discusión sobre las decisiones arquitecturales de una aplicación Web implementada bajo JSF's.

1.2 Entregables del proyecto

A continuación se indican y describen cada uno de los artefactos generados y utilizados por el proyecto y que constituyen los entregables. Esta lista constituye la configuración de RUP desde la perspectiva de artefactos para este proyecto.

1) **Plan de Desarrollo del Software**
   Documento donde se describe todo el proceso de desarrollo del proyecto

2) **Glosario**
   Es un documento que define los principales términos usados en el proyecto. Permite establecer una terminología consensuada.

3) **Modelo de Casos de Uso**
   El modelo de Casos de Uso presenta las funciones del sistema y los actores que hacen uso de ellas. Se representa mediante Diagramas de Casos de Uso.

4) **Modelo de Análisis y Diseño**
   Este modelo establece la realización de los casos de uso en clases y pasando desde una representación en términos de análisis (sin incluir aspectos de implementación) hacia una de diseño (incluyendo una orientación hacia el entorno de implementación), de acuerdo al avance del proyecto.

5) **Modelo de Datos**
   Previendo que la persistencia de la información del sistema será soportada por una base de datos relacional, este modelo describe la representación lógica de los datos persistentes, de acuerdo con el enfoque para modelado relacional de datos. Para expresar este modelo se utiliza un Diagrama de Clases (donde se utiliza un profile UML para Modelado de Datos, para conseguir la representación de tablas, claves, etc.) .

6) **Modelo de Implementación**
   Este modelo es una colección de componentes y los subsistemas que los contienen. Estos componentes incluyen: ficheros ejecutables, ficheros de código fuente, y todo otro tipo de ficheros necesarios para la implantación y despliegue del sistema. (Este modelo es sólo una versión preliminar al final de la fase de Elaboración, posteriormente tiene bastante refinamiento).

7) **Producto**
   Los ficheros del producto empaquetados y almacenadas en un CD con los mecanismos apropiados para facilitar su instalación. El producto, a partir de la primera iteración de la fase de Construcción es desarrollado incremental e iterativamente, obteniéndose una nueva release al final de cada iteración.

1.3 Organización del Proyecto

1.3.1 Participantes en el Proyecto

El proyecto será realizado en su totalidad por Xavier Belzunce Flores, asumiendo los roles típicos en el desarrollo de cualquier proyecto (jefe de proyecto, analista, programador, etc.).

A su vez, éste será tutelado por la consultora Verònica Peña Pastor de la Universitat Oberta de Catalunya.

1.4 Gestión del Proceso

1.4.1 Plan del Proyecto

En esta sección se presenta la organización en fases e iteraciones y el calendario del proyecto.

1.4.2 Plan de las Fases

El desarrollo se llevará a cabo en base a fases con una o más iteraciones en cada una de ellas. La siguiente tabla muestra una la distribución de tiempos y el número de iteraciones de cada fase (para las fases de Construcción y Transición es sólo una aproximación muy preliminar)

Fase	**Nro. Iteraciones**	**Duración**
Fase de Inicio	1	1 semana
Fase de Elaboración	1	5 semanas
Fase de Construcción	1	5 semanas

Los hitos que marcan el final de cada fase se describen en la siguiente tabla.

Descripción	**Hito**
Fase de Inicio	En esta fase desarrollarán los requisitos del producto. Los principales casos de uso serán identificados y se hará un refinamiento del Plan de Desarrollo del Proyecto. La aceptación del artefacto Plan de Desarrollo marcan el final de esta fase.
Fase de Elaboración	En esta fase se analizan los requisitos y se desarrolla un prototipo de arquitectura (incluyendo las partes más relevantes y / o críticas del sistema). Al final de esta fase, todos los casos de uso correspondientes a requisitos que serán implementados en la fase de Construcción deben estar analizados y diseñados (en el Modelo de Análisis / Diseño). La revisión y aceptación del prototipo de la arquitectura del sistema marca el final de esta fase.
Fase de Construcción	Durante la fase de construcción se construye el producto produciendo una release a la cual se le aplican las pruebas y se valida. El hito que marca el fin de esta fase es la versión de la release 1.0, con la capacidad operacional del producto que se haya considerado como crítica, lista para ser entregada. .

1.4.3 Calendario del Proyecto

A continuación se presenta un calendario de las principales tareas del proyecto incluyendo sólo las fases de Inicio, Elaboración y Construcción. Como se ha comentado, el proceso iterativo e incremental de RUP está caracterizado por la realización en paralelo de todas las disciplinas de desarrollo a lo largo del proyecto, con lo cual la mayoría de los artefactos son generados muy tempranamente en el proyecto pero van desarrollándose en mayor o menor grado de acuerdo a la fase e iteración del proyecto. La siguiente figura ilustra este enfoque, en ella lo ensombrecido marca el énfasis de cada disciplina (workflow) en un momento determinado del desarrollo.

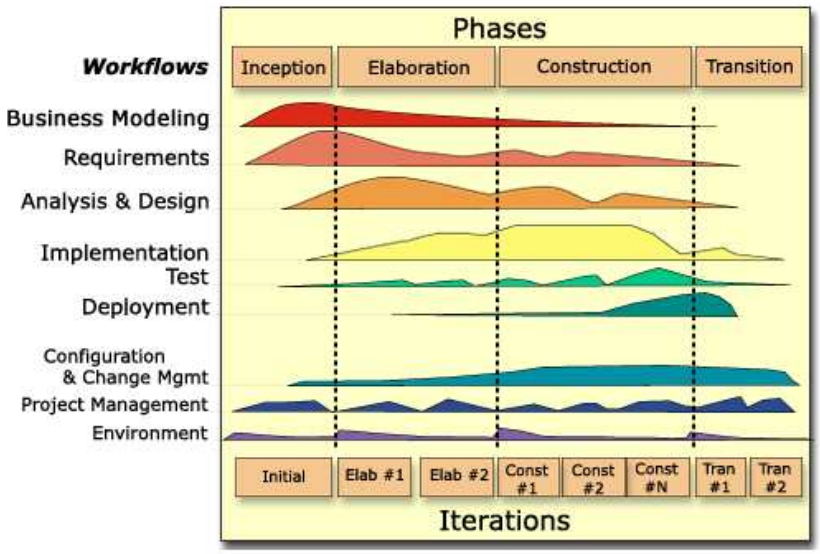

Figura 3. Fases e iteraciones del proceso

Para este proyecto se ha establecido el siguiente calendario. La fecha de finalización indica cuándo el artefacto en cuestión tiene un estado de completitud suficiente para someterse a revisión y aprobación, pero esto no quita la posibilidad de su posterior refinamiento y cambios. El final de la primera fase de elaboración, que engloba requisitos, análisis y diseño, finaliza la semana 5 coincidiendo con la fecha de entrega de la segunda PAC. Para la fase de construcción, el modelo de despliegue marca el final de la fase coincidiendo con la fecha de entrega de la tercera PAC.

Disciplinas / Artefactos generados o modificados	Comienzo	Finalización
Requisitos		
Glosario	Semana 1 17/03	Semana 1 17/03
Modelo de Casos de Uso	Semana 1 18/03	Semana 2 24/03
Especificación de Casos de Uso	Semana 1 18/03	Semana 2 24/03
Análisis/Diseño		
Modelo de Análisis/Diseño	Semana 2 25/03	Semana 4 11/04
Modelo de Datos	Semana 4 12/04	Semana 5 14/04
Implementación		
Prototipos de Interfaces de Usuario	Semana 5 15/04	Semana 5 18/04
Modelo de Implementación	Semana 6 21/04	Semana 9 15/05
Pruebas		

Casos de Pruebas Funcionales	Semana 9 15/05	Semana 9 17/05
Despliegue		
Modelo de Despliegue	Semana 9 18/05	Semana 10 19/05
Gestión de Cambios y Configuración	Durante todo el proyecto	

1.5 Seguimiento y Control del Proyecto

Control de Plazos

El calendario del proyecto tendrá un seguimiento y evaluación a definir por el consultor.

Control de Calidad

Los defectos detectados en las revisiones y formalizados también en una Solicitud de Cambio tendrán un seguimiento para asegurar la conformidad respecto de la solución de dichas deficiencias Para la revisión de cada artefacto y su correspondiente garantía de calidad se utilizarán las guías de revisión y checklist (listas de verificación) incluidas en RUP.

1.6 Planes y guías de aplicación

La gestión del proyecto se llevará a cabo bajo la metodología de Rational Unified Process. Para el análisis y diseño se utilizará la notación UML con la herramienta StartUML.

La aplicación se desarrollará con la tecnología JavaServer Faces. Como IDE se trabajará con netBeans 6.0.1 (JDK 1.5), utilizando como servidor de aplicaciones GlassFish 2.1 de SUN, que ya viene integrado con netBeans. Como gestor de base de Datos se utilizará Derby, que viene integrada con GlassFish. Para la persistencia, se implementará con Oracle TopLink, y Spring como marco de trabajo para abstraer el acceso a datos.

En las siguientes secciones, veremos un estudio detallado de las tecnologías implicadas en el desarrollo de la aplicación, presentaremos la fases de elaboración y explicando con detalle el modelo de casos de uso, de análisis y diseño. También estudiaremos la fase de construcción, donde expondremos las diferentes soluciones dadas para la final elaboración de la aplicación.

2. La tecnología JavaServer Faces

La tecnología JavaServer Faces es un marco de trabajo de interfaces de usuario del lado de servidor para aplicaciones Web basadas en tecnología Java.

Los principales componentes de la tecnología JavaServer Faces son:

1. Un API y una implementación de referencia para: representar componentes UI y manejar su estado; manejo de eventos, validación del lado del servidor y conversión de datos; definir la navegación entre páginas; soportar internacionalización y accesibilidad; y proporcionar extensibilidad para todas estas características.

2. Una librería de etiquetas JavaServer Pages (JSP) personalizadas para dibujar componentes UI dentro de una página JSP.

Este modelo de programación bien definido y la librería de etiquetas para componentes UI facilita de forma significativa la tarea de la construcción y mantenimiento de aplicaciones Web con UIs del lado del servidor.

Con un mínimo esfuerzo, podemos:

- Conectar eventos generados en el cliente a código de la aplicación en el lado del servidor.
- Mapear componentes UI a una página de datos del lado del servidor.
- Construir un UI con componentes reutilizables y extensibles.
- Grabar y restaurar el estado del UI más allá de la vida de las peticiones de servidor.

Como se puede apreciar en la siguiente figura, el interface de usuario que creamos con la tecnología JavaServer Faces (representado por myUI en el gráfico) se ejecuta en el servidor y se renderiza en el cliente.

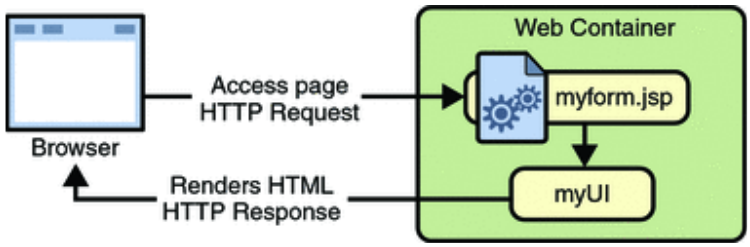

Figura 4. Arquitectura JSF's.

La página JSP, myform.jsp, dibuja los componentes del interface de usuario con etiquetas personalizadas definidas por la tecnología JavaServer Faces. El UI de la aplicación Web (representado por myUI en la magen) maneja los objetos referenciados por la página JSP:

- Los objetos componentes que mapean las etiquetas sobre la página JSP.
- Los oyentes de eventos, validadores, y los conversores que está registrados en los componentes.
- Los objetos del modelo que encapsulan los datos y las funcionalidades de los componentes específicos de la aplicación.

2.1 Beneficios de la tecnología JavaServer Faces

Una de las grandes ventajas de la tecnología JavaServer Faces es que ofrece una clara separación entre el comportamiento y la presentación. Las aplicaciones Web construidas con tecnología JSP conseguían parcialmente esta separación. Sin embargo, una aplicación JSP no puede mapear peticiones HTTP al manejo de eventos específicos de componentes o manejar elementos UI como objetos con estado en el servidor. La tecnología JavaServer Faces nos permite construir aplicaciones Web que implementan unas separaciones entre el comportamiento y la presentación tradicionalmente ofrecidas por arquitectura UI del lado del cliente.

La separación de la lógica de la presentación también le permite a cada miembro del equipo de desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo, y proporciona un sencillo modelo de programación para enlazar todas las piezas. Por ejemplo, los Autores de páginas sin experiencia en programación pueden usar las etiquetas de componentes UI de la tecnología JavaServer Faces para enlazar código de la aplicación desde dentro de la página Web sin escribir ningún script.

Otro objetivo importante de la tecnología JavaServer Faces es mejorar los conceptos familiares de componente-UI y capa-Web sin limitarnos a una tecnología de script particular o un lenguaje de marcas. Aunque la tecnología JavaServer Faces incluye una librería de etiquetas JSP personalizadas para representar componentes en una página JSP, los APIs de la tecnología JavaServer Faces se han creado directamente sobre el API JavaServlet. Esto nos permite hacer algunas cosas: usar otra tecnología de presentación junto a JSP, crear nuestros propios componentes personalizados directamente desde las clases de componentes, y generar salida para diferentes dispositivos cliente.

Pero lo más importante, la tecnología JavaServer Faces proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario, y manejar eventos.

2.2 ¿Qué es una aplicación JavaServerFaces?

En su mayoría, las aplicaciones JavaServer Faces son como cualquier otra aplicación Web Java. Se ejecutan en un contenedor Servlet Java, y típicamente contienen:

- Componentes JavaBeans (llamados objetos del modelo en tecnología JavaServer Faces) conteniendo datos y funcionalidades específicas de la aplicación.
- Oyentes de Eventos.
- Páginas, cómo páginas JSP.
- Clases de utilidad del lado del servidor, como beans para acceder a las bases de datos.

Además de estos ítems, una aplicación JavaServer Faces también tiene:

- Una librería de etiquetas personalizadas para dibujar componentes UI en una página.
- Una librería de etiquetas personalizadas para representar manejadores de eventos, validadores, y otras acciones.
- Componentes UI representados como objetos con estado en el servidor.
- Validadores, manejadores de eventos y manejadores de navegación.
- Toda aplicación JavaServer Faces debe incluir una librería de etiquetas personalizadas que define las etiquetas que representan componentes UI y una librería de etiquetas para representar otras acciones importantes, como validadores y manejadores de eventos. La implementación de JavaServer Faces proporciona estas dos librerías.

La librería de etiquetas de componentes elimina la necesidad de codificar componentes UI en HTML u otro lenguaje de marcas, resultando en componentes completamente reutilizables. Y, la librería "core" hace fácil registrar eventos, validadores y otras acciones de los componentes.

La librería de etiquetas de componentes puede se la librería html_basic incluida con la implementación de referencia de la tecnología JavaServer Faces, o podemos definir nuestra propia librería de etiquetas que dibuje componentes personalizados o que dibuje una salida distinta a HTML.

Otra ventaja importante de las aplicaciones JavaServer Faces es que los componentes UI de la página están representados en el servidor como objetos con estado. Esto permite a la aplicación manipular el estado del componente y conectar los eventos generados por el cliente a código en el lado del servidor.

Finalmente, la tecnología JavaServer Faces nos permite convertir y validar datos sobre componentes individuales y reportar cualquier error antes de que se actualicen los datos en el lado del servidor.

2.3 Pasos para crear una aplicación con JavaServer Faces

Desarrollar una sencilla aplicación JavaServer Faces requiere la realización de estos pasos:

1. Desarrollar los objetos del modelo, los que contendrán los datos.
2. Añadir las declaraciones del bean controlado al fichero de configuración de la aplicación.
3. Crear las páginas utilizando las etiquetas de componentes UI y las etiquetas "core".
4. Definir la navegación entre las páginas.

2.3.1 Desarrollar los objetos del modelo. Bean Manejado y Bean de respaldo

JSF presenta dos nuevos términos: managed bean (bean manejado) y backing bean (bean de respaldo). JSF proporciona una fuerte facilidad de bean manejado. Los objetos JavaBean manejados por una implementación JSF se llaman beans manejados. Un bean manejado describe como se crea y se maneja un bean. No tiene nada que ver con las funcionalidades del bean.

El bean de respaldo define las propiedades y las lógicas de manejo asociadas con los componentes UI utilizados en la página. Cada propiedad del bean de respaldo está unida a un ejemplar de un componente o a su valor. Un bean de respaldo también define un conjunto de métodos que realizan funciones para el componente, como validar los datos del componente, manejar los eventos que dispara el componente, y realizar el procesamiento asociado con la navegación cuando el componente se activa.

Una típica aplicación JSF acopla un bean de repaldo con cada página de la aplicación. Sin embargo, algunas veces en el mundo real, forzar una relación uno-a-uno entre el bean de respaldo y la página no es la solución ideal. Puede causar problemas como la duplicación de código. En el escenario del mundo real, varias páginas podrían necesitar compartir el mismo bean de respaldo detrás de la escena.

Un objeto view (vista) es un objeto modelo utilizado específicamente en la capa de presentación. Contiene los datos que debe mostrar en la capa de la vista y la lógica para validar la entrada del usuario, manejar los eventos, e interactuar con la capa de lógica-de-negocio. El bean de respaldo es el objeto vista en una aplicación basada en JSF. En comparación con la aproximación ActionForm y Action de Struts, el desarrollo con beans de respaldo de JSF sigue unas mejores prácticas de diseño orientado a objetos. Un bean de respaldo no sólo contiene los datos a ver, también el comportamiento relacionado con esos datos. En Struts, Action y ActionForm contienen los datos y la lógica por separado.

2.3.2 Añadir las declaraciones del bean controlado

Después de desarrollar los beans utilizados en la aplicación, necesitamos añadir declaraciones para ellos en el fichero de configuración de la aplicación.. Aquí tenemos la declaración de bean controlado para UserNumberBean:

```
<managed-bean>
  <managed-bean-name>UserNumberBean</managed-bean-name>
    <managed-bean-class>
      guessNumber.UserNumberBean
    </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

La implementación de JavaServer Faces procesa este fichero en el momento de arrancada de la aplicación e inicializa el UserNumberBean y lo almacena en el ámbito de sesión. Entonces el bean estará disponible para todas las páginas de la aplicación. Para aquellos que estén familiarizados con versiones anteriores, esta facilidad de "bean controlado" reemplaza la utilización de la etiqueta jsp:useBean.

2.3.3 Crear las páginas

La creación de las páginas implica distribuir los componentes UI en las páginas, mapear los componentes a los datos de los objetos del modelo, y añadir otras etiquetas importantes (como etiquetas del validador) a las etiquetas de los componentes.

2.3.4 Definir la navegación por las páginas

Otra posibilidad que tiene el desarrollador de la aplicación es definir la navegación de páginas por la aplicación, como qué página va después de que el usuario pulse un botón para enviar un formulario.

El desarrollador de la aplicación define la navegación por la aplicación mediante el fichero de configuración, el mismo fichero en el que se declararon los beans manejados. Aquí están las reglas de navegación definidas para el ejemplo guessNumber:

```
<navigation-rule>
  <from-tree-id>/greeting.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/response.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-tree-id>/response.jsp</from-tree-id>
    <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/greeting.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
```

Cada regla de navegación define cómo ir de una página (especificada en el elemento from-tree-id) a otras páginas de la aplicación. El elemento navigation-rule puede contener cualquier número de elemento navigation-case, cada uno de los cuales define la página que se abrirá luego (definida por to-tree-id) basándose en una salida lógica (definida mediante from-outcome).

La salida se puede definir mediante el atributo action del componente UICommand que envía el formulario, como en el ejemplo guessNumber:

```
<h:command_button id="submit" action="success" label="Submit" />
```

La salida también puede venir del valor de retorno del método invoke de un objeto Action. Este método realiza algún procesamiento para determinar la salida. Un ejemplo es que el método invoke puede chequear si la password que el usuario ha introducido en la página corresponde con la del fichero. Si es así, el método invoke podría devolver "success"; si no es así, podría devolver "failure". Un salida de "failure" podría resultar en la recarga de la página de logon. Una salida de "success" podría resultar en que se mostrara una página con las actividades de la tarjeta de crédito del usuario, por ejemplo.

2.4 Ciclo de vida de una página JavaServer Faces

El ciclo de vida de una página JavaServer Faces page is similar al de una página JSP: El cliente hace una petición HTTP de la página y el servidor responde con la página traducida a HTML. Sin embargo, debido a las características extras que ofrece la tecnología JavaServer Faces, el ciclo de vida proporciona algunos servicios adicionales mediante la ejecución de algunos pasos extras.

Los pasos del ciclo de vida se ejecutan dependen de si la petición se originó o no desde una aplicación JavaServer Faces y si la respuesta es o no generada con la fase de renderizado del ciclo de vida de JavaServer Faces

La mayoría de los usuarios de la tecnología JavaServer Faces no necesitarán conocer a fondo el ciclo de vida de procesamiento de una petición. Sin embargo, conociendo lo que la tecnología JavaServer Faces realiza para procesar una página, un desarrollador de aplicaciones JavaServer Faces no necesitará preocuparse de los problemas de renderizado asociados con otras tecnologías UI. Un ejemplo sería el cambio de estado de los componentes individuales. Si la selección de un componente como un checkbox afecta a la apariencia de otro componente de la página, la tecnología JavaServer Faces manejará este evento de la forma apropiada y no permitirá que se dibuje la página sin reflejar este cambio.

La siguiente figura ilusta los pasos del ciclo de vida petición-respuesta JavaServer Faces

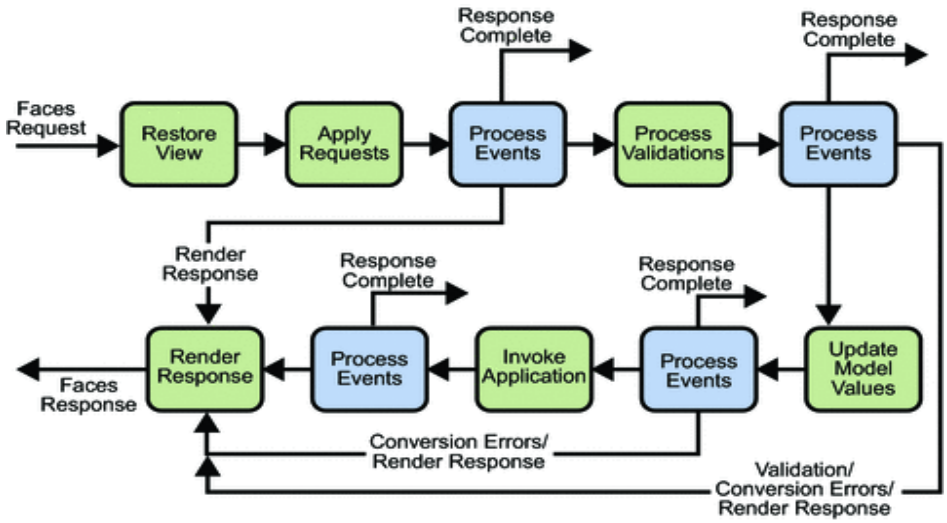

Figura 5. Ciclo de vida JSF's.

2.4.1 Reconstituir el Árbol de Componentes

Cuando se hace una petición para una página JavaServer Faces, como cuando se pulsa sobre un enlace o un botón, la implementación JavaServer Faces comienza el estado Reconstituir el Árbol de Componentes. (restore view phase)
Durante esta fase, la implementación JavaServer Faces construye el árbol de componentes de la página JavaServer Faces, conecta los manejadores de eventos y los validadores y graba el estado en el FacesContext.

2.4.2 Aplicar Valores de la Petición

Una vez construido el árbol de componentes, cada componente del árbol extrae su nuevo valor desde los parámetros de la petición con su método decode. Entonces el valor es almacenado localmente en el componente. Si falla la conversión del valor, se genera un mensaje de error asociado con el componente y se pone en la cola de FacesContext. Este mensaje se mostrará durante la fase Renderizar la Respuesta, junto con cualquier error de validación resultante de la fase Procesar Validaciones.

Si durante esta fase se produce algún evento, la implementación JavaServer Faces emite los eventos a los oyentes interesados.

En este punto, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete. En el caso del componente userNumber de la página greeting.jsp, el valor es cualquier cosa que el usuario introduzca en el campo. Como la propiedad del objeto del model unida al componente tiene un tipo Integer, la implementación JavaServer Faces convierte el valor de un String a un Integer.

En este momento, se han puesto los nuevos valores en los componentes y los mensajes y eventos se han puesto en sus colas.

2.4.3 Procesar Validaciones

Durante esta fase, la implementación JavaServer Faces procesa todas las validaciones registradas con los componentes del árbol. Examina los atributos del componente que especifican las reglas de validación y compara esas reglas con el valor local almacenado en el componente. Si el valor local no es válido, la implementación JavaServer Faces añade un mensaje de error al FacesContext y el ciclo de vida avanza directamente hasta la fase Renderizar las Respuesta para que la página sea dibujada de nuevo incluyendo los mensajes de error. Si había errores de conversión de la fase Aplicar los Valores a la Petición, también se mostrarán.

En este momento, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si se han disparado eventos durante esta fase, la implemetnación JavaServer Faces los emite a los oyentes interesados.

En la página greeting.jsp, la implementación JavaServer Faces procesa el validador sobre la etiqueta input_number de UserNumber. Verifica que el dato introducido por el usuario en el campo de texto es un entero entre 0 y 10. Si el dato no es válido, o ocurrió un error de converisón durante la fase Aplicar los Valores a la Petición, el procesamiento salta a la fase Renderizar las Respuesta, durante la que se dibujará de nuevo la página greeting.jsp mostrando los mensajes de error de conversión o validación en el componente asociado con la etiqueta output_errors.

2.4.4 Actualizar los Valores del Modelo

Una vez que la implementación JavaServer Faces determina que el dato es válido, puede pasar por el árbol de componentes y configurar los valores del objeto de modelo correspondiente con los valores locales de los componentes. Sólo se actualizarán los componentes que tengan expresiones valueRef. Si el dato local no se puede convertir a los tipos especificados por las propiedades del objeto del modelo, el ciclo de vida avanza directamente a la fase Renderizar las Respuesta, durante la que se dibujará de nuevo la página mostrando los errores, similar a lo que sucede con los errores de validación.

En este punto, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete. Si se han disparado eventos durante esta fase, la implemetnación JavaServer Faces los emite a los oyentes interesados.

En esta fase, a la propiedad userNumber del UserNumberBean se le da el valor del componente userNumber.

2.4.5 Invocar Aplicación

Durante esta fase, la implementación JavaServer Faces maneja cualquier evento a nivel de aplicación, como enviar un formulario o enlazar a otra página.
En este momento, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

La página greeting.jsp del ejemplo guessNumber tiene asociado un evento a nivel de aplicación con el componente Command. Cuando se procesa este evento, una implementación de ActionListener por defecto recupera la salida, “success”, desde el atributo action del componente. El oyente pasa la salida al NavigationHandler por defecto. Y éste contrasta la salida con las reglas de navegación definidas en el fichero de configuración de la aplicación para determinar qué página se debe mostrar luego.

Luego la implementación JavaServer Faces configura el árbol de componentes de la respuesta a esa nueva página. Finalmente, la implementación JavaServer Faces transfiere el control a la fase Renderizar la Respuesta.

2.4.6 Renderizar la Respuesta

Durante esta fase, la implementación JavaServer Faces invoca las propiedades de codificación de los componentes y dibuja los componentes del árbol de componentes grabado en el FacesContext.

Si se encontraron errores durante las fases Aplicar los Valores a la Petición, Procesar Validaciones o Actualizar los Valores del Modelo, se dibujará la página original. Si las páginas contienen etiquetas output_errors, cualquier mensaje de error que haya en la cola se mostrará en la página.

Se pueden añadir nuevos componentes en el árbol si la aplicación incluye renderizadores personalizados, que definen cómo renderizar un componente. Después de que se haya renderizado el contenido del árbol, éste se graba para que las siguientes peticiones puedan acceder a él y esté disponible para la fase Reconstituir el Árbol de Componentes de las siguientes llamadas.

3. Spring Framework

Spring es un marco de trabajo de aplicaciones J2EE, publicado bajo la Apache Software License. Spring proporciona servicios para todas las capas arquitecturales, pero es particularmente potente en cuanto a la persistencia proporcionando una aproximación consistente para acceder a datos. La capa de servicios de persistencia de Spring se acompaña con una implementación del patrón de diseño J2EE Data Access Objects (DAO).

Spring contiene muchas características que le dan una funcionalidad muy basta; dichas características están organizadas en siete grandes módulos como se puede observar en el diagrama de abajo. Esta sección comenta someramente las características de cada módulo.

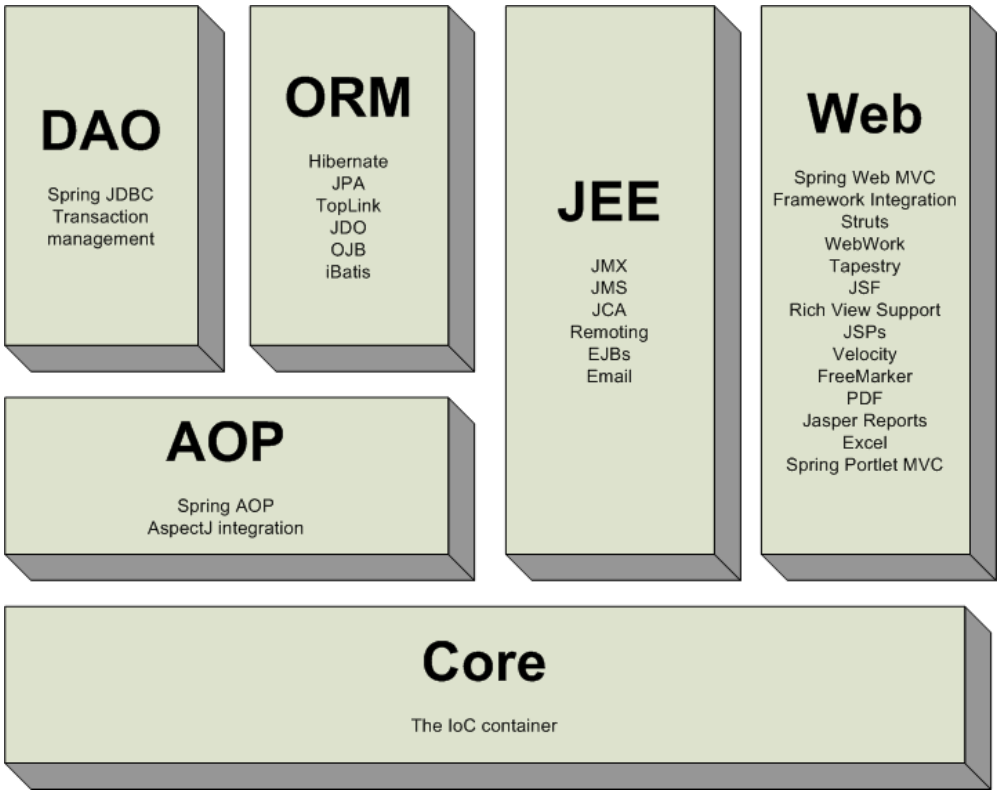

Figura 6. Modulos Spring Framework

- El módulo Core o "Núcleo" es la parte fundamental del framework ya que provee toda la funcionalidad de Inyección de Dependencias permitiéndote administrar la funcionalidad del contenedor de beans. El concepto básico de este módulo es el BeanFactory, que implementa el patrón de diseño Factory (fábrica) eliminando la necesidad de crear singletons programáticamente permitiéndote desligar la configuración y especificación de las dependencias de tu lógica de programación.

- Encima del módulo core se encuentra el módulo Context (Contexto), el cual te provee de herramientas para acceder a los beans de una manera elegante, similar a un registro JNDI. El paquete de contexto hereda sus características del paquete de beans y añade soporte para mensajería de texto, como son resource bundles (para internacionalización), propagación de eventos, carga de recursos y creación transparente de contextos por contenedores (como el

contenedor de servlets, por ejemplo).

- El paquete DAO provee una capa de abstracción de JDBC que elimina la necesidad de teclear código JDBC tedioso y redundante así como el parseo de códigos de error específicos de cada proveedor de base de datos. También, el paquete JDBC provee de una manera de administrar transacciones tanto declarativas como programáticas, no solo para clases que implementen interfaces especiales, sino tambien para POJOs.

- El paquete ORM provee capas de integración para APIs de mapeo objeto - relacional, incluyendo, JDO, Hibernate, TopLink e iBatis. Usando el paquete ORM puedes usar esos mapeadores en conjunto con otras características que Spring ofrece, como la administración de transacciones mencionada con anterioridad.

- El paquete AOP provee una implementación de programación orientada a aspectos compatible con AOP Alliance, permitiéndote definir pointcuts e interceptores de métodos para desacoplar el código de una manera limpia implementando funcionalidad que por lógica y claridad debería estar separada. Usando metadatos a nivel de código fuente se pueden incorporar diversos tipos de información y comportamiento al código, un poco similar a los atributos de .NET

- El paquete Web provee características básicas de integración orientadas a la Web, como funcionalidad multipartes (para realizar la carga de archivos), inicialización de contextos mediante servlet listeners y un contexto de aplicación orientado a Web. Cuando se usa Spring junto con WebWork o Struts, este es el paquete que te permite una integración sencilla.

- El paquete Web MVC provee de una implementación Modelo - Vista - Controlador para las aplicaciones Web. La implementación de Spring MVC permite una separación entre código de modelo de dominio y las formas Web y permite el uso de otras características de Spring Framework como lo es la validación.

3.1 Escenarios de uso

Con los bloques descritos en la sección anterior uno puede usar Spring en una multitud de escenarios, desde applets hasta aplicaciones empresariales complejas usando la funcionalidad de Spring para el manejo transaccional y el framework Web.

3.1.1 Aplicación Web Típica

Una aplicación Web típica usa gran parte de las características de Spring. Por medio de los TransactionProxyFactoryBeans la aplicación Web se vuelve totalmente transaccional, tal y como pudiera ser si se usaran las transacciones administradas por el contenedor como lo manejan los Enterprise JavaBeans. Toda la lógica de negocio puede ser implementada usando POJOs, administrados por el contenedor de Inyección de Dependencias de Spring. Servicios adicionales como el envío de mails y la validación, independientes a la capa Web te permiten escoger donde y cuando ejecutar las reglas de validación. El soporte ORM de Spring está integrado con Hibernate, JDO, TopLink e iBatis. Si usas el HibernateDaoSupport, puedes reutilizar tus mapeos de Hibernate existentes. Los controladores de formas te permiten integrar la capa Web con el modelo de dominio, eliminando la necesidad de ActionForms o cualquier clase que transforme los parámetros HTTP en valores para tu modelo de dominio.

3.1.2 Capa intermedia de Spring con un framework WEB alternativo

A veces las circunstancias actuales no te permiten cambiarte completamente a un framework diferente. Spring NO TE OBLIGA a utilizar todo lo que provee, es decir, no se trata de una solución todo o nada. Existen frontends como Webwork, Struts, Tapestry o JSF que permiten integrarse perfectamente a una capa intermedia basada en Spring, permitiéndote usar todas las características transaccionales que Spring ofrece. Lo único que necesitas es ligar tu lógica de negocios usando un ApplicationContext e integrar tu capa Web mediante un WebApplicationContext.

3.1.3 Escenario de acceso a recursos remotos

Cuando necesitas acceder a código existente vía web services, puedes utilizar las clases Hessian-, Rmi- o JaxRpcProxyFactory. El habilitar acceso remoto a aplicaciones existentes de pronto no es tan difícil.

3.1.4 EJBs - Envolviendo POJOs existentes

Spring provee una capa de acceso y una capa de abstracción para Enterprise JavaBeans, permitiéndote reutilizar POJOs existentes y envolverlos en Stateless Session Beans, para ser utilizados en aplicaciones Web robustas y escalables, que puedan necesitar seguridad declarativa.

3.2 Spring y JavaServer Faces

3.2.1 Contenedores de Inversión de Control y el patrón de Inyección de Dependencias

Supongamos que una clase A para realizar su trabajo necesita unos datos. Esos datos se los pide a una clase B que los lee de un fichero. La forma directa de realizar esto es que la clase A haga un new de la clase B. Esto, sin embargo, hace que la clase A sea menos reutilizable, ya que depende totalmente de la clase B y sólo podrá leer los datos de un fichero en el formato que entiende la clase B.

Este patrón de inyectables nos dice que hagamos una interfaceB con los métodos interesantes de la clase B y hagamos que la clase B implemente esa interfaceB. Por otro lado, la clase A en vez de hacer directamente el new de B, recibe en algún método -el constructor o un método set()- la interfaceB. De esta forma, alguien fuera de la clase A hace el new de B y se lo pasa a A. Haciéndolo así, si algún día cambiamos el formato del fichero o bien deseamos leer los datos de una base de datos, sólo tendremos que hacernos una clase OtraB que implemente la interrfaceB y pasársela a A. La clase A, sin tocar su código, será capaz de leer los datos del nuevo fichero o de la base de datos.

A esto lo llama inversión de control o inyección de dependencia. En vez de ser la clase A la que decide de quien depende, alguien desde fuera le "inyecta" la dependencia a través de un método set().

La inyección de dependencias radica en resolver las dependencias de cada clase (atributos) generando los objetos cuando se arranca la aplicación y luego inyectarlos en los demás objetos que los necesiten a través de métodos set o bien a través del constructor, pero estos objetos se instancia una vez, se guardan en una factoría y se comparten por todos los usuarios (al menos en el caso de Spring) y nos evitamos tener que andar extendiendo clases o tumbar el servidor de la BD.

Spring soporta inyección de dependencias a través del constructor y a través de métodos set pero se aconseja hacerlo a través de métodos set. Vamos a ver que habría que hacer para utilizar inyección de dependencias con Spring a través de métodos set.

```
public class Clase {
        private Clase2 atributo1;

        public Clase(){}

        public void setAtributo1(Clase2 atributo1){
                this.atributo1 = atributo1;
        }
}

public class Clase2 {
```

```
—IMPLEMENTACIÓN—
}
```

Declaramos nuestros beans en el contexto de la aplicación

```
< bean id="atributo1" class="org.mipaquete.bean.Clase2"><  /bean>

< bean id="atributo" class="org.mipaquete.bean.Clase">
        < property name=" atributo1" ref="atributo1" / >
< /bean>
```

El atributo ref hace referencia al id del bean creado.

Al arrancar la aplicación estos beans serán creados y todas sus dependencias resueltas y en todas las partes de código en las que utilicemos un atributo de la clase Clase2 bastara con que le llamemos atributo1 y añadamos a la clase un método set y ese objeto será inyectado en tiempo de ejecución.

A la inyección de dependencias también se le llama Inverse of Control o simplemente IoC.

Ésta no es la única forma de eliminar la dependencia entre la clase de la aplicación y la implementación. El otro patrón que se puede utilizar para hacer esto es Service Locator.

3.2.2 Integración de Spring con JavaServer Faces

El modo recomendado para integrar Spring con JSF es configurar el DelegatingVariableResolver de Spring en el faces-context.xml. Para ello utilizaremos el patrón de inyección de dependencias de Spring. Los elementos <application> <variable-resolver> de un fichero faces-config.xml permiten a una aplicación basada en Faces registrar clases personalizadas para sustituir la implementación estándar de VariableResolver. El DelegatingVariableResolver de Spring primero delega al resolver original de la implementación subyacente de JSF, para luego delegar al WebApplicationContext raíz de Spring.Esto permite configurar los Spring Beans como propiedades gestionadas por los Managed Beans de JSF.

Por ejemplo, dada la siguiente configuración del faces-config:

```
<faces-config>
    <application>
        <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-
        resolver>
    </application>

  <managed-bean>
        <managed-bean-name>productController</managed-bean-name>
        <managed-bean-class>compos.management.web.ProductController</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
                <property-name>productManager</property-name>
                <value>#{productManager}</value>
        </managed-property>
  </managed-bean>
</faces-config>
```

El bean productManager implementa la lógica de negocio y es donde haríamos uso de los DAO’s. Este Bean lo tenemos declarado en Spring y lo inyectamos a un Backing Bean, productController, declarado en JSF.

4. Persistencia. Spring + TopLink

Toda aplicación J2EE necesita acceder a una (o más) bases de datos relacionales, por eso no es pretencioso decir que una de las decisiones más importantes al seleccionar la arquitectura de una aplicación J2EE es cómo accederá la aplicación a los datos persistentes: La estrategia de persistencia no sólo puede determinar el rendimiento de la aplicación, sino que también influirá enormemente en la cantidad de esfuerzo requerido para desarrollar y mantener la aplicación; y a menos que tome las decisiones de diseño correctas desde el principio, podría ser difícil revisar esta parte del diseño después de haber terminado la aplicación.

Vamos a ver como dos marcos de trabajos diferentes (pero complementarios) pueden trabajar de forma conjunta o independiente para ayudarnos a superar la complejidad del acceso a datos persistentes en aplicaciones J2EE, más específicamente:

- Oracle TopLink, un poderoso marco de trabajo de persistencia objeto-relacional que proporciona un mecanismo productivo y altamente flexible para acceder a datos relacionales.

- Spring, un marco de tabajo de aplicaciones J2EE, publicado bajo la Apache Software License. Spring proporciona servicios para todas las capas arquitecturales, pero es particularmente potente en cuanto a la persistencia proporcionando una aproximación consistente para acceder a datos. La capa de servicios de persistencia de Spring se acompaña con una implementación del patrón de diseño J2EE Data Access Objects (DAO).

Utilizar juntos TopLink y la capa DAO de Spring puede ofrece una aproximación muy productiva y de alto rendimiento para persistir Plain Old Java Objects (POJOs), que son objetos normalmes Java que no son JavaBeans, ni Beans de entidad ni de sesión, en bases de datos relacionales.

Esta aproximación es una alternativa a la tradicional solución de persistencia de objetos J2EE; los Beans de Entidad de EJB 2.x; que han caído en desuso por muchas razones, entre las que se incluye un modelo de programación embarullado, una baja productividad de desarrollo, y un mal testeo. Además, el lenguaje de consultas de EJB (EJBQL) en su forma sin adornos (sin extensiones propietarias) está muy limitado. Sin embargo, lo más importante es que los beans de entidad presentan muchas restricciones al diseño OO evitando el uso de herencia real, por lo que no son los más ideales para modelos de persistencia sofisticados. (Por otro lado, los POJOs son especialmente útiles para crear un modelo de objetos para problemas de dominio específicos).

La aproximación a la persistencia presentada permitirá persistir POJOs sin restricciones en el diseño OO, que según nuestro punto de vista es una mejor estrategia que los EJBs.

Aunque nos enfoquemos en algunas extensiones de TopLink y Spring, las tecnologías que generalmente representan; mapeo objeto-relacional, implementación de capas de persistencia como abstracciones DAO, separación de la configuración mediante la Inversión de Control (IoC); ofrecen lecciones de arquitectura para los desarrolladores J2EE. Dichas tecnologías son capaces de persistir modelos de dominio orientados a objetos que capturan importantes conceptos de negocio. Normalmente se dice que ofrecen persistencia transparente, la habilidad que tiene un producto de mapeo objeto-relacional de manipular directamente datos almacenados en una base de datos relacional (utilizando un lenguaje de programación de objetos).

Veremos algunas de esas capacidades y ofreceremos guías generales, las mejores prácticas y estrategias básicas para persistencia J2EE para utilizar sin importar las herramientas específicas o los marcos de trabajo que utilice.

Empecemos con una rápida introducción de algunos de los retos del desarrollo de aplicaciones que abarcan los mundos de Java y de las bases de datos relacionales.

4.1 La necesidad de automatizar el acceso a datos desde Java

El API JDBC define una forma estándar para que el código Java se comunique con bases de datos relacionales, pero es de un nivel tan bajo que su uso no es productivo para los desarrolladores de aplicaciones. Se puede crear código JDBC de forma más productiva y menos propensa a errores utilizando una capa de abstracción sobre JDBC (como los paquetes JDBC de Spring o un buen marco de trabajo

JDBC propietario, por ejemplo).

Sin embargo, el desarrollador aún debe realizar mucho trabajo sucio de bajo nivel. Por ejemplo, sacar los datos de la base de datos requiere seleccionar valores en PreparedStatements de JDBC y extraerlos requiere el uso de ResultSets.

Las herramientas de abstracción (como iBATIS SQL Maps) que funcionan a un nivel un poco más alto que los marcos de trabajo JDBC puede automatizar la mayor parte del mapeo JDBC-a-SQL. Sin embargo, los desarrolladores siguen siendo los responsables de escribir el SQL, detectar cuando los objetos persistidos está "sucios" (no están sincronizados con la base de datos), y escribir el extenso código Java para almacenar objetos.

Algunas veces trabajar a ese nivel tan bajo es apropiado. Algunas aplicaciones no requieren más automatización que la proporcionada por dichas aproximaciones. Pero la mayoría si lo necesitan. Lo que se necesita es el nivel de automatización proporcionado por las sofisticadas herramientas de Mapeo Objeto-Relacional (ORM).

4.2 Introducción al mapeo O-R

ORM ayuda a reducir la llamada diferencia de impedancia Objeto-Relacional; el abismo entre el modelo orientado a objetos de una aplicación Java bien diseñada (basada en el modelado de cosas y conceptos del mundo real) y el modelo relacional de un esquema de base de datos (basado en aproximaciones matemáticas para almacenar datos). Este abismo es sorprendentemente ancho. Una aproximación ingenua; el simple mapeo de una clase a una tabla; falla en muchos aspectos. Por ejemplo:

- Las relaciones entre objetos Java se mapean como uniones en la base de datos, y son costosas de seguir a menos que el mapeo esté optimizado.
- Los objetos normalmente participan en árboles de herencia, un concepto externo al modelo relacional.
- El modelo relacional estándar no proporciona los tipos extensibles de un lenguaje como Java.
- Si las relaciones se escriben completamente en SQL, sus relaciones con objetos Java podrían no ser tan claras.

El ORM lo realiza un marco de trabajo de persistencia, que sabe como consultar la base de datos para recuperar objetos Java, y cómo persistir dichos objetos en su representación en las tablas y columnas de la base de datos. Los mapeos se definen en meta datos, típicamente ficheros XML. Algunas herramientas populares proporcionan herramientas gráficas para hacer más sencilla la construcción de mapeos.

Con ORM, la automatización se extiende no sólo al mapeo de objetos a filas y columnas de la base de datos, sino también a la detección de objetos sucios y la escritura en la base de datos con el menor número de actualizaciones SQL; requerimientos que son apropiados para la mayoría de las aplicaciones y se pueden cumplir más fácilmente utilizando marcos de trabajo.

4.3 Beneficios principales de ORM

ORM tiene una larga historia, y no es específico de Java. Sin embargo, se ha vuelto particularmente popular entre los desarrolladores Java. ORM tiene tantos propósitos que juntos pueden añadir una enorme ganancia de productividad. Específicamente:

1. ORM elimina el requerimiento de escribir SQL para cargar y persistir el estado de los objetos. Aunque aún se tiene que escribir consultas, una buena herramienta ORM debería hacerlo más sencillo. Y se verá liberado de la codificación asociada con JDBC.
2. ORM le permite crear un modelo de dominio apropiado sin muchos de los problemas asociados con el modelo relacional. Puede pensar en términos de objetos, en vez de en filas y columnas.
3. ORM puede realizar detección automática de cambios, eliminando una tarea propensa a errores del ciclo de vida de los desarrolladores.
4. ORM puede mejorar la portabilidad entre bases de datos reduciendo la dependencia del SQL específico de la base de datos, que se puede abstraer detrás de la herramienta ORM.

El efecto de ORM sobre el rendimiento de una aplicación dependerá del escenario. Para operaciones de actualización basadas en set, el rendimiento de ORM será menor que el JDBC directo, pero en muchos casos ORM puede mejorar el rendimiento en comparación a la codificación manual de código JDBC, permitiendo el ajuste fino de estrategias como las relaciones que serían difíciles de codifica a mano.

Algunas veces, como hace TopLink, ORM va agarrado de la mano de sofisticados mecanismos de caché que pueden mejorar el rendimiento. La mejora que obtendrá del caché dependerá de la naturaleza de su aplicación.

4.4 Cuando y como elegir ORM

ORM tiene sus escépticos. Especialmente en el espacio .NET ORM ha sido mirado con sospecha, y recientemente este escepticismo también se ha expresado en voz baja en el mundo Java.

En vez de teorizar, dibujemos nuestra propia experiencia práctica, a lo largo de numerosos proyectos J2EE. Usado adecuadamente, ORM puede reducir el esfuerzo de desarrollo y mejorar el mantenimiento. No es algo inhabitual ver un ahorro del 30 ó 40% en la cantidad de código Java necesario al adoptar una solución ORM, en vez de JDBC. Esto es una gran ganancia, porque todo el esfuerzo ahorrado se puede invertir en implementar características de negocio. Es simplemente irresponsable el ignorar el beneficio potencial disponible a través del uso de ORM en las aplicaciones correctas, en el entorno adecuado.

Las experiencias negativas con herramientas ORM suelen ser el resultado de intentar usarlo cuando no es apropiado. ORM produce buenos resultados siempre que no lo fuerce donde no cabe. Aquí tenemos algunas guías básicas:

- Entienda su base de datos objetivo:
  No piense que usar ORM le permite ignorar el SQL y el modelo de búsquedas de su base datos. El mapeo Objeto-Relacional es una herramienta para hacer lo que usted quiera hacer más fácil; no le libera de la necesidad de entender qué es lo que usted quiere hacer. (No apreciar los matices de la base de datos y su SQL probablemente es la principal causa de problemas de resultados con ORM).

- No tenga miedo de usar SQL si es necesario:
  Funciona bien en muchos escenarios. Un buen producto ORM como TopLink le permitirá lanzar consultas SQL. Pero algunas veces necesitará realizar actualizaciones basadas en set o invocar a procedimientos almacenados.

- Haga su investigación antes de comprometerse con un producto de ORM:
  No todos los productos ORM son iguales. Asegúrese de evaluar dos o tres alternativas desarrollando un perfil de conceptos que refleje sus necesidades. Este ejercicio le ayudará a asegurar que su utilización de ORM es apropiada en términos de rendimiento. Al igual que en el desarrollo empresarial en general, es vital mitigar cualquier riesgo de rendimiento en los inicios del proyecto. Además, las funciones de mapeo de su herramienta ORM no deberían suponer mucha sobrecarga.

- Sepa cuándo utilizar ORM:
  ORM funciona particularmente bien para aplicaciones OLTP que actualizan entidades individualmente y realizan operaciones basadas en set de forma poco frecuente, como las aplicaciones que actualizan los registros de clientes y sus pedidos asociados de forma individual.

- Y, cuando no utilizarlo:
  ORM no es siempre la mejor opción. No es apropiado para:

  - Aplicaciones que realizan frecuentes actualizaciones de numerosos registros.

  - Aplicaciones OLAP, porque normalmente las bases de datos ofrecen mencanismos de programación ativos. (En general, J2EE podría no ser la mejor elección para OLAP).
    Las bases de datos o entornos operativos repletos de código SQL y llamadas a procedimientos almacenados como único mecanismo para recuperar y actualizar datos. En dichos casos, una aproximación basada en JDBC podría ser la mejor opción; iBATIS

SQL Maps también brilla en estos escenarios. (Observe sin embargo, que un buen producto ORM, como TopLink, puede funcionar con la mayoría de los esquemas existentes y con procedimientos almacenados, por eso aún podría considerar la utilización de ORM en dichos escenarios).

- Aplicaciones que se sirven mejor con aproximaciones directas basadas en SQL, en dichas aplicaciones la lógica de negocio está codificada dentro de la base de datos, o forzadas por restricciones de integridad, y que le dan a los usuarios una "ventana de datos", permitiendo que el usuario los edite. En aplicaciones como éstas, ORM tiene menos que ofrecer porque los objetos en general, tienen menos que ofrecer; más allá de una ilusión de orientación a objetos, se puede ganar muy poco modelando las tablas de la base de datos como objetos de dominio.

Hoy en día hay varias buenas herramientas ORM a su disposición, incluyendo tanto comerciales (TopLink, el producto ORM para Java más antiguo, y varias implementaciones de JDO) como de código abierto (como Hibernate o implementaciones de JDO). Simplemente elija una de ellas, no es necesario que se enrrolle. (En el pasado, era muy común desarrollar marcos de trabajo ORM personales, pero hoy en día, no tiene sentido: es un enorme gasto cuyos resultados nunca son tan buenos como las soluciones genéricas comerciales o de código abierto). Echemos un vistazo más de cerca a la utilización de TopLink y veamos un ejemplo de utilización de un marco de trabajo ORM, y los beneficios de productividad que proporciona.

4.5 El Marco de Trabajo Spring, ORM y la Persistencia

El marco de trabajo Spring es uno de los marcos de trabajo de código abierto líderes en aplicaciones J2EE que proporciona servicios para todas las capas arquitecturales.

Al contrario que los marcos de trabajo de una sola capa (como Struts), Spring es una aplicación marco de trabajo que facilita una aproximación clara y por capas a la arquitectura de una aplicación. Spring no impone esta arquitectura, pero hace más sencillo seguir las buenas prácticas. La arquitectura típica de una aplicación Spring consta de la capa de presentación, la capa de servicios, el interface DAO y la capa de implementación, y una capa de objetos de dominio persistentes.

En términos de persistencia de objetos, es la capa de interfaces DAO la que dirige el acceso a los datos, implementa las consultas y otras operaciones de persistencia. Echemos un vistazo en profundidad a esa capa, primero en términos genéricos y luego veremos en el código que la capa DAO de Spring se puede utilizar en conjunción con una herramienta ORM (como TopLink).

4.6 Beneficios de la capa de abstracción DAO

Un interface DAO bien diseñado puede ocultar totalmente la tecnología de persistencia específica a los objetos de la capa de servicios. Por ejemplo, puede implementar el mismo interface DAO con TopLink o Hibernate, sin impactar en el código llamante.

Esto es bueno, porque reduce la dependencia de una herramienta ORM particular, y proporciona un grado de aprovechamiento futuro. Los conceptos de acceso datos no cambiarán en los próximos años, pero las tecnologías que implementan el DAO si lo harán. Incluso aún más importante, esto significa que usted puede probar los objetos de la capa de servicios sin acceder a una base de datos, simulando DAOs.

Como mecanismo general (junto con las líneas de Eric Evans en la noción de Repositorios (puede encontrar más información en Domain-Driven Design) los interfaces DAO normalmente ofrecen estos tipos de métodos:

- Buscadores, para recuperar ejemplares de objetos persistentes, estos métodos se implementan utilizando el interface de consultas de la herramienta ORM.
- Métodos para grabar, para persistir nuevos objetos.
- Métodos para borrar, para borrar objetos persistidos en la base de datos.
- Funciones agregadoras, para contar objetos u otras funciones agregadoras. Consultar la base de datos mediante la agregación normalmente es mucho más rápido que interactuar con una gran cantidad de entidades persistentes.

4.7 Una Mirada en Profundidad a la Capa DAO de Spring

En una arquitectura Spring típica, los DAOs son objetos del patrón Strategy, que aíslan los objetos de la capa de servicio de la tecnología específica para obtener y grabar los objetos persistentes.

Spring soporta varias herramientas ORM y marcos de trabajo de persistencia, incluyendo (aunque no limitado a) TopLink, Hibernate, JDO (Java Data Objects), iBATIS SQL Maps, y Apache OJB (ObjectRelational Bridge).

Spring proporciona una aproximación consistente para trabajar con todas estas herramientas, pero no intenta abstraer complemente la herramienta ORM, porque hacer eso sería contraproductivo. Así, por ejemplo, las implementaciones de interfaces DAO usará el API de consulta nativo de la herramienta ORM: Spring no intenta crear una abstracción de consultas, que sería menos poderoso que los productos líderes en ORM.

4.8 Cómo puede influir Spring en cualquier ORM

Para que los interfaces DAO sean independientes de la tecnología de persistencia, necesitamos resolver algunos problemas importantes. Spring proporciona aquí una ayuda interesante, especialmente en las áreas de manejo de excepciones: adquisición y liberación de recursos: y control de transaciones.

4.8.1 El Manejo de Excepciones de Spring

Dos características clave del manejo de excepciones de Spring le permiten permanecer independiente de la tecnología de persistencia. Primero, utiliza un árbol basado en clases de excepciones genéricas de acceso a datos (la Figura 4 muestra las clases más importantes del árbol) que pueden ocultar cualquier mecanismo de excepciones de la ORM. Este árbol de excepciones se puede extender fácilmente. Así, por ejemplo, puede escribir código de una capa de servicio para manejar DataIntegrityViolationException (violación de una restricción de una clave primaria), sin preocuparse de la heramienta de acceso a datos utilizada, que podría ser JDBC, TopLink, JDO, Hibernate u otro marco de trabajo soportado. Este árbol es extensible incluso sin modificar el código de Spring, si desea añadir mapeos para condiciones de error únicas de su base de datos o su herramienta de persistencia.

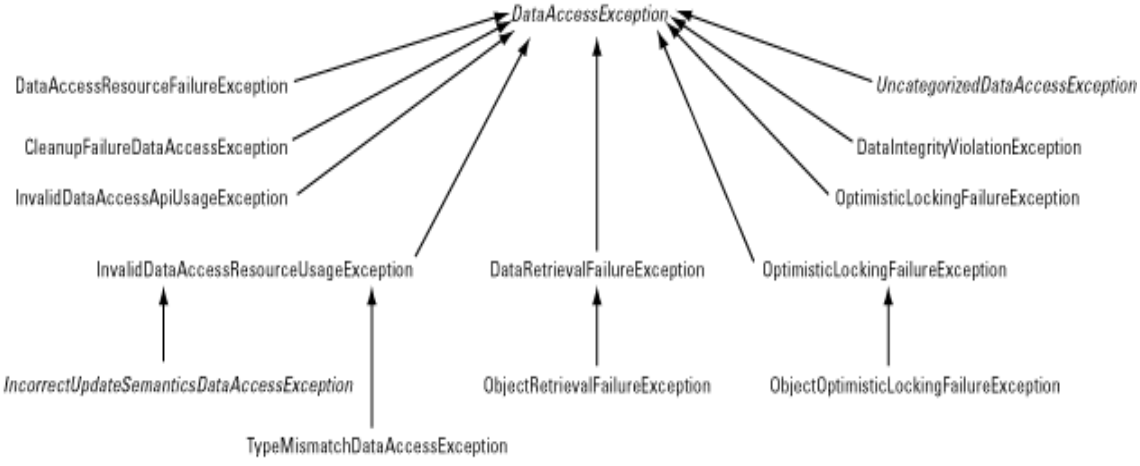

Figura7. Diagrama UML del árbol de clases de excepciones de Spring.

Segundo, la aproximación al manejo de excepciones de Spring usa excepciones no chequeadas. La utilización de este tipo de excepciones funciona particularmente bien en conjunción con un árbol de excepciones, porque sólo es necesario capturar una subclase particular que podría ser recuperable. Capturar subclases de excepciones chequeadas es menos útil, porque aún se necesita capturar la clase base. (Observe que JDBC siempre ha sido un API raro, en términos de su utilización de excepciones chequeadas: TopLink y JDO, por ejemplo, usan excepciones no chequeadas. En su versión 3, Hibernate cambiará de excepciones chequeadas a no chequeadas.

4.8.2 Adquisición y Liberación de Recursos

Uno de los mayores retos de trabajar con cualquier tecnología de acceso a datos es que se debe adquirir y liberar correctamente recursos como las conexiones JDBC o las sesiones ORM incluso si ocurren errores.

Una sesión ORM es una unidad de trabajo transacional manejada por la herramienta ORM. Una sesión contiene un conjunto de objetos asociados con la transación, algunos de los cuales podrían haber sido modificados durante el curso de la transación, y a partir de los cuales la herramienta ORM genera SQL para actualizar la base de datos. Por lo tanto, el control de sesiones es primordial para la integridad de los datos).

Spring resuelve el problema de la adquisición y liberación de recursos utilizando una aproximación de retrollamadas. El marco de trabajo es el responsable de adquirir y liberar los recursos, mientras que el desarrollador de la aplicación immplementa código en un método de retrollamada que realiza la operación de persistencia necesaria con el recurso deseado. Así los desarrolladores de la aplicación se enfocan en las necesidades únicas de su aplicación, en vez de escribir código de fontanero.

Las clases de ayuda que proporcionan esta funcionalidad son plantillas condicionales, y Spring proporciona una aproximación consistente entre muchos APIs, donde cada adquisición y liberación de recurso es un problema, como JDBC, JNDI y JMS.

4.8.3 Control de Transaciones y unión de Threads

La estrategia de adquisición y liberación de recursos de Spring está integrada con su soporte de control de transaciones. Es importante obtener el mismo recurso para cada operación de la misma transación. Mientras JTA y JCA podrían proporcionar soporte de este tipo en un entorno totalmente J2EE (dependiendo de la herramienta de persistencia), Spring proporciona soporte para ello en cualquier entorno, incluyendo entornos J2SE. La configuración que necesita Spring es mucho más simple que la que necesita JCA.

Spring realiza esto uniendo recursos como una conexión JDBC o una sesión de TopLink o de Hibernate al thread actual. Muchos desarrolladores de aplicaciones habrán implementado dicho soporte en sus propias aplicaciones. Sin embargo, la solución de Spring no sólo tiene el mérito de ser mucho más sofisticada que las soluciones caseras (con soporte para suspensión de transaciones y muchos otros valores añadidos complejos de implementar), sino que elimina la necesidad de escribir y mantener dicho código personalizado. De nuevo para los problemas genéricos es mejor una solución genérica.

4.9 Clases y métodos de conveniencia

Spring proporciona clases de conveniencia para simplificar el trabajo con cada herramienta de persistencia soportada.

Por ejemplo, las clases de soporte para JDBC, TopLink, e Hibernate proporcionan métodos de consulta y grabado que reducen dichas operaciones a una única línea de código, como la siguiente (una consulta Hibernate utilizando la clase de conveniencia HibernateTemplate de Spring):

```
List customers = hibernateTemplate.find("from Customer c where c.age > ?", age);
```

O está (una consulta TopLink usando la clase de conveniencia TopLinkTemplate de Spring):

```
List customers =
toplinkTemplate.findByNamedQuery(Customer.class,"findAllCustomersOlderThan",args);
```

Dichos métodos de conveniencia puede reducir significativamente la cantidad de código para acceder a datos requerida por una aplicación que utiliza el API nativo de la herramienta de persistencia y trata con la adquisición y liberación de recursos.

Spring y TopLink son una pareja particularmente buena. TopLink es un producto complejo, por eso la arquitectura consistente y la receta de "plantillas" que trae Spring pueden proporcionar a los usuarios unas guías muy útiles.

Cuando utilice Spring con TopLink, debe asegurarse de:

- Utilizar consultas con nombre. Esta ha sido una buena práctica ampliamente recomendada por los consultores de TopLink.
- Use la clase TopLinkTemplate para obtener y usar sesiones TopLink, en vez de escribir su propio código para manejar sesiones TopLink. Esto le ahorrará el tiempo y el esfuerzo y asegurará un manejo de errores consistente en su aplicación.
- Sigua la práctica arquitectural normal de Spring, y oculte el uso de TopLink detrás de los interfaces DAO.

5. Fase de Elaboración

Uno de los aspectos más interesantes de Java es el ritmo al que evolucionan tanto sus tecnologías como las arquitecturas diseñadas para soportarlas. Cada poco tiempo aparecen nuevas librerías, herramientas, frameworks, patrones, etc, y se hace muy difícil elegir entre ellos y, sobre todo, combinarlos de una manera eficiente que aproveche toda su flexibilidad y potencia.

En esta sección, explica la satisfactoria experiencia del diseño de onLineStore, con la adopción de varias de las últimas tecnologías más útiles disponibles en Java, cubriendo todos los aspectos, desde los puramente técnicos hasta los metodológicos –así como los problemas que han surgido- para lograr este objetivo.

Este proceso ha sido una evolución hacia un sistema totalmente modular, conformado por frameworks especializados líderes en su área (JavaServer Faces, Spring, etc.), con una arquitectura que facilita un desarrollo conducido por el modelo y con una separación limpia en capas -en donde pueden participar desarrolladores especializados-.

5.1 Diseño de la arquitectura de alto nivel

La siguiente fase del diseño de una aplicación Web es el diseño de la arquitectura de alto nivel. Esto implica subdividir la aplicación en componentes funcionales y dividir estos componentes en capas. El diseño de la arquitectura de alto nivel es neutral a las tecnologías utilizadas.

5.1.1 Arquitectura multicapas

Una arquitectura multicapa divide todo el sistema en distintas unidades funcionales: cliente, presentación, lógica-de-negocio, integración, y persistencia. Esto asegura una división clara de responsabilidades y hace que el sistema sea más mantenible y extensible. Los sistemas con tres o más capas se han probado como más escalables y flexibles que un sistema cliente-servidor, en el que no existe la capa central de lógica-de-negocios.

La capa del cliente es donde se consumen y presentan los modelos de datos. Para una aplicación Web, la capa cliente normalmente es un navegador Web. Los clientes pequeños basados en navegadores no contienen lógica de presentación; se trata en la capa de presentación.

La capa de presentación expone los servicios de la capa de lógica-de-negocio a los usuarios. Sabe cómo procesar una petición de cliente, cómo interactuar con la capa de lógica de negocio, y cómo seleccionar la siguiente vista a mostrar.

La capa de la lógica-de-negocio contiene los objetos y servicios de negocio de la aplicación. Recibe peticiones de la capa de presentación, procesa la lógica de negocio basada en las peticiones, y media en los accesos a los recursos de la capa de persistencia. Los componentes de la capa de lógica-de-negocio se benefician de la mayoría de lo servicios a nivel de sistema como el control de seguridad, de transaciones y de recursos.

La capa de integración es el puente entre la capa de lógica-de-negocio y la capa de persistencia. Encapsula la lógica para interactuar con la capa de persistencia. Algunas veces a la combinación de las capas de integración y de lógica-de-negocio se le conoce como capa central.

Los datos de la aplicación persisten en la capa de persistencia. Contiene bases de datos relacionales, bases de datos orientadas a objetos, y sistemas antiguos.

5.2 Justificación de la arquitectura elegida

5.2.1 Elección de capas y componentes

El objetivo principal de la arquitectura es separar, de la forma más limpia posible, las distintas capas de desarrollo, con especial atención a permitir un modelo de domino limpio y a la facilidad de mantenimiento y evolución de las aplicaciones. Otros elementos importantes son la facilidad del despliegue y el empleo de las mejores tecnologías disponibles en la actualidad –en contraposición al continuismo con opciones que se consideran anticuadas a día de hoy-.

Se desea una arquitectura que permita trabajar en capas. Para lograr esto hemos elegido el patrón MVC (Modelo-Vista-Controlador) que permite una separación limpia entre las distintas capas de una aplicación.

Para la capa de presentación (la vista) se buscaba un framework que nos proporcionase una mayor facilidad en la elaboración de pantallas, mapeo entre los formularios y sus clases en el servidor, la validación, conversión, gestión de errores, de una forma sencilla y –sobre todo- fácil de mantener. Para esta capa se ha elegido JavaServer Faces.

En la capa de negocio y persistencia, se decidió desde el primer momento no emplear EJB’s por su elevado coste de desarrollo y mantenimiento así como su falta de flexibilidad y coste en tiempo para los cambios.

Se optó por una solución basada en servicios (no necesariamente servicios Web, aunque permitiendo su integración de forma limpia) que trabajaban contra un modelo de dominio limpio. La persistencia de las clases se sustenta en DAO’s (Objetos de Acceso a Datos), manteniendo aislada la capa de persistencia de la capa de negocio. Tanto los servicios como los DAO’s así como el propio modelo son realmente POJOs (clases simples de Java), con la simplicidad que conllevan y sin dependencias reales con ningún framework concreto. Para realizar esta integración se ha elegido Spring.

Para la capa de persistencia se pensó en utilizar alguna herramienta ya existente, que permitiese realizar el mapeo objeto-relacional de una forma cómoda pero potente, sin tener que implementarlo directamente mediante JDBC. Esto último conllevaría, por ejemplo, un esfuerzo importante en un caso de cambio de

base de datos, en la gestión de la caché, la utilización de carga perezosa, etc. La herramienta elegida finalmente fue TopLink. A continuación, presentamos el diagrama de componentes de la aplicación:

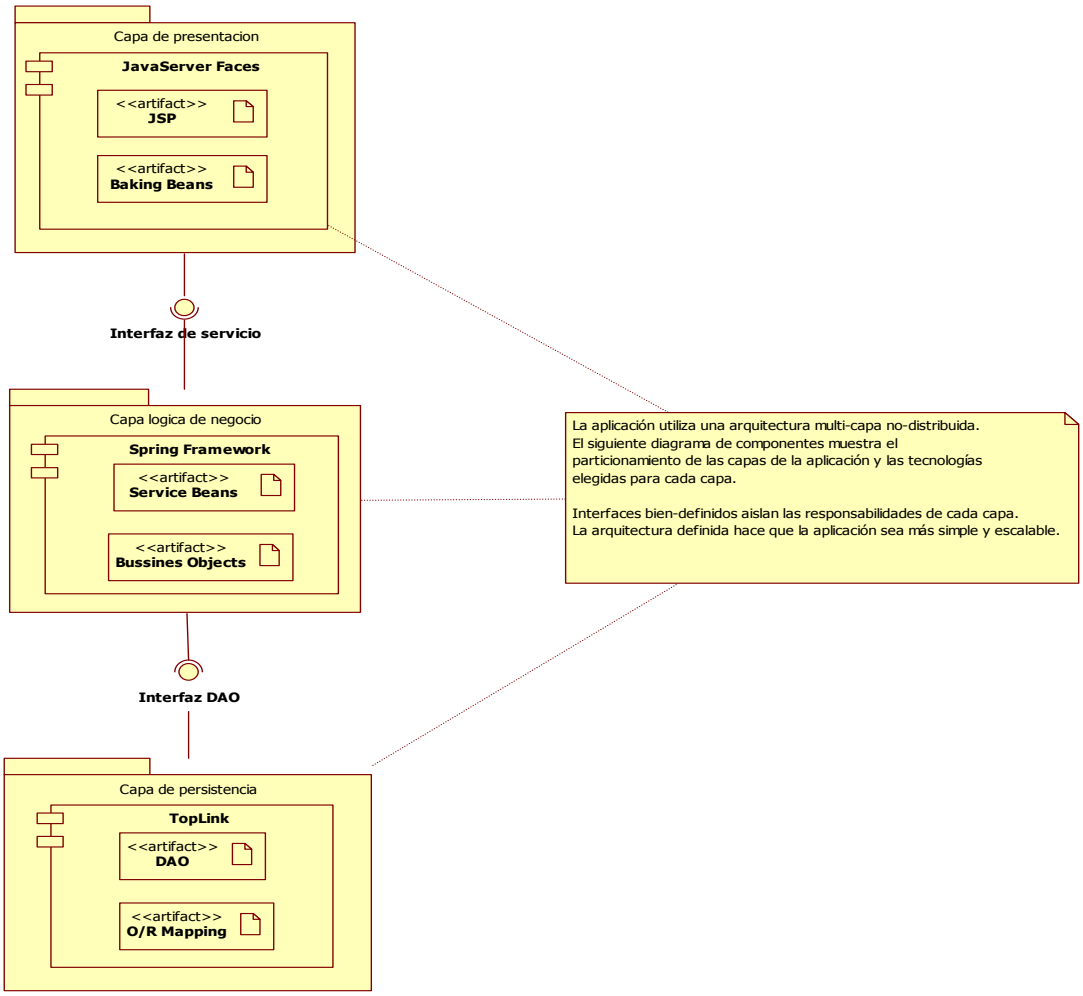

Figura 8. Diagrama de componentes de la arquitectura diseñada

5.2.2 Beneficios de la arquitectura diseñada

La primera ventaja se deriva de la modularidad del diseño. Cada una de las partes empleadas (JSF para la vista, Spring para la integración y TopLink para la persistencia) es intercambiable de forma sencilla y limpia por otras soluciones disponibles. Por ejemplo, para la vista se emplea JavaServer Faces, pero nada impide emplear también una aplicación de escritorio mediante Swing o SWT sin tener que tocar ni una sola línea de código de las restantes capas. Es más, nada impediría que se pudiese disponer de una aplicación con una parte de la capa de presentación en JSF y otra parte, para otro tipo de usuarios, en Swing, ambas funcionando a la vez y compartiendo todo el resto del código (lógica de negocio, persistencia, integración, seguridad, etc).

De igual forma, si se desean cambiar elementos de la capa de persistencia empleando otro Framework para el mapeo diferente de TopLink –o sencillamente no utilizar ninguno- tan sólo serían necesarios cambios en esa capa.

De la misma manera se podrían sustituir cualquiera de las otras capas. El diseño se ha hecho reduciendo al mínimo posible las dependencias entre ellas, y utilizando interfaces que conectan las diferentes capas.

5.3 Frameworks empleados

5.3.1 JavaServer Faces

JavaServer Faces es el estándar presentado por Sun para la capa de presentación Web. Forma parte de la especificación J2EE 5 -que deberán cumplir todos los servidores de aplicaciones- y se erige como una evolución natural de los frameworks actuales hacia un sistema de componentes.

Es un estándar sencillo que aporta los componentes básicos de las páginas Web además de permitir crear componentes más complejos (menús, pestañas, árboles, etc). Ya hay disponibles diferentes implementaciones de la especificación, tanto comerciales como de código abierto, así como librerías de componentes adicionales que amplían la funcionalidad de esos componentes iniciales.

JSF ha sido acogida por la comunidad como "el framework que hacía falta". Muchos de los proyectos de código abierto y las compañías con más influencia lo han identificado como el framework de presentación Web del futuro. JBoss ha integrado directamente JSF con EJB3 mediante el proyecto Seam (abanderado además por Gavin King, el líder del equipo de desarrollo Hibernate). IBM lo ha incorporado como mecanismo de presentación estándar para su entorno, desarrollando no sólo el soporte completo en su IDE, sino nuevos componentes. Oracle es otra de las compañías que más ha apostado por esta tecnología, ofreciendo la que posiblemente sea la mayor oferta de componentes propios.

Dentro de los "pesos pesados" en Java, nombres que han guiado a la industria en el pasado como Matt Raible, Rick Hightower, David Geary, el mencionado Gavin King y por supuesto el propio creador de Struts y ahora arquitecto de JSF, Craig McClanahan.

5.3.2 Spring

Spring es un conjunto de librerías "a la carta" de entre las que podemos escoger aquellas que faciliten el desarrollo de nuestra aplicación. Entre sus posibilidades más potentes está su contenedor de Inversión de Control (Inversión de Control, también llamado Inyección de Dependencias, es una técnica alternativa a las clásicas búsquedas de recursos vía JNDI. Permite configurar las clases en un archivo XML y definir en él las dependencias. De esta forma la aplicación se vuelve muy modular y a la vez no adquiere dependencias con Spring, la introducción de aspectos, plantillas de utilidades para Hibernate, TopLink, iBatis y JDBC así como la integración con JSF.
Es uno de los proyectos más sorprendentes en el panorama actual en Java en el grado en que ayuda a que los diferentes componentes que forman una aplicación trabajen entre sí, pero no establece apenas dependencias consigo mismo. Esta es la primera característica de este framework. Sería posible retirarlo sin prácticamente cambiar líneas de código. Lo único que sería necesario es, lógicamente, añadir la funcionalidad que provee, ya sea con otro framework similar o mediante nuestro código.

A nivel de soporte de la comunidad, Spring es uno de los proyectos con más actividad, con desarrollos dentro y fuera del propio Framework. Actualmente dispone de soporte comercial a través de Interface21, la empresa creadora, así como otros fabricantes que dan soporte en su área.

5.3.3 TopLink

Toda aplicación J2EE necesita acceder a una (o más) bases de datos relacionales, por eso no es pretencioso decir que una de las decisiones más importantes al seleccionar la arquitectura de una aplicación J2EE es cómo accederá la aplicación a los datos persistentes: La estrategia de persistencia no sólo puede determinar el rendimiento de la aplicación, sino que también influirá enormemente en la cantidad de esfuerzo requerido para desarrollar y mantener la aplicación; y a menos que se tomen las decisiones de diseño correctas desde el principio, podría ser difícil revisar esta parte del diseño después de haber terminado la aplicación.

TopLink probablemente sea el marco de trabajo ORM más maduro. Primero fue desarrollado en Smalltalk

en 1994. Desde 1997, su foco principal ha sido Java. Hoy en día TopLink se puede utilizar tanto fuera como dentro de un servidor de aplicaciones J2EE. Aunque ahora es propiedad y está soportado por Oracle, funciona con cualquiera de las principales bases de datos, no sólo con Oracle. Funciona con todos los servidores de aplicaciones J2EE (o incluso fuera de ellos).

Utilizar juntos TopLink y la capa DAO de Spring puede ofrece una aproximación muy productiva y de alto rendimiento para persistir Plain Old Java Objects (POJOs).

5.4 La Capa de Presentación y JavaServer Faces

La capa de presentación recoge la entrada del usuario, presenta los datos, controla la navegación por las páginas y delega la entrada del usuario a la capa de la lógica-de-negocio. La capa de presentación también puede validar la entrada del usuario y mantener el estado de sesión de la aplicación. En las siguientes secciones, discutiremos las consideraciones de diseño y los patrones de la capa de presentación.

5.4.1 Model-View-Controller (MVC)

MVC es el patrón de diseño arquitectural recomendado para aplicaciones interactivas Java. MVC separa los conceptos de diseño, y por lo tanto decrementa la duplicación de código, la centralización del control y hace que la aplicación sea más extensible. MVC también ayuda a los desarrolladores con diferentes habilidades a enfocarse en sus habilidades principales y a colaborar a través de interfaces claramente definidos. MVC es el patrón de diseño arquitectural para la capa de presentación.

JSF encaja bien en la arquitectura de la capa de presentación basada en MVC. Ofrece una clara separación entre el comportamiento y la presentación. Une los familiares componentes UI con los conceptos de la capa-Web sin limitarnos a una tecnología de script o lenguaje de marcas particular.

Los beans que hay tras JSF son la capa de modelo (en una sección posterior hablaremos más de estos beans). También contienen acciones, que son una extensión de la capa del controlador y delegan las peticiones del usuario a la capa de la lógica-de-negocio. Desde la perspectiva de la arquitectura general de la aplicación, también se puede referir a la capa de lógica-de-negocio como la capa del modelo. Las páginas JSP con etiquetas JSF personalizadas son la capa de la vista. El Servlet Faces proporciona la funcionalidad del controlador.

5.5 La Capa de Lógica-de-Negocio y el Marco de Trabajo Spring

Los objetos y servicios de negocio existen en la capa de lógica-de-negocio. Un objeto de negocio no sólo contiene datos, también la lógica asociada con ese objeto específico. En la aplicación **onLineStore** se han identificado tres objetos de negocio: Producto, Categoría, y Usuario.

Los servicios de negocio interactúan con objetos de negocio y proporcionan una lógica de negocio de más alto nivel. Se definirá una capa de interface de negocio formal, que contenga los interfaces de servicio que el cliente utilizará directamente. POJO, con la ayuda del marco de trabajo Spring, implementará la capa de lógica-de-negocio de la aplicación **onLineStore**. Hay dos servicios de negocio: **CatalogoService** contiene la lógica de negocio relacionada con el manejo del catálogo de productos, y **UserService** contiene la lógica de manejo del usuario.

5.6 La Capa de persistencia y TopLink

TopLink es un marco de trabajo de mapeo O/R que evita la necesidad de utilizar el API JDBC. TopLink soporta la mayoría de los sistemas de bases de datos SQL, proporcionando un puente elegante entre los mundos objeto y relacional. Ofrece facilidades para recuperación y actualización de datos, control de transaciones, repositorios de conexiones a bases de datos y soporte para Java annotations. TopLink es menos invasivo que otros marcos de trabajo de mapeo O/R. Esto nos permite desarrollar objetos persistentes siguiendo el lenguaje común de Java: incluyendo asociación, herencia, polimorfismo, composición y el marco de trabajo Collections de Java. Los objetos de negocio de la aplicación son POJO y no necesitan implementar ningún interface específico de TopLink.

5.6.1 *Data Acces Object (DAO)*

En la aplicación se utiliza el patrón DAO. Este patrón abstrae y encapsula todos los accesos a la fuente de datos. La aplicación tiene dos interfaces DAO: **CatalogoDao** y **UserDao**. Sus clases de implementación, **CatalogoDAOTopLinkImpl** y **UserDAOTopLinkImpl** contienen lógica específica de TopLink para manejar los datos persistentes.

5.7 Proceso

A continuación veremos el proceso a seguir para el diseño de los casos de uso de la aplicación mediante los correspondientes diagramas de clase y actividad. También presentaremos el diseño de la base de datos y el modelo de navegación de la aplicación.

5.7.1 *Casos de uso*

Se utiliza el análisis de los casos de uso para acceder a los requisitos funcionales de la aplicación. En la siguiente figura se puede ver el diagrama de casos de uso de la aplicación:

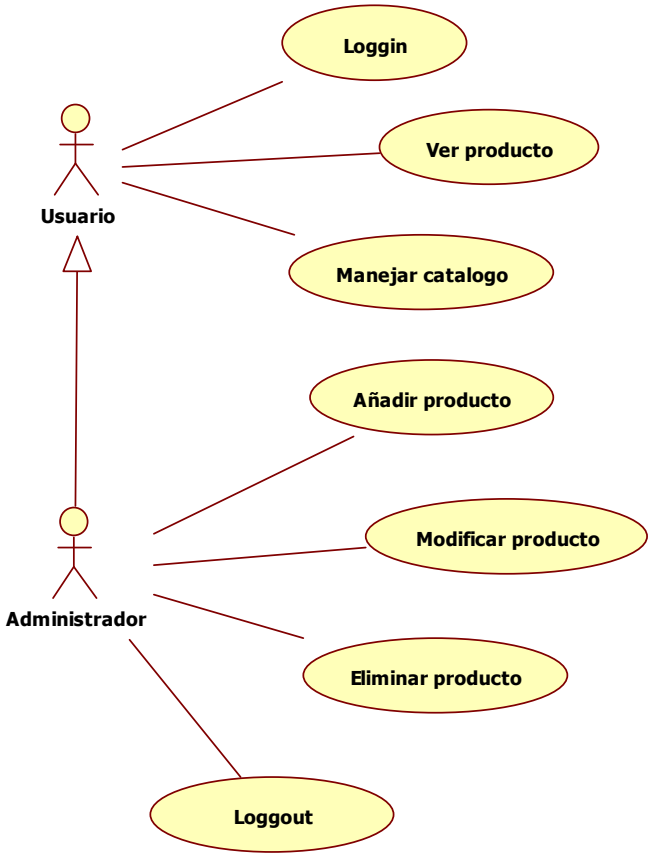

Figura 9. Diagrama de casos de uso de la aplicación OnlineStore

Un diagrama de casos de uso identifica los actores en un sistema y las operaciones que podrían realizar.

En la aplicación se deben implementar siete casos de uso. El actor Usuario puede navegar por el catálogo de productos y ver los detalles de esos productos. Una vez que el Usuario entra en el sistema, se convierte en el actor Administrador, que puede crear nuevos productos, editar productos existentes, y borrar productos obsoletos. Para ver más en detalle los casos de uso modelados en la aplicación, consultar el documento **Casos de Uso del proyecto**.

5.7.2 Modelo de navegación

La siguiente figura muestra todas las páginas de OnLineStore y las transiciones entre ellas:

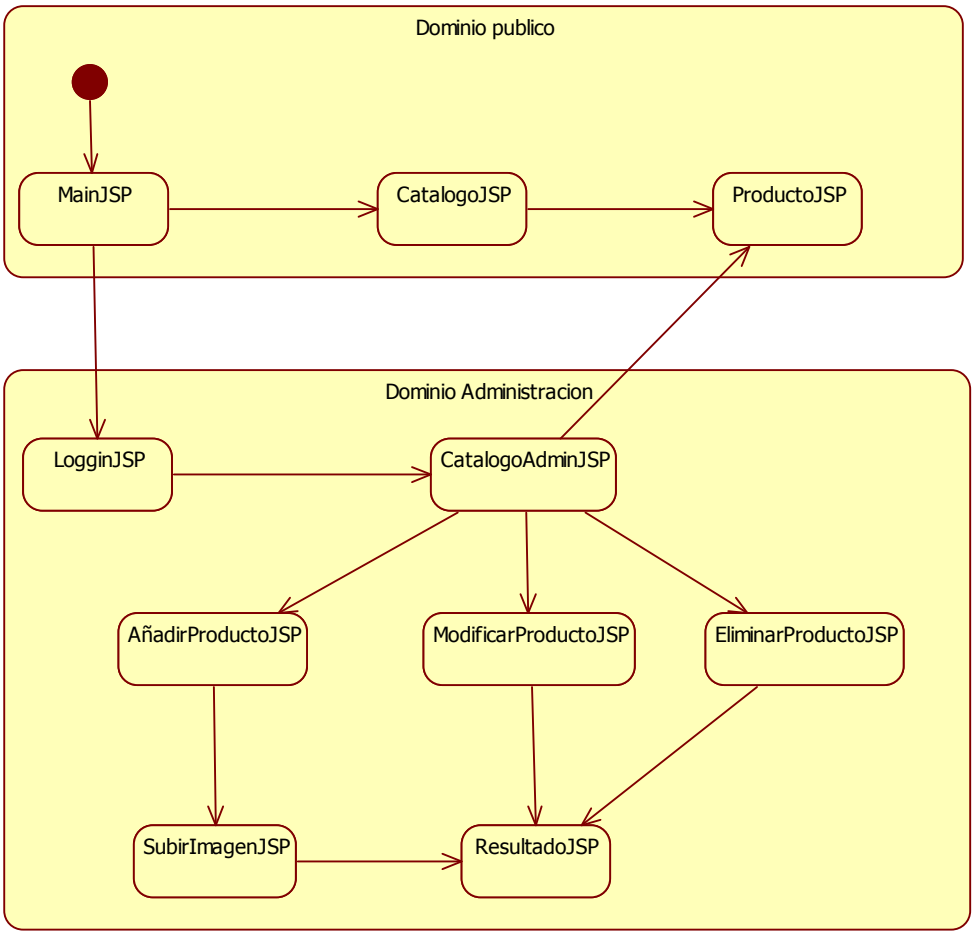

Figura 10. Mapa del modelo de navegación

La aplicación tiene dos grupos de páginas: públicas por Internet y de administración por intranet. La intranet es accesible sólo cuando el usuario se 'logea' en el sistema. CatalogoAdminJSP es un catalogo especial que sólo pueden ver los administradores. Contiene enlaces para crear, editar y borrar productos.

5.7.3 Diseño de los casos de uso

Los propósitos de diseñar un caso de uso son:

1. Identificar las clases del diseño necesarias para realizar el flujo de eventos del caso de uso.
2. Distribuir el comportamiento del caso de uso en objetos del diseño.
3. Refinar los requerimientos en las operaciones de los subsistemas y/o sus interficies.
4. Definir/refinar requerimientos en las operaciones de las clases de diseño.

En este documento mostraremos y comentaremos con detalle el diseño de dos casos de usos. El caso de uso Loggin y el caso de uso añadir producto**. Para ver el modelado y el diseño de los restantes casos de uso, consultar el modelo UML onLineStore.uml.**

5.7.3.1 Identificar las clases del diseño participantes

En este paso se identifican las clases del diseño necesarias para realizar el caso de uso.

En el diagrama de clases de la realización del caso de uso del diseño deben mostrarse las clases del diseño que participan en la realización del caso de uso y las relaciones entre ellas. Para cada clase expresaremos únicamente los atributos y operaciones que se utilizan en la realización. Una clase del diseño puede aparecer en más de una realización del diseño del caso de uso, pero en este diagrama solo deben aparecer los atributos, operaciones y asociaciones que se utilicen en la realización.

El diagrama de clases del diseño participantes se irá enriqueciendo con más clases, operaciones, atributos y asociaciones a medida que realicemos los diagramas de interacción de los objetos

5.7.3.2 Describir Interacciones entre Objetos del Diseño

Para cada realización de caso de uso, debemos ilustrar las interacciones entre sus clases del diseño participantes mediante diagramas de secuencia y diagramas de clases del diseño participantes (ver apartado anterior).

A medida que detallamos los diagramas de secuencia, encontramos nuevos caminos alternativos que el caso de uso puede tomar. Estos caminos pueden ser descritos como notas a los diagramas o bien como nuevos Diagramas de Secuencia.

En este punto presentaremos el diseño de dos de los casos de uso más significativos de la aplicación. Gracias a la arquitectura empleada y al diseño basado en interfaces, todos los casos de uso siguen un mismo patrón de modelado, siendo las clases de diseño participantes y los diagramas de secuencia muy parecidos. Con esto conseguimos un patrón de desarrollo modular y uniforme, basado en componentes e interfaces.

5.7.3.3 Diseño del caso de uso Loggin

A continuación vamos a presentar el diseño del caso de uso loggin. Para ello, hemos utilizado un diagrama de clases con las clases que participan en el caso de uso, y un diagrama de secuencia para ilustrar las interacciones entre las clases participantes.

La siguiente figura ilustra el diagrama de clases participantes del caso de uso:

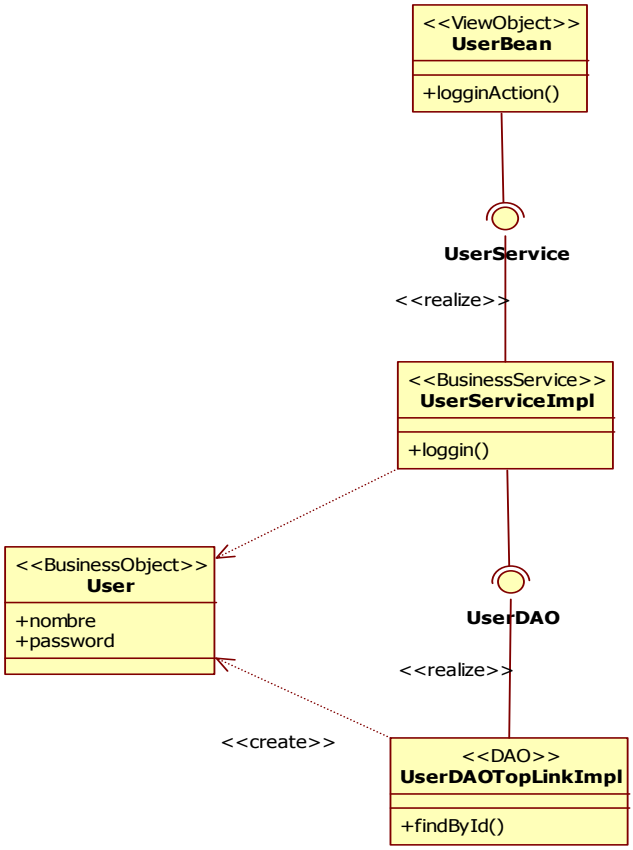

Figura 11. Diagrama de clases para el caso de uso Loggin

Se modelan 4 clases y dos interfaces. Como se puede ver, el diseño de las clases y sus relaciones encaja perfectamente en el modelo de componentes que mostramos en la arquitectura diseñada. La clase UserBean está modelada como un ViewObject. Un ViewObject es un objeto modelo utilizado específicamente en la capa de presentación. Contiene los datos que debe mostrar en la capa de la vista y la lógica para validar la entrada del usuario, manejar los eventos, e interactuar con la capa de lógica-de-negocio. El bean de respaldo es el objeto vista en una aplicación basada en JSF. En este caso contiene la operación logginAction, que se activa cuando el usuario pulsa el botón de loggin en la página correspondiente de la aplicación.

El bean UserBean delega las peticiones del usuario a la capa de la lógica-de-negocio. Se define una capa de interface de negocio formal (UserService), que contiene los interfaces de servicio que el UserBean utilizará directamente. La implementación corre a cargo de la clase UserServideImpl, modelada como un BussinesService. Esta clase contiene la lógica de negocio relacionada con el manejo del usuario y, mediante la operación loggin, es la encargada de acceder a la capa de persistencia.

Mediante la interfaz DAO UserDao y su clase de implementación, UserDAOToplinkImpl, manejamos los datos persistentes del objeto de negocio User. Definimos un método, findByID, que es el encargado de recuperar de la base de datos el objeto User, con el que la clase de servicio trabajará.

A continuación, mostramos el diagrama de secuencia del caso de uso Loggin:

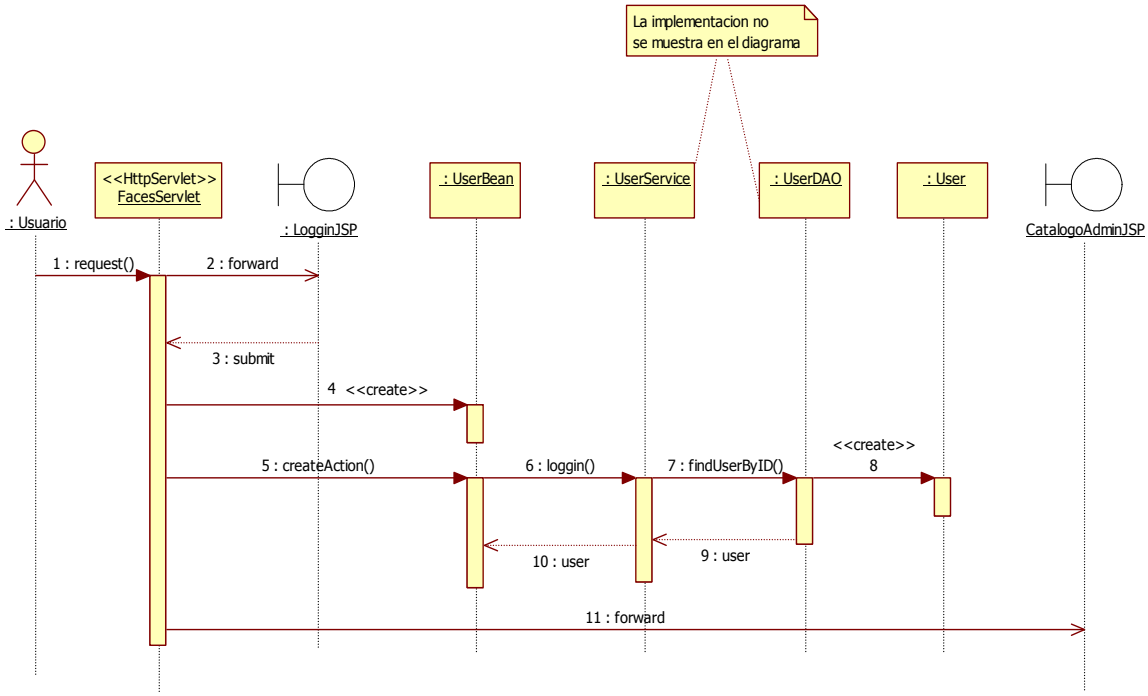

Figura 12. Diagrama de secuencia del UC Loggin

El diagrama de secuencia ilustra las interacciones entre las clases del diseño participantes. En este caso, el usuario realiza una petición, que es capturada por el FacesServlet, que actúa como FrontEnd. Éste es el encargado de servir la JSP correspondiente. Cuando el usuario introduce sus datos y pulsa el botón de loggin, se realiza el submit del formulario. El bean de respaldo UserBean se activa con esta acción y delega en el UserService la acción de loggin. Esta clase es la encargada, mediante el DAO userDAO, de recuperar el usuario de la base de datos. Si el usuario existe, se servirá la siguiente JSP, CatalogoAdminJSP

5.7.3.4 Diseño del caso de uso Añadir producto

El diagrama de clases participantes en este caso de uso es muy parecido al anterior. En él, participa el ViewObject ProductoBean que contiene los datos del formulario añadirProductoJSP y las acciones y eventos para interactuar con la capa de lógica de negocio, delegando las peticiones, en este caso la acción crearProducto.

La implementación de la clase CatalogoServideImpl, modelada como un BussinesService, contiene la lógica de negocio relacionada con el manejo del catálogo y, mediante la operación añadirProducto, es la encargada de acceder a la capa de persistencia a través de la interfaz DAO CatalogoDao y su clase de implementación, CatalogoDAOToplinkImpl. Con ella, manejamos los datos persistentes del objeto de negocio Producto. Definimos un método, saveProductByID, que es el encargado de almacenar en la base de datos el objeto Producto creado.

Veamos el diagrama de clases participantes:

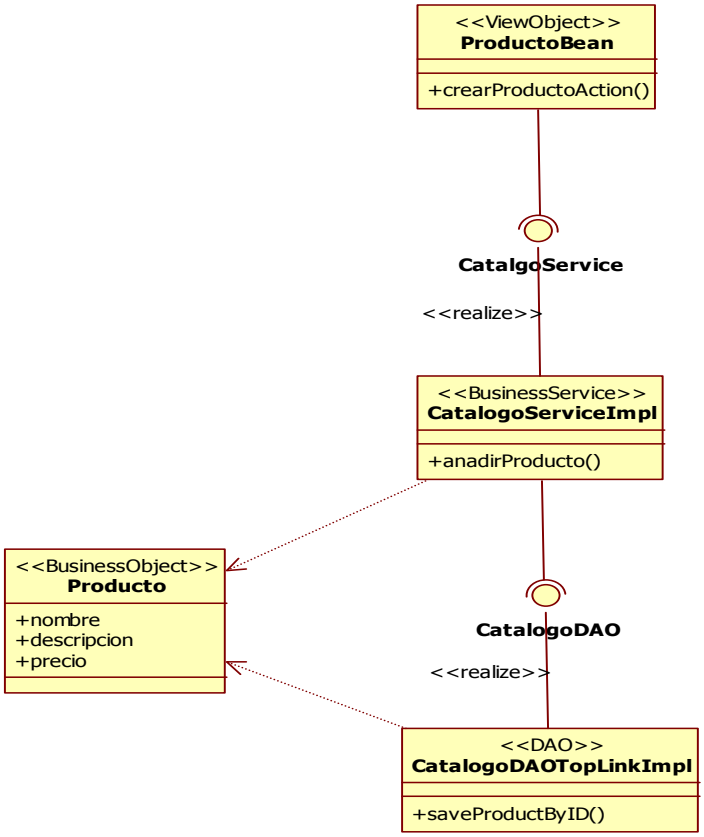

Figura 12. Diagrama de clases participantes UC añadir producto

Y este es el diagrama de secuencia que ilustra las interacciones entre las clases del diseño participantes:

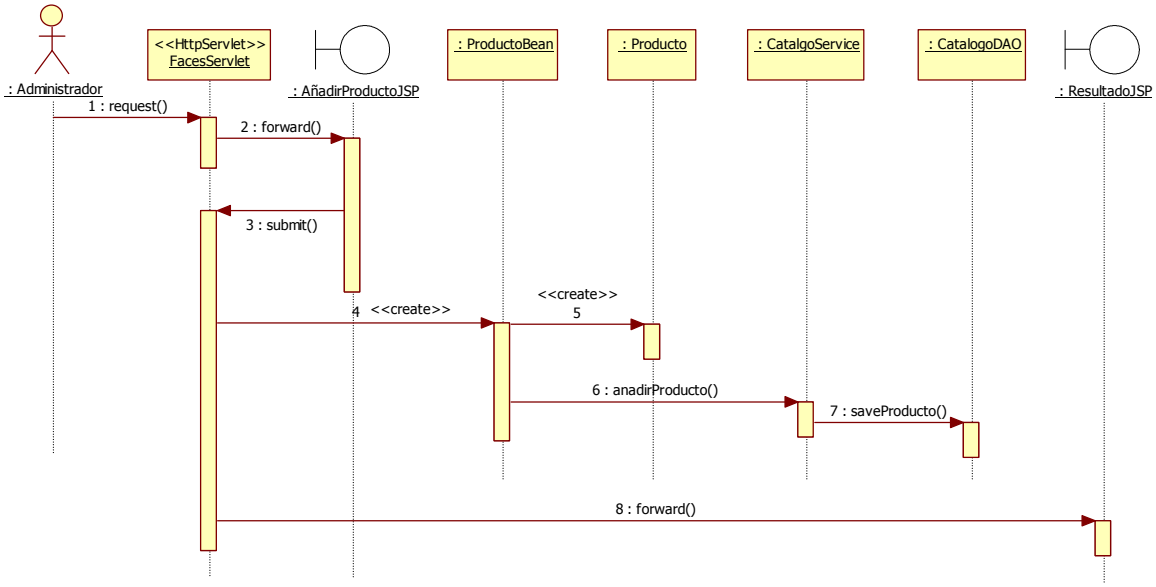

Figura 13. Diagrama de secuencia UC Añadir producto.

5.7.4 Diseño de la base de datos

Una vez definidas las clases del diseño, debe realizarse un modelo de persistencia. En este modelo deben definirse cómo las clases del diseño persistentes se mapean a las tablas de las bases de datos y la definición de estas tablas relacionales.

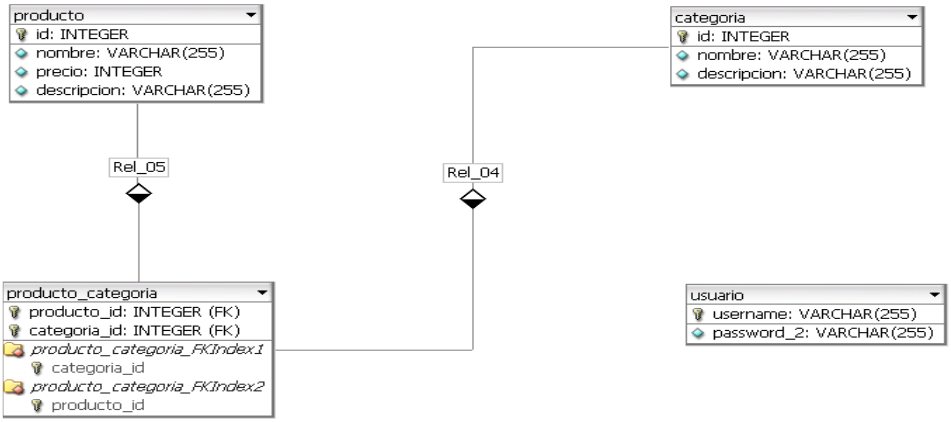

Figura 14: Diseño de la base de datos.

En la anterior figura vemos el diseño de las tablas de la aplicación. Se diseñan 4 tablas, las dos en las que persisten los objetos de negocio Usuario y Producto que hemos modelado y la tabla categoría que permite clasificar un producto dentro de una categoría. La tabla relación producto_categoria relaciona un producto con una categoría. Nótese que un producto siempre debe existir al menos en una categoría.

6. Fase de Construcción

6.1 Implementación de la capa de presentación

La implementación de la capa de presentación implica crear las páginas JSP, definir la navegación por las páginas, crear y configurar los beans de respaldo, e integrar JSF con la capa de lógica de negocio. Vamos a ver todo esto tomando como ejemplo la implementación del caso de uso "Crear Producto":

Página JSP:
createProduct.jsp es la página para crear un nuevo producto. Contiene componentes UI y conecta los componentes al ProductoBean.

Navegación de páginas:
La navegación por la aplicación se define en el fichero de configuación de la aplicación, faces-config.xml. Aquí están las reglas de navegación definidas para CreateProduct:

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>createProduct</from-outcome>
    <to-view-id>/createProduct.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

<navigation-rule>
  <from-view-id>/createProduct.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/uploadImage.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>retry</from-outcome>
    <to-view-id>/createProduct.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>cancel</from-outcome>
    <to-view-id>/productList.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Bean de respaldo:

El ProductoBean no sólo contiene los mapeos de propiedades a los datos de los componentes UI de la página, también tiene tres acciones: createAction, editAction, y deleteAction. Aquí está el código para el método createAction():

```
public String createAction() {
 this.logger.debug("createAction is invoked");

 try {
    Producto product = ProductoBuilder.crearProducto(this);
```

```
            this.getCatalogoService().guardarProducto(product);

            //store the current product id inside the session bean.
            //for the use of image uploader.
            FacesUtils.getSessionBean().setCurrentProductId(this.id.toString());

            //remove the productList inside the cache
            this.logger.debug("remove ProductListBean from cache");
            FacesUtils.resetManagedBean(BeanNames.PRODUCTO_LIST_BEAN);

        } catch (IDProductoDuplicadoException de) {
            String msg = "El producto con ID "+ this.id +"ya existe en la base de datos";
            this.logger.info(msg);
            FacesUtils.addErrorMessage(msg);

            return NavigationResult.RETRY;
        } catch (Exception e) {
            String msg = "Error al guardar el producto";
            this.logger.error(msg, e);
            FacesUtils.addErrorMessage(msg + ": Error de aplicación");

            return NavigationResult.FAILURE;
        }
        String msg = "El producto con ID " + this.id + " ha sido creado correctamente.";

        this.logger.debug(msg);
        FacesUtils.addInfoMessage(msg);

        return NavigationResult.SUCCESS;
    }
```

Dentro de la acción, se construye un objeto de negocio Producto basado en las propiedades de ProductoBean. Finalmente, createProduct pide su delegado al CatalogoService, que está en la capa de lógica-de-negocio, la acción de guardar el producto en la base de datos..

Declaración del Bean Manejado:

El ProductoBean se debe configurar en el fichero de configuración de recursos JSF, faces-config.xml:

```
<managed-bean>
 <description>
    Backing bean that contains product information.
 </description>
 <managed-bean-name>productoBean</managed-bean-name>
 <managed-bean-class>com.webapp.view.bean.ProductoBean</managed-bean-class>
 <managed-bean-scope>request</managed-bean-scope>
 <managed-property>
    <property-name>id</property-name>
    <value>#{param.productId}</value>
 </managed-property>
 <managed-property>
    <property-name>catalogoService</property-name>
    <value>#{catalogoService}</value>
 </managed-property>
</managed-bean>
```

Se configura el ProductoBean para tener un ámbito de petición, lo que significa que la implementación JSF

crea un nuevo ejemplar de ProductoBean por cada petición si se ha referenciado dentro de la página JSP. La propiedad ID-managed se rellena con el parámetro productId de la petición. La implementación JSF obtiene el parámetro desde la petición y selecciona la propiedad manejada.

6.2 Integración entre las capas de presentación y de lógica de negocio

La integración con las dos capas se realiza con Spring.

El modo recomendado para integrar Spring con JSF es configurar el ***DelegatingVariableResolver*** de Spring en **el faces-config.xml**. Los elementos *<application> <variable-resolver>* de un fichero faces-config.xml permiten a una aplicación basada en Faces registrar clases personalizadas para sustituir la implementación estándar de VariableResolver. El DelegatingVariableResolver de Spring primero delega al resolver original de la implementación subyacente de JSF, para luego delegar al webApplicationContext raíz de Spring.

Esto permite configurar los Spring Beans como propiedades gestionadas por los beans manejados de JSF. Por ejemplo, el catalogoService Spring Bean está configurado como una managed property del bean manejado productoBean :

Ejemplo de código de: faces-context.xml

```
<application>
        <variable-resolver>
                org.springframework.web.jsf.DelegatingVariableResolver
        </variable-resolver>
</application>
<managed-bean>
(….)
    <managed-bean-name>productoBean</managed-bean-name>
    <managed-property>
            <property-name>catalogoService</property-name>
            <value>#{catalogoService}</value>
    </managed-property>
</managed-bean>
```

El catalogoService, y su implementación CatalogoDAO, se define como un Spring bean en el fichero de configuración /WEB-INF/applicationContext.xml :

Ejemplo de código de: applicationContext.xml

```
<bean id="catalogoDAO" class="com.webapp.model.dao.toplink.CatalogoDAOImpl"/>

  <bean id="catalogoService" class="com.webapp.model.service.impl.CatalogoServiceImpl">
    <property name="catalogoDAO">
      <ref bean="catalogoDAO"/>
    </property>
  </bean>
```

<property name=”catalogoService“> se refiere al método setCatalogoService de productoBean. El webApplicationContext de Spring “inyecta” el Spring Bean catalogoService dentro de la property catalogoService del bean manejado productoBean:

Ejemplo de código de: ProductoBean.java

```
public class ProductoBean {
private CatalogoService catalogoService;

public void setCatalogoService(CatalogoService catalogoService) {
      this.catalogoService = catalogoService;
      this.init();
 }
```

6.3 Capa de lógica de negocio

La tarea de esta capa consiste en definir los objetos de negocio, creando los interfaces de servio con sus implementaciones, y conectando los objetos con Spring.

Objetos de Negocio:
Como JPA proporciona la persistencia, los objetos de negocio Producto, Usuario y Categoria deben proporcionar métodos get y set para todos sus campos.

Servicios de Negocio:
El interface CatalogoService define todos los servicios relacionados con el control del catálogo:

```
public interface CatalogoService {

   public Producto obtenerProductoByID(Integer anId) throws CatalogoException;
   public List obtenerProductosTodos() throws CatalogoException;
   public Categoria obtenerCategoriaByID(Integer id) throws CatalogoException;
   public List obtenerCategoriasTodas() throws CatalogoException;
   public void guardarProducto(Producto aProducto)throws CatalogoException;
   public void eliminarProducto (Producto aProducto)throws CatalogoException;
   public Producto modificarProducto (Producto aProducto)throws CatalogoException;

}
```

6.4 Usando la JavaPersitence API (JPA) con Spring

El Spring bean CatalogDAO usa el objeto EntityManager Query de la Java Persistence API para devolver una lista de items. El CatalogDAO anota el campo private EntityManager em; con @PersistenceContext, que hace que un entity manager sea inyectado. (Nótese que usar la anotación @PersistenceContext es el mismo método para inyectar un Entity Manager para un EJB 3.0 Session Bean.)

Ejemplo de código de: CatalogDAO.java

```
@Repository
@Transactional
public class CatalogoDAOImpl implements CatalogoDAO {

   @PersistenceContext(unitName = "onLineStorePU")
   private EntityManager em;

   public List <Producto> getAllProducts() {
      Query q = em.createQuery("select distinct product from Producto product order by product.id");
      return q.getResultList();
   }
```

La Java Persistence Query APIs se usa para crear y ejecutar consultas que puedan devolver una lista de resultados.

En el siguiente código, se muestra la clase de entidad Producto que se “mapea” con la tabla PRODUCTO que guarda las instancias de tipo producto. Éste es un objeto de entidad típico de la Java Persistence. Hay dos requisitos para una entidad:

1. anotar la clase con la anotación @Entity.
2. anotar el identificador de la clave primaria con @Id

```
@Entity
@Table(name = "producto")
public class Producto implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", nullable = false)
    private Integer id;
    @Column(name = "nombre")
    private String nombre;
    @Column(name = "descripcion")
    private String descripcion;
    @Column(name = "precio")
    private Double precio;
    @OneToMany
    @JoinTable(name = "producto_categoria", joinColumns =
    {@JoinColumn(name = "producto_id", unique = true)},
                                inverseJoinColumns = {@JoinColumn(name = "categoria_id")})
    private Set<Categoria> categorias;
```

Los campos nombre, descripción… son mapeos básicos de los campos del objeto a las columnas del mismo nombre en la tabla de la base de datos. Las relaciones O/R con la entidad Categoria mediante la tabla de relación PRODUCTO_CATEGORIA también son anotadas.

7. Conclusiones

Con el proyecto de final de carrera onLineStore, hemos conseguido el principal objetivo que nos marcamos al inicio, es decir, realizar un estudio detallado del framework JavaServer Faces para el desarrollo de aplicaciones Web, y su integración con otros marcos de trabajo, específicamente, Spring y TopLink. Con el desarrollo de onLineSotre, hemos cubierto todas las fases del diseño de una aplicación Web, incluyendo el descubrimiento de los requisitos del negocio, el análisis, la selección de tecnologías, la arquitectura de alto nivel, y el diseño a nivel de la implementación. Hemos discutido las ventajas y desventajas de las tecnologías utilizadas y se han esbozado aproximaciones para diseñar algunos de los aspectos clave de la aplicación.

A nivel personal, estoy muy satisfecho de la experiencia. En mi ámbito laboral, tengo amplia experiencia en el desarrollo de aplicaciones Web bajo la arquitectura J2EE con Struts, así que trabajar con este nuevo Framework, que cada vez más se está imponiendo en el mercado como solución para el desarrollo de aplicaciones Web, ha sido muy positivo. JSF es posterior a la herramienta de soporte para el desarrollo de aplicaciones Web Apache Struts, por lo que se nutre de su experiencia y mejora algunas sus deficiencias. JSF gana en flexibilidad del Controlador y manejo de eventos, navegación, desarrollo de páginas, integración y extensibilidad. También cabe destacar que el soporte de JSF en IDEs como Eclipse, Netbeans, etc. es mucho mejor. Otro punto a favor, a mi entender, es que JSF es parte de Java EE, Struts no.

Otro de los temas interesantes abordados en el proyecto, es la persistencia, utilizando TopLink JPA y las “annotations”. Es una forma elegante y muy productiva y de alto rendimiento para persistir Plain Old Java Objects (POJOs) y atacar el acceso a datos desde Java. Además, trabajando de forma conjunta con Spring, hemos conseguido superar la complejidad del acceso a datos persistentes en aplicaciones J2EE. Esta parte, para mi, es la más compleja del proyecto. Llegar a configurar el aplicationContext de la aplicación, y llegar a entender el manejo y declaración de transacciones, ha sido uno de los puntos que más trabajo ha llevado. Por otro lado, queda muchísimo por profundizar en este ámbito. Cuestiones como el manejo y declaración de transacciones y su configuración en la aplicación, a un nivel mucho más complejo, son temas que quedan por abordar en el futuro.

8. Glosario

8.1 Administrador

Persona que asume el rol de administrador de la aplicación. Es un Usuario registrado en el sistema con capacidad para llevar a cabo acciones de edición sobre el catálogo de productos.

8.2 Catálogo

Lista ordenada y sistematizada de productos relacionados con un fenómeno en particular.

8.3 Categoría

Cada uno de los grupos básicos en los que puede incluirse o clasificarse un producto. Se definirán 4 categorías en la aplicación.

8.4 Producto

Llamamos producto a cualquier ítem que se encuentre en el catálogo de la aplicación y que es manejado por éste.

8.5 Usuario

Persona física que asume el rol de usuario público en la aplicación. Solo puede manejar el catálogo para ver o buscar productos. Si el usuario se “loggea” y está dado de alta en el sistema, adquiere privilegios de administrador.

9. Bibliografía

Using Oracle TopLink with the Spring Framework. [en línea] *Lonneke Dikmans*. Oracle. Mayo 2007
< http://www.oracle.com/technology/pub/articles/dikmans-spring-toplink.html >

Build a Web Application (JSF) Using JPA. [en línea] Oracle. Agosto 2006
< http://www.oracle.com/technology/products/ias/toplink/jpa/tutorials/jsf-jpa-tutorial.html >

Persistencia y POJOs. [en línea] Rod Johnson y Jim Clark. Programacion.net. 21 jul. 2003
< http://www.programacion.net/java/articulo/jap_j2eemaster_9 >

Using Java Persistence in a Web Application. [en línea]. NetBeans.org.
< http://www.netbeans.org/kb/60/web/customer-book.html >

Introduction to the Spring Framework. [en línea]. Rod Johnson. The Server Side. Mayo 2005
< http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework >

Spring into JavaServer Faces. [en línea]. Michael Klaene. Gamelan.
< http://www.developer.com/java/ent/article.php/10933_3602061_1>

The Java EE 5 Tutorial. [en línea]. Varios autores. Sun Microsystems.
< http://java.sun.com/javaee/5/docs/tutorial/doc/>

Combining JavaServer Faces Technology, Spring, and the Java Persistence API,
and Supporting Tokens and Issued Token Delegation in WSIT. [en línea]. Carol McDonald
<http://java.sun.com/mailers/techtips/enterprise/2007/TechTips_Aug07.html#1>

Using the Java Persistence API (JPA) with Spring 2.0. [en línea] Mike Keith; Rod Johnson. Abril 2007
<http://java.sys-con.com/read/366275.htm/>

Getting Started With JPA in Spring 2.0. [en línea]. Mark Fisher. Mayo 2006
< http://blog.springsource.com/main/2006/05/30/getting-started-with-jpa-in-spring-20/>

Introducción a la Tecnología JavaServer Faces. [en línea].Sun microsystems.
<http://www.programacion.net/java/tutorial/jsf_intro/>

Sample Application using JSF, Spring 2.0, and Java Persistence APIs . [en línea].Java.net. Junio 2007
<http://weblogs.java.net/blog/caroljmcdonald/archive/2007/06/sample_applicat.html/>