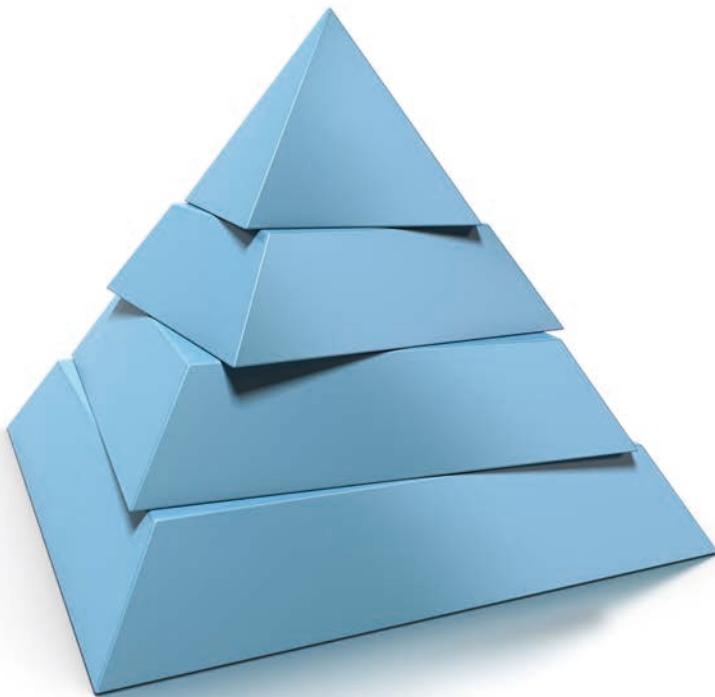

Arquitectura *java* Sólida

Cecilio Álvarez Caules
Sun Certified Enterprise Architect



ARQUITECTURA JAVA SÓLIDA

Cecilio Álvarez Caules

© 2012 Cecilio Álvarez Caules. Todos los derechos reservados.

ISBN : 978-1-291-16766-5

Agradecimientos

Este libro esta dedicado a la comunidad de desarrolladores JEE de Cantabria puesto que muchos de los capítulos se basan en reflexiones fundamentadas en las distintas preguntas que se me han formulado estos años en los cursos y consultorías realizados para su entorno. Dichas aportaciones me han proporcionado un animo esencial para escribir este libro.

Agradezco al grupo de arquitectura JEE del gobierno de Cantabria las distintas colaboraciones en las que hemos tomado parte estos años, ya que el presente libro recoge un cúmulo de ideas extraídas directamente de éstas.

Agradezco, igualmente, al equipo humano de Consultec los distintos proyectos en los que he participado estos últimos años : me han hecho crecer como arquitecto y me han ayudado a obtener una visión más neutral a nivel tecnológico sobre las distintas plataformas y frameworks existentes.

Mi agradecimiento va también para las aportaciones de las personas con las que trabajé mis primeros años de profesional en Mundivia. Esta etapa me ayudó a adquirir la visión necesaria sobre la orientación de mi futuro perfil profesional, algo que con el paso del tiempo ha sido clave en la evolución de mi carrera.

Gracias, así mismo, a Jesús Martín Fernández y a Miguel Blanchard Rodríguez por las distintas charlas e intercambios de ideas que hemos tenido estos años sobre JEE , además de los enfoques y distintas visiones que me han venido aportado.

Agradecer a Jose Manuel San Emeterio Perez de Emcanta los distintos procesos de certificación en los que participamos ambos ,que me permitieron definir el punto de partida de los contenidos del libro.

Es también necesario agradecer a Jordi Álvarez Caules y a Ana Patricia Fernández del Llano la revisión del documento y las ideas objetivas y sinceras que me transmitieron sobre cómo mejorar el contenido.

Agraceder a Olga Pelaez Tapia la revisión completa del documento a nivel de claridad de contenidos .Ya que sin ella tengo claro que muchas ideas no habrían quedado claras.

Por último, agradezco a mis padres la educación en la cultura del conocimiento que me han proporcionado, ya que sin ella no me hubiera convertido en arquitecto.

ARQUITECTURA JAVA SÓLIDA	2
Cecilio Álvarez Caules	2
Agradecimientos.....	4
Introducción	14
1. Conocimientos previos.....	14
2. Requerimientos de software.....	15
3. Instalación del entorno	15
4. Configuración del entorno	16
Resumen.....	20
1.HTML.....	22
1. Construir un formulario HTML	23
2. Validaciones de JavaScript.....	24
3. Añadir formato	25
4. Uso correcto de etiquetas.....	26
5. Accesibilidad de la pagina.....	27
6. Uso de XHTML como estándar.....	27
7. Uso de estándares DOM	28
8. Uso de JavaScript Degradable.....	30
Resumen.....	32
2.Java Server Pages	34

Arquitectura Java

1. Creación de una tabla Libros	34
2. Instalar el driver JDBC	35
3. Creación de la página “InsertarLibro.jsp”	36
4. Creación de la pagina MostrarLibros.jsp	38
Resumen	42
3.DRY y JSP	44
1. Añadir nueva clase	45
2. Modificar paginas JSP	48
3. Añadir Filtro por categoría	50
4. El principio DRY y las consultas SQL	52
5. El principio DRY métodos y parametros	55
6. ResultSets vs Listas de objetos	57
7. Uso de interfaces a nivel de Libro	59
8. Cierre de conexiones y reflection.....	61
4.Editar, Borrar y Filtrar	66
1. Añadir enlace de borrar	67
2. Añadir método borrar.....	67
3. Añadir página BorrarLibro.jsp.....	68
4. Añadir link Edición	69
5. Método de búsqueda por clave	70
6. Añadir formulario de edición de código.....	70
7. Añadir método salvar.....	73

8. Añadir pagina SalvarLibro.jsp	73
9. Añadir método buscarPorCategoria.....	75
Resumen.....	80
5.Manejo de excepciones	82
1. Flujo de excepciones y cláusulas catch	83
2. Manejo de las cláusulas throw, throws y flujo de excepciones.....	84
3. Creación y conversión de excepciones	86
4. Excepciones anidadas.....	90
5. Excepciones RunTime	92
6. Crear Pagina de Error	94
7. Modificar fichero web.xml	96
Resumen.....	97
6.Log4J.....	98
1. Instalación de log4j	99
2. Introducción a log4j	100
3. Construcción de un ejemplo sencillo.....	101
4. Mensajes de error y niveles.....	104
5. Manejo de Log4j.properties.....	107
6. Uso de log4j en nuestra aplicación.....	109
Resumen.....	111
7.El principio SRP y el modelo MVC	112
1. Responsabilidades de la aplicación y el principio SRP	113

2. Construir un servlet controlador	116
3. Mapeo de Servlet.....	117
4. Inserción con modelo MVC	121
5. Borrar en modelo MVC.....	123
6. Editar en modelo MVC.....	125
7. Filtrado en Modelo MVC	127
Resumen	131
8.JSTL	132
1. Instalación de JSTL	133
2. Introducción a JSTL.....	133
3. Etiquetas Básicas JSTL	134
4. Modificar MostrarLibro.jsp	135
5. Modificar Formulario Insertar.....	136
6. Modificar FormularioEditarLibro.jsp	137
Resumen	137
9.El principio OCP y modelo MVC 2.....	138
1. El principio OCP y el Controlador.....	139
2. El principio OCP y el patrón Command	140
3. Creación de una acción principal	142
4. Crear jerarquía de acciones	143
5. Api de reflection y el principio OCP.....	147
Resumen	150

10.Hibernate	152
1. Introducion al concepto de framework de persistencia.....	153
2. Instalación de Hibernate	154
3. Introducción a Hibernate	155
4. Configuración Hibernate	158
5. Insertar Libro con Hibernate	161
6. Seleccionar Objetos con Hibernate	164
7. Seleccionar un único Objeto	165
8. Borrar un objeto.....	166
9. Filtrar objetos con Hibernate	167
10. Migrar nuestra aplicación a Hibernate	168
11. Hibernate y Convención sobre Configuración.....	172
12. DRY e Hibernate.....	174
11.Hibernate y Relaciones	180
1. Construir la clase categoría y mapearla.....	182
2. Modificar la Capa de presentación y soportar la clase Categoría	184
3. Creación de relaciones entre clases.....	186
4. Relaciones y persistencia con Hibernate	188
5. Relaciones y capa de presentación	194
Resumen.....	197
12.Java Persistence API.....	198
1. Introducción al API de JPA	200

2. Migración de Aplicación a JPA.....	204
3. Manejo de Excepciones.....	207
Resumen	208
13.El principio ISP y el patrón DAO	210
Objetivos :	213
1. Crear las clases LibroDAO y CategoriaDAO.....	213
2. Uso de interfaces en componentes DAO	218
3. El principio DRY y el patrón GenericDAO.....	221
4. Rendimiento.....	228
Resumen	228
14.El principio de inversión de control y patrón factory	230
1. Crear Factorías e implementar el principio de IOC,.....	232
2. El principio DRY y el patrón Abstract Factory	237
Resumen	243
15.El Principio DRY y el patrón servicio	244
1. El principio DRY y el acceso a la capa de persistencia	245
2. Creación de una clase de Servicio	247
Resumen	251
16.El principio IOCy el framework Spring.....	252
1. Creación de ejemplo de factorias	255
2. Instalación de Spring.....	261
3. Instalación de Spring en nuestra aplicación.	266

Resumen.....	269
17.Inyeccion de Dependencia y Spring framework.....	270
1. Introducción al principio de Inyección de Dependencia.....	275
2. Spring e inyección de dependencia.....	280
3. Spring y factoría para aplicaciones web	283
4. Spring inyección de dependencia y Capas DAO	283
Resumen.....	287
18.El principio DRY y Spring Templates.....	288
1. El patrón Template.....	290
2. Spring y plantillas.....	297
3. Spring Herencia Plantillas y JDADOSSupport.....	302
Resumen.....	304
19.Programación Orientada a Aspecto (AOP)	306
1. Introducción a AOP.....	308
2. Usando Proxies con Spring	314
3. Configuración de proxies y transacciones.....	319
Resumen.....	326
20.Uso de anotaciones y COC.....	328
1. @PersistenceContext y EntityManager	329
2. @Repository y manejo de excepciones	332
3. Anotación @Service	333
4. Anotaciones @AutoWired.....	334

Arquitectura Java

Resumen	337
21.Java Server faces	338
1. Introducción a JSF.....	339
2. Instalación de JSF.....	340
3. Hola Mundo con JSF.....	342
4. Construir un combo con JSF	346
5. Crear Tabla con JSF	350
Resumen	352
21.Migración a Java Server Faces	354
1. Añadir controlador de JSF y eliminar el existente	355
2. Creación ManagedBean	358
3. Crear MostrarLibro.xhtml	362
4. Borrar Libro	365
5. Inserción Libro	366
6. Funcionalidad de Edición.....	368
7. Filtrar por categorias	371
8. Mejoras Expression Language 2.0.....	372
9. Integración de Spring.....	374
10. JSF 2 y Business Objects	375
Resumen	375
23.Servicios Web y JAX-WS.....	376
1. Introducción a programación distribuida	376

2. Introducción a Servicios Web	380
3. Servicio Web	382
4. Publicación del servicio web.....	387
5. Instalación de Apache CXF	387
6. Configuración del framework.....	388
Resumen.....	391
24.Administración y pools	392
1. El framework Spring y Pools.....	393
2. Pool de conexiones y Tomcat.....	394
3. Configuración de Spring vía JNDI.....	396
Resumen.....	399
25.Conclusiones	400
¿Que es lo mas importante?	400
1. JEE un ecosistema	402
Resumen.....	403
Bibliografía.....	404

Introducción

La idea original de escribir este libro viene de las distintas formaciones y consultorías sobre JEE que he realizado en los últimos años. En la plataforma JEE existen muchos libros que tratan sobre EJBs, JPA, Spring etc. Pero casi siempre estos libros se centran en un único producto. Así pues, para muchas personas resulta realmente difícil adquirir una visión global de cómo estos productos se integran unos con otros y conforman una solución empresarial. El objetivo de este libro no trata de hacernos expertos en Hibernate o en Spring sino de ser capaces de comenzar a manejar cada uno de estos frameworks y adquirir una visión global como arquitectos de cómo construir una aplicación empresarial .Para ello vamos a construir desde cero una aplicación JEE pequeña que irá evolucionando según vayamos avanzando capítulos y se apoyará en los distintos frameworks o estándares existentes a la hora de abordar las distintas problemáticas.

1. Conocimientos previos

Para poder abordar los conceptos que se explican en este libro no será necesario tener unos conocimientos amplios de la plataforma Java y de JEE .Ahora bien, sí es recomendable conocer un mínimo del lenguaje Java, así como unos mínimos de JEE ,concretamente los conceptos de Servlet y JSP. Por último, serán necesarios conocimientos sobre el lenguaje HTML.

2. Requerimientos de software

Una vez que tenemos claros los conocimientos necesarios para abordar con garantías los distintos capítulos, vamos a listar el software que utilizaremos a la hora de crear nuestra aplicación.

- Open JDK 1.6 o JDK 1.6
- FireFox
- Web Developer
- Eclipse JEE
- Tomcat 7
- MySQL
- Ubuntu 10.10 o Windows 7

3. Instalación del entorno

Antes de comenzar a desarrollar nuestra aplicación debemos abordar una serie de pasos previos relativos a la instalación.

Instalación JDK 1.6: En este caso simplemente nos bajamos el jdk 1.6 para Windows o bien lo instalamos desde el gestor de paquetes synaptic en caso de usar Ubuntu (open jdk).

Firefox y WebDeveloper : No hay mucho que decir en esta sección simplemente obtendremos el navegador, lo instalamos y añadimos el plugin de web developer correspondiente.

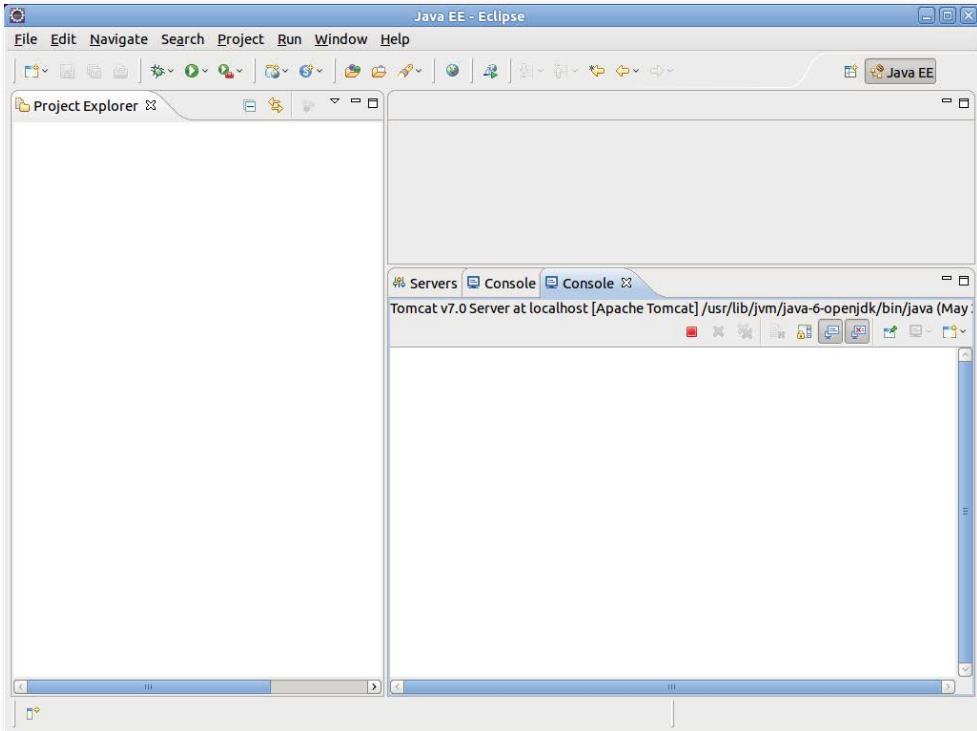
Eclipse JEE : Obtendremos el entorno de desarrollo eclipse y lo descomprimimos en un directorio cualquiera ya que, al estar desarrollado en java y tener instalado el jdk ,simplemente con pulsar sobre su icono el entorno se lanzara sin mayores necesidades de instalación.

Tomcat 7: Obtendremos el servidor web Tomcat de la pagina de apache y lo descomprimimos en un directorio paralelo al que tenemos ubicado eclipse JEE

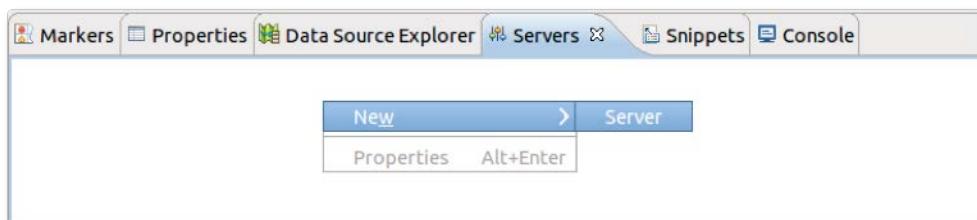
MySQL: Bajamos el instalador e instalamos el servidor de MySQL con las opciones por defecto .Añadimos las GUI Tools o el MySQL Workbench para una gestión más sencilla del servidor .En Ubuntu podemos instalar MySQL y sus herramientas gráficas a través de Synaptic.

4. Configuración del entorno

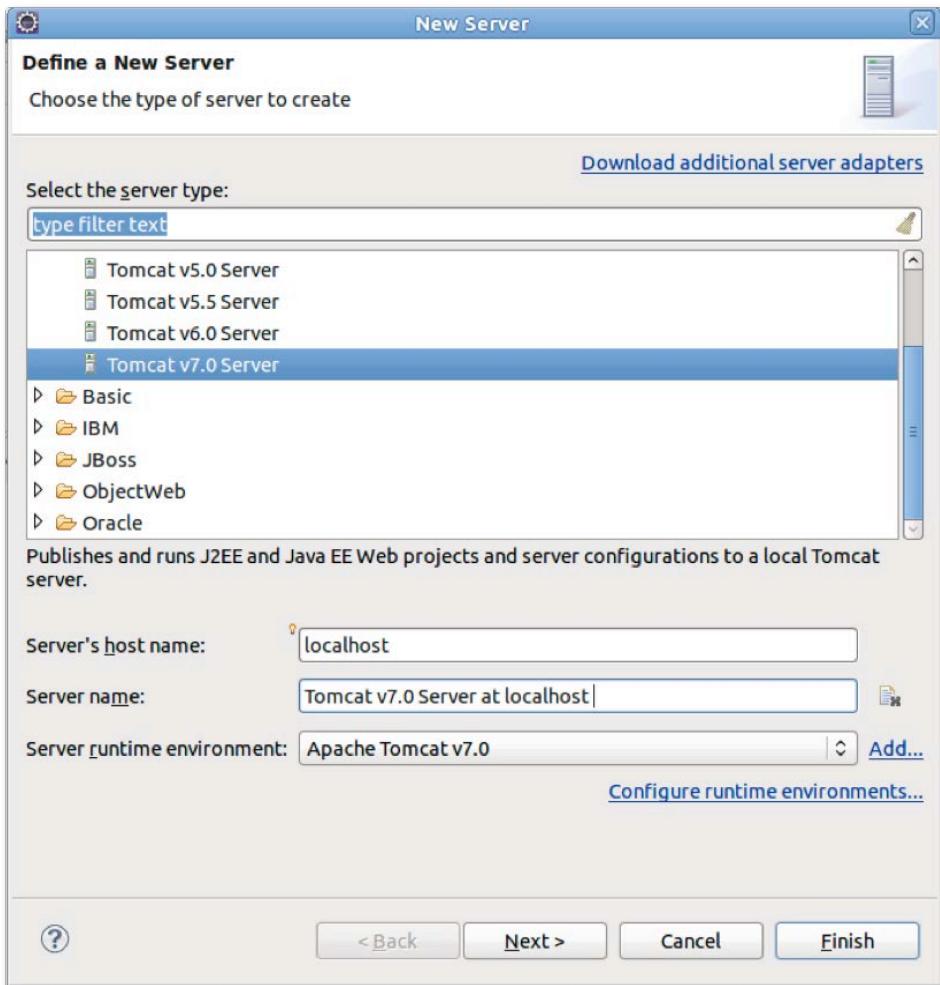
Hemos instalado ya todo el software que necesitamos y acabamos de abrir el eclipse (ver imagen)



Es momento de integrar Eclipse JEE con Tomcat 7. Esto puede realizarse de una forma muy sencilla en la pestaña de servers, pulsando botón derecho **New>Server** (ver imagen)

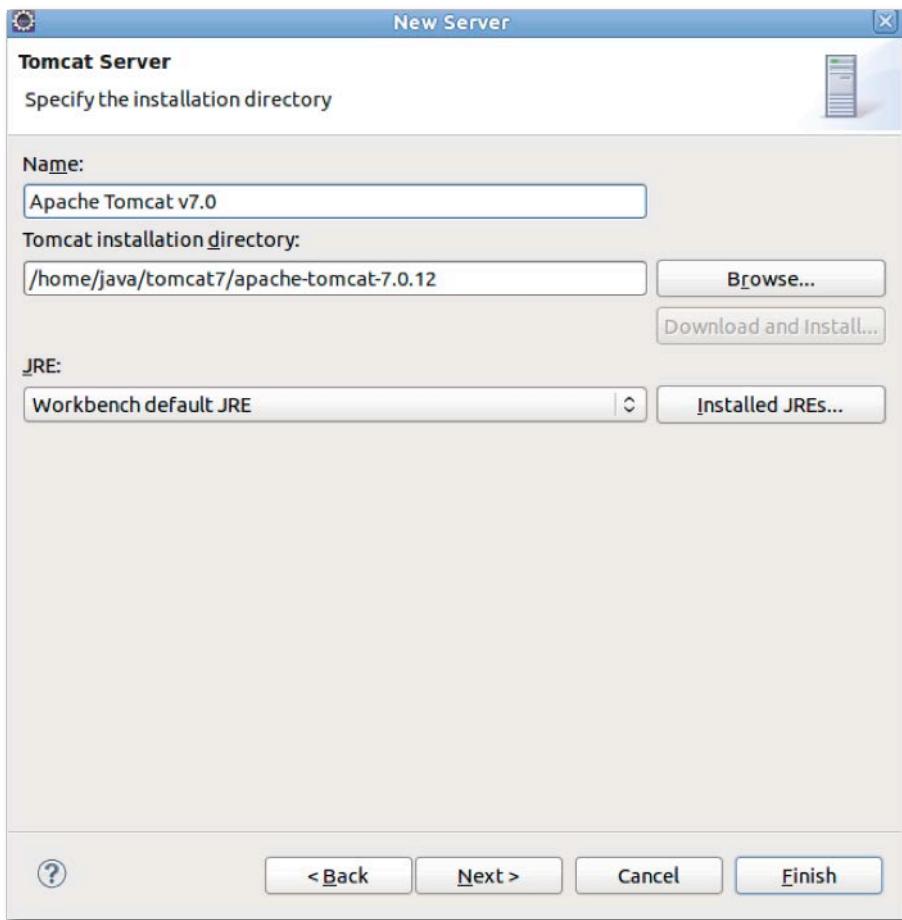


Al pulsar esta opción pasaremos a elegir el tipo de servidor que deseamos añadir a nuestro entorno de desarrollo (Tomcat 7) (ver imagen).

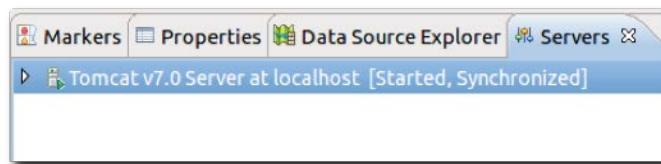


Elegido el tipo de servidor, nos solicitará que especifiquemos en qué directorio se encuentra instalado (ver imagen).

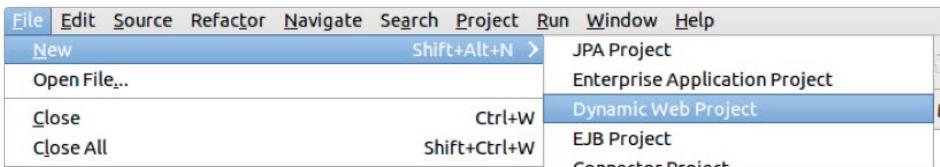
Arquitectura Java



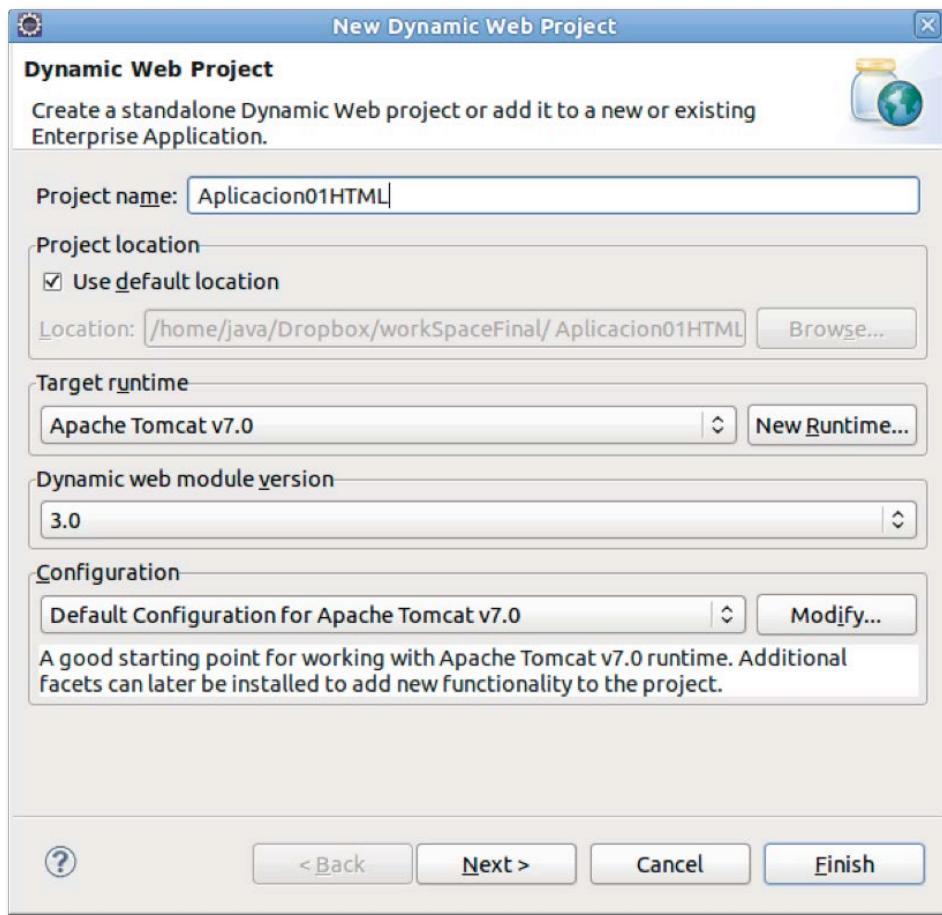
Seleccionamos el directorio y podemos pulsar “Finish” . En la pestaña de servidores nos aparecerá el nuevo servidor para que podamos utilizarlo.



Configurado el servidor , vamos a proceder a construir nuestra primera aplicación web y desplegarla en dicho servidor. Para ello usaremos el menú de **File>New>Dynamic Web Project** y crearemos un nuevo proyecto web (ver imagen) .



Una vez que hayamos elegido el tipo de proyecto, eclipse nos solicitará que lo alojemos en alguno de los servidores que tenemos definidos. En nuestro caso Tomcat 7 (ver imagen)



Tras realizar estas operaciones, dispondremos en eclipse de un nuevo proyecto web sobre JEE con el que podremos comenzar a construir nuestra aplicación como se muestra seguidamente.



Resumen

Este capítulo ha servido para introducir los objetivos del libro así como para configurar el entorno de desarrollo que vamos a utilizar en capítulos posteriores. En el próximo capítulo comenzaremos con el desarrollo de la aplicación.

1. HTML

En este capítulo vamos a comenzar a construir una pequeña aplicación web sobre JEE que nos ayudará a gestionar una colección de libros. La aplicación se encargará de añadir, borrar y modificar los distintos libros que tenemos e irá evolucionando durante el transcurso de los distintos capítulos según se vayan abordando las distintas tecnologías de la plataforma JEE.

Comenzaremos entonces por la parte más sencilla: la construcción de un formulario HTML a través del cuál insertaremos la información del libro que deseamos guardar en la aplicación.

Objetivo:

- Crear un formulario HTML que nos permita dar de alta nuevos libros en la aplicación.

Tareas:

1. Construir un formulario HTML que contenga los campos necesarios para dar de alta un nuevo libro (ISBN, título, categoría).
2. Construcción de las validaciones de JavaScript que el formulario necesita para poder gestionar la información.
3. Aplicar formato al formulario utilizando una hoja de estilo CSS.
4. Revisión del uso correcto de etiquetas HTML en el formulario.
5. Uso de etiquetas de accesibilidad en el formulario.
6. Uso de XHTML como estándar en el formulario.
7. Uso de DOM como estándar a nivel de JavaScript.
8. Uso de JavaScript degradable en el formulario.

Acabamos de definir el conjunto de tareas a realizar .Quizá en este momento no tengamos claras todas ellas pero se irán clarificando según las vayamos abordando.

1. Construir un formulario HTML

La primera tarea que debemos llevar a cabo es sencilla y se trata de construir un formulario HTML para dar de alta nuevos libros .Para ello vamos a definir el conjunto inicial de campos que va a contener .

Campos:

- ISBN
- Título
- Categoría

Una vez definidos los campos, es momento de construir el formulario utilizando etiquetas HTML (ver imagen)

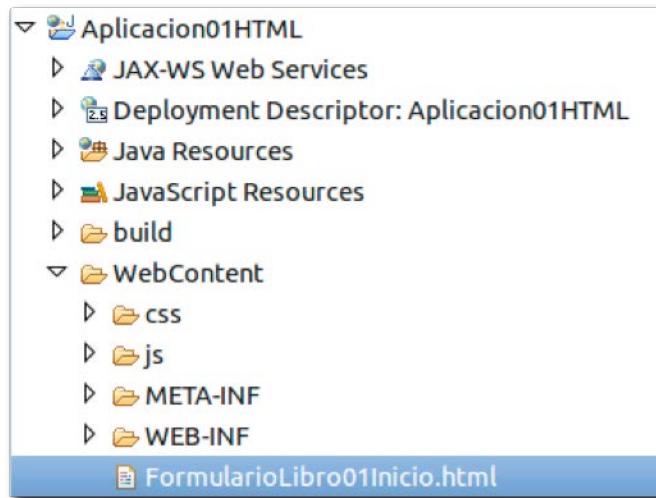
Código 1.1: (FormularioLibro01Inicio.html)

```
<html>
<head>
<title>Formulario Libro</title>
</head>
<body>
<h1>Formulario alta Libro</h1>
<form>
ISBN:
<input type="text" name="isbn"><br>
Título:
<input type="text" name="titulo"><br>
Categoría:
<input type="text" name="categoria"><br>
<input type="button" value="Insertar" >
</form>
</body>
</html>
```

Nuestra primera versión del formulario es realmente sencilla, como se muestra (ver imagen).

The screenshot shows a web browser window with the URL <http://localhost:8080/Applicacion01HTML/FormularioLibro01Ini>. The page title is "Formulario alta Libro". Below the title are three input fields: "ISBN:" with a placeholder, "Titulo:" with a placeholder, and "Categoria:" with a placeholder. At the bottom is a blue "Insertar" button.

El formulario se encuentra ubicado dentro de un proyecto web de eclipse (ver imagen)



2. Validaciones de JavaScript

El objetivo de este libro es ayudarnos a tomar buenas decisiones orientadas a la arquitectura, las validaciones que vamos a usar en el formulario serán sencillas y en este caso únicamente nos encargaremos de validar si el ISBN esta vacío. Vamos a ver a continuación el código de JavaScript que se encarga de esta funcionalidad dentro de la página.

Código 1.2: (FormularioLibro02JavaScript.html)

```
<html>
<head>
<script type="text/javascript">
    function validacion() {
        if (document.forms[0].isbn.value == "") {
            alert("datos no validos");
        } else {
            document.forms[0].submit();
        }
    }
</script>
<title>Formulario Libro</title>
</head>
<body>
<h1>Formulario alta Libro</h1>
<form>
ISBN:
<input type="text" name="isbn"><br>
Título:
<input type="text" name="titulo"><br>
Categoría:
<input type="text" name="categoria"><br>
<input type="button" value="Insertar" onclick="validacion()">
</form>
</body>
</html>
```

Hemos añadido una función de validación y un evento “onclick” al formulario, el cuál a partir de este momento obliga a que introduzcamos el ISBN .

3. Añadir formato

Para añadir formato a nuestro formulario usaremos una hoja de estilo mínima que simplemente presentará el texto del formulario en negrita (ver imagen)

Código1.3 :(FormularioLibro03CSS.html)

```
<style type="text/css">
font-weight:bold;
</style>
```

4. Uso correcto de etiquetas

Es frecuente encontrarnos con situaciones en las que, aunque se usa una tecnología, ésta no se conoce en profundidad. Éste es el caso de la página que acabamos de crear, ya que usamos una etiqueta **<h1>** de cabecera para asignar un título al formulario y etiquetas **
** para separar el contenido, algo que no es correcto, ya que existen etiquetas más específicas que pueden realizar estas tareas. Tales etiquetas son, en concreto:

<field></field>y <legend></legend>:Etiquetas encargadas de agrupar un conjunto de campos de un formulario y asignar a este un título o leyenda.

<p></p>:Etiqueta que se encarga de definir un párrafo

Así pues, es preciso modificar las etiquetas de nuestro formulario de la siguiente forma.

Código1.4: (FormularioLibro04UsoEtiquetas.html)

```
<form action="insertarLibro.jsp">
<fieldset>
<legend>Formulario alta libro</legend>
<p>ISBN:<br/>
<input type="text" name="isbn">
</p>
<p>
    Titulo:<br/>
    <input type="text" name="titulo">
</p>
<p>
    Categoría:<br/>
    <input type="text" name="categoria">
</p>
<p>
    <input type="button" value="Insertar" onclick="validacion()">
</p>
</fieldset>
</form>
```

Como podemos observar, no solo hemos añadido las nuevas etiquetas sino que también hemos eliminado las que no eran correctas "**<h1>**".

5. Accesibilidad de la pagina

En estos momentos nuestra pagina HTML carece de las etiquetas habituales de accesibilidad, como por ejemplo la etiqueta **<label>**, las cuáles definidas en el formulario permiten una mejor accesibilidad a usuarios discapacitados. Para solventar este problema vamos a añadirlas a nuestro código (ver código).

Código 1.5: (FormularioLibro05Accesibilidad.html)

```
<fieldset>
<legend>Formulario alta libro</legend>
<p><label for="isbn">ISBN:</label>
<input type="text" name="isbn"/>
</p>
<p>
<label for="titulo">Titulo:</label>
<input type="text" name="titulo"/>
</p>
<p>
<label for="categoria">Categoria :</label>
<input type="text" name="categoria"/>
</p>
<p>
<input type="button" value="Insertar" onclick="validacion()"/>
</p>
</fieldset>
```

Después de añadir estas etiquetas, podemos pasar a las siguientes tareas encargadas de mejorar el uso de estándares a nivel de HTML.

6. Uso de XHTML como estándar.

XHTML es hoy por hoy el estándar que se utiliza a la hora de construir páginas HTML , en espera de que HTML 5 se imponga en el mercado. Para conseguir que nuestra página web cumpla con los requerimientos de XHTML, necesitamos realizar las siguientes modificaciones a nuestra página.

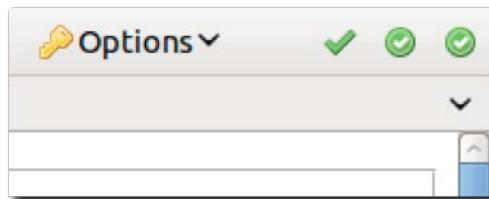
1. Añadir declaración XML al documento
2. Añadir DOCTYPE al documento
3. Añadir espacio de nombres (xml namespace)
4. Todas las etiquetas deben ser declaradas en minúsculas
5. Toda etiqueta abierta debe ser cerrada

Aunque en principio parecen bastantes cambios, son rápidos de aplicar. Vamos a ver como queda modificado nuestro código para cumplir con XHTML.

Código 1.6: (FormularioLibro06XHTML.html)

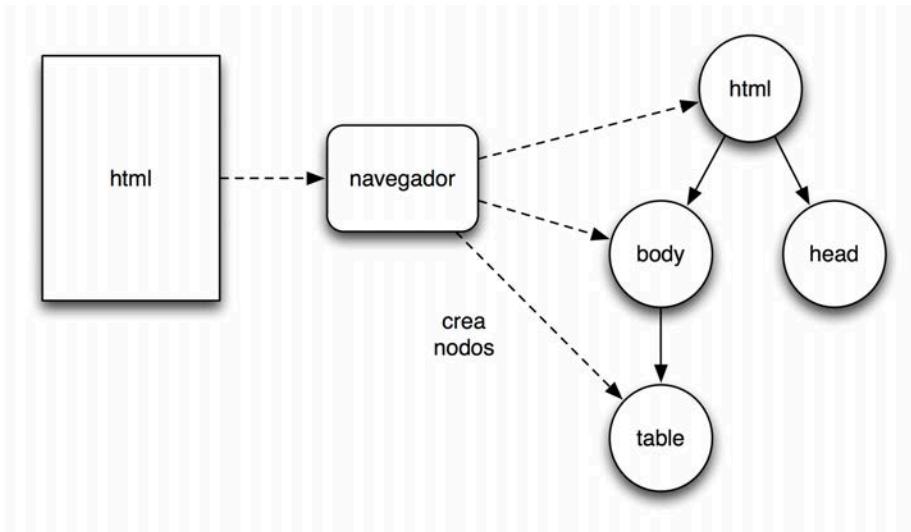
```
<?xml version="1.0" encoding="UTF-8"?>1
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"DTD/xhtml1-strict.dtd">2
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="es" lang="es">3
<head>
<!--omitimos JavaScript y cabecera-->
<title>Ejemplo01</title>
</head>
<body>
<form >
<fieldset>
<legend>Formulario alta libro</legend>
<!--Omitimos etiquetas que no varían-->
<p>
<input type="button" value="Insertar" onclick="validacion()"/>
</p>
</fieldset></form>
</body>
</html>
```

Hemos añadido una serie de cabeceras al documento que permiten validar el formulario como XHTML y nos hemos encargado también de abrir y cerrar correctamente las etiquetas HTML. En estos momentos podemos nuestro documento cumple los estándares algo que podemos comprobar si utilizamos la herramienta WebDeveloper de Firefox como muestra la siguiente imagen.



7. Uso de estándares DOM

DOM o Document Object Model es el estándar que usan todos los navegadores para representar en memoria la página como una serie de nodos enlazados entre sí en forma de árbol (ver imagen).



DOM es usado por JavaScript para acceder a los distintos elementos de la página. En nuestro caso, el uso de DOM afecta a la función de validación que hemos construido en JavaScript. Para facilitar y estandarizar el uso de DOM en nuestra página, modificaremos algunas de las etiquetas de nuestro formulario HTML y les añadiremos atributos de identificador “id” como muestra el siguiente bloque de código.

Código 1.7: (FormularioLibro07DOM.html)

```

<p><label for="isbn">ISBN:</label>
<input id="isbn" type="text" name="isbn"/></p>
<p>
<label for="titulo">Titulo:</label>
<input id="titulo" type="text" name= "titulo"/>
</p><p>
<label for="categoria">Categoria :</label>
<input id="categoria" type="text" name="categoria"/>
</p>
  
```

Una vez realizados estos cambios a nivel de estructura HTML, podremos usar el API de DOM de JavaScript para acceder de una forma más directa a los distintos elementos de la pagina .En este caso, haremos uso de la siguiente instrucción.

Código 1.8: (FormularioLibro07DOM.html)

```
document.getElementById("identificador");
```

Arquitectura Java

Dicha instrucción sirve para acceder a un elemento determinado del documento a través de su identificador ; de esta forma, la función de JavaScript pasará a tener la siguiente estructura.

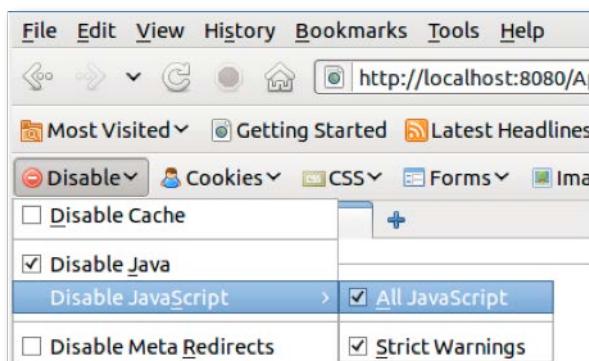
Código1.9: (FormularioLibro08JavaScriptDOM.html)

```
function validacion() {  
    var isbn= document.getElementById("isbn");  
    var miformulario=document.getElementById("miformulario");  
    if(isbn.value==""){  
        alert("datos no validos");  
        return false;  
    }else{  
        miformulario.submit();  
    }  
}  
</script>
```

El uso de DOM a nivel de JavaScript visiblemente nos permite ganar en claridad y sencillez a la hora de construir nuestras funciones de validación. Llegados a este punto, hemos terminado de definir las tareas asociadas al uso de estándares. La última tarea pendiente está orientada a mejorar el propio funcionamiento de la página en el caso de que el navegador no soporte JavaScript (usuarios con discapacidad).

8. Uso de JavaScript Degradable

Es el momento de ver cómo nuestro formulario se comporta en situaciones adversas como puede ser el caso de que el navegador no disponga de JavaScript. Para simular esta situación, usaremos el plugin **WebDeveloper** que permite deshabilitar el JavaScript de una página en concreto . La siguiente imagen muestra deshabilitar JavaScript usando esta herramienta.



Tras deshabilitar el JavaScript, nos podemos dar cuenta de que el botón del formulario ha dejado de funcionar. Esto se debe a que nuestro formulario no ha sido diseñado para soportar una degradación correcta de JavaScript y deja de funcionar en ausencia de éste. Para conseguir que el formulario se degrade es necesario realizar dos modificaciones.

1. Cambiar el tipo de botón que estamos usando por tipo “**submit**”.
2. Cambiar el evento de tipo “**onclick**” a tipo “**onsubmit**” y ubicarlo a nivel del propio formulario

A continuación se muestran las modificaciones que se precisan realizar en código.

Código 1.10:(FormularioLibro09JavascriptDegradable.html)

```
<body>
<form action="destino.html" onsubmit="return validacion();"
<!--código omitido-->
<input type="submit" value="Insertar" />
</form>
```

Una vez realizadas estas modificaciones el formulario funcionará de forma correcta en el caso de que el navegador no soporte JavaScript. En estos momentos disponemos de una versión final de nuestro formulario. A continuación, pasaremos a modificar la CSS de éste para que encaje de forma más natural con las últimas modificaciones realizadas. A continuación se muestra el código de la nueva hoja.

Código1.11 : (formato.css)

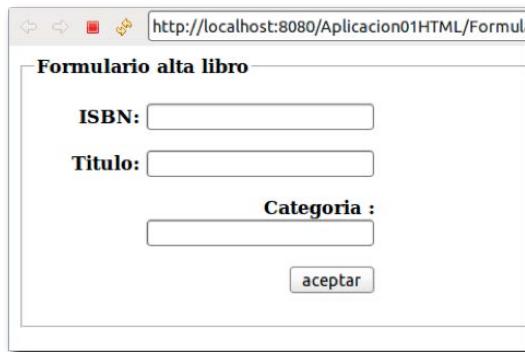
```
legend,label {
    font-weight:bold;
}
p {
    text-align:right;
    width:300px;
}
```

Por ultimo modificaremos la página para que externalice correctamente la hoja de estilo y la función de JavaScript.

Código1.12: (validacion.js)

```
<head>
<link rel="stylesheet" type="text/css" href="css/formato.css" />
<script type="text/javascript" src="js/validacion.js" >
</script>
</head>
```

Una vez realizadas estas modificaciones podemos ver el resultado final



Resumen

Este capítulo nos ha servido para construir un formulario HTML que usaremos mas adelante en nuestra aplicación JEE .Pero sobre todo nos ha servido para irnos familiarizando con la estructura de contenidos que tendrán los distintos capítulos.

2. Java Server Pages

En el capítulo anterior hemos construido un formulario HTML. En este capítulo nos encargaremos de construir las primeras páginas JSP de nuestra aplicación, éstas se encargarán de guardar los datos de nuestros libros en base de datos, así como de mostrar una lista con los libros que hemos almacenado en ésta. A continuación se detallan los objetivos del capítulo.

Objetivos:

- Crear la pagina “InsertarLibro.jsp” que se encargará de insertar libros en nuestra base de datos.
- Crear la pagina “MostrarLibros.jsp” que se encargará de presentar una lista con los libros almacenados en la base de datos.

Tareas:

1. Construcción de la tabla Libros en un servidor MySQL.
2. Instalación de un driver JDBC nativo para acceder a la base de datos desde Java.
3. Creación de la pagina “InsertarLibro.jsp”.
4. Creación de la pagina “MostrarLibros.jsp”.

1. Creación de una tabla Libros

Para realizar esta tarea necesitamos haber instalado previamente un servidor MySQL y añadido un esquema denominado “**arquitecturajava**” a éste. Una vez realizadas estas dos operaciones elementales, usaremos la herramienta MySQL QueryBrowser para crear una tabla con los siguientes campos:

- Isbn: varchar (10).
- Titulo: varchar (30).
- Categoría: varchar (30).

A continuación se muestra una imagen con la tabla ya creada dentro del esquema.



Tras crear la tabla en la base de datos, la siguiente tarea que nos ayuda a instalar un driver JDBC.

2. Instalar el driver JDBC.

Al crear la tabla en el servidor MySQL será preciso acceder a ella desde nuestra aplicación web , para ello necesitaremos instalar en la aplicación un driver JDBC nativo para MySQL. El driver se puede obtener de la siguiente url.

- <http://www.mysql.com/downloads/connector/j/>

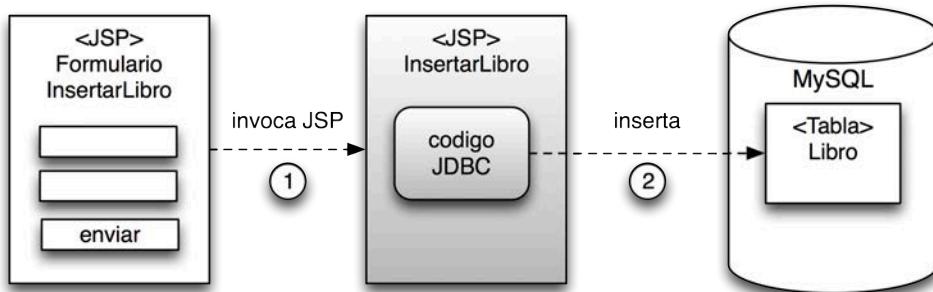
Después de obtener el driver (formato tar.gz para Ubuntu) lo vamos a descomprimir y así obtendremos el fichero **mysql-connector-java-5.1.12-bin.jar** que contiene las clases que nos permiten conectarnos a MySQL desde Java. Tras obtener este fichero, lo añadiremos a la carpeta **lib** de nuestra aplicación web (ver imagen)



Terminadas ya estas primeras tareas orientadas más hacia la administración que hacia el desarrollo, podemos comenzar a construir las páginas de nuestra aplicación.

3. Creación de la página “InsertarLibro.jsp”

La página “InsertarLibro.jsp” recogerá los datos enviados por el formulario construido en el capítulo anterior e insertará un nuevo registro en la base de datos .El siguiente diagrama muestra la relación entre los tres elementos.



Como podemos observar hemos renombrado el fichero del capítulo anterior como

- FormularioInsertarLibro.jsp

Asignándole .jsp para que, a partir de estos momentos, todas las páginas comparten la misma extensión. Es el momento de pasar a construir la página “InsertarLibro.jsp”, a continuación se muestra su código fuente.

Código 2.1: (InsertarLibro.jsp)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!-- sentencias de import necesarias para jdbc-->
<%@ page import="java.sql.Connection"%>
<%@ page import="java.sql.Statement"%>
<%@ page import="java.sql.DriverManager"%>
<%@ page import="java.sql.SQLException"%>

<%
    //1
    String isbn = request.getParameter("isbn");
    String titulo = request.getParameter("titulo");
    String categoria = request.getParameter("categoria");

    Connection conexion = null;
    Statement sentencia = null;
```

```

int filas=0;
try {
    //2
    Class.forName("com.mysql.jdbc.Driver");
    conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost/arquitecturajava",
        "root",
        "java");

    sentencia = conexion.createStatement();
    //3
    String consultaSQL = "insert into Libros (isbn,titulo,categoría) values ";
    consultaSQL += "(" + isbn + "," + titulo + "," + categoria + ")";
    //4
    filas = sentencia.executeUpdate(consultaSQL);

    response.sendRedirect("MostrarLibros.jsp");

} catch (ClassNotFoundException e) {

    System.out.println("Error en la carga del driver"
        + e.getMessage());

} catch (SQLException e) {
    System.out.println("Error accediendo a la base de datos"
        + e.getMessage());
} finally {
    //5
    if (sentencia != null) {

        try {sentencia.close();}
        catch (SQLException e)
            {System.out.println("Error cerrando la sentencia" +
                e.getMessage());}
    }
    if (conexion != null) {

        try {conexion.close();}
        catch (SQLException e)
            {System.out.println("Error cerrando la conexion" +
                e.getMessage());}
    }
}
}%>

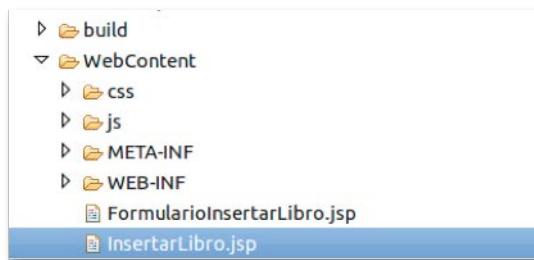
```

Es evidente que el código de la pagina aunque sencillo, es también bastante extenso ya que se encarga de gestionar la conexión a la base de datos y posterior ejecución de una consulta .A continuación se enumeran las principales operaciones que el código realiza.

Arquitectura Java

1. Lee la información que proviene de FormularioInsertarLibro.html usando el objeto request de JSP.
2. Crea un objeto de tipo Connection(conexión) y un objeto de tipo Statement (sentencia)
3. Crea una consulta SQL de inserción con los datos del libro
4. Ejecuta la sentencia con su SQL
5. Cierra los recursos (conexión ,sentencia etc)

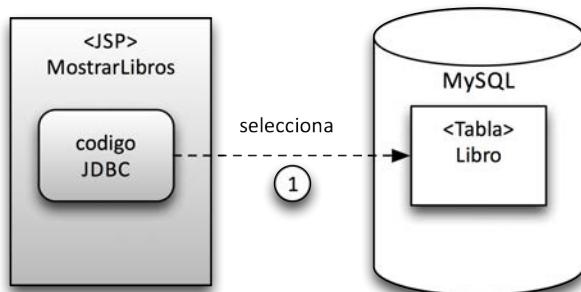
A continuación se muestra una imagen de la estructura del proyecto y el nuevo fichero que acabamos de crear.



Terminada esta primera página, es momento de abordar el segundo objetivo del capítulo.

4. Creación de la pagina MostrarLibros.jsp

La página “MostrarLibros.jsp” se encargará de mostrar una lista completa de todos los libros que tenemos almacenados en la base de datos. Para ello hará uso del API de JDBC (ver imagen).



Éste es el código fuente de la página.

Código 2.2:MostrarLibros.jsp

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ page import="java.sql.Connection" %>
<%@ page import="java.sql.Statement" %>
<%@ page import="java.sql.DriverManager" %>
<%@ page import="java.sql.SQLException" %>
<%@ page import="java.sql.ResultSet" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Lista de Libros</title>
</head>
<body>
<%
Connection conexion=null;
Statement sentencia=null;
ResultSet rs=null;
try {
    Class.forName("com.mysql.jdbc.Driver");
    //1
    conexion =
DriverManager.getConnection("jdbc:mysql://localhost/arquitecturajava",
                                "root",
                                "java");

    sentencia= conexion.createStatement();
    //2
    String consultaSQL= "select isbn,titulo,categoría from Libros";
    //3 y 4
    rs=sentencia.executeQuery(consultaSQL);
    //5
    while(rs.next()) { %>

        <%=rs.getString("isbn")%>
        <%=rs.getString("titulo")%>
        <%=rs.getString("categoría")%>
        <br/>

    <% } %>
}catch (ClassNotFoundException e) {

    System.out.println("Error en la carga del driver")
}

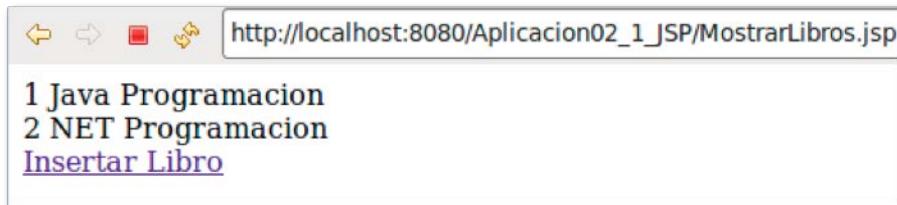
```

```
+ e.getMessage());  
  
}catch (SQLException e) {  
  
    System.out.println("Error accediendo a la base de datos"  
                       + e.getMessage());  
}  
finally {  
    //6  
    if (rs != null) {  
  
        try {rs.close();} catch (SQLException e)  
        {System.out.println("Error cerrando el resultset" + e.getMessage());}  
  
    }  
  
    if (sentencia != null) {  
  
        try {sentencia.close();} catch (SQLException e)  
        {System.out.println("Error cerrando la sentencia" + e.getMessage());}  
  
    }  
    if (conexion != null) {  
  
        try {conexion.close();} catch (SQLException e)  
        {System.out.println("Error cerrando la conexion" + e.getMessage());}  
    }  
}  
%>  
<a href="FormularioInsertarLibro.jsp">Insertar Libro</a>  
</body></html>
```

Podemos apreciar que el código fuente de MostrarLibro.jsp realiza las siguientes tareas.

1. Crea un objeto conexión y un objeto sentencia.
2. Crea una consulta SQL de selección para todos los libros de la tabla.
3. Ejecuta la sentencia con su SQL.
4. Devuelve un *ResultSet* con todos los registros.
5. Recorre el *ResultSet* y lo imprime en html.
6. Cierra los recursos (conexión ,sentencia, etc).

Una vez creadas ambas páginas, podemos invocar la página MostrarLibros.jsp a través del navegador y así ver cuál es la lista inicial de libros que presenta(hemos insertado 2).



Tras cargar esta página, podremos pulsar en el enlace que nos redirige al formulario de inserción (ver imagen)

Formulario alta libro

ISBN:

Título:

Categoría :

En el momento en que pulsemos al botón de insertar un nuevo libro, éste será insertado en la tabla Libros y la aplicación volverá a mostrar la página MostrarLibros.jsp (ver imagen).



Arquitectura Java

Por último mostramos la estructura de ficheros final del capítulo.



Resumen

En este capítulo hemos progresado al añadir una nueva funcionalidad a nuestra aplicación, que en estos momentos ya es capaz de insertar y seleccionar registros en la base de datos.

3.DRY y JSP

En el capítulo anterior hemos construido la funcionalidad de insertar y listar libros . Sin embargo, si revisamos el código de nuestras páginas, veremos que gran parte es idéntico en ambas. Ésto es muy habitual en el desarrollo de aplicaciones y genera problemas de mantenimiento pues si en algún momento hay que cambiar parte del código, en vista de que lo hemos repetido en todas las páginas, será necesario realizar las modificaciones una por una . Para solventar este problema, en este capítulo introduciremos uno de los principios mas importantes de ingeniería de software **el principio DRY**.

DRY : DRY o Don't Repeat Yourself implica que, cualquier funcionalidad existente en un programa debe existir de forma única en él , o lo que es lo mismo, no debemos tener bloques de código repetidos.

Objetivos:

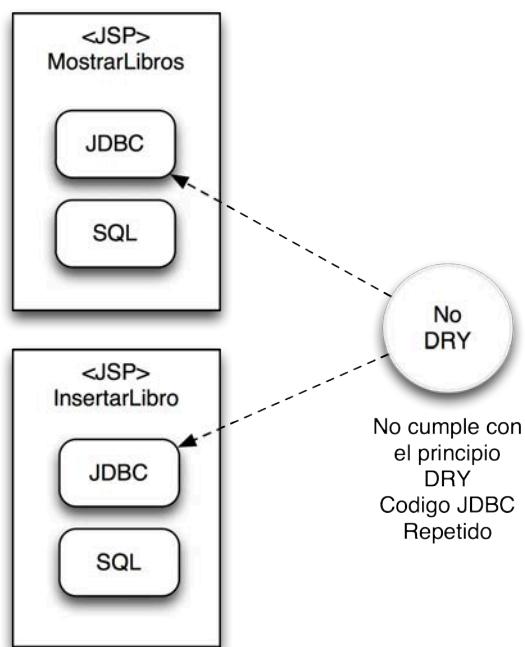
- Aplicar el principio DRY a las páginas que hemos construido hasta este momento,eliminando cualquier repetición de código.
- Avanzar en la construcción de la aplicación y añadir un desplegable de categorías. Revisar el uso del principio DRY apoyándonos en dicho desplegable

Tareas:

1. Aplicar el principio DRY y crear una nueva clase que ayude a eliminar el código repetido JDBC de las páginas.
2. Modificar las páginas JSP para que deleguen parte de su funcionalidad en la nueva clase.
3. Añadir desplegable de categorías para seguir profundizando en la aplicación del principio DRY.
4. El principio DRY y las consultas SQL.
5. El principio DRY métodos y parámetros
6. ResultSet vs listas de objetos
7. Uso de interfaces a nivel de libro
8. Cierre de conexiones usando reflection

1. Añadir nueva clase

En nuestro caso nos vamos a centrar en eliminar el código JDBC repetido en nuestras páginas (ver imagen).



Arquitectura Java

Para ello vamos a una nueva clase que se va a denominar DataBaseHelper y nos ayudara a gestionar mejor el código JDBC. Esta clase implementará los siguientes métodos:

public static ResultSet seleccionarRegistros(String sql): Método que se encargará de ejecutar una consulta SQL de selección y devolvernos un conjunto de registros con una estructura de ResultSet.

public static void modificarRegistro(String sql) :Método que se encargará de ejecutar cualquier consulta SQL de modificación (insert,update,delete etc) y devolvernos un entero con el numero de filas afectadas.

A continuación se muestra su código fuente.

Código 3.1: (DataBaseHelper.java)

```
package com.arquitecturajava;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DataBaseHelper {

    private static final String DRIVER = "com.mysql.jdbc.Driver";
    private static final String URL = "jdbc:mysql://localhost/arquitecturajava";
    private static final String USUARIO = "root";
    private static final String CLAVE = "java";

    public int modificarRegistro(String consultaSQL) {

        Connection conexion = null;
        Statement sentencia = null;
        int filasAfectadas = 0;

        try {

            Class.forName(DRIVER);
            conexion = DriverManager.getConnection(URL,
                                                USUARIO, CLAVE);

            sentencia = conexion.createStatement();
            filasAfectadas = sentencia.executeUpdate(consultaSQL);

        } catch (ClassNotFoundException e) {
```

```

        System.out.println("Error driver" + e.getMessage());
    } catch (SQLException e) {

        System.out.println("Error de SQL" + e.getMessage());
    } finally {

        if (sentencia != null) {

            try {sentencia.close();} catch (SQLException e) {}

        }
        if (conexion != null) {

            try {conexion.close();} catch (SQLException e) {}
        }
    }
    return filasAfectadas;
}
public ResultSet seleccionarRegistros(String consultaSQL) {

    Connection conexion = null;
    Statement sentencia = null;
    ResultSet filas = null;
    try {
        Class.forName(DRIVER);

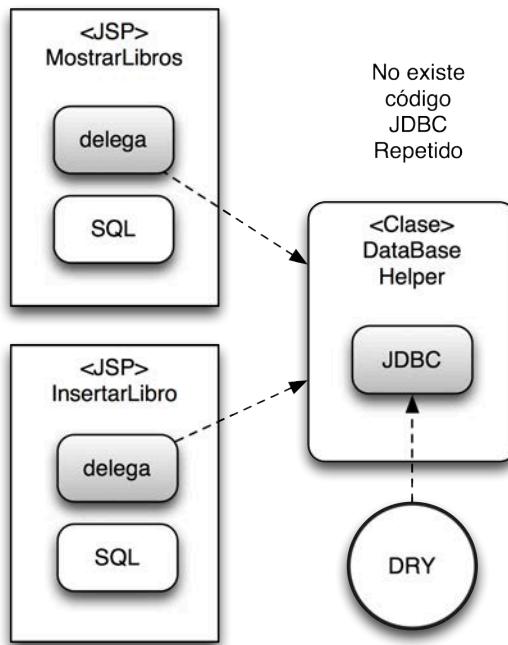
        conexion = DriverManager.getConnection(URL,
                                              USUARIO, CLAVE);

        sentencia = conexion.createStatement();

        filas = sentencia.executeQuery(consultaSQL);
    } catch (ClassNotFoundException e) {
        System.out.println("Error Driver" + e.getMessage());
    } catch (SQLException e) {
        System.out.println("Error de SQL " + e.getMessage());
    }
    return filas;
}
}

```

Una vez que esta clase esté construida, las páginas JSP de nuestra aplicación simplificarán significativamente el código que contienen y delegarán en la clase DataBaseHelper para realizar la mayor parte de las operaciones JDBC (ver imagen).



Acabamos de construir la clase DataBaseHelper, es momento de pasar a modificar las páginas JSP para que deleguen en ella.

2. Modificar páginas JSP

Tras la construcción de la nueva clase, modificaremos el código de las páginas que se ven afectadas InsertarLibro.jsp y MostrarLibros.jsp (ver código).

Código 3.2: (InsertarLibro.jsp)

```

<%@page import="com.arquitecturajava.DataBaseHelper"%>
<%
    String isbn= request.getParameter("isbn");
    String titulo= request.getParameter("titulo");
    String categoria= request.getParameter("categoria");
    //realizo la consulta usando el DBHelper y el código queda simplificado
    String consultaSQL= "insert into Libros (isbn,titulo,categoría) values ";
    consultaSQL+= "(" +isbn+ "," +titulo + "," +categoria+ ")";
    DataBaseHelper db= new DataBaseHelper();
    int filas=db.modificarRegistro(consultaSQL);
    response.sendRedirect("MostrarLibros.jsp");
%>
  
```

Como podemos ver la página InsertarLibros.jsp queda visiblemente simplificada .Sin embargo, no ocurre lo mismo con MostrarLibros.jsp, (ver imagen).

Código 3.3: (MostrarLibros.jsp)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>

<%@ page import="java.sql.ResultSet"%>
<%@ page import="java.sql.SQLException"%>
<%@page import="com.arquitecturajava.DataBaseHelper"%>
<!--omitimos cabecera-->
<body>
<%
    ResultSet rs = null;
    try {
        String consultaSQL = "select isbn,titulo,categoría from Libros";
        DataBaseHelper helper = new DataBaseHelper();
        rs = helper.seleccionarRegistros(consultaSQL);
        while (rs.next()) {
%
<%=rs.getString("isbn")%>
<%=rs.getString("titulo")%>
<%=rs.getString("categoría")%>
<br />
<%
    }
}
    catch (SQLException e) {

        System.out.println("Error accediendo a la base de datos"
            + e.getMessage());
    } finally {

        if (rs != null) {

            try {
                rs.close();
            } catch (SQLException e) {
                System.out.println("Error cerrando el resultset"
                    + e.getMessage());
            }
        }
%
<a href="FormularioInsertarLibro.jsp">Insertar Libro</a>
</body>
</html>
```

Para poder simplificar aún más el código de la página MostrarLibros.jsp precisaremos apoyarnos de nuevo en el principio DRY. Sin embargo, con tan poco código construido, es difícil ver cómo aplicarlo con mayor coherencia. Así pues vamos a añadir una nueva funcionalidad a nuestra aplicación para en adelante ver las cosas con mayor claridad.

3. Añadir Filtro por categoría

Vamos a añadir un desplegable de categorías a la página MostrarLibros.jsp que nos permitirá posteriormente realizar filtrados sobre la lista dependiendo de la categoría que elijamos (ver imagen).



Para conseguir que nuestra página de MostrarLibros muestre el desplegable tendremos que añadirle el siguiente bloque de código.

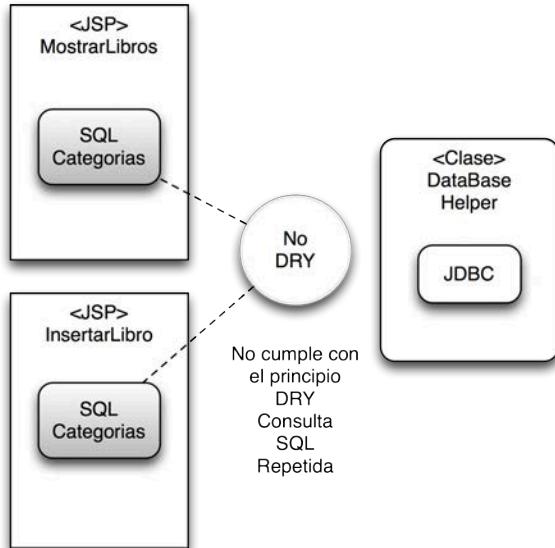
Código 3.4: (MostrarLibros.jsp)

```
<select name="categoria">
<option value="seleccionar">seleccionar</option>
<%
ResultSet rs=null;
try {
    String consultaSQL = "select distinct(categoría) from Libros";
    DataBaseHelper helper = new DataBaseHelper();
    rs=helper.seleccionarRegistros(consultaSQL);
    while(rs.next()) { %>
        <option value="<%=">%>rs.getString("categoria")%></option>
        <% } %>
    }
</select>
<br/>
```

En principio esta acción no es muy complicada, sin embargo si revisamos el formulario de inserción, nos daremos cuenta de que hay diseñarlo como desplegable (ver imagen).

The screenshot shows a web browser window with the URL http://localhost:8080/Applicacion03_2_JSPSQLRe. The page title is "Formulario alta libro". The form contains three text input fields labeled "ISBN:", "Titulo:", and "Categoria :". The "Categoria :" field has a dropdown menu open, showing the option "Programacion". Below the form is a single button labeled "Insertar".

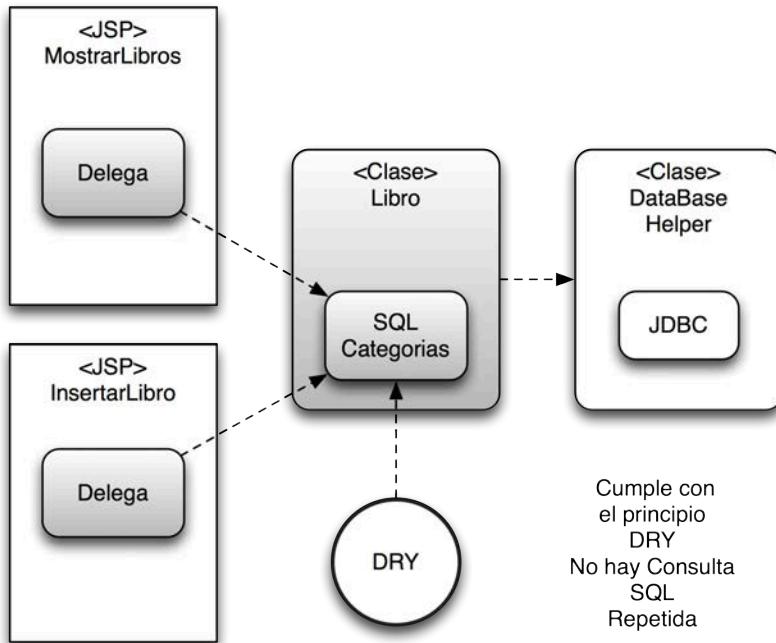
En la página de MostrarLibros.jsp utilizaremos el desplegable para filtrar la lista de libros por categoría, mientras que el formulario permitirá al usuario seleccionar las categorías de forma más sencilla, evitando cualquier tipo de error de escritura al estar prefijadas. Ahora bien, la consulta SQL está repetida en ambas páginas como se muestra en la siguiente imagen.



Por lo tanto nos encontramos en una situación similar a la anterior , sólo que en esta ocasión en vez de tener código JDBC repetido tenemos consultas SQL.

4. El principio DRY y las consultas SQL

Para evitar la repetición de SQLs a lo largo de las distintas páginas JSP (con los problemas de mantenimiento asociados) nos apoyaremos en la idea anterior y crearemos una nueva clase que se encargue de gestionar las consultas. Denominaremos a la nueva clase **Libro** : ésta almacenará todas las consultas que manejen los datos que la tabla Libro contiene. A continuación se muestra una imagen sobre cómo encaja la nueva clase en la estructura que ya teníamos definida.



Clarificado así donde encaja la nueva clase en nuestra arquitectura, vamos a mostrar su código fuente.

Código 3.5 : (Libro.java)

```
public class Libro {
    public static ResultSet buscarTodasLasCategorias() {
        String consultaSQL = "select distinct(categoría) from Libros";
        DataBaseHelper helper = new DataBaseHelper();
        ResultSet rs = helper.seleccionarRegistros(consultaSQL);
        return rs;
    }
    public static void insertar(String isbn, String título, String categoría) {
        String consultaSQL = "insert into Libros (isbn,título,categoría) values ";
        consultaSQL += "(" + isbn + "," + título + "," + categoría + ")";
        DataBaseHelper helper = new DataBaseHelper();
        helper.modificarRegistro(consultaSQL);
    }
    public static ResultSet buscarTodos() {
        String consultaSQL = "select isbn,título,categoría from Libros";
        DataBaseHelper helper = new DataBaseHelper();
        ResultSet rs = helper.seleccionarRegistros(consultaSQL);
        return rs;
    }
}
```

Arquitectura Java

Como podemos ver, la clase alberga tres métodos.

1. buscarTodos()
2. buscarPorCategoria()
3. insertar()

Cada uno de ellos se encarga de ejecutar una de las consultas necesarias para la tabla Libros. Una vez creada esta clase, las páginas JSP podrán delegar en ella y eliminar las consultas SQL de las páginas. A continuación se muestra la pagina “InsertarLibro.jsp” como ejemplo del uso de la clase Libro en las páginas JSP.

Código 3.6: (InsertarLibro.jsp)

```
<%@page import="com.arquitecturajava.Libro"%>
<% String isbn= request.getParameter("isbn");
   String titulo= request.getParameter("titulo");
   String categoria= request.getParameter("categoria");
   //realizo la consulta usando el DBHelper y el código queda simplificado
   Libro.insertar(isbn,titulo,categoria);
   response.sendRedirect("MostrarLibros.jsp");%>
```

Hemos eliminado las consultas SQL de nuestras páginas aplicando el principio DRY. Sin embargo es buen momento para volver a revisar este principio de cara a la nueva clase Libro que hemos construido. Si nos fijamos en la firma del método insertar,

Código 3.7: (Libro.java)

```
public static void insertar(String isbn, String título, String categoría)
```

nos podemos dar cuenta de que probablemente no sea el único método que incluya estos parámetros ya que el método modificar y el método borrar tendrán parámetros similares cuando se construyan:

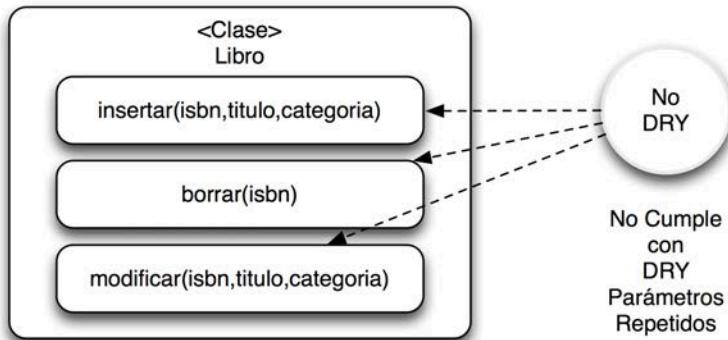
Código 3.8: (Libro.java)

```
public static void editar(String isbn, String título, String categoría)
public static void borrar(String isbn)
```

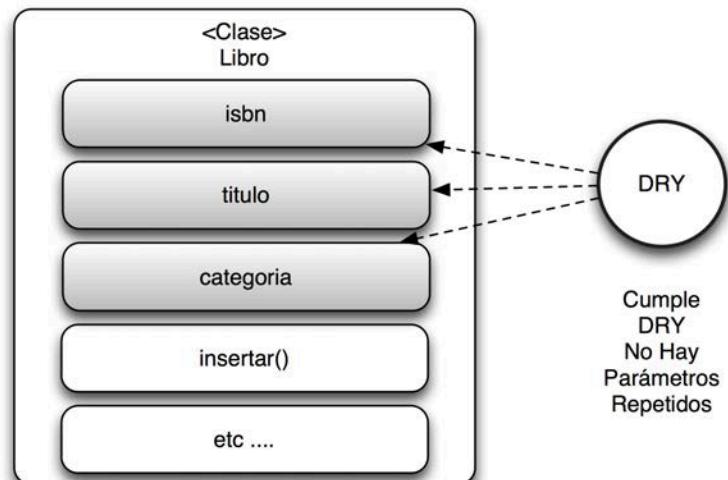
Parece claro que todos comparten un mismo grupo de parámetros. Es una ocasión perfecta para volver a aplicar el principio DRY y eliminar esta repetición de parámetros.

5. El principio DRY métodos y parametros

En programación orientada a objeto, una clase siempre se compone de propiedades y métodos o funciones. En nuestro caso, la clase Libro no dispone de propiedades y de ahí los problemas con todos los parámetros(ver imagen).



Por lo tanto, vamos a modificar nuestra clase para que disponga de las propiedades necesarias y los métodos se puedan apoyar en ellas evitando repeticiones innecesarias. La siguiente imagen clarifica la nueva estructura de la clase y como ésta se apoya en el principio DRY.



Una vez tenemos claro las modificaciones que afectaran a la clase, vamos a ver su código fuente.

Código 3.9: (Libro.java)

```
public class Libro {  
    private String isbn;  
    private String titulo;  
    private String categoria;  
    public String getIsbn() {  
        return isbn;  
    }  
    public void setIsbn(String isbn) {  
        this.isbn = isbn;  
    }  
    public String getTitulo() {  
        return titulo;  
    }  
    public void setTitulo(String titulo) {  
        this.titulo = titulo;  
    }  
    public String getCategoría() {  
        return categoria;  
    }  
    public void setCategoría(String categoria) {  
        this.categoría = categoria;  
    }  
    public Libro(String isbn, String titulo, String categoria) {  
        this.isbn = isbn;  
        this.titulo = titulo;  
        this.categoría = categoria;  
    }  
    public static ResultSet buscarTodasLasCategorías() {  
        String consultaSQL = "select distinct(categoría) from Libros";  
        DataBaseHelper helper = new DataBaseHelper();  
        ResultSet rs = helper.seleccionarRegistros(consultaSQL);  
        return rs;}  
    public void insertar() {  
        String consultaSQL = "insert into Libros (isbn,titulo,categoría) values ";  
        consultaSQL += "(" + this.isbn + "," + this.titulo + "," +  
                     + this.categoría + ")";  
        DataBaseHelper helper = new DataBaseHelper();  
        helper.modificarRegistro(consultaSQL);  
    }  
    public static ResultSet buscarTodos() {  
        String consultaSQL = "select isbn,titulo,categoría from Libros";  
        DataBaseHelper helper = new DataBaseHelper();  
        ResultSet rs = helper.seleccionarRegistros(consultaSQL);  
        return rs;  
    }}}
```

Visto así el código fuente, es momento de mostrar las modificaciones que debemos realizar a nivel de las páginas JSP. En este caso volveremos a apoyarnos en la página InsertarLibro.jsp.

Código 3.10: (InsertarLibro.jsp)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@page import="com.arquitecturajava.Libro"%>
<% String isbn= request.getParameter("isbn");
    String titulo= request.getParameter("titulo");
    String categoria= request.getParameter("categoria");
    Libro libro= new Libro(isbn,titulo,categoria);
    libro.insertar();
    response.sendRedirect("MostrarLibros.jsp");
%>
```

Vemos que el método insertar ya no necesita recibir ningún tipo de parámetro ya que obtiene toda la información necesaria de las propiedades que son asignadas a nivel del constructor del objeto. El avance ha sido significativo, sin embargo todavía quedan algunas cuestiones pendientes. Nos referimos a los métodos de búsqueda que en estos momentos devuelven ResultSets (ver código).

Código 3.11(Libro.java)

```
public ResultSet buscarTodos();
public ResultSet buscarCategorias();
```

6. ResultSets vs Listas de objetos

Para las páginas JSP sería mucho más sencillo a la hora de trabajar el recibir una lista de objetos (lista de Libros) ya que no tendría que encargarse cerrar el ResultSet y gestionar las posibles excepciones. Por lo tanto, nuestra tarea será modificar los métodos de búsqueda para que devuelvan estructuras de tipo lista .Vamos a ver como estos cambios se implementan a nivel de código.

.

Arquitectura Java

Código 3.12: (Libro.java)

```
public static ArrayList<Libro> buscarTodos() {  
    String consultaSQL = "select isbn,titulo,categoría from Libros";  
    DataBaseHelper helper = new DataBaseHelper();  
    ResultSet rs = helper.seleccionarRegistros(consultaSQL);  
    ArrayList<Libro> listaDeLibros= new ArrayList<Libro>();  
    try {  
        while (rs.next()) {  
            listaDeLibros.add(new Libro(rs.getString("isbn"),  
                                         rs.getString("titulo"),  
                                         rs.getString("categoria")));  
        }  
    } catch (SQLException e) {  
        System.out.println(e.getMessage());  
    }  
    return listaDeLibros;  
}
```

Una vez modificado el método `buscarTodos()` para que devuelva una lista de libros, la pagina `MostrarLibros.jsp` podrá apoyarse en el nuevo método a la hora de mostrar los datos.

Código 3.13: (Libro.java)

```
<%  
ArrayList<Libro> listaDeLibros=null;  
listaDeLibros=Libro.buscarTodos();  
for(Libro libro:listaDeLibros){ %>  
    <%=libro.getIsbn()%>  
    <%=libro.getTitulo()%>  
    <%=libro.getCategoría()%>  
    <br/>  
    <% }  
%>
```

Así, ya no existen referencias a JDBC en la pagina, el mismo cambio se puede aplicar al método de `buscarTodasLasCategorías()`. A continuación se muestra su código fuente.

Código 3.14: (Libro.java)

```

public static ArrayList<String> buscarTodasLasCategorias() {
    String consultaSQL = "select distinct(categoría) as categoria
                           from Libros";
    DataBaseHelper helper = new DataBaseHelper();
    ResultSet rs = helper.seleccionarRegistros(consultaSQL);
    ArrayList<String> listaDeCategorias= new ArrayList<String>();
    String categoria=null;
    try {
        while (rs.next()) {
            categoria= rs.getString("categoria");
            listaDeCategorias.add(categoria);
        }
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
    return listaDeCategorias;
}

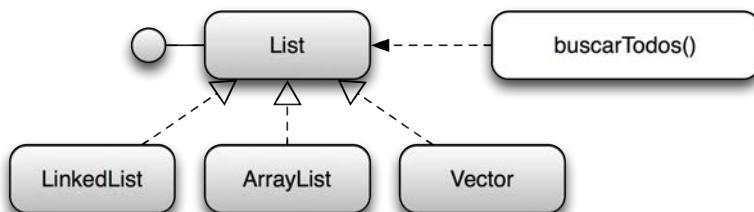
```

7. Uso de interfaces a nivel de Libro

En estos momentos disponemos de métodos de búsqueda de Libros o Categorías que devuelven un ArrayList de un tipo determinado <Libro> o <String> (ver código).



Si queremos tener una mayor flexibilidad en el parámetro de retorno, se debe cambiar el tipo ArrayList por el **Interface List** (ver código).



Arquitectura Java

A continuación se muestra como queda el método una vez modificado.

Código 3.15: (Libro.java)

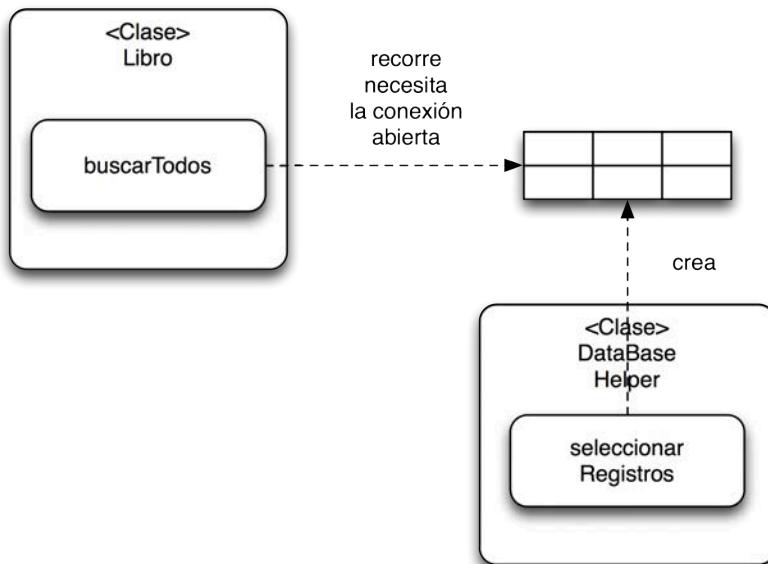
```
public static List<Libro> buscarTodos() {  
    String consultaSQL = "select isbn,titulo,categoría from Libros";  
    DataBaseHelper helper = new DataBaseHelper();  
    ResultSet rs = helper.seleccionarRegistros(consultaSQL);  
    List<Libro> listaDeLibros= new ArrayList<Libro>();  
    try {  
        while (rs.next()) {  
            listaDeLibros.add(new Libro(rs.getString("isbn"),  
                                         rs.getString("titulo"),  
                                         rs.getString("categoria")));  
        } catch (SQLException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
    return listaDeLibros;  
}
```

Nos puede parecer que hemos terminado de construir la clase libro y sí es cierto que no tenemos que modificar nada a nivel de estructura general. Ahora bien, si revisamos el código, veremos que el método *seleccionarRegistros* de la clase *DataBaseHelper* no cierra la conexión cuando nos devuelve el *ResultSet* (ver código).

Código 3.16: (DataBaseHelper.java)

```
public ResultSet seleccionarRegistros(String consultaSQL) {  
    Connection conexion = null;  
    Statement sentencia = null;  
    ResultSet filas = null;  
    try {  
        Class.forName(DRIVER);  
        conexion = DriverManager.getConnection(URL, USUARIO,  
                                              CLAVE);  
        sentencia = conexion.createStatement();  
        filas = sentencia.executeQuery(consultaSQL);  
    } catch (ClassNotFoundException e) {  
        System.out.println("Error de Driver" + e.getMessage());  
    } catch (SQLException e) {  
        System.out.println("Error de SQL " + e.getMessage());  
    }  
    return filas;  
}
```

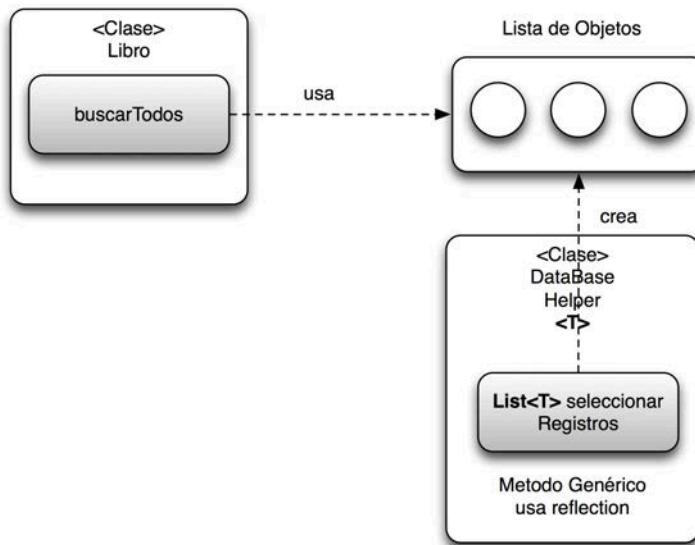
Ésto lamentablemente es necesario ahora ya que si cerramos la conexión a este nivel la clase Libro no podrá recorrer el ResultSet y cargar la lista de Libros (ver imagen).



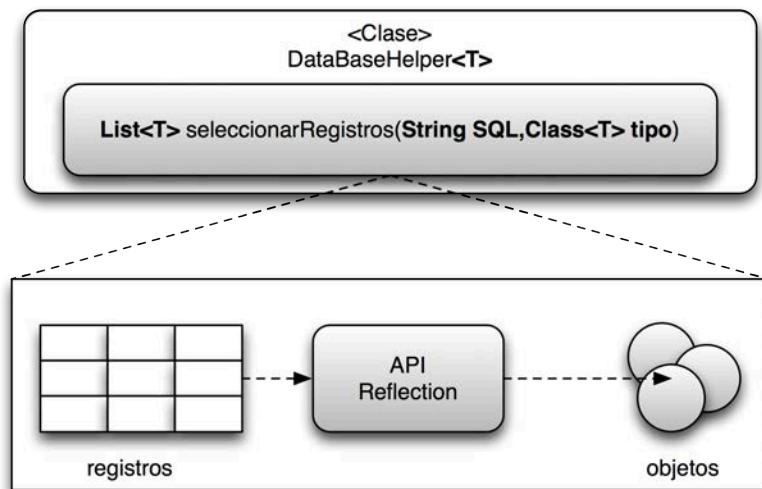
8. Cierre de conexiones y reflection.

Para poder solventar el problema de cierre de conexiones, vamos a pasar a modificar la clase DataBaseHelper: la convertiremos en una clase Genérica de tal forma que ella misma, dependiendo del tipo de clase que reciba como parámetro en los métodos de búsqueda, nos generará una lista de objetos de ese tipo concreto (ver imagen).

Arquitectura Java



Ahora bien, para que la clase DataBaseHelper pueda hacer uso de genéricos y sea capaz de construir distintos tipos de listas de objetos, hay que usar el API de reflection de Java: será la encargada de, a partir de una consultas SQL cualquiera, generar automáticamente una lista de objetos del tipo que nosotros solicitamos (ver imagen).



A continuación se muestra el código fuente que se encarga de realizar esta conversión **no lo vamos a explicar detalladamente, su importancia es muy relativa pues en**

capítulos posteriores utilizaremos un framework de persistencia para realizar este tipo de operaciones. Aun así, a continuación se muestra el código una vez modificado.

Código 3.17: (DataBaseHelper.java)

```
public class DataBaseHelper<T> {
    public List<T> seleccionarRegistros(String consultaSQL, Class clase) {
        Connection conexion = null;
        Statement sentencia = null;
        ResultSet filas = null;
        List<T> listaDeObjetos = new ArrayList<T>();
        try {
            Class.forName(DRIVER);
            conexion = DriverManager.getConnection(URL, USUARIO, CLAVE);
            sentencia = conexion.createStatement();
            filas = sentencia.executeQuery(consultaSQL);
            while (filas.next()) {
                T objeto = (T) Class.forName(clase.getName()).newInstance();
                Method[] metodos = objeto.getClass().getDeclaredMethods();
                for (int i=0;i<metodos.length;i++) {
                    if (metodos[i].getName().startsWith("set")) {
                        metodos[i].invoke(objeto,
                            filas.getString(metodos[i].getName().substring(3)));
                    }
                    if (objeto.getClass().getName().equals("java.lang.String")) {
                        objeto = (T) filas.getString(1);
                    }
                }
                listaDeObjetos.add(objeto);
            }
        } catch (Exception e) {
            System.out.println("Error al seleccionar registros" + e.getMessage());
        }
        finally {
            if (sentencia != null) {
                try {sentencia.close();} catch (SQLException e) {}
            }
            if (conexion != null) {
                try {conexion.close();} catch (SQLException e) {}
            }
        }
        return listaDeObjetos;
    }
}
```

Arquitectura Java

Comprobamos que ahora sí se realiza un correcto cierre de conexiones. Terminada esta operación, los métodos de la clase Libro que utilizan el DataBaseHelper<T> quedarán muy simplificados(ver código).

Código 3.18: (Libro.java)

```
public static List<Libro> buscarTodos() {
    String consultaSQL = "select isbn,titulo,categoría from Libros";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    List<Libro> listaDeLibros = helper.seleccionarRegistros(consultaSQL,
                                                             Libro.class);
    return listaDeLibros;
}
public static List<String> buscarTodasLasCategorias() {
    String consultaSQL = "select distinct(categoría) as categoria from Libros";
    DataBaseHelper<String> helper = new DataBaseHelper<String>();
    List<String> listaDeCategorias = helper.seleccionarRegistros(consultaSQL,
                                                               String.class);
    return listaDeCategorias;
}
```

Tras estos cambios, vamos a mostrar cómo queda el código final de la pagina MostrarLibros.jsp en lo que a las listas se refiere.

Código 3.19: (MostrarLibros.jsp)

```
<select name="categoria">
<% List<String> listaDeCategorias=null;
listaDeCategorias=Libro.buscarTodasLasCategorias();
for(String categoria:listaDeCategorias) { %>
    <option value="<%=>categoría%>"><=%=categoria%></option>
<% } %>
</select>
<%List<Libro> listaDeLibros=null;
listaDeLibros=Libro.buscarTodos();
for(Libro libro:listaDeLibros){ %>
    <=%=libro.getIsbn()%><=%=libro.getTitulo()%><=%=libro.getCategoría()%>
    <br/>
<% }
%>
<a href="FormularioInsertarLibro.jsp">Insertar Libro</a>
</body></html>
```

Resumen

En este capítulo nos hemos centrado en explicar el principio DRY y aplicarlo en nuestra aplicación. Como resultado, han aparecido dos nuevas clases Libro y DataBaseHelper. La arquitectura de nuestra aplicación ha aumentado su nivel de complejidad pero como contraprestación hemos reducido significativamente la engorrosa repetición de código , reduciéndose también el esfuerzo de desarrollo.

4.Editar, Borrar y Filtrar

En el capítulo anterior hemos introducido el principio DRY. En este capítulo nos centraremos en añadir una nueva funcionalidad a nuestra aplicación. Veremos las capacidades de edición, borrado y filtrado de Libros ayudándonos a asentar los conceptos aprendidos en el capítulo anterior.

Objetivos :

- Añadir la funcionalidad que nos permita borrar libros.
- Añadir funcionalidad que nos permita editar libros.
- Añadir un filtro por categorías a los libros.

Tareas:

1. Añadir un nuevo enlace de borrar a “MostrarLibros.jsp” .
2. Añadir el método borrar() a la clase Libro.
3. Añadir una nueva página “BorrarLibro.jsp” que se encargue de ejecutar la funcionalidad de borrar.
4. Añadir un nuevo enlace de edición a MostrarLibros.jsp.
5. Añadir el método buscarPorClave() que busque un libro por ISBN a la clase Libro.
6. Añadir una nueva página (FormularioEditarLibro.jsp) que muestre los datos del libro que deseamos modificar.
7. Añadir el método salvar() a la clase libro que guarda las modificaciones realizadas.
8. Añadir una nueva pagina SalvarLibro.jsp que se encargue de ejecutar la funcionalidad de salvar
9. Añadir el método de buscarPorCategoria() a la clase Libro

10. Modificar la pagina MostrarLibros.jsp para que se apoye en el método buscarPorCategoria.

1. Añadir enlace de borrar

Para que nuestra aplicación sea capaz de borrar libros, debemos añadir un nuevo enlace a la página MostrarLibros.jsp (ver código).



Para ello, necesitamos añadir una nueva línea de código a “MostrarLibros.jsp” como se muestra en el código.

Código 4.1: (MostrarLibro.jsp)

```
<%=libro.getIsbn()%>
<%=libro.getTitulo()%>
<%=libro.getCategoría()%>
<a href="BorrarLibro.jsp?isbn=<%=libro.getIsbn()%>">Borrar</a>
<br/>
```

2. Añadir método borrar

Al crear el enlace, es necesario añadir un nuevo método a la clase Libro que se encargue de borrar el registro de la tabla. A continuación se muestra el código de este método.

Código 4.2: (Libro.java)

```
public void borrar() {  
  
    String consultaSQL = "delete from Libros where isbn='"+ this.isbn+"';"  
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();  
    helper.modificarRegistro(consultaSQL);  
}
```

Una vez modificada la clase Libro, procederemos a construir la página BorrarLibro.jsp.

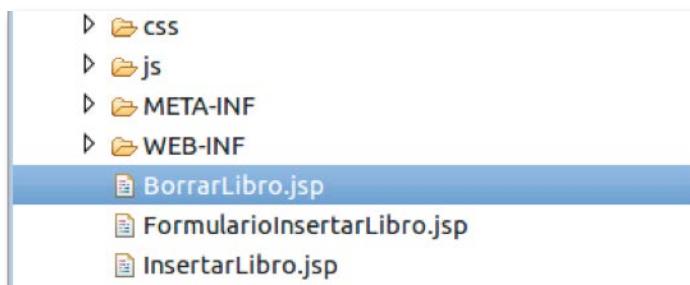
3. Añadir página BorrarLibro.jsp

La página que se encarga de borrar los libros es muy sencilla, simplemente obtiene el ISBN de la petición, se encarga de crear un objeto libro e invocar al método borrar. Veamos su código fuente.

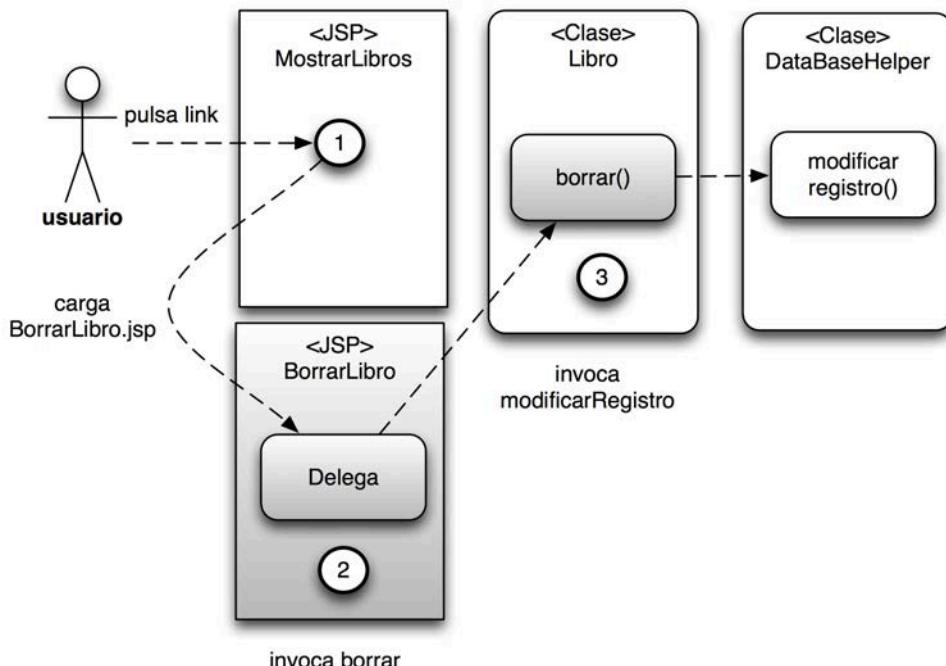
Código 4.3:(BorrarLibro.jsp)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"  
    pageEncoding="UTF-8"%>  
<%@page import="com.arquitecturajava.Libro"%>  
<%  
    String isbn= request.getParameter("isbn");  
    Libro libro= new Libro(isbn);  
    libro.borrar();  
    response.sendRedirect("MostrarLibros.jsp");  
%>
```

Tras todos estos pasos, dispondremos de una nueva página en nuestro proyecto (ver imagen).



Con estas primeras tres tareas complementamos el primer objetivo del capítulo: que nuestra aplicación sea capaz de borrar Libros. A continuación se muestra una imagen del flujo entre páginas y clases que ayuda a clarificar los pasos realizados.



Es el momento de pasar a las siguientes tareas que se encargaran del segundo objetivo: la edición de Libros.

4. Añadir link Edición

Como en el caso de borrar, la primera tarea será la de añadir un nuevo enlace a la pagina "MostrarLibros.jsp" que permita editar.



A continuación se muestran las modificaciones del código fuente.

Código 4.4:(MostrarLibro.jsp)

```
<%=libro.getIsbn()%>
<%=libro.getTitulo()%>
<%=libro.getCategoría()%>
<a href="BorrarLibro.jsp?isbn=<%=libro.getIsbn()%>">Borrar</a>
<a href="FormularioEditarLibro.jsp?isbn=<%=libro.getIsbn()%>">Editar</a>
<br/>
```

Añadido de esta forma el enlace, pasaremos a definir la siguiente tarea.

5. Método de búsqueda por clave

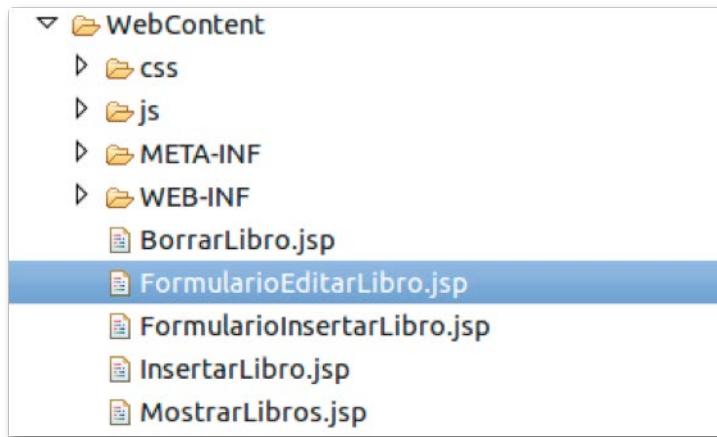
Para poder editar un registro necesitaremos primero seleccionarlo, para ello crearemos un nuevo método para la clase Libro que se denominara **buscarPorClave()**. Este método nos buscará un libro a través de su ISBN. A continuación se muestra el código fuente del nuevo método.

Código 4.5:(Libro.java)

```
public static Libro buscarPorClave (String isbn) {
    String consultaSQL = "select isbn,titulo,categoría from Libros where
                           isbn='"+isbn+"'";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    List<Libro> listaDeLibros =
        helper.seleccionarRegistros(consultaSQL,Libro.class);
    return listaDeLibros.get(0);
}
```

6. Añadir formulario de edición de código.

Una vez añadido el método de buscarPorClave, pasaremos a añadir una nueva página a nuestra aplicación denominada “FormularioEditarLibro.jsp” (ver imagen).



Finalizada así esta operación, podremos apoyarnos en el método que acabamos de construir para que la nueva página formularioEditarLibro.jsp muestre los datos del registro que nos interesen. Veamos como quedaría:



Una vez tenemos claro qué datos debe presentar la pagina vamos a ver su código fuente.

Arquitectura Java

Código 4.6: (MostrarLibro.jsp)

```
<%@ page import="java.util.List" %>
<%@page import="com.arquitecturajava.Libro"%>
<%Libro libro= Libro.buscarPorClave(request.getParameter("isbn"));%>
<!-- cabecera y javascript omitido-->
<body>
<form id="formularioEdicion" action="SalvarLibro.jsp">
<fieldset>
<legend>Formulario alta libro</legend>
<p><label for="isbn">ISBN:</label>
<input type="text" id="isbn" name="isbn" value="<%=libro.getIsbn()%>" /></p>
<p><label for="titulo">Titulo:</label><input type="text" id="titulo" name="titulo" value="<%=libro.getTitulo()%>" /></p>
<p><label for="categoria">Categoria :</label>
<select name="categoria">
<%
List<String> listaDeCategorias=null;
listaDeCategorias=Libro.buscarTodasLasCategorias();
for(String categoria:listaDeCategorias) {

    if (libro.getCategoría().equals(categoria)) { %>
<option value="<%=categoria%>" selected="selected">
<%=categoria%></option>
    <%}else{ %>
<option value="<%=categoria%>"><%=categoria%></option>
    <% }
    } %>
</select>
<br/>
</p>
<p><input type="submit" value="Salvar" /></p>
</fieldset>
</form>
</body>
</html>
```

Como vemos, el código es idéntico al formulario de Insertar con el que hemos trabajado en capítulos anteriores, con la única salvedad del siguiente bloque de código.

Código4.7: (MostrarLibro.jsp)

```
Libro libro=Libro.buscarPorClave(request.getParameter("isbn"))
```

Esta línea es la encargada de buscarnos el libro que deseamos editar y guardarlo en la variable “libro”. Una vez disponemos de los datos del libro simplemente los mostramos

a través de las etiquetas <input> y <select> usando scriptlet de jsp a la hora de asignar la propiedad value, como muestra el siguiente bloque de código.

Código 4.8: (MostrarLibro.jsp)

```
<input type="text" id="isbn" name="isbn" value="<%={libro.getIsbn()}%>" /></p>
```

7. Añadir método salvar

Una vez que hemos construido este formulario, podemos modificar los datos del libro que previamente hemos seleccionado. Realizados estos cambios, es necesario guardarlos en la base de datos. Para ello añadiremos a la clase Libro el siguiente método:

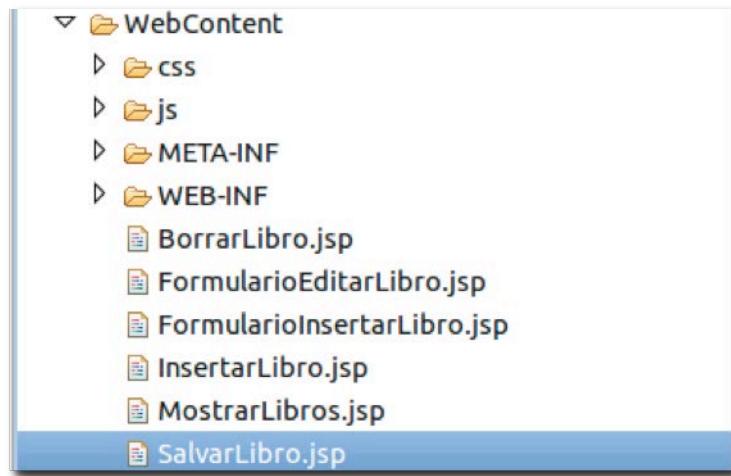
Código4. 9: (Libro.java)

```
public void salvar() {
    String consultaSQL = "update Libros set titulo='"+ this.titulo+"',
    categoria='"+ categoria+"' where isbn='"+ isbn+"'";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    helper.modificarRegistro(consultaSQL);
}
```

Por último, es preciso ligar el método salvar con el formulario de edición que acabamos de construir. La siguiente tarea abordara ésto.

8. Añadir pagina SalvarLibro.jsp

Para ligar el formulario y el método salvar añadiremos una nueva página a la aplicación denominada SalvarLibro.jsp (ver imagen).

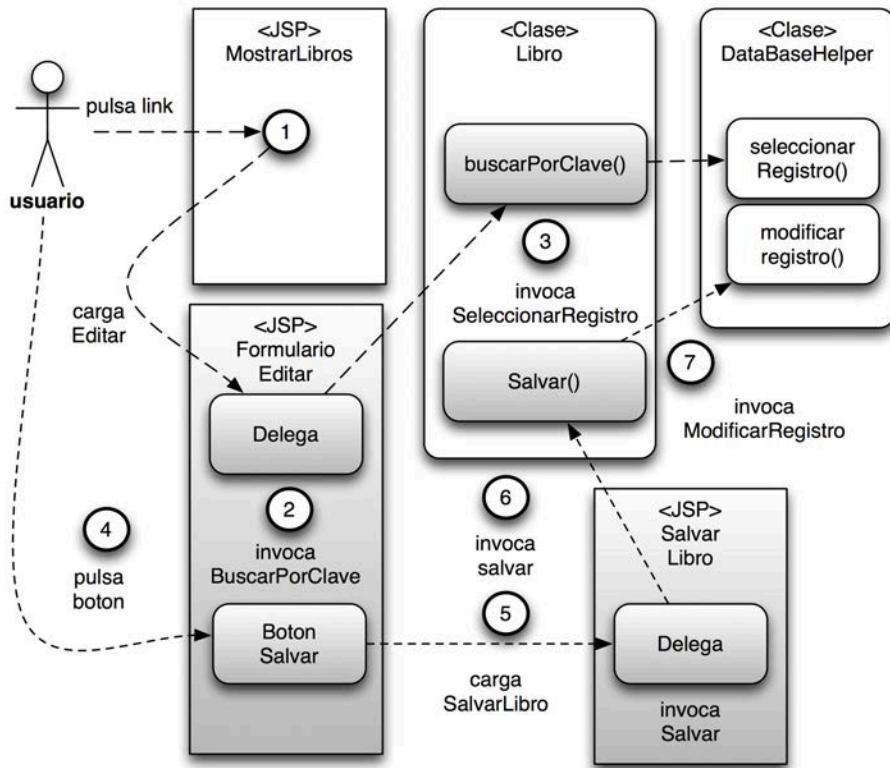


A continuación se muestra el código fuente de esta página

Código 4.9: (SalvarLibro.jsp)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@page import="com.arquitecturajava.Libro"%>
<% String isbn= request.getParameter("isbn");
    String titulo= request.getParameter("titulo");
    String categoria= request.getParameter("categoria");
    Libro libro= new Libro(isbn,titulo,categoria);
    libro.salvar();
    response.sendRedirect("MostrarLibros.jsp");%>
```

Una vez construidos todos los métodos, vamos a ver una imagen final aclaratoria.



9. Añadir método buscarPorCategoria.

Vamos a dar un último paso y añadir la funcionalidad de filtro por categoría a nuestra aplicación. Para ello, lo primero que debemos hacer es añadir un nuevo método a la clase Libro que nos devuelva a una lista de libros para una categoría concreta. Vamos a ver el código fuente de este método .

Código 4.10: (Libro.java)

```

public static List<Libro> buscarPorCategoria (String categoria) {
    String consultaSQL = "select isbn,titulo,categoria from Libros where
        categoria='"+ categoria+"'";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    List<Libro> listaDeLibros = helper.
        seleccionarRegistros(consultaSQL,Libro.class);
    return listaDeLibros;
}
  
```

Arquitectura Java

Una vez construido este nuevo método, hay que modificar la página MostrarLibros.jsp para que pueda invocarlo. Para ello añadiremos a esta página un nuevo botón de filtrado(ver imagen).



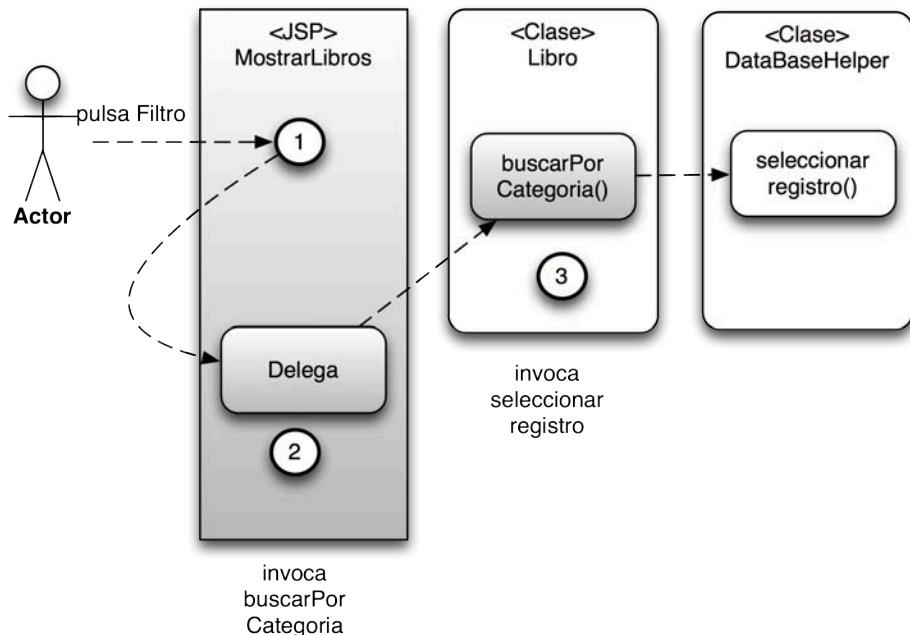
Vamos a mostrar a continuación el nuevo código de la página MostrarLibros.jsp orientado a las modificaciones susceptibles de realizarse a nivel del filtro.

Código 4.11: (MostrarLibro.jsp)

```
List<Libro> listaDeLibros=null;
if (request.getParameter("categoria")==null ||
    request.getParameter("categoria").equals("seleccionar")) {
    listaDeLibros=Libro.buscarTodos();
}
else {
    listaDeLibros=Libro.
        buscarPorCategoria(request.getParameter("categoria"));

}
for(Libro libro:listaDeLibros){ %>
    <%=libro.getIsbn()%>
    <%=libro.getTitulo()%>
    <%=libro.getCategoría()%>
    <a href="BorrarLibro.jsp?isbn=<%=libro.getIsbn()%>">Borrar</a>
    <a href="FormularioEditarLibro.jsp?isbn=<%=libro.getIsbn()%>">
        Editar</a>
    <br/>
    <% } %>
```

En este caso la operación no es muy complicada y simplemente selecciona el método `buscarTodos()` o `buscarPorCategoria()`, dependiendo de si pasamos parámetros o no. A continuación se muestra una imagen aclaratoria de la ligazón existente entre la página jsp y las distintas clases que hemos construido.



Llegados a este punto, hemos terminado el último objetivo del capítulo. A continuación se muestra el código fuente de la clase Libro que se ha visto sucesivamente modificada.

Arquitectura Java

Código 4.12: (Libro.java)

```
package com.arquitecturajava;
import java.util.List;
public class Libro {
    private String isbn;
    private String titulo;
    private String categoria;
    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getCategoria() {
        return categoria;
    }
    public void setCategoria(String categoria) {
        this.categoria = categoria;
    }
    public Libro(String isbn) {
        super();
        this.isbn = isbn;
    }
    public Libro() {
        super();
    }
    public Libro(String isbn, String titulo, String categoria) {
        super();
        this.isbn = isbn;
        this.titulo = titulo;
        this.categoria = categoria;
    }
    public static List<String> buscarTodasLasCategorias() {
        String consultaSQL = "select distinct(categoria) as categoria from
                                Libros";
        DataBaseHelper<String> helper = new DataBaseHelper<String>();
        List<String> listaDeCategorias = helper.seleccionarRegistros(
            consultaSQL, String.class);
        return listaDeCategorias;
    }
}
```

```

public void insertar() {

    String consultaSQL = "insert into Libros (isbn,titulo,categoría) values ";
    consultaSQL += "(" + this.isbn + "," + this.titulo + "," +
                   + this.categoría + ")";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    helper.modificarRegistro(consultaSQL);
}

public void borrar() {

    String consultaSQL = "delete from Libros where isbn=" + this.isbn
                           + "";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    helper.modificarRegistro(consultaSQL);
}

public void salvar() {

    String consultaSQL = "update Libros set titulo=" + this.titulo
                           + ", categoria=" + categoria + " where isbn=" + isbn + "";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    helper.modificarRegistro(consultaSQL);
}

public static List<Libro> buscarTodos() {

    String consultaSQL = "select isbn,titulo,categoría from Libros";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    List<Libro> listaDeLibros = helper.seleccionarRegistros(consultaSQL,
                                                             Libro.class);
    return listaDeLibros;
}

public static Libro buscarPorClave(String isbn) {
    String consultaSQL = "select isbn,titulo,categoría from Libros where
                           isbn=" + isbn + "";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    List<Libro> listaDeLibros = helper.seleccionarRegistros(consultaSQL,
                                                             Libro.class);
    return listaDeLibros.get(0);
}

public static List<Libro> buscarPorCategoria(String categoría) {

    String consultaSQL = "select isbn,titulo,categoría from Libros where
                           categoría=" + categoría + "";
    DataBaseHelper<Libro> helper = new DataBaseHelper<Libro>();
    List<Libro> listaDeLibros = helper.seleccionarRegistros(consultaSQL,
                                                             Libro.class);
    return listaDeLibros;
}
}

```

Resumen

En este capítulo hemos afianzado conceptos mientras añadíamos distintas funcionalidades a nuestra aplicación. Con ello hemos conseguido una clase Libro que se encarga de realizar todas las consultas SQL que la Tabla Libros necesita. A estas clases que mapean todas las operaciones sobre una tabla concreta se las denomina clases ActiveRecord, ya que implementan el patrón Active Record (definido en *Patterns of Enterprise Application Architecture*, de Martin Fowler) . La conclusión de este capítulo y del anterior es que aplicar principios de ingeniería de software como el principio DRY suele llevar a diseños basados en patrones.

5. Manejo de excepciones

En el capítulo anterior hemos añadido funcionalidad a la aplicación y construido una clase Libro que cumple con el patrón ActiveRecord. A partir de este capítulo no añadiremos mucha más funcionalidad sino que nos centraremos en ir aplicando distintos refactorings que permitan un mejor comportamiento de la aplicación. En este capítulo nos encargaremos de mejorar la gestión de excepciones .

Objetivos:

- Simplificar el manejo de excepciones de la aplicación

Tareas:

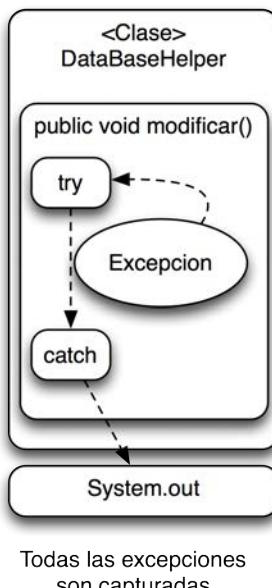
1. Revisión de el flujo de excepciones y uso de la cláusulas try/catch.
2. Manejo de la cláusula throw , throws y flujo de excepciones.
3. Creación y conversión de excepciones.
4. Anidamiento de excepciones.
5. Excepciones de Tipo RunTime.
6. Creación de una página jsp de error.
7. Modificación del fichero web.xml.

1. Flujo de excepciones y cláusulas catch

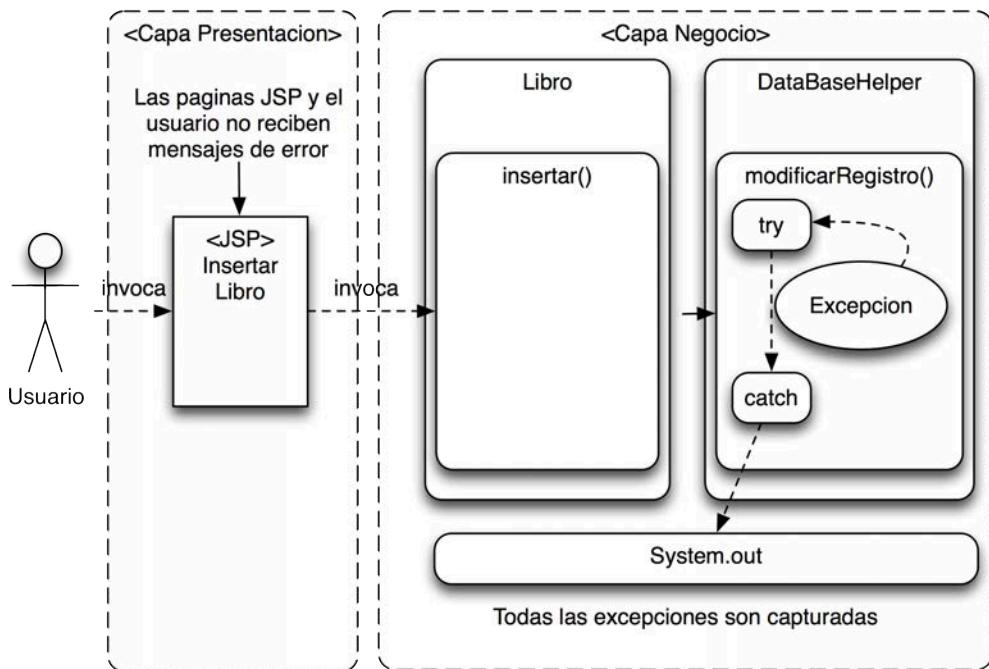
Como punto de partida, vamos a analizar cómo se gestionan las excepciones en la aplicación en estos momentos. Para ello, nos vamos a apoyar en los siguientes ficheros.

- Clase DataBaseHelper
- Clase Libro
- MostrarLibros.jsp

Si revisamos el código de manejo de excepciones que tiene la clase DataBaseHelper, veremos que se encarga de capturar todas las excepciones que se producen usando cláusulas try/catch e imprimiendo los errores por la consola (ver imagen).



Aunque esta gestión en un primer momento pueda parecernos correcta, presenta un problema: al capturar todos los mensajes de error, éstos nunca llegan a las páginas JSP y por lo tanto, un usuario habitual de la aplicación no recibirá ningún mensaje por muchos errores que se produzcan (ver imagen).



Si los usuarios no reciben ningún mensaje de error, no podrán avisar a los desarrolladores de los problemas que la aplicación les presenta; a lo sumo, podrán indicar que la aplicación no realiza una funcionalidad cualquiera, pero no nos podrán aportar más información. Si deseamos que los usuarios puedan aportarnos información adicional, es necesario que reciban unos mensajes de error claros. La siguiente tarea aborda esta problemática.

2. Manejo de las cláusulas throw, throws y flujo de excepciones.

Para conseguir que los usuarios puedan recibir información sobre los errores de la aplicación es esencial realizar las siguientes modificaciones.

1. En la clase DataBaseHelper una vez capturada la excepción y realizado log del mensaje de error, volveremos a lanzar las mismas excepciones utilizando la cláusula **throw**.
2. En la clase Libro modificaremos la firma de cada **método** para que no capture ninguna excepción y éstas puedan fluir hasta las páginas JSP.
3. En cada una de las páginas JSP añadiremos cláusulas try/catch para capturar las excepciones.

A continuación se muestra cómo queda el método modificarRegistro de la clase DataBaseHelper una vez aplicados los cambios .El resto de métodos será similar.

Código 5.1: (DataBaseHelper.java)

```
public int modificarRegistro(String consultaSQL) throws ClassNotFoundException,
SQLException {
try {
.....
} catch (SQLException e) {
    System.out.println("Error de SQL" + e.getMessage());
    throw e;
}
.....
```

Realizado el primer cambio, podemos ver cómo la cláusula throw relanza la excepción y cómo la clausula throws define qué conjunto de excepciones está soportada. Del mismo modo que hemos modificado la clase DataBaseHelper modificaremos a continuación los métodos de la clase Libro. Los cuáles únicamente marcan el método con las excepciones que puede producir.

Código 5.2: (Libro.java)

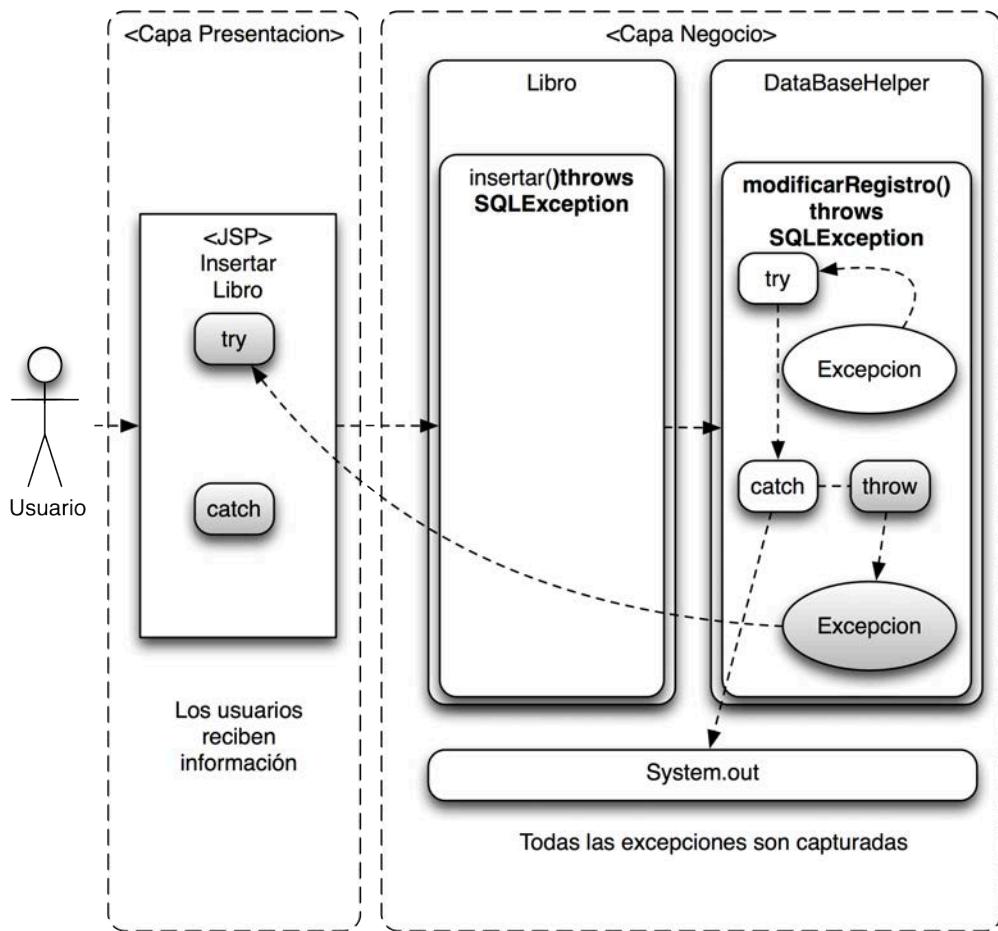
```
public void insertar() throws ClassNotFoundException, SQLException {
.....
}
```

El último paso es modificar nuestra página JSP de inserción y capturar las excepciones con un bloque try/catch .

Código5.3: (MostrarLibro.jsp)

```
try {
    LibroAR libro = new LibroAR(isbn, titulo, categoria);
    libro.insertar();
} catch (Exception e) {%
    <%=e.getMessage()%>
<%}%>
```

A continuación se muestra un diagrama aclaratorio de cómo todos estos cambios trabajan unidos para conseguir que los mensajes de error lleguen al usuario.



3. Creación y conversión de excepciones

Hemos avanzado en nuestra gestión de excepciones y ahora los usuarios reciben información sobre los errores que se producen. Sin embargo, si revisamos la firma del nuevo método insertar de la clase Libro, nos encontraremos con lo siguiente.

Código 5.4: (Libro.java)

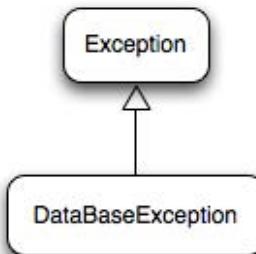
```
public void insertar() throws ClassNotFoundException, SQLException
```

Como podemos ver, la cláusula throws lanza varios tipos de excepciones. Estas cláusulas se repetirán método tras método por todo nuestro código y en algunos casos serán más complejas, por ejemplo en el método buscarTodos:

Código 5.5: (Libro.java)

```
public static List<Libro> buscarTodos() throws ClassNotFoundException, SQLException,
InstantiationException, IllegalAccessException, InvocationTargetException { }
```

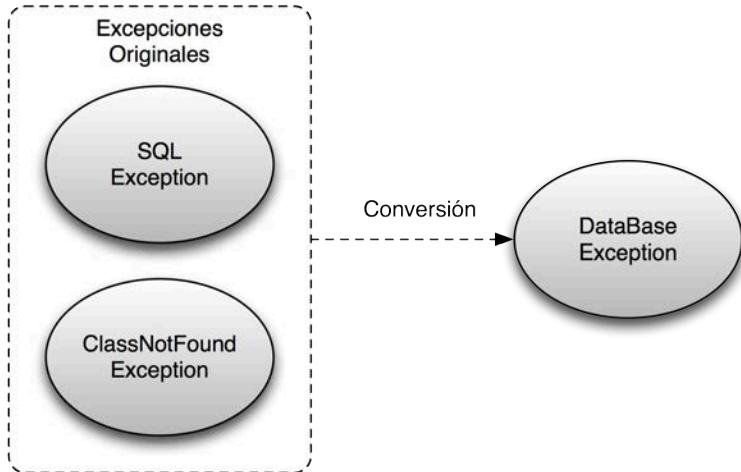
Para evitar este problema y simplificar el manejo de complicados conjuntos de excepciones, construiremos para nuestra aplicación una nueva excepción denominada “*DataBaseException*”. A continuación mostramos el diagrama UML que nos permite ver de qué clase hereda, así como su código fuente.



Código 5.6: (DataBaseException.java)

```
package com.arquitecturajava;
public class DataBaseException extends Exception {
    private static final long serialVersionUID = 1L;
    public DataBaseException() {
        super();
    }
    public DataBaseException(String message, Throwable cause) {
        super(message, cause);
    }
    public DataBaseException(String message) {
        super(message);
    }
    public DataBaseException(Throwable cause) {
        super(cause);
    }
}
```

Una vez creada esta nueva excepción, modificaremos el código fuente de nuestra clase *DataBaseHelper* para que convierta todas las excepciones que produce JDBC a nuestro nuevo tipo de excepción (ver imagen).



A continuación se muestra el código fuente modificado de uno de los métodos de la clase DataBaseHelper.

Código 5.7: (DataBaseHelper.java)

```

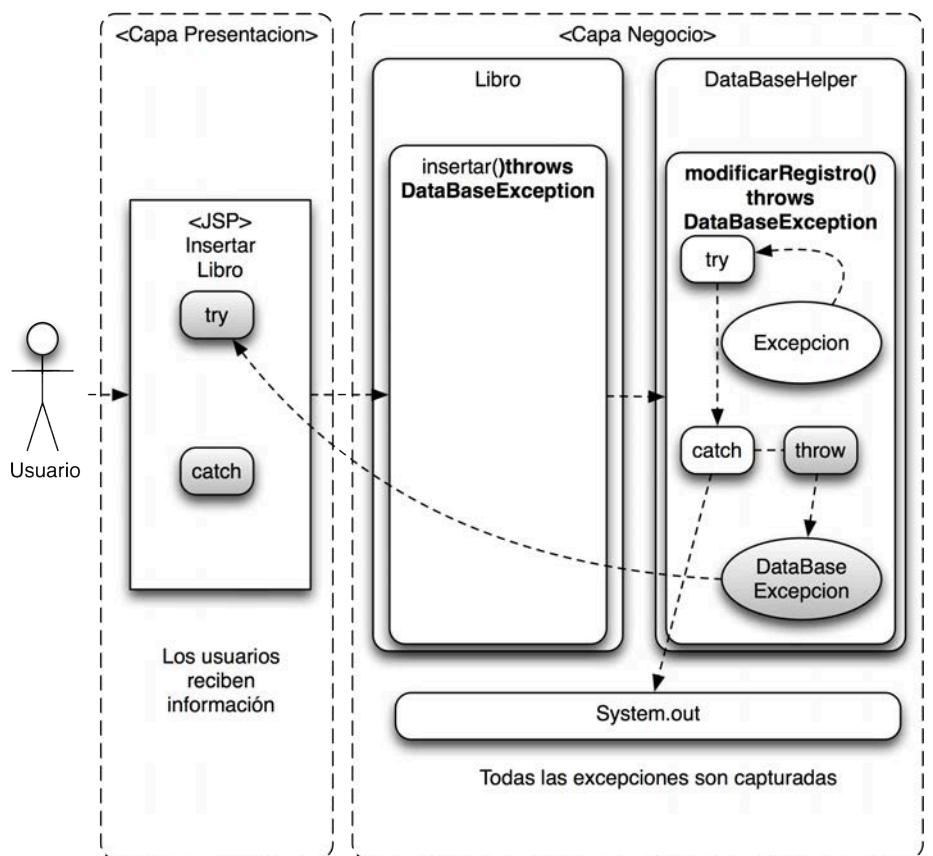
public int modificarRegistro(String consultaSQL) throws DataBaseException {
    Connection conexion = null;
    Statement sentencia = null;
    int filasAfectadas = 0;
    try {
        Class.forName(DRIVER);
        conexion = DriverManager.getConnection(URL, USUARIO,
        CLAVE);
        sentencia = conexion.createStatement();
        filasAfectadas = sentencia.executeUpdate(consultaSQL);
    } catch (ClassNotFoundException e) {
        System.out.println("Clase no encontrada" + e.getMessage());
        throw new DataBaseException("Clase no encontrada");
    } catch (SQLException e) {
        System.out.println("Error de SQL" + e.getMessage());
        throw new DataBaseException("Error de SQL");
    } finally {
        if (sentencia != null) {
            try {sentencia.close();} catch (SQLException e) {}
        }
        if (conexion != null) {
            try {conexion.close();} catch (SQLException e) {}
        }
    }
    return filasAfectadas;
}
    
```

Una vez realizadas estas modificaciones, los ficheros JSP simplemente realizaran una captura sencilla (ver código).

Código 5.8: (InsertarLibro.jsp)

```
<% try {
    String isbn= request.getParameter("isbn");
    String titulo= request.getParameter("titulo");
    String categoria= request.getParameter("categoria");
    Libro libro= new Libro(isbn,titulo,categoria);
    libro.insertar();
    response.sendRedirect("MostrarLibros.jsp");
} catch (DataBaseException e) {
    out.println(e.getMessage());
}
%>
```

A continuación se muestra la imagen que aglutina los últimos cambios.



4. Excepciones anidadas

Tras conseguir homogeneizar la gestión de excepciones creando nuestra nueva clase DataBaseException, nos damos cuenta que cuando se produce un error recibimos el mensaje de “Error de SQL” (ver imagen).

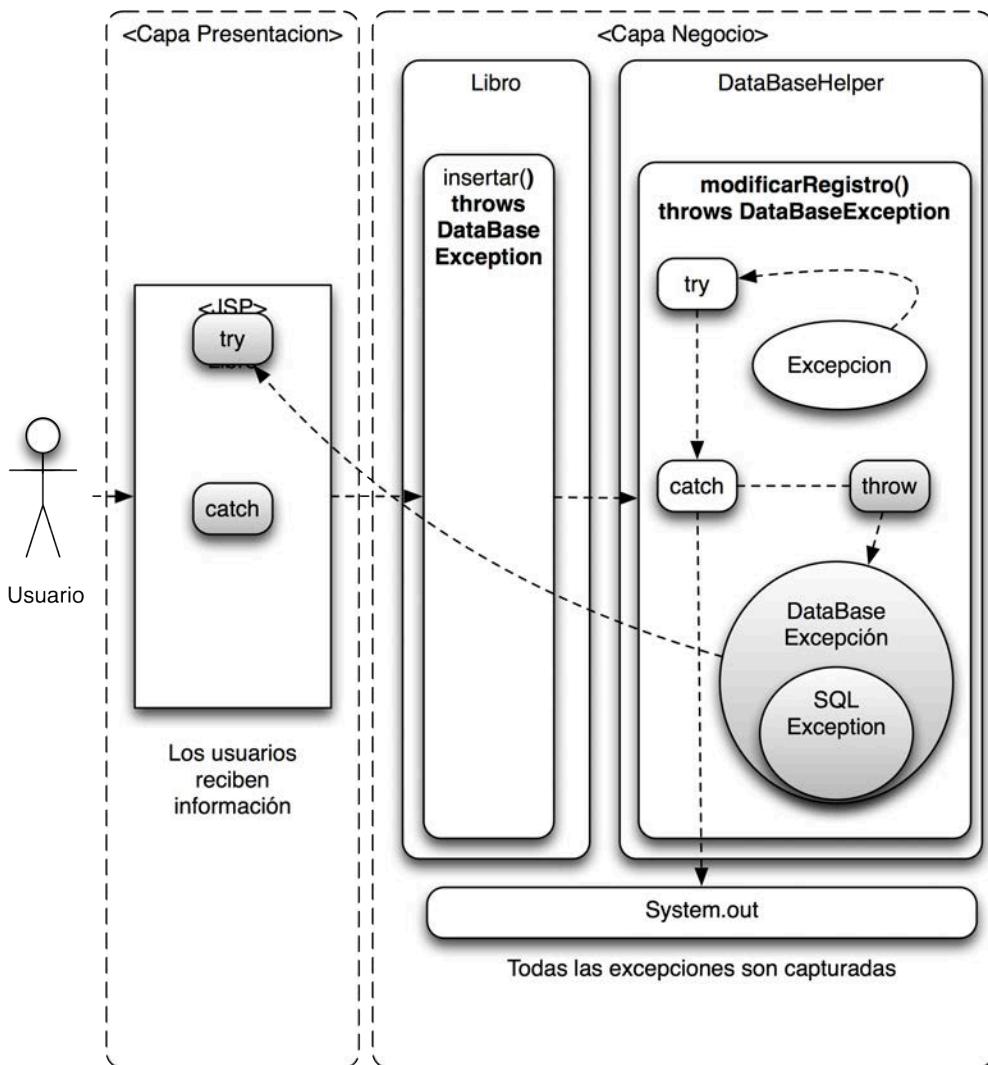


Sin embargo, aunque aparece el mensaje de error no es el mensaje de error original que se produjo. Ésto se debe a que hemos construido nuevas excepciones substituyendo las excepciones originales por éstas , perdiendo así los mensajes de error originales del api de java . De esta forma se complica sobremanera saber cuál es el error real que nuestra aplicación ha producido. Para poder acceder a la información original del error, debemos rediseñar nuestro sistema de excepciones para que soporte excepciones anidadas . Este tipo de excepciones es capaz de mostrar su mensaje de error, así como el mensaje de error original que las apis de java produjeron (ver código)

Código 5.9: (DataBaseHelper.java)

```
 } catch (ClassNotFoundException e) {  
     System.out.println("Error de acceso al driver" + e.getMessage());  
     throw new DataBaseException("Error de SQL",e);  
 }
```

A continuación se muestra un diagrama que contiene una excepción de tipo DataBaseException la cual tiene anidada una excepción de SQL para ayudar a clarificar.

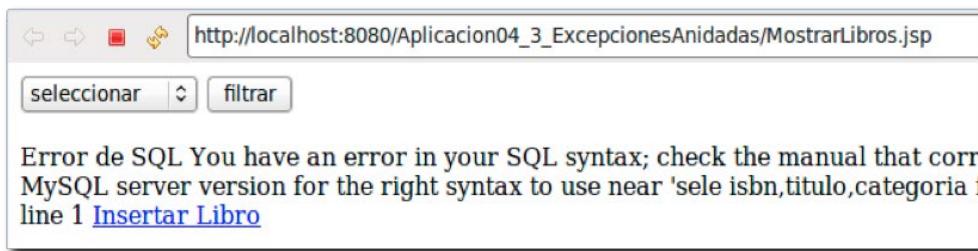


Una vez realizado este cambio, es necesario realizar una modificación en el fichero JSP que se encarga de capturar las excepciones que provienen de la capa de negocio, ya que debemos obtener la información de la excepción original que se produjo.

Código 5.10: (InsertarLibro.jsp)

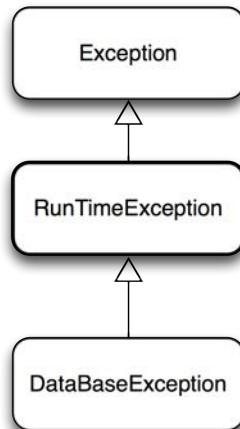
```
try {  
    LibroAR libro = new LibroAR(isbn, titulo, categoria);  
    libro.insertar();  
} catch (DataBaseException e) {  
    <%=e.getMessage()%>  
    <%=e.getException().getMessage()%>  
<%}%>
```

Las excepciones anidadas nos han ayudado a guardar la información original de los errores (ver imagen).

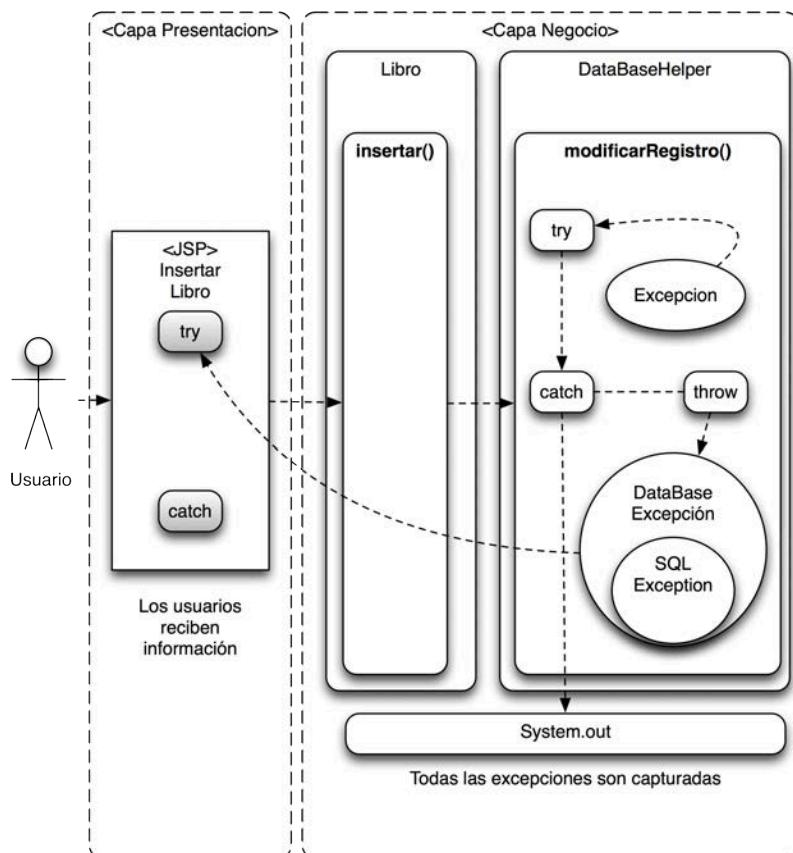


5. Excepciones RunTime

Queremos avanzar en la simplificación del manejo de excepciones, para ello modificaremos nuestra clase DataBaseException para que, en vez de que su clase padre sea la clase Excepción, lo sea la clase RunTimeException (ver imagen).



Al hacerlo de esta manera, podremos eliminar todas las cláusulas throws de nuestras clases ya que este tipo de excepciones java no obliga a capturarlas y pueden fluir libremente por todo el código como muestra la figura.



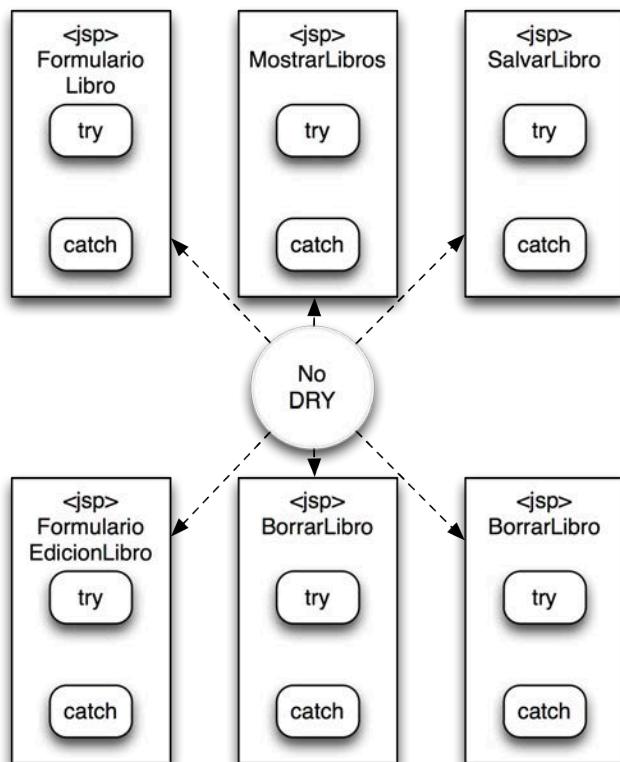
Una vez realizado este cambio, los métodos de nuestras clases quedarán definidos de la siguiente forma:

Código 5.11: (DataBaseHelper.jsp)

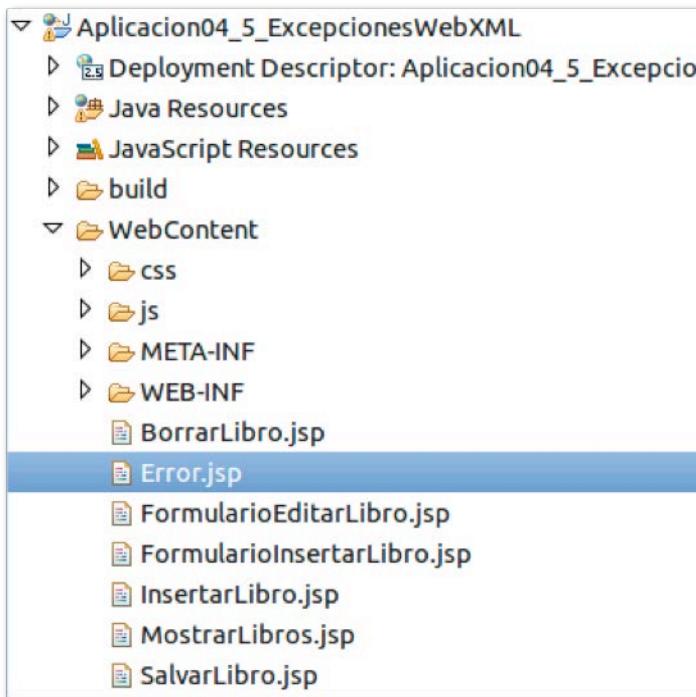
```
public int modificarRegistro(String consultaSQL) // no hay excepciones  
public void insertar() //no hay excepciones
```

6. Crear Pagina de Error

Parece que hemos terminado, sin embargo si revisamos el estado de nuestras páginas JSP respecto a la captura de excepciones, nos daremos cuenta de que todas las páginas contienen el mismo bloque try/catch (ver imagen).



Este es otro caso típico de repetición de código, donde podemos hacer uso del principio DRY para extraer la responsabilidad de la gestión de errores de la página JSP y centralizarla en otra nueva página denominada página de error, la cuál se encargará de gestionar los errores (ver imagen).



Vamos a ver a continuación el código fuente de esta página que tiene como peculiaridad el usar a nivel de directiva @page del atributo isErrorPage que la identifica como pagina de error.

Código 5.12: (Error.jsp)

```
<%@ page isErrorPage="true"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html><head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title></head>
<body>
Ha ocurrido un error en la aplicacion :<%=exception.getMessage()%>
Error Interno:<%=exception.getCause().getMessage()%>
</body></html>
```

Una vez creada la página JSP, debemos configurar nuestra aplicación web para que se apoye en esta nueva página de error. Para ello, modificaremos el fichero web.xml.

7. Modificar fichero web.xml

El fichero web.xml es el fichero de configuración de toda aplicación web java y se encuentra ubicado en la carpeta WEB-INF . A continuación se muestra el bloque de código que añadiremos a éste para dar de alta correctamente la página de error en la aplicación.

Código 5.13: (web.xml)

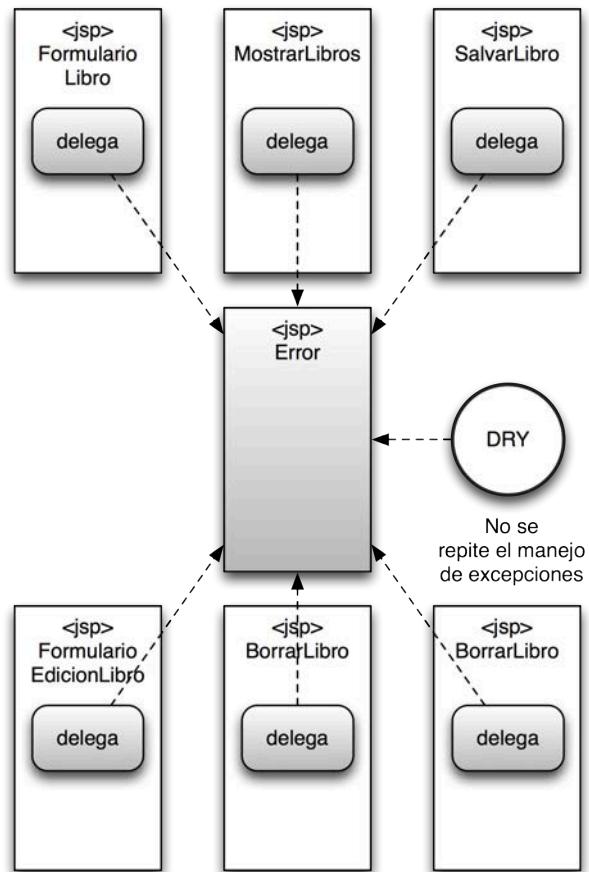
```
<web-app>
<error-page>
<exception-type>java.lang.RuntimeException</exception-type>
<location>/Error.jsp</location>
</error-page>
.....
</web-app>
```

Realizada esta operación, las páginas ya no contendrán bloques try/catch (ver código)

Código 5.14: (InsertarLibro.jsp)

```
<%
    String isbn = request.getParameter("isbn");
    String titulo = request.getParameter("titulo");
    String categoria = request.getParameter("categoria");
    LibroAR libro = new LibroAR(isbn, titulo, categoria);
    libro.insertar();
    response.sendRedirect("ListaLibros.jsp");
%>
```

Modificado el fichero web.xml y las páginas jsp, todas delegan ahora en la página de error.jsp a la hora de gestionar los distintos errores, cumpliendo con el principio DRY (ver imagen).



Resumen

En este capítulo nos hemos centrado en simplificar el manejo de excepciones de la aplicación, para lo cuál hemos realizado bastantes cambios, entre los que destacan:

1. Uso de excepciones de tipo `RunTimeException` para simplificar el flujo de excepciones de la aplicación.
2. Uso del principio DRY y construcción de una pagina de Error para centralizar la gestión de excepciones.

6.Log4J

En el capítulo anterior nos hemos encargado de simplificar la gestión de excepciones de nuestra aplicación. Sin embargo, siempre ligado a la gestión de excepciones se encuentra la gestión de los ficheros de log . Hasta este momento no hemos tratado el caso a fondo y nos hemos limitado imprimir mensajes por la consola con bloques de código del siguiente estilo.

Código 6.1: (DataBaseHelper.jsp)

```
System.out.println(e.getMessage())
```

Es evidente que esta forma de trabajar no es la adecuada cuando se trata de aplicaciones reales. Así pues, en este capítulo nos encargaremos de introducir un API de log y modificaremos nuestra aplicación para que haga uso de ella . En nuestro caso vamos a usar log4j que es un standard de facto en la plataforma Java. De esta manera, los objetivos del capítulo serán:

Objetivos:

- Introducción y manejo de log4j
- Actualización de la aplicación para que use log4j.

Tareas:

1. Instalación del api de log4j.
2. Introducción del api de log4j.
3. Construcción de un ejemplo sencillo.
4. Mensajes de error y niveles.

5. Manejo de Log4.properties.
6. Integración de log4j en la aplicación.

1. Instalación de log4j

Para poder trabajar con el api de log4j debemos obtener el producto de la web descargándolo de la siguiente url.

- <http://www.apache.org/dyn/closer.cgi/logging/log4j/1.2.16/apache-log4j-1.2.16.tar.gz>

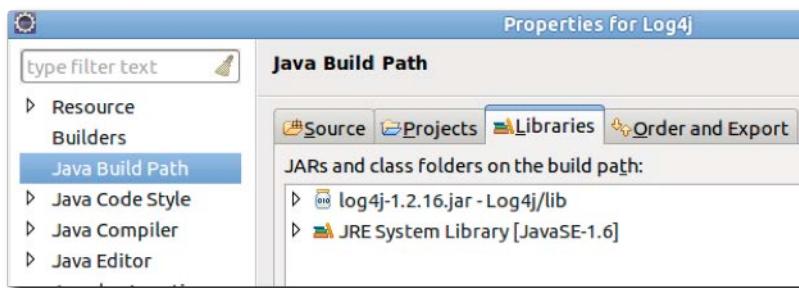
Una vez obtenido crearemos un nuevo proyecto **Java (No java Web)** en el eclipse, ya que lo primero que vamos a hacer es un conjunto de ejemplos sencillos para introducir al lector a este api. A continuación se muestra una imagen del proyecto.



Una vez creado el proyecto, añadiremos una carpeta **lib** a éste y ubicaremos en ella el fichero **log4j-1.2.16.jar**. El cual obtenemos del paquete tar.gz que acabamos de bajar. Ubicado el fichero en la carpeta, lo mapeamos como se muestra en la figura.



Realizada esta operación, únicamente deberemos referenciar la librería en nuestro proyecto para tenerla a nuestra disposición. Para ello, accederemos a las propiedades del proyecto y en el apartado de java build path mapearemos la librería (ver imagen).

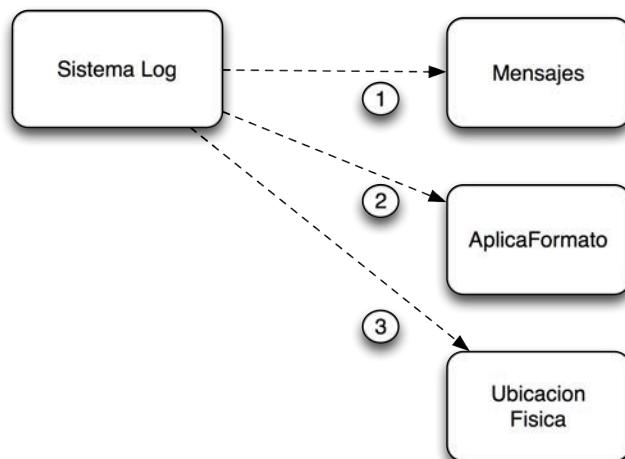


Una vez instalada la librería, podemos comenzar a utilizarla. La siguiente tarea nos introducirá en ella.

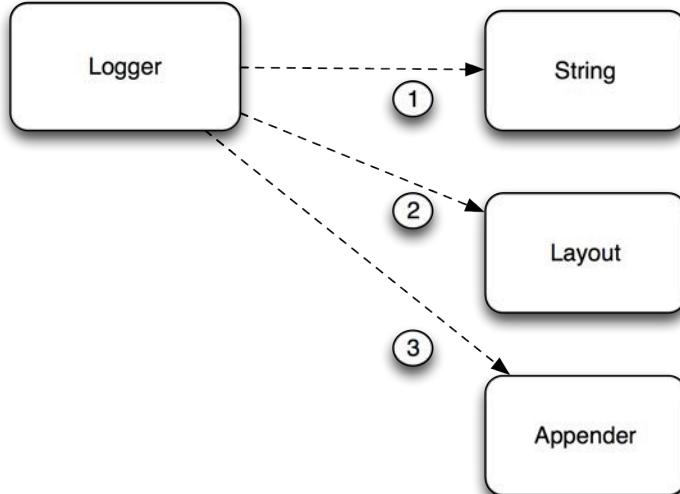
2. Introducción a log4j

El API de log4j no es complicada de manejar pero sí que es en muchos casos poco conocida por parte de los desarrolladores. A continuación realizaremos una introducción a sus conceptos principales.

Si nos preguntamos cómo funciona un sistema de log mucha gente nos responderá con algo como lo siguiente: un sistema de log genera mensajes con un formato determinado y los almacena en una ubicación para posterior análisis .La siguiente imagen muestra la idea a nivel conceptual.



En el caso del api de log4j la estructura es muy similar y está compuesta por el siguiente diagrama de clases, que sirve para paralelizar los conceptos anteriores.



Vamos a comentar a continuación cada una de estas clases.

- **String** : Clase del API de JSE que representa el concepto de mensaje (o cadena) a nivel del API de log4j.
- **Layout** : Clase del API de log4j que se encarga de seleccionar el formato en el cuál los mensajes son impresos.
- **Appender**: Clase del API de log4j que hace referencia al recurso en el cuál los mensajes de error han de ser escritos (Consola, Fichero, Base de Datos etc.). Se apoyará en la clase Layout para asignar un formato a los mensajes
- **Logger**: Clase del API de log4j que se encarga de la gestión de los mensajes de error y los imprime al appender o appenders que se le hayan asignado, utilizando el formato que se defina en el Layout.

Aunque las definiciones son sencillas, no hay nada mejor que un ejemplo concreto de código para clarificar como funciona Log4j .

3. Construcción de un ejemplo sencillo

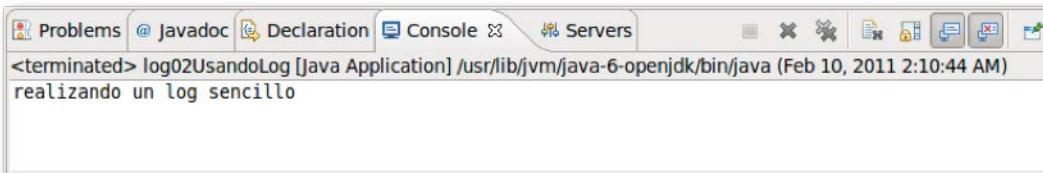
Vamos a construir un ejemplo elemental de uso de log4j. Para ello hemos construido el siguiente bloque de código que hace un uso del api y que pasaremos a analizar.

Arquitectura Java

Código 6.2: (Log2UsandoLog.java)

```
package com.arquitecturajava.logs;
import org.apache.log4j.ConsoleAppender;
import org.apache.log4j.Logger;
import org.apache.log4j.PatternLayout;
public class log02UsandoLog {
    public static void main(String[] args) {
        PatternLayout patron= new PatternLayout("%m %n");
        ConsoleAppender consola= new ConsoleAppender(patron);
        Logger log= Logger.getLogger("milog");
        log.addAppender(consola);
        log.info("realizando un log sencillo");
    }
}
```

Una vez construido el código vamos a ver que resultado muestra su ejecución



Como podemos ver, imprime una línea de texto por la consola. Vamos a continuación a analizar cada una de las líneas de nuestro código.

Código 6.3: (Log2UsandoLog.java)

```
PatternLayout patron= new PatternLayout("%m %n");
```

Esta clase define el formato en el cuál el mensaje ha de ser impreso. A continuación, se explican cada uno de los elementos % o patrones que se han utilizado.

- **%m** : Obliga a que se imprima el texto del mensaje
- **%n** : Obliga a que cada vez que se imprima un mensaje haya un retorno de carro y se salte de línea

Una vez definido el tipo de layout, es momento de construir el appender o recurso en el que se van a imprimir los distintos mensajes de error , el appender recibirá un layout como parámetro.

Código6.4: (Log2UsandoLog.java)

```
ConsoleAppender consola= new ConsoleAppender(patron);
```

En este caso hemos construido el appender más sencillo que imprime los mensajes a la consola cumpliendo con el patron especificado. El siguiente paso será la creación de un Logger u objeto de log que se encarga de gestionar estos mensajes. Este logger debe tener un nombre ("milog").

Código 6. 5: (Log2UsandoLog.java)

```
Logger log= Logger.getLogger("milog");
```

Una vez creado el objeto de log, es necesario asignarle el appender al cuál va a escribir en este caso la consola.

Código 6.6: (Log2UsandoLog.java)

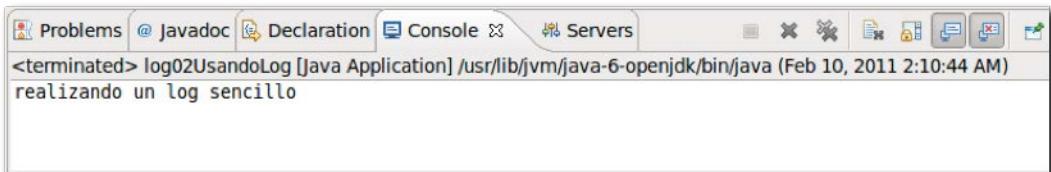
```
log.addAppender(consola);
```

Por ultimo, únicamente nos queda invocar al objeto de log y mandarle imprimir un mensaje a través del método info, que más adelante explicaremos detalladamente.

Código 6.7: (Log2UsandoLog.java)

```
log.info("realizando un log sencillo");
```

El resultado de todo este bloque de código es que, al ejecutar el programa, se imprimirá un mensaje por la consola (ver imagen).



Este sencillo ejemplo nos ha servido para introducir el funcionamiento del api de log4j. Ahora seguiremos avanzando y haremos especial hincapié en los niveles de mensajes de error.

4. Mensajes de error y niveles.

Acabamos de ver que el objeto de log se encarga de escribir un mensaje en la consola con el siguiente código.

Código 6.8: (Log2UsandoLog.java)

```
log.info("realizando un log sencillo");
```

Sin embargo, no queda muy claro por qué el método se denomina info y no por ejemplo message(), ya que simplemente imprime un mensaje. Ésto se debe a que el api de log4j soporta varios niveles de impresión de mensaje, dependiendo de si es un mensaje sin importancia o es un mensaje crítico para la aplicación. A continuación se muestra los distintos niveles de mensaje soportados y su uso.

Logger.fatal()	Imprime información de errores que hacen fallar completamente a la aplicación.
Logger.error()	Imprime información sobre errores que ha generado la aplicación pero no son fatales.
Logger.warn()	Imprime información sobre situaciones atípicas en la aplicación.
Logger.info()	Imprime información sobre el flujo de la aplicación.
Logger.debug()	Almacena información importante a nivel de debug de la aplicación

Una vez que tenemos claros los distintos tipos de mensaje que una aplicación puede imprimir apoyándose en el api de log4j ,vamos a crear un nuevo ejemplo con el siguiente bloque de código.

Código 6.9: (Log2Niveles.java)

```
public class Log02Niveles {
    public static void main(String[] args) {
        PatternLayout patron = new PatternLayout("%m %n");
        ConsoleAppender consola = new ConsoleAppender(patron);
        Logger log = Logger.getLogger("milog");
        log.addAppender(consola);
        log.fatal("realizando un log sencillo nivel FATAL");
        log.error("realizando un log sencillo nivel ERROR");
        log.warn("realizando un log sencillo nivel WARN");
        log.info("realizando un log sencillo nivel INFO");
        log.debug("realizando un log sencillo nivel DEBUG");
    }
}
```

Realizada esta operación, si ejecutamos el programa, podremos ver por consola el siguiente resultado:

```
<terminated> log02Niveles [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java
realizando un log sencillo nivel FATAL
realizando un log sencillo nivel ERROR
realizando un log sencillo nivel WARN
realizando un log sencillo nivel INFO
realizando un log sencillo nivel DEBUG
```

Todos los mensajes de error se han imprimido de forma correcta. Ahora bien, algo que no hemos comentado al introducir el api de log4j es que estos mensajes se encuentran relacionados de forma jerárquica (ver imagen).

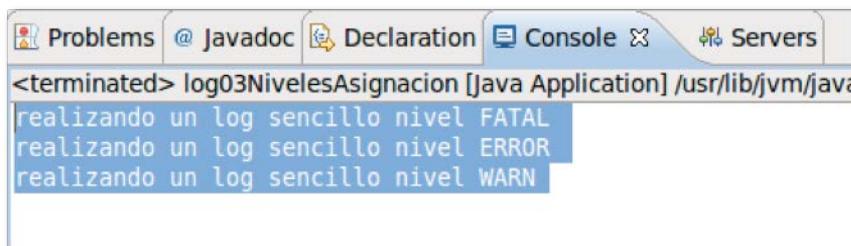


De manera que se puede configurar el sistema de log para que sólo se impriman por la consola los mensajes que pertenezcan a un elemento de la jerarquía o superior. Vamos a añadir el siguiente bloque de código a nuestro programa.

Código 6.11: (Log2Niveles.java)

```
logger.setLevel(Level.WARN);
```

Con este código solo serán impresos los de nivel WARN o superiores (ver imagen)



Ésto nos permitirá definir en nuestro código qué mensajes y con qué niveles se asignan. Seguidamente se muestra una función de ejemplo y algunos usos de los distintos niveles de log.

Código6.12: (Log2Niveles.java)

```

public static double calcularImporte(double importe) {
    Logger log = crearLog();
    log.info("entramos en la funcion");
    if (importe < 0) {
        log.warn("el importe no puede ser negativo");
        throw new RuntimeException("operacion no soportada con el
                                    importe");
    } else {
        if (importe > 0 && importe < 100) {
            log.info("compra con el 10 % de descuento");
            return importe * 0.90;
        } else {
            log.info("compra con el 20% de descuento");
            return importe * 0.80;
        }
    }
    log.debug("el importe calculado es :" + importe);
}

```

Es bastante probable que, cuando estemos desarrollando el programa, deseemos que el nivel de log se encuentre en modo DEBUG de tal forma que ayude a los desarrolladores a solventar cualquier tipo de problema a través de sus trazas de mensaje. Ahora bien, cuando el sistema se encuentre en producción, preferiremos que sólo se almacenen trazas de errores críticos y modificaremos el nivel de traza de log a error.

Hasta este momento hemos utilizado el API de Log4J a nivel de nuestro código tanto para realizar las tareas de log como para configurar el log en sí .En los entornos reales esta configuración de log donde uno asigna el formato del mensaje, los appenders y en nivel de log no se realiza a nivel de código, sino que se utiliza un fichero de propiedades sencillo denominado Log4j.properties. En la siguiente tarea se aborda el manejo de este fichero.

5. Manejo de Log4j.properties

Una vez que hemos entendido como funciona el api de log4j es mucho más sencillo hablar del fichero log4j.properties, que es el encargado de asignar los niveles de log ,appenders y layouts en una aplicación real de tal forma que esta configuración pueda ser modificada por un administrador sin tener que tocar el código de la aplicación Vamos a examinar detalladamente el siguiente fichero log4j.properties, que usaremos en nuestra aplicación.

Arquitectura Java

Código 6.13: (log4j.properties)

```
log4j.rootLogger = DEBUG, AppenderDeConsola
log4j.appender.AppenderDeConsola=org.apache.log4j.ConsoleAppender
log4j.appender.miAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.miAppender.layout.conversionPattern=%m%n
```

En este punto ya es mucho mas sencillo entender cada línea, aun así las vamos a ir comentando una por una.

Código 6.14: (log4j.properties)

```
log4j.rootLogger = DEBUG, AppenderDeConsola
```

La primera línea define el logger principal a nivel de DEBUG y le asigna un appender denominado “AppenderDeConsola” .Vamos a analizar la siguiente linea.

Código 6.15: (log4j.properties)

```
log4j.appender.AppenderDeConsola=org.apache.log4j.ConsoleAppender
```

Esta línea se encarga de definir qué tipo de appender usa nuestro log. En este caso se trata de un sencillo appender de consola que imprimirá los mensajes por la consola de la aplicación. Las últimas dos líneas se encargan de definir el tipo de layout que se va a utilizar (ver imagen).

Código 6.16 (log4j.properties)

```
log4j.appender.miAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.miAppender.layout.conversionPattern=%m%n
```

En nuestro caso el Layout es de tipo “Pattern” el cuál define un patrón a aplicar sobre la estructura del mensaje . En este caso %m es un patrón que indica que el texto del mensaje ha de ser impreso y por ultimo %n define que una vez impreso el texto del mensaje, se imprima un retorno de carro .Existen muchas más opciones de formato con este Layout, pero no las vamos a tratar aquí.

Lo último que nos queda por comentar es que el nombre log4j que aparece en cada una de las líneas hace referencia al nombre de nuestro logger “log4j” .Este nombre puede ser cambiado por cualquier otro, como por ejemplo.

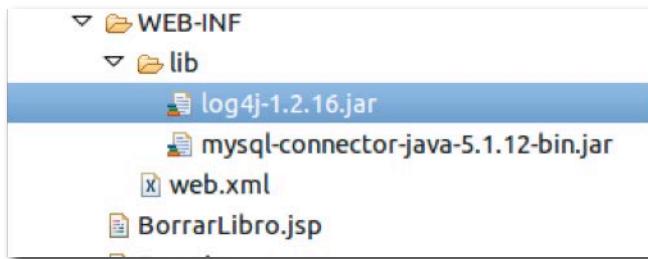
Código 6.17 (log4j.properties)

```
com.arquitecturajava
```

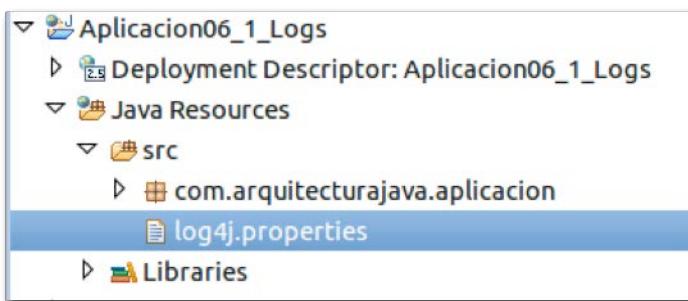
El cuál hace referencia al nombre de los packages de nuestro proyecto y será el nombre que se utilizará a nivel de la aplicación. La siguiente tarea se encarga de configurar Log4j en nuestra aplicación.

6. Uso de log4j en nuestra aplicación.

Una vez que hemos visto como funciona el api de log4j ,podremos hacer uso del mismo en nuestra aplicación. Para ello instalaremos la librería de log4j en el directorio **lib** de nuestra aplicación web (ver imagen).



Añadida la librería, hay que añadir el fichero de configuración de log4j con sus parametrizaciones . Este fichero se ubicará en la carpeta src de nuestro proyecto de eclipse (ver imagen).



Realizada esta operación, definimos el contenido del fichero que en este caso, por simplificar, será idéntico al anteriormente comentado.

Arquitectura Java

Código 6.18 (log4j.properties)

```
log4j.rootLogger = DEBUG, miappender
log4j.logger.milog = com.arquitecturajava.logs
log4j.appender.miappender=org.apache.log4j.ConsoleAppender
log4j.appender.miappender.layout=org.apache.log4j.PatternLayout
log4j.appender.miappender.layout.conversionPattern=%m%n
```

Una vez realizada esta operación, simplemente nos queda modificar nuestra clase DataBaseHelper para que se encargue de substituir las líneas de System.out por el uso del api de log (ver imagen).

Código 6.19 (DataBaseHelper.java)

```
public class DataBaseHelper<T> {
    private static final Logger log = Logger.getLogger(DataBaseHelper.class
        .getPackage().getName());
    public int modificarRegistro(String consultaSQL) {
        .....
        try {
            .....
        } catch (ClassNotFoundException e) {
            log.error("Error de acceso al driver" + e.getMessage());
            throw new DataBaseException("Error de SQL", e);
        } catch (SQLException e) {
            log.error("Error de SQL" + e.getMessage());
            throw new DataBaseException("Error de SQL", e);
        } finally {
            if (sentencia != null) {
                try {
                    sentencia.close();
                } catch (SQLException e) {
                    log.error("Error con la sentencia" +
                        e.getMessage());
                }
            }
            if (conexion != null) {
                try {
                    conexion.close();
                } catch (SQLException e) {
                    log.error("Error cerrando la conexion" +
                        e.getMessage());
                }
            }
        }
        return filasAfectadas;
    }
}
```

Resumen

En este capítulo hemos tomado contacto con el funcionamiento del API de log4j , el estándar de facto de la plataforma JEE a la hora de gestionar los mensajes de log de una aplicación . Lo hemos configurado para que funcione en nuestro proyecto.

7. El principio SRP y el modelo MVC

Hemos avanzando en estos capítulos en la construcción de nuestra aplicación, realizando diversos refactorings que han mejorado su diseño. En estos momentos disponemos de los siguientes componentes:

- **Páginas JSP** : Se encargan de crear la capa de presentación
- **Clases Java** : Se encargan de crear la capa de negocio y persistencia

El uso del principio **DRY** nos ha servido de guía en el diseño de nuestra aplicación. Aun así, quedan muchas cosas por hacer y el principio **DRY** no podrá ayudarnos en todas ellas. Es momento de seguir progresando en el diseño .Para ello, vamos a introducir un nuevo principio: el **principio SRP**.

SRP (Simple Responsibility Principle): toda clase o componente debe tener una única responsabilidad y todas sus funciones deben orientarse hacia ésta. Otra forma de enfocarlo es : una clase, al tener una única responsabilidad, sólo debe ser alterada a través de un cambio en dicha responsabilidad. Esta definición puede parecer confusa en un primer momento pero el concepto se irá aclarando paulatinamente con las explicaciones del capítulo.

Objetivos

Aplicar el principio SRP a nuestra aplicación

Tareas :

1. Responsabilidades de la aplicación y el principio SRP
2. Construcción de un servlet controlador
3. Mapeo de Servlet
4. Servlet Controlador y funcionalidad
5. Inserción en modelo MVC
6. Borrar en el modelo MVC
7. Editar en el modelo MVC
8. Filtrar en el modelo MVC

1. Responsabilidades de la aplicación y el principio SRP

Si revisamos las responsabilidades nuestro código fuente, podremos encontrarnos con lo siguiente :

1. **Paginas JSP (Capa Presentación)** : Responsabilidad de presentar información al usuario
2. **Clases Java (Capa de Persistencia)** : Responsabilidad de persistir los datos en la base de datos.

Parece que nuestras clases cumplen con los requisitos del principio SRP ya que tienen asociada una única responsabilidad. Ahora bien, si repasamos las distintas páginas JSP de nuestra aplicación, podremos encontrarnos con la página BorrarLibro.jsp y su código fuente:

Código 7.1 (BorrarLibro.jsp)

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@page import="com.arquitecturajava.Libro"%>
<%@ page import="com.arquitecturajava.DataBaseException" %>
<%
    String isbn= request.getParameter("isbn");
    Libro libro= new Libro(isbn);
    libro.borrar();
    response.sendRedirect("MostrarLibros.jsp");
%>
```

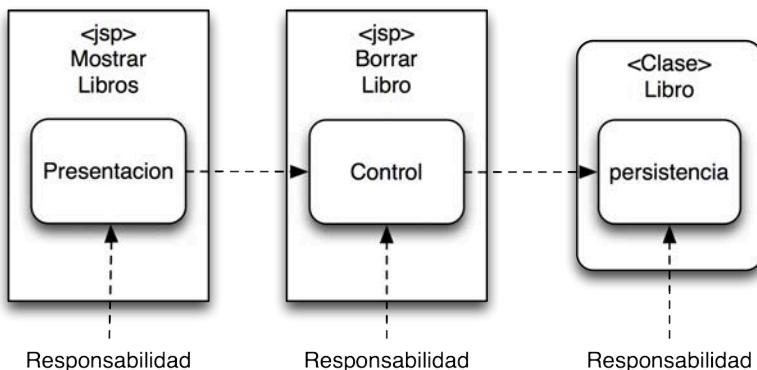
Arquitectura Java

En principio al ser una página JSP se debería encargar de presentar información al usuario. Sin embargo, es evidente que esta página no incluye ni una sola etiqueta html y no presenta ninguna información, por tanto, su responsabilidad no es presentar datos. Puede parecernos que se trata de una clase cuya responsabilidad es borrar datos, sin embargo, si revisamos el propio código de la página, veremos como la página instancia un objeto de tipo Libro y es éste el encargado de borrar el registro de la base de datos (ver código).

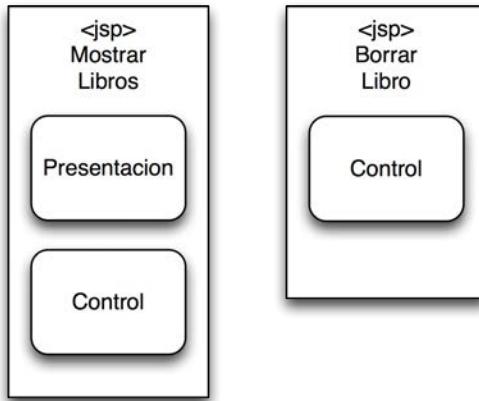
Código 7.2 (BorrarLibro.jsp)

```
Libro libro= new Libro(isbn);
libro.borrar();
```

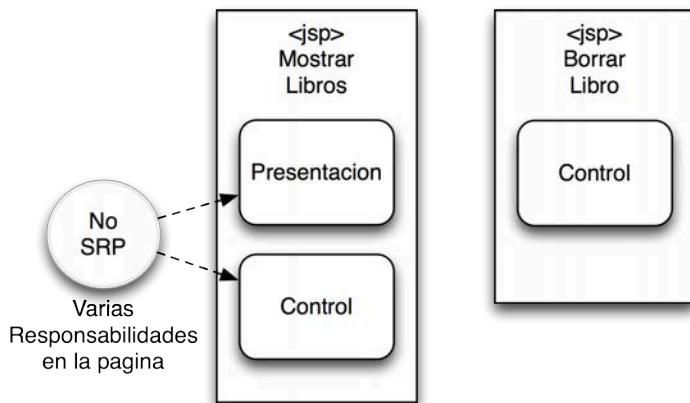
Así pues ¿cuál es realmente la responsabilidad de esta página? Aparentemente no tiene responsabilidades, pero si así fuera, podríamos eliminarla sin perjuicio para nuestra aplicación. Es evidente que si eliminamos la página, no podremos borrar los registros. Ésto se debe a que la pagina sí tiene una responsabilidad y ésta es la de **control**. Es decir ,se encarga de definir qué página se carga en cada momento, así como qué clase y qué método se invoca (ver código).



Así ,nos encontramos con una nueva responsabilidad dentro de nuestra aplicación: el control. Esta responsabilidad no es expresada únicamente por la página BorrarLibro.jsp. sino que prácticamente por todas las páginas de nuestra aplicación. Por ejemplo la página MostrarLibros.jsp se encarga de controlar a qué métodos de la clase Libro debemos invocar para presentar los datos. Así pues la responsabilidad de control se encuentra distribuida entre distintas páginas de diversas maneras (ver imagen).



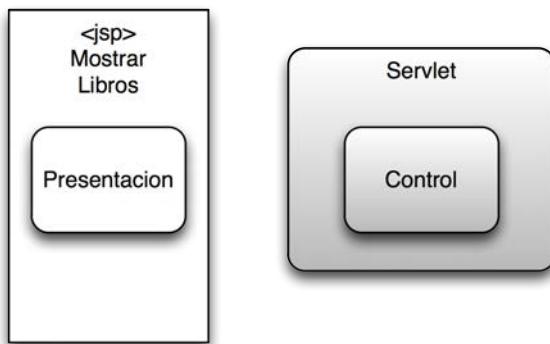
Ahora bien, en el caso de la página MostrarLibros.jsp nos encontramos ante un claro incumplimiento del principio SRP ya que la página en cuestión se hace cargo de dos responsabilidades claramente independientes: la de control y la de presentación (ver imagen).



Para solventar este problema, debemos extraer la responsabilidad de la página y ubicarla en otro componente; pero no podrá ser ninguno de los componentes que actualmente tenemos definidos ya que éstos ya tienen asociada una responsabilidad: las páginas la presentación y las clases la de persistencia. Para poder extraer esta responsabilidad tendremos que apoyarnos en un nuevo tipo de componente : un **Servlet**.

2. Construir un servlet controlador

Un servlet es una clase Java que es capaz de generar código html como lo hace una páginas JSP pero también es capaz de gestionar la comunicación entre varias de estas páginas. Esta capacidad es la que vamos a utilizar en nuestro caso a la hora de reubicar la responsabilidad de control (ver imagen).



Una vez clarificado que el servlet será el encargado de la responsabilidad de control, estaremos cumpliendo con el principio SRP en el cuál cada componente sólo tiene una única responsabilidad. Vamos a pasar ya a construir el servlet e implementar una primera versión de su método doGet() que será el encargado de la comunicación entre las distintas páginas.

Código 7.3 (ControladorLibros.java)

```
package com.arquitecturajava.aplicacion;
//omitimos imports
public class ControladorLibros extends HttpServlet {
    private static final long serialVersionUID = 1L;
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response) throws
ServletException, IOException {
        RequestDispatcher despachador = null;
        List<Libro> listaDeLibros = Libro.buscarTodos();
        List<String> listaDeCategorias = Libro.buscarTodasLasCategorias();
        request.setAttribute("listaDeLibros", listaDeLibros);
        request.setAttribute("listaDeCategorias", listaDeCategorias);
        despachador = request.getRequestDispatcher("MostrarLibros.jsp");
        despachador.forward(request, response);
    }
}
```

3. Mapeo de Servlet

Una vez creado el servlet, debemos añadir un mapeo de éste a través del fichero web.xml para dejarlo completamente configurado y poder acceder al mismo desde la siguiente url /ControladorLibros.do

Código 7.4 (web.xml)

```
< servlet >
< description ></ description >
< display-name > ControladorLibros </ display-name >
< servlet-name > ControladorLibros </ servlet-name >
< servlet-class > com.arquitecturajava.aplicacion.ControladorLibros </ servlet-class >
< / servlet >
< servlet-mapping >
< servlet-name > ControladorLibros </ servlet-name >
< url-pattern > /ControladorLibros </ url-pattern >
< / servlet-mapping >
```

4. Servlet Controlador y funcionalidad

Realizadas estas dos operaciones (construcción y mapeo), es momento de explicar cada una de las líneas que este servlet contiene, las cuáles se encargan de la funcionalidad de control . Vamos a comenzar con las dos primeras:

Código 7.5 (ControladorLibros.java)

```
List<Libro> listaDeLibros = Libro.buscarTodos();
List<String> listaDeCategorias = Libro.buscarTodasLasCategorias();
```

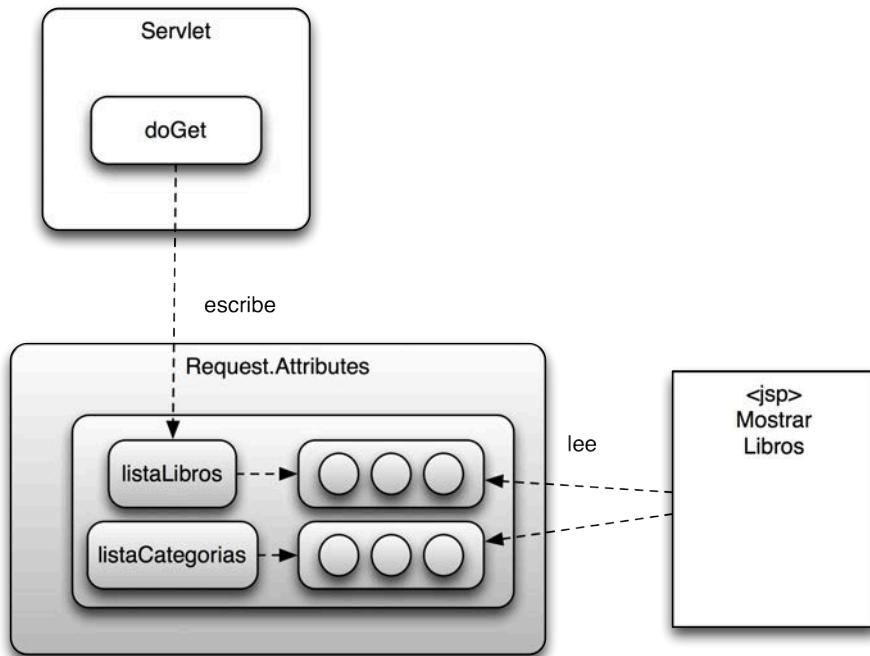
Estas líneas delegan en la capa de persistencia y cargan la información que la página MostrarLibros.jsp necesita en dos variables: listaDeLibros y listaDeCategorias, definiendo parte de la responsabilidad de control. Vamos a analizar las siguientes líneas.

Código 7.6 (ControladorLibros.java)

```
request.setAttribute("listaDeLibros", listaDeLibros);
request.setAttribute("listaDeCategorias", listaDeCategorias);
```

Arquitectura Java

Estas líneas se encargan de almacenar ambas listas en el objeto request, que es accesible tanto por el servlet controlador como por las páginas JSP y hace una función de intermediario (ver imagen).



Una vez el objeto request incluye ambas listas , usaremos el siguiente bloque de código para redirigir la petición hacia la página MostrarLibros.jsp.

Código 7.7 (ControladorLibros.java)

```
despachador = request.getRequestDispatcher("MostrarLibros.jsp");
despachador.forward(request, response);
```

Realizado este último paso , las páginas jsp no necesitan ya usar para nada la capa de persistencia y pueden obtener la información que necesitan mostrar del propio objeto request que sirve de intermediario (ver imagen).

Código 7.8 (MostrarLibro.jsp)

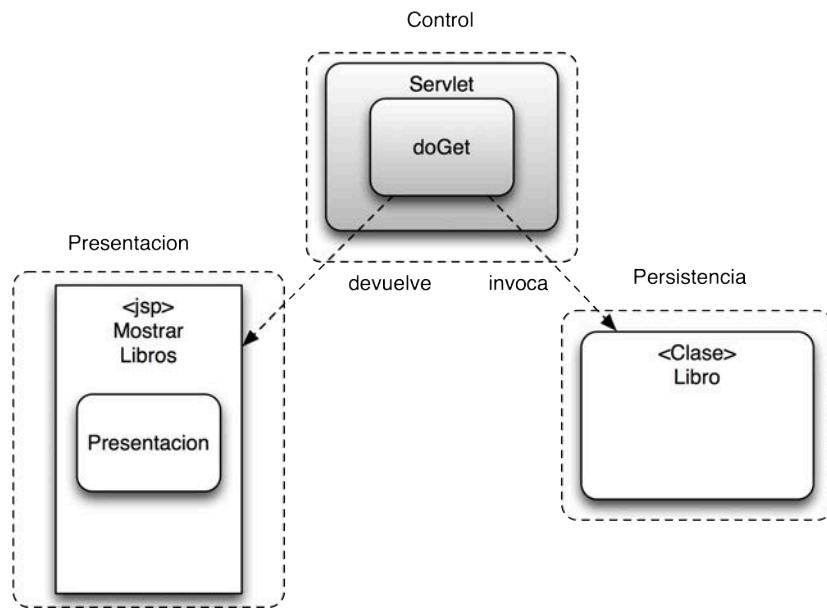
```

<select name="categoria">
<option value="seleccionar">seleccionar</option>
<%
    List<String> listaDeCategorias=null;
    listaDeCategorias=(List<String>)request.getAttribute("listaDeCategorias")
    for(String categoria:listaDeCategorias) {
        if (categoria.equals(request.getParameter("categoria"))){
            %
        <option value=<%=categoria%> " selected ><%=categoria%></option>
        <% } else { %>
        <option value=<%=categoria%>"><%=categoria%></option>
        <% } %>
    </select>
    <input type="submit" value="filtrar">
</form>
<br/>
<%
List<Libro> listaDeLibros
    =(List<Libro>)request.getAttribute("listaDeLibros");
for(Libro libro:listaDeLibros){ %>
<%=libro.getIsbn()%>
<%=libro.getTitulo()%>
<%=libro.getCategoría()%>
<a href="BorrarLibro.do?isbn=<%=libro.getIsbn()%>">Borrar</a>
<a href="FormularioEditarLibro.do?isbn=<%=libro.getIsbn()%>">Editar</a>
<br/>
<% }
%>

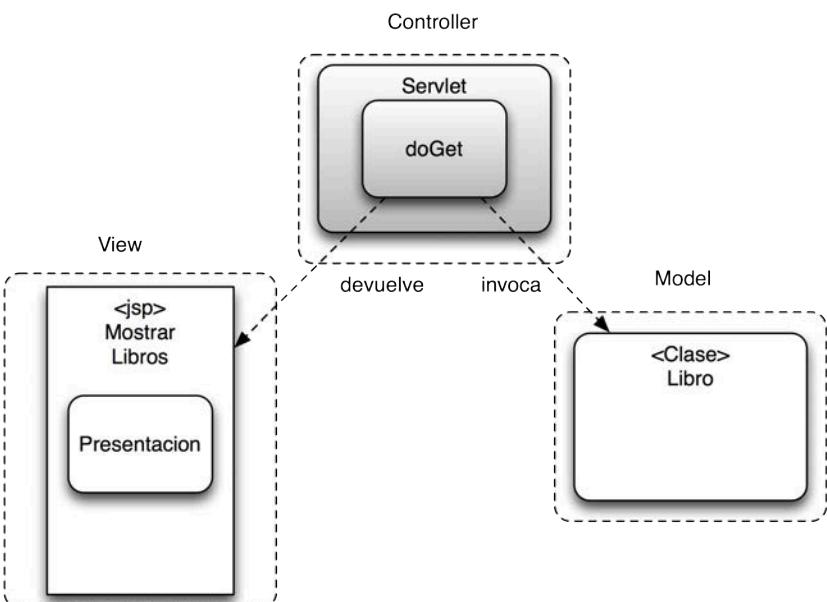
```

Una vez realizados estos refactorings hemos añadido un nuevo componente a nuestra aplicación que se encarga del control de la misma. A continuación se muestra un diagrama con la nueva arquitectura de la aplicación y las distintas responsabilidades.

Arquitectura Java



Hemos usado el principio SRP para separar responsabilidades en la aplicación y de forma indirecta hemos diseñado una arquitectura que cumple con el patrón MVC en el cual cada componente tiene un rol cerrado (ver imagen).



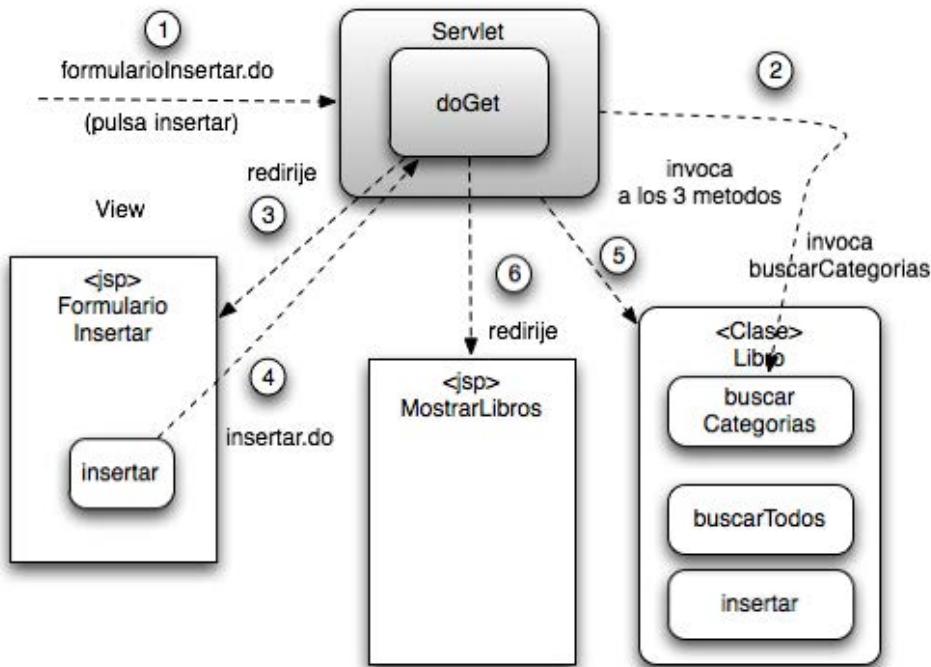
Vamos a seguir avanzando en la construcción de un Modelo MVC para nuestra aplicación.

4. Inserción con modelo MVC

En esta tarea vamos a modificar la aplicación para que tanto la parte de mostrar como la de insertar estén construidas sobre un modelo MVC. Para ello definiremos las siguientes tres URLs asociadas al controlador.

- MostrarLibros.do
- FormularioLibroInsertar.do
- InsertarLibro.do

Dependiendo de la URL que se solicite al Controlador por ejemplo ControladorLibros/MostrarLibros.do el controlador realizará una operación u otra. Para clarificar el funcionamiento del controlador vamos a mostrar un diagrama con el flujo de trabajo cuando el usuario pulsa el botón de insertar en la página MostrarLibros.jsp



Vamos a comentar el diagrama paso a paso para clarificar del todo cuál es la lógica que define la funcionalidad de mostrar y insertar libros.

1. Pulsamos en el botón de insertar libro y solicitamos la siguiente url al controlador **FormularioLibroInsertar.do**
2. El controlador invoca a la clase Libro y carga la lista de categorías

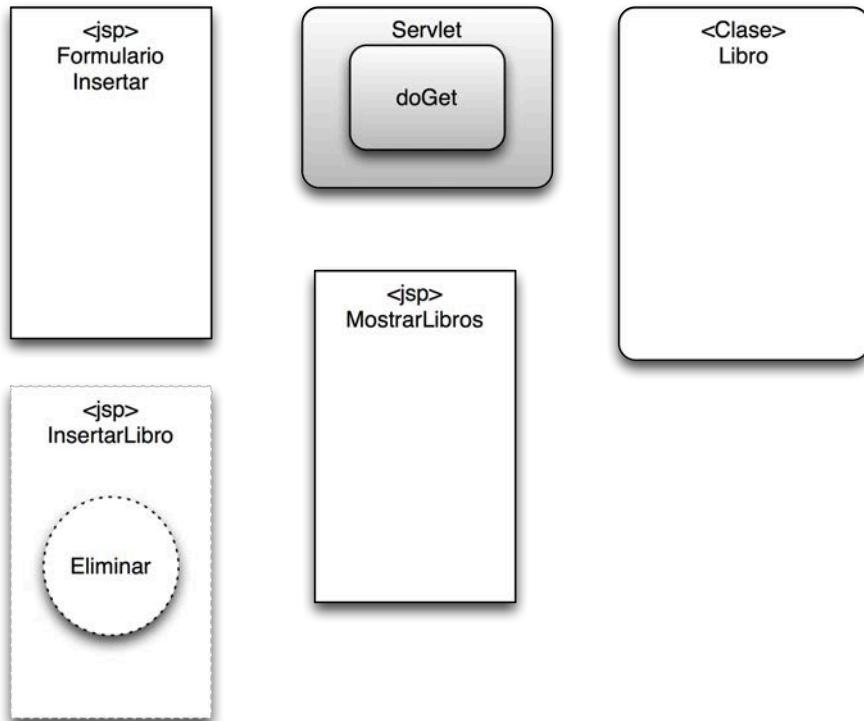
3. El controlador nos redirige a la pagina FormularioLibro.jsp y muestra la lista de categorías cargada.
4. Rellenamos el formulario e invocamos al controlador pasando como url **insertarLibro.do**
5. El controlador invoca la la clase Libro e inserta los datos en la base de datos. Una vez realizada esta operación, invoca los métodos de buscarTodos y buscarCategorias
6. El controlador nos redirige a MostrarLibros.do que carga la pagina MostrarLibros.jsp con los datos que acabamos de cargar

Una vez aclaradas cuáles son las distintas peticiones que realizaremos en esta tarea al servlet Controlador, vamos a ver el código fuente del método doGet() modificado para incluir las distintas opciones:

Código 7. 9 (ControladorLibros.java)

```
RequestDispatcher despachador = null;
if (request.getServletPath().equals("/MostrarLibros.do")) {
    List<Libro> listaDeLibros = Libro.buscarTodos();
    List<String> listaDeCategorias =
        Libro.buscarTodasLasCategorias();
    request.setAttribute("listaDeLibros", listaDeLibros);
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    despachador = request.getRequestDispatcher("MostrarLibros.jsp");
} elseif (request.getServletPath().equals("/FormularioInsertarLibro.do")) {
    List<String> listaDeCategorias=null;
    listaDeCategorias=Libro.buscarTodasLasCategorias();
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    despachador=request.getRequestDispatcher("FormularioInsertarLibro.jsp");
} else {
    String isbn = request.getParameter("isbn");
    String titulo = request.getParameter("titulo");
    String categoria = request.getParameter("categoria");
    Libro libro = new Libro(isbn, titulo, categoria);
    libro.insertar();
    despachador = request.getRequestDispatcher("MostrarLibros.do");
}
```

Tras modificar el servlet, podremos darnos cuenta de que toda la funcionalidad que se encontraba ubicada en la página InsertarLibro.jsp se ha reubicado a nivel del servlet y la página insertarLibro.jsp; así podrá ser eliminada (ver imagen).



5. Borrar en modelo MVC

Es el momento de progresar y aplicar el modelo MVC a la funcionalidad de borrar. Para ello añadiremos una nueva URL a nuestro modelo.

- BorrarLibro.do

El código que se encargará de gestionar la petición de esta url es el siguiente:

Código 7.10 (ControladorLibros.java)

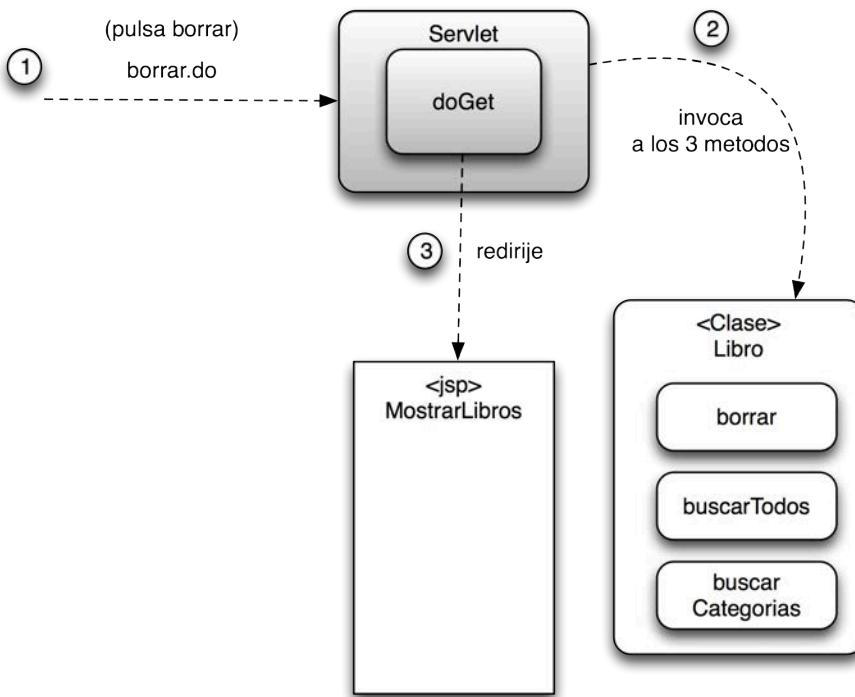
```
else {
    String isbn = request.getParameter("isbn");
    Libro libro = new Libro(isbn);
    libro.borrar();
    despachador = request.getRequestDispatcher("MostrarLibros.do");
}
```

Para que el código funcione correctamente, debemos modificar el enlace de la pagina MostrarLibros.jsp (ver imagen).

Código 7.11 (ControladorLibros.java)

```
<a href="BorrarLibro.do?isbn=<%=libro.getIsbn()%>">Borrar</a>
```

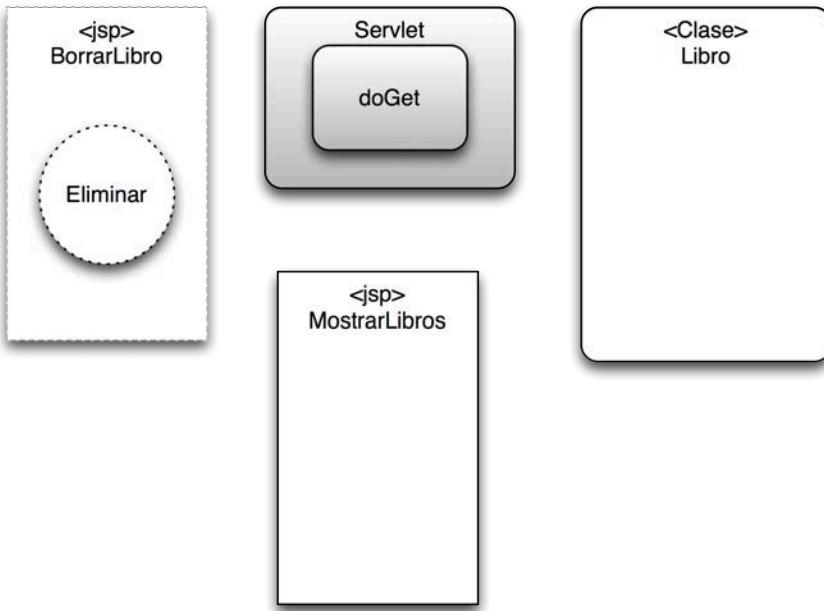
A continuación se muestra el nuevo flujo de la aplicación.



Esta es la explicación detallada de cada una de las peticiones:

1. Pulsamos en el botón de borrar a nivel de uno de los registros e invitamos al controlador con BorrarLibro.do
2. El controlador invoca a la clase active record a su método borrar y borra el registro. Una vez hecho ésto, invoca buscarTodos y buscarCategorías para cargar los datos necesarios.
3. El controlador nos redirige a la página MostrarLibros.jsp

Una vez completada la tarea de borrar a través del modelo MVC, nos daremos cuenta de que la página BorrarLibro.jsp ya no es necesaria (ver imagen).



6. Editar en modelo MVC

La tarea de editar presenta grandes similitudes con la tarea de insertar y nos permitirá de nuevo eliminar otra página (EditarLibro.jsp) de la aplicación. Para construir la funcionalidad de edición necesitaremos añadir una nueva URL a la gestión del controlador **FormularioEditarLibro.do** que nos redirigirá a la pagina FormularioEditarLibro.do. Vamos a ver a continuación el código:

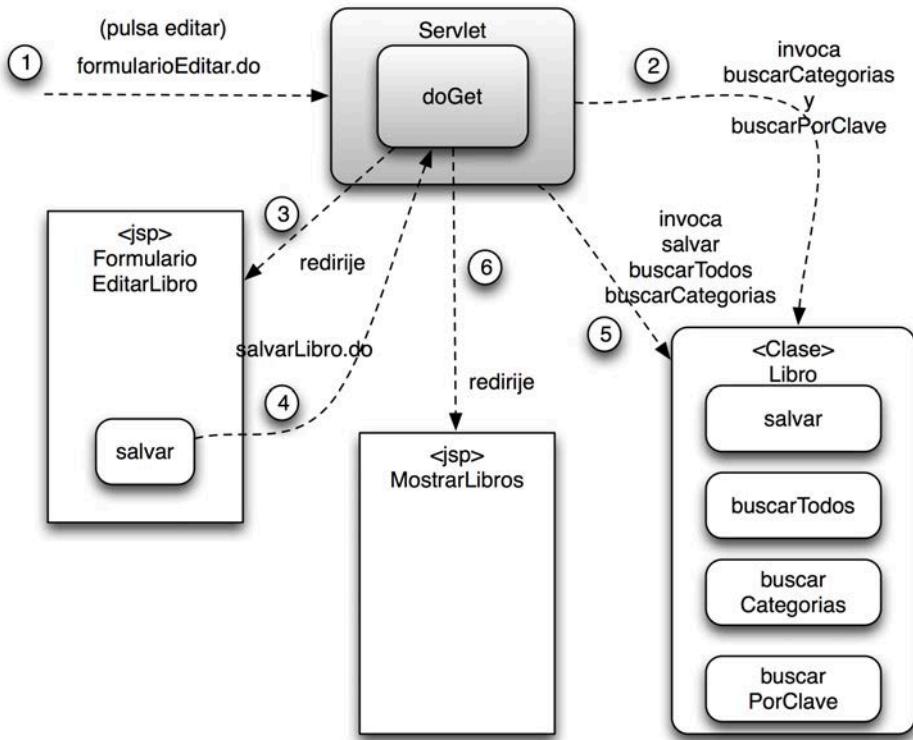
Código 7.12 (ControladorLibros.java)

```

} else if (request.getServletPath().equals("/FormularioEditarLibro.do")){
    String isbn = request.getParameter("isbn");
    List<String> listaDeCategorias = Libro.buscarTodasLasCategorias();
    Libro libro = Libro.buscarPorClave(request.getParameter("isbn"));
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    request.setAttribute("libro", libro);
    despachador = request.getRequestDispatcher("FormularioEditarLibro.jsp");
}

```

Una vez creado este bloque de código la aplicación ha de ser capaz de modificar los datos existentes en el formulario y salvar los cambios (ver imagen).



A continuación se muestra el bloque de código que se encargaría de aceptar los cambios y salvarlos.

```

}else {

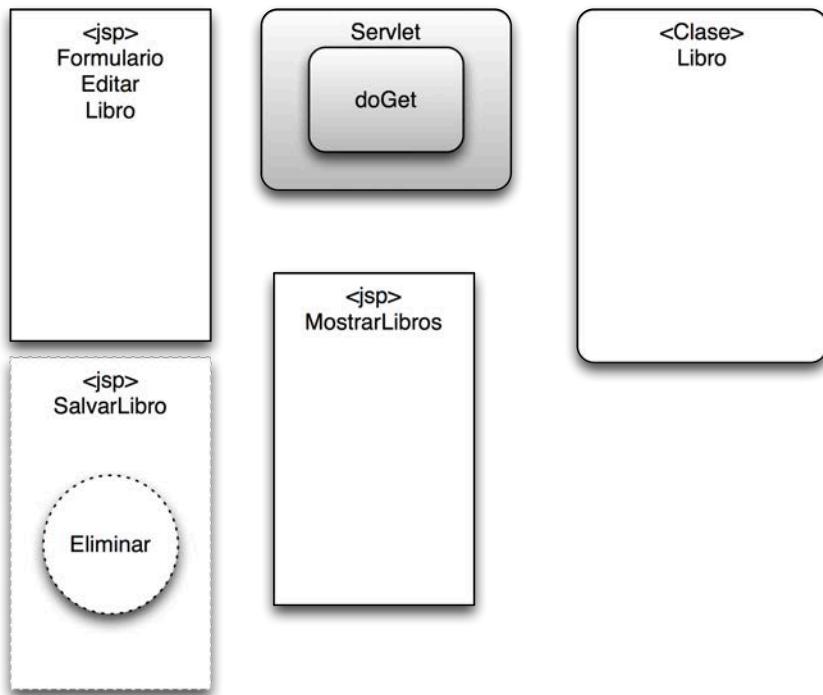
    String isbn = request.getParameter("isbn");
    String titulo = request.getParameter("titulo");
    String categoria = request.getParameter("categoria");
    Libro libro = new Libro(isbn, titulo, categoria);
    libro.salvar();
    despachador = request.getRequestDispatcher("MostrarLibros.do");
}
  
```

Vamos a explicar minuciosamente cada una de las peticiones de la tarea de edición.

1. Pulsamos en el botón de editar a nivel de uno de los registros e invitamos al controlador con FormularioEditarLibro.do

2. El controlador invoca a la clase Libro a su método buscarPorClave
3. El controlador nos redirige a la pagina FormularioEditarLibro.jsp, mostrando los datos a modificar.
4. Modificamos los datos del formulario, pulsamos al botón de salvar e invocamos a SalvarLibro.do.
5. El controlador invoca al método salvar para actualizar los datos del Libro. Después de hacer esto, como siempre invoca buscarTodos y buscarCategorias para cargar los datos que la página MostrarLibros.jsp necesita.
6. El controlador nos redirige a MostrarLibros.jsp

Una vez hemos terminado de gestionar la tarea de edición, nos sovrará la pagina SalvarLibro.jsp ya que su funcionalidad la cubre el controlador, por lo tanto podremos eliminarla (ver imagen).



7. Filtrado en Modelo MVC

Complementando a las modificaciones anteriores nos encontramos con la tarea de filtrado. Ésta permite organizar y seleccionar los libros por categoría para ello añadiremos una nueva url que acceda al controlador y sea la encargada de filtrar.

- Filtrar.do

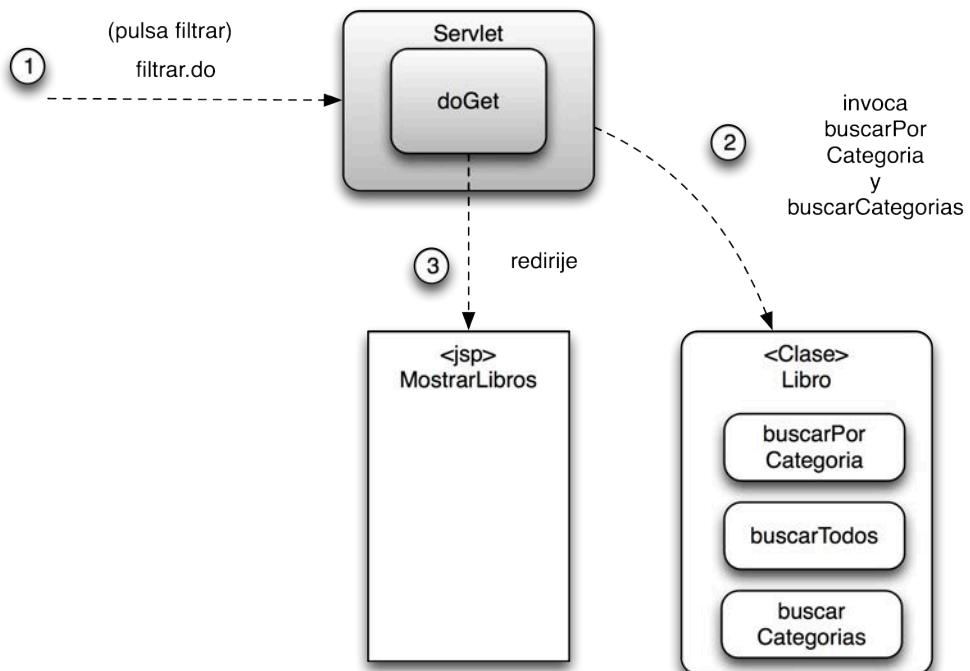
Arquitectura Java

Una vez añadida esta url y modificada la pagina MostrarLibros.jsp, podremos añadir el siguiente bloque de código al controlador de tal forma que se encargue de realizar el filtrado.

Código 7.13 (ControladorLibros.java)

```
List<Libro> listaDeLibros = null;
List<String> listaDeCategorias = Libro.buscarTodasLasCategorias();
if (request.getParameter("categoria") == null ||
request.getParameter("categoria").equals("seleccionar")) {
listaDeLibros = Libro.buscarTodos();
} else {
listaDeLibros = Libro.buscarPorCategoria(request
.getParameter("categoria"));
}
request.setAttribute("listaDeLibros", listaDeLibros);
request.setAttribute("listaDeCategorias", listaDeCategorias);
despachador = request.getRequestDispatcher("MostrarLibros.jsp");
}
```

A continuación se muestra un diagrama de flujo con la lógica que se ejecuta durante la operación de filtrado.



Cada una de las peticiones de la tarea de edición en explicación pormenorizada:

1. Pulsamos en el botón de filtrar a nivel del desplegable.
2. El controlador invoca a la clase Libro a su método buscarPorCategoría. Una vez hemos esto, el Controlador invoca al método buscarCategorias para cargar los datos necesarios para la párra MostrarLibros.jsp.
3. El controlador nos redirige a la página MostrarLibros.jsp, aplicando el filtro.

Hemos terminado de modificar las páginas y de construir nuestro controlador usando el principio SRP y reubicando las responsabilidades de control a nivel del servlet controlador .A continuación se muestra el código fuente del servlet completo para ayudar a clarificar todas las modificaciones realizadas.

```
public class ControladorLibros extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response) throws ServletException, IOException {
        RequestDispatcher despachador = null;

        if (request.getServletPath().equals("/MostrarLibros.do")) {

            List<Libro> listaDeLibros = Libro.buscarTodos();
            List<String> listaDeCategorias = Libro.
                buscarTodasLasCategorias();
            request.setAttribute("listaDeLibros", listaDeLibros);
            request.setAttribute("listaDeCategorias", listaDeCategorias);
            despachador = request.
                getRequestDispatcher("MostrarLibros.jsp");
        } else if (request.getServletPath().equals(
                    "/FormularioInsertarLibro.do")) {

            List<String> listaDeCategorias = null;
            listaDeCategorias = Libro.buscarTodasLasCategorias();
            request.setAttribute("listaDeCategorias", listaDeCategorias);
            despachador = request
                .getRequestDispatcher("FormularioInsertarLibro.jsp");
        } else if (request.getServletPath().equals("/InsertarLibro.do")) {

            String isbn = request.getParameter("isbn");
            String titulo = request.getParameter("titulo");
            String categoria = request.getParameter("categoria");
            Libro libro = new Libro(isbn, titulo, categoria);
            libro.insertar();
            despachador = request.
                getRequestDispatcher("MostrarLibros.do");
        } else if (request.getServletPath().equals("/BorrarLibro.do")) {
            String isbn = request.getParameter("isbn");
            Libro libro = new Libro(isbn);
            libro.borrar();
        }
    }
}
```

```

        despachador = request.
            getRequestDispatcher("MostrarLibros.do");
    } else if (request.getServletPath().equals("/FormularioEditarLibro.do")) {
        String isbn = request.getParameter("isbn");
        List<String> listaDeCategorias = Libro.
            buscarTodasLasCategorias();
        Libro libro = Libro
            .buscarPorClave(request.getParameter("isbn"));
        request.setAttribute("listaDeCategorias", listaDeCategorias);
        request.setAttribute("libro", libro);

        despachador = request
            .getRequestDispatcher("FormularioEditarLibro.jsp");

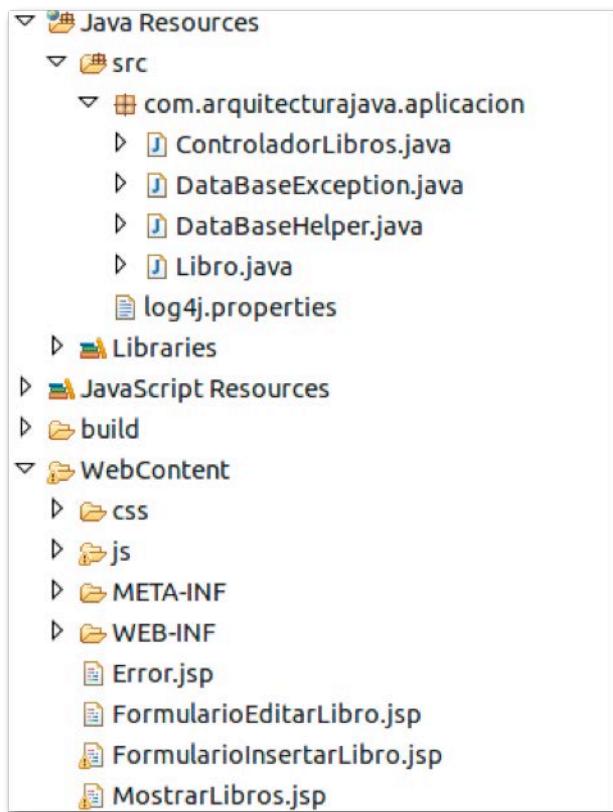
    } else if (request.getServletPath().equals("/SalvarLibro.do")) {

        String isbn = request.getParameter("isbn");
        String titulo = request.getParameter("titulo");
        String categoria = request.getParameter("categoria");
        Libro libro = new Libro(isbn, titulo, categoria);
        libro.salvar();
        despachador = request.
            getRequestDispatcher("MostrarLibros.do");
    } else {
        List<Libro> listaDeLibros = null;
        List<String> listaDeCategorias =
            Libro.buscarTodasLasCategorias();
        if (request.getParameter("categoria") == null
            ||
            request.getParameter("categoria").equals("seleccionar")) {
            listaDeLibros = Libro.buscarTodos();
        } else {
            listaDeLibros = Libro.buscarPorCategoria(request
                .getParameter("categoria"));
        }
        request.setAttribute("listaDeLibros", listaDeLibros);
        request.setAttribute("listaDeCategorias", listaDeCategorias);
        despachador = request.
            getRequestDispatcher("MostrarLibros.jsp");
    }
    despachador.forward(request, response);
}
}

```

Resumen

Una vez finalizado el controlador, hemos reubicado las responsabilidades de nuestra aplicación, organizándolas de acuerdo al principio SRP. Como resultado, hemos podido separar mejor cada una de las capas y eliminar parte de los ficheros JSP que teníamos. Por ultimo, queda por ver como ha quedado la estructura física de ficheros una vez completada la evolución a un modelo MVC . La siguiente imagen muestra el resultado.



8.JSTL

Acabamos de aplicar el patrón MVC a nuestra aplicación, apoyándonos en el principio SRP ,aun así un desarrollador puede decidir no usar este patrón de diseño y seguir usando scriptlet de JSP en sus páginas. Para evitar este problema, en este nuevo capítulo modificaremos las páginas de nuestra aplicación para que use Java Standard Tag Library (JSTL) como librería de etiquetas, la cuál nos ayudará a simplificar el desarrollo y evitara el uso de scriptlet, obteniendo unas paginas más claras.

Objetivos:

- Utilizar JSTL en la aplicación

Tareas:

1. Instalación de JSTL
2. Introducción a JSTL
3. Etiquetas Básicas JSTL
4. Modificar MostrarLibro.jsp
5. Modificar FormularioInsertarLibro.jsp
6. Modificar FormularioEditarLibro.jsp

1. Instalación de JSTL

Antes de comenzar a introducirnos en JSTL, necesitamos instalar librerías adicionales a nuestro servidor Tomcat. Para ello accederemos a la siguiente url.

- <http://jstl.java.net/download.html>

De la cuál podremos obtener las siguientes librerías

- jstl.jar
- standard.jar

Una vez obtenidas las librerías, las ubicaremos en el directorio WEB-INF/lib de nuestra aplicación (ver imagen).



Realizado este paso, habremos terminado de instalar JSTL en nuestra aplicación y podremos comenzar a trabajar con ello.

2. Introducción a JSTL

En estos momentos nuestras páginas JSP mezclan el uso de etiquetas con scriptlet. JSTL nos permitirá homogeneizar este comportamiento, permitiéndonos el uso de etiquetas html y la substitución del scriptlet por un conjunto de etiquetas de servidor, de tal forma que las nuevas páginas JSP únicamente estarán compuestas por etiquetas (ver imagen).

```
<JSP>  
MostrarLibros
```

```
<html>  
<%  
scriptlet  
%>
```

```
<JSP>  
MostrarLibros
```

```
<html>  
<c:out>
```

A continuación se muestra un ejemplo sencillo de las diferencias entre ambas tecnologías a la hora de imprimir información html

JSP

Código 8.1:

```
<%if (edad>18) {>  
    Eres una persona adulta con edad <%=edad%>  
<%}>%>
```

JSTL

Código 8.2:

```
<c:if test="${edad>18}">  
    Eres una persona adulta ${edad}  
</c:if>
```

Como podemos ver con JSTL únicamente tenemos grupos de etiquetas y texto plano lo que hará más sencillo a los maquetadores trabajar con las páginas.

3. Etiquetas Básicas JSTL

Antes de modificar nuestra aplicación para que soporte JSTL, vamos a explicar las etiquetas elementales de JSTL que nuestra aplicación va a necesitar.

La siguiente etiqueta imprime una variable JSP en la página:

Código 8.3:

```
 ${variable}
```

La etiqueta <forEach> recorre colección de objetos ubicados en una lista e imprime sus propiedades

Código 8.4:

```
<c:forEach var="objeto" items="${lista}">
    ${objeto.propiedad}
</c:forEach>
```

Se encarga de imprimir un bloque de información dependiente de una condición predeterminada.

Código 8.5:

```
<c:if test="${edad>30}">
    Eres una persona adulta ${edad}
</c:if>
```

Una vez vistas las etiquetas esenciales, vamos a modificar nuestras páginas.

4. Modificar MostrarLibro.jsp

A continuación vamos a usar dos etiquetas <c:forEach> para modificar la pagina y eliminar completamente el scriptlet. A continuación se muestra el código.

Código 8.6: (MostrarLibro.jsp)

```
<form name="filtroCategoria" action="filtrar.do"><select  
    name="categoria">  
        <option value="seleccionar">seleccionar</option>  
        <c:forEach var="categoria" items="${listaDeCategorias}">  
            <option value="${categoria}">${categoria}</option>  
        </c:forEach>  
</select><input type="submit" value="filtrar"></form>  
<br />  
<c:forEach var="libro" items="${listaDeLibros}">  
    ${libro.isbn}${libro.titulo}${libro.categoría}  
    <a href="BorrarLibro.do?isbn=${libro.isbn}">borrar</a>  
    <a href="FormularioEditarLibro.do?isbn=${libro.isbn}">editar</a>  
    <br />  
</c:forEach>  
<a href="FormularioInsertarLibro.do">InsertarLibro</a>
```

Después de realizar esta tarea, pasaremos a modificar la página FormularioInsertar.jsp.

5. Modificar Formulario Insertar

Será suficiente con crear otro bucle `<c:forEach>` encargado de cargar la lista de categorías.

Código 8.7: (FormularioLibro.jsp)

```
<form id="formularioInsercion" action="InsertarLibro.do">  
<fieldset><legend>FormularioaltaLibro</legend>  
<p><label for="isbn">ISBN:</label>  
<input type="text" id="isbn" name="isbn" /></p>  
<p><label for="titulo">Titulo:</label>  
<input type="text" id="titulo" name="titulo" /></p>  
<p><label for="categoria">Categoria :</label>  
<select name="categoria">  
    <c:forEach var="categoria" items="${listaDeCategorias}">  
        <option value="${categoria}">${categoria}</option>  
    </c:forEach>  
</select><br />  
</p><p><input type="submit" value="Insertar" /></p>  
</fieldset>  
</form>
```

Finalizada esta operación, nos queda una última página por modificar

6. Modificar FormularioEditarLibro.jsp

Esta página es muy similar a la página anterior, aunque tendrá más bloques de código JSTL al tener que presentar el formulario con la información para editar ya cargada.

Código 8.8: (FormularioLibroEdicion.jsp)

```
<form id="formularioEdicion" action="ModificarLibro.do">
<fieldset><legend>FormularioalLibro</legend>
<p><label for="isbn">ISBN:</label><input type="text" id="isbn"
   name="isbn" value="${libro.isbn}" /></p>
<p><label for="titulo">Título:</label><input type="text"
   id="titulo" name="titulo" value="${libro.titulo}" /></p>
<p><label for="categoria">Categoria :</label><select
   name="categoria">
   <c:forEach var="categoria" items="${listaDeCategorias}">
      <option value="${categoria}">${categoria}</option>
   </c:forEach>
</select><br />
</p>
<p><input type="submit" value="Salvar" /></p>
</fieldset>
</form>
```

Resumen

Una vez finalizadas las modificaciones, nuestra aplicación ha dejado de usar scriptlet en el código de las páginas JSP y utiliza etiquetas JSTL, reforzando el modelo MVC que construimos en el capítulo anterior.

9. El principio OCP y modelo MVC 2

En los dos capítulos anteriores hemos modificado nuestra aplicación para que, usando el principio SRP, la división de responsabilidades sea mas natural y use JSTL como tecnología de capa de presentación. En este capítulo introduciremos otro principio importante: **el principio OCP** o principio de apertura y cierre cuya definición es:

OCP (Open Closed Principle): El principio OCP o principio de apertura y cierre, se define como: todo código desarrollado para una aplicación debe estar cerrado a las modificaciones y abierto a la extensibilidad. Expresado de otra manera: debemos poder añadir nueva funcionalidad a la aplicación sin tener que alterar el código ya construido. Seguidamente nos encargaremos de aplicar este principio en nuestra aplicación.

Objetivos

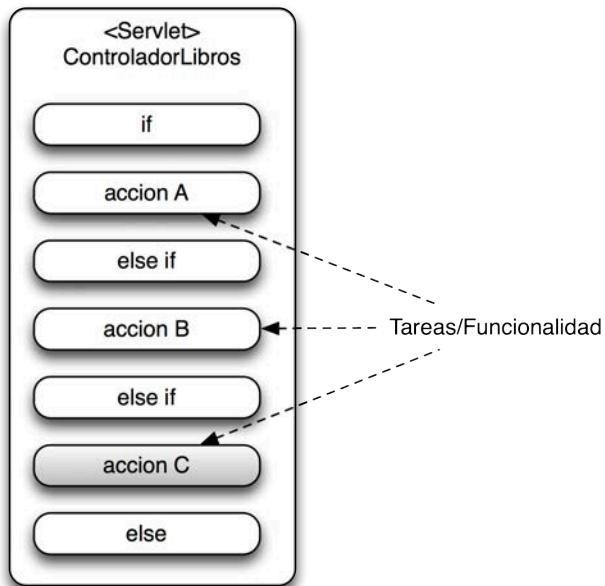
- Aplicar el principio OCP a nuestra aplicación

Tareas

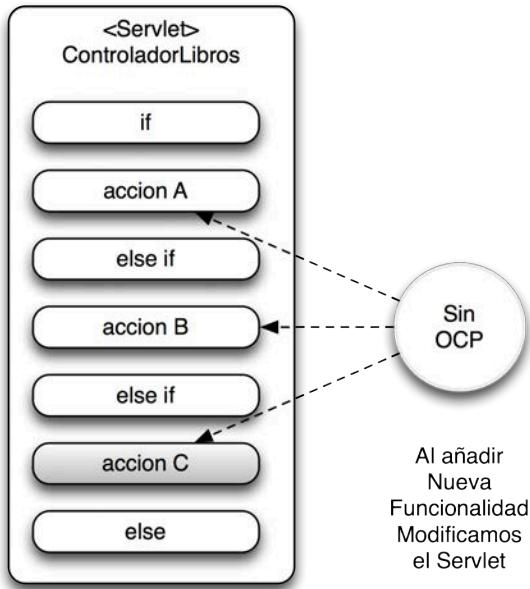
1. El principio OCP y el Controlador
2. El principio OCP y el patron Command
3. Creación de una accion principal
4. Crear una jerarquia de acciones
5. Api de reflection y principio OCP

1. El principio OCP y el Controlador

En estos momentos tenemos construida una aplicación sobre el modelo MVC y su diseño es bastante sólido. Después de conocer la definición del principio OCP nos percatamos de que cada vez que añadimos una mínima funcionalidad nueva a nuestra aplicación estamos obligados a modificar el Servlet controlador de esta y añadir una o varias sentencias if –else if que implementen la nueva funcionalidad (ver imagen).



Así pues parece claro que el controlador es un elemento que no está para nada cerrado a las modificaciones, sino más bien al contrario: cualquier modificación por pequeña que sea de la aplicación afecta directamente al controlador y clarifica que no cumple con el principio OCP (ver imagen).

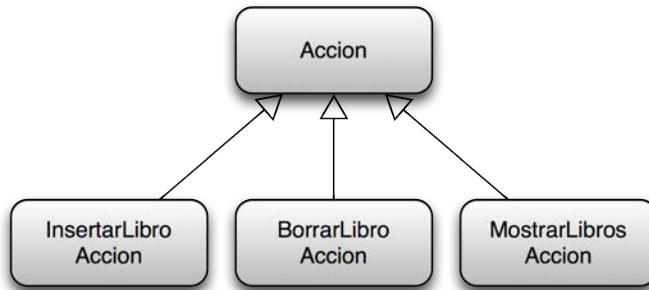


2. El principio OCP y el patrón Command

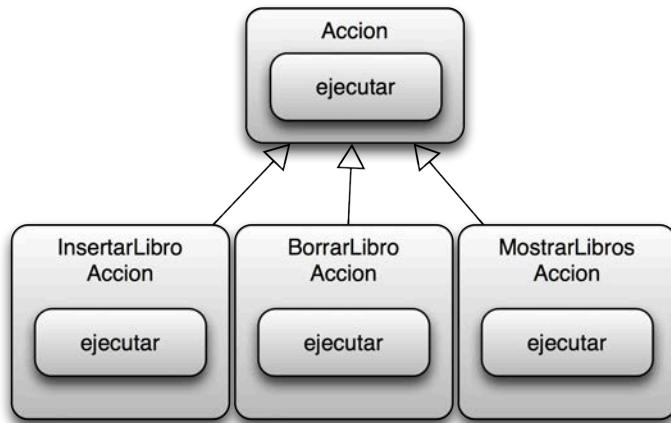
Vamos a apoyarnos en el principio OCP para comenzar a rediseñar el controlador de tal forma que podamos añadir nueva funcionalidad a la aplicación sin que éste se vea afectado. Para rediseñar el controlador y que pueda cumplir con el principio OCP vamos extraer el concepto de **Acción/Tarea** del mismo y ubicarlo en una jerarquía de clases completamente nueva . Cada una de estas clases tendrá las siguientes características:

1. Heredarán de una clase común denominada Acción.
2. Todas dispondrán del método ejecutar en el cual ejecutaran la funcionalidad a ellas asignada.
3. Tendrán un nombre que haga referencia a la funcionalidad que se encargan de ejecutar.

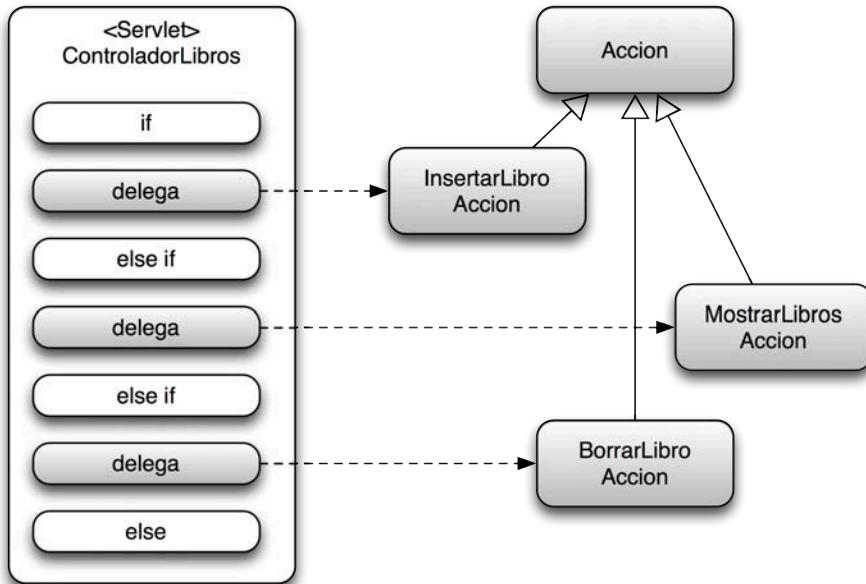
A continuación se muestra la imagen con una parte de la jerarquía de las acciones.



En principio nos puede parecer que se trata de un sencillo grupo de clases sin embargo la jerarquía de acciones se construirá apoyándonos en un patrón de diseño denominado Comando. Este patrón obliga cada clase a definir un método ejecutar en el cuál ubicaremos la funcionalidad que deseamos (ver imagen).



Definida la jerarquía de clases, podremos extraer la funcionalidad que se encuentra ubicada en el controlador y reubicarla en el método ejecutar de cada una de las clases . A partir de este momento, el controlador delegará en las acciones (ver imagen).



3. Creación de una acción principal

Definido el nuevo funcionamiento del controlador, vamos a ver el código fuente de la clase Acción.

Código 9.1: (Accion.java)

```

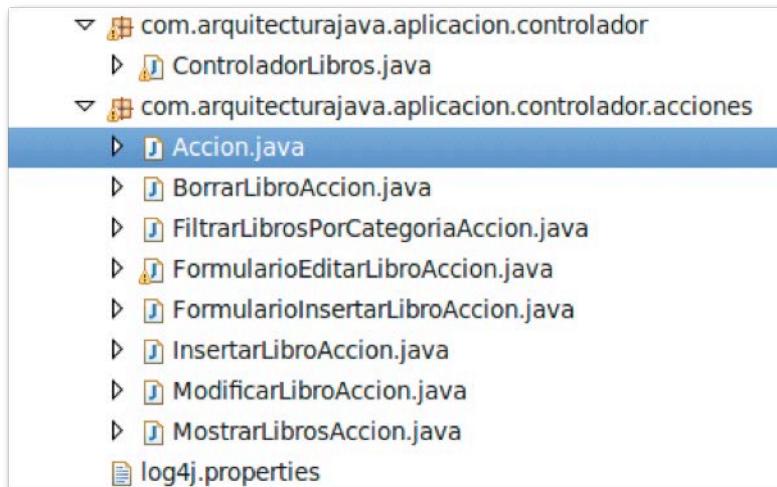
package com.arquitecturajava.aplicacion.controlador.acciones;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public abstract class Accion {
    public abstract String ejecutar(HttpServletRequest request,
                                    HttpServletResponse response);
}
    
```

Creada esta clase, podemos ver como el método ejecutar recibe dos parámetros.

- HttpServletRequest
- HttpServletResponse

Son los mismos parámetros que utilizamos a nivel del servlet. De esta forma nuestro servlet controlador podrá delegar en el conjunto de acciones que vemos a continuación.



4. Crear jerarquía de acciones

A partir de ahora usaremos acciones para implementar la funcionalidad de cada una de los distintos bloques de código. Vamos a ver el código fuente de una acción en concreto que nos ayude a entender qué parte de la funcionalidad existente cubren las acciones y qué otra funcionalidad cubre el controlador. A continuación se muestra el código fuente de la Accion InsertarLibro.

Código 9.2: (InsertarLibroAccion.java)

```
public class InsertarLibroAccion extends Accion{
    public String ejecutar(HttpServletRequest request,
                           HttpServletResponse response) {
        String isbn = request.getParameter("isbn");
        String titulo = request.getParameter("titulo");
        String categoria = request.getParameter("categoria");
        Libro libro = new Libro(isbn, titulo, categoria);
        libro.insertar();
        return "MostrarLibros.do";
    }
}
```

Hemos creado cada una de las distintas acciones extrayendo el código que existía en las cláusulas if-elseif-else del método doGet() del controlador, quedando éste muy simplificado ya que únicamente tiene que decidir qué acción crea y luego invocar al método ejecutar de ésta (ver código).

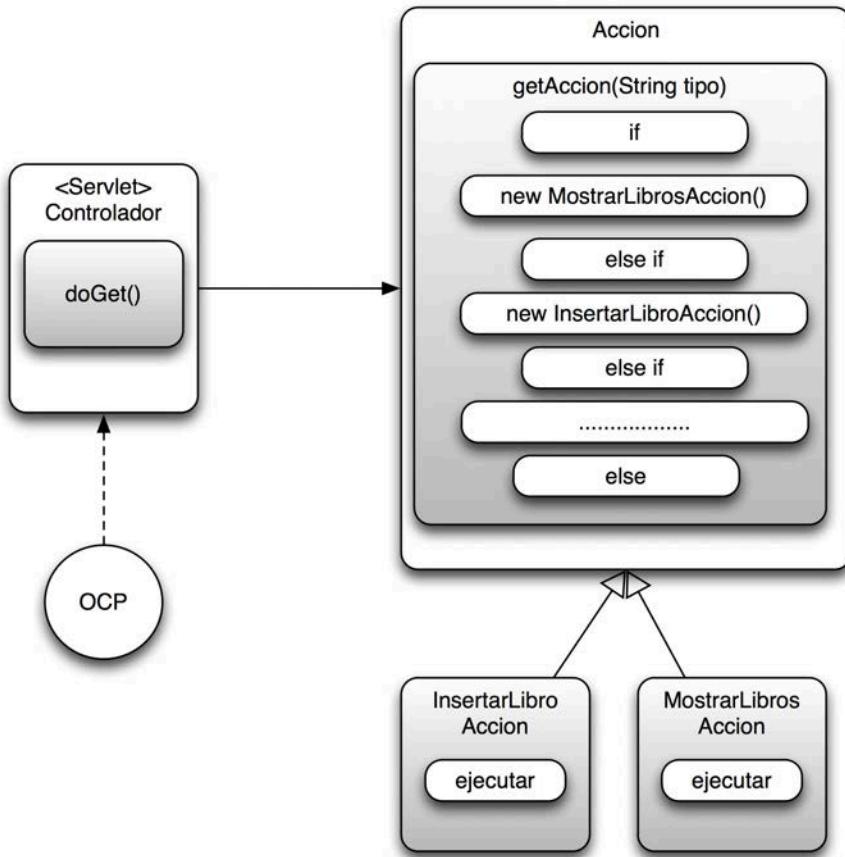
Código 9.3: (ControladorLibros.java)

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) throws ServletException, IOException {
    RequestDispatcher despachador = null;
    Accion accion = null;
    if (request.getServletPath().equals("/MostrarLibros.do")) {
        accion = new MostrarLibrosAccion();
    } elseif (request.getServletPath().equals(
              "/FormularioInsertarLibro.do")) {
        accion = new FormularioInsertarLibroAccion();
    } elseif (request.getServletPath().equals("/InsertarLibro.do")) {
        accion = new InsertarLibroAccion();
    } elseif (request.getServletPath().equals("/BorrarLibro.do")) {
        accion = new BorrarLibroAccion();
    } elseif (request.getServletPath().equals("/FormularioEditarLibro.do")) {
        accion = new FormularioEditarLibroAccion();
    } elseif (request.getServletPath().equals("/ModificarLibro.do")) {
        accion = new ModificarLibroAccion();
    } else {
        accion = new FiltrarLibrosPorCategoriaAccion();
    }
    despachador = request.getRequestDispatcher(
                  ....accion.ejecutar(request,response));
    despachador.forward(request, response);
}
```

Con este refactoring hemos mejorado bastante nuestra aplicación y aunque cada vez que añadimos una nueva funcionalidad, debemos modificar el controlador, la mayor parte del nuevo código se encuentra en cada una de las acciones.

El siguiente paso a realizar será extraer el propio bloque if else del controlador y ubicarlo en la clase Acción de tal forma que el controlador ya no tenga ninguna responsabilidad en cuanto a qué acción ha de crear y esta responsabilidad quede completamente delegada a la clase Acción.

Para este paso construiremos un nuevo método en la clase Acción denominado getAccion() que se encargue de crear las distintas acciones. Estos métodos habitualmente se les denominan métodos factoría ya que cumplen con el patrón factory. A continuación se muestra un diagrama aclaratorio así como el código fuente de este método.



Código 9. 4: (Accion.java)

```

public static Accion getAccion(String tipo) {
    Accion accion=null;
    if (tipo.equals("/MostrarLibros.do")) {
        accion = new MostrarLibrosAccion();
    } elseif (tipo.equals(
        "/FormularioInsertarLibro.do")) {
        accion = new FormularioInsertarLibroAccion();
    } elseif (tipo.equals("/InsertarLibro.do")) {
        accion = new InsertarLibroAccion();
    }
    .....
    return accion;
}

```

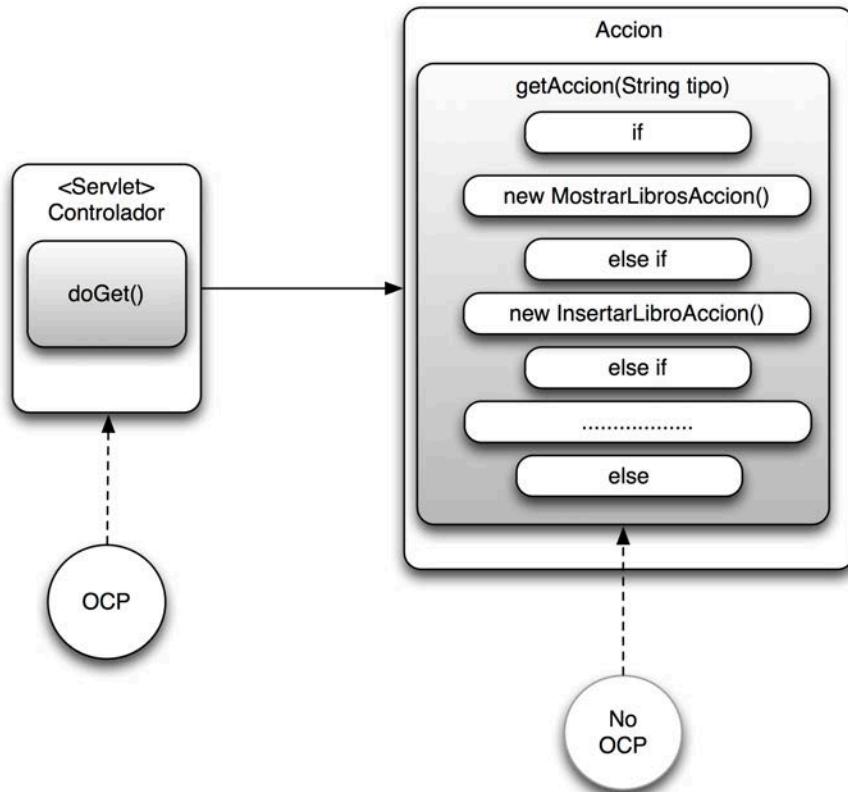
Arquitectura Java

Como podemos ver, es este método ahora el encargado de crear cada una de las acciones, simplificando sobremanera el controlador que tenemos desarrollado, ya que a partir de este momento el controlador queda con el siguiente código:

Código 9.5: (ControladorLibros.java)

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) throws ServletException,
                     IOException {
    RequestDispatcher despachador = null;
    Accion accion = null;
    String url= request.getServletPath();
accion= Accion.getAccion(url.substring(1,url.length()-3));
    despachador = request.getRequestDispatcher(accion.ejecutar(request,
                response));
    despachador.forward(request, response);
}
```

Una vez realizados estos cambios, no tenemos que modificar el servlet nunca más para poder añadir nuevas acciones a nuestra aplicación: por lo tanto el servlet controlador cumple con el principio OCP, aunque no ocurre lo mismo con la clase action, como se muestra en el siguiente diagrama.



En estos momentos debemos modificar la clase Acción y su método estático `getAccion()` cada vez que deseemos añadir nuevas acciones . Es por esto que todavía nos queda trabajo que realizar para que también esta clase cumpla con el principio OCP.

5. Api de reflection y el principio OCP

Para solventar este problema es necesario fijarse en el nombre de las distintas URL que pasamos al controlador, como es por ejemplo la siguiente:

Código 9.7

FormularioInsertarLibro.do

Arquitectura Java

Después de obtener esta URL, podemos transformarla en el siguiente texto ejecutando un par de funciones de formato, con lo cual nos queda un texto idéntico al nombre de una clase Acción concreta.

Código 9.8:

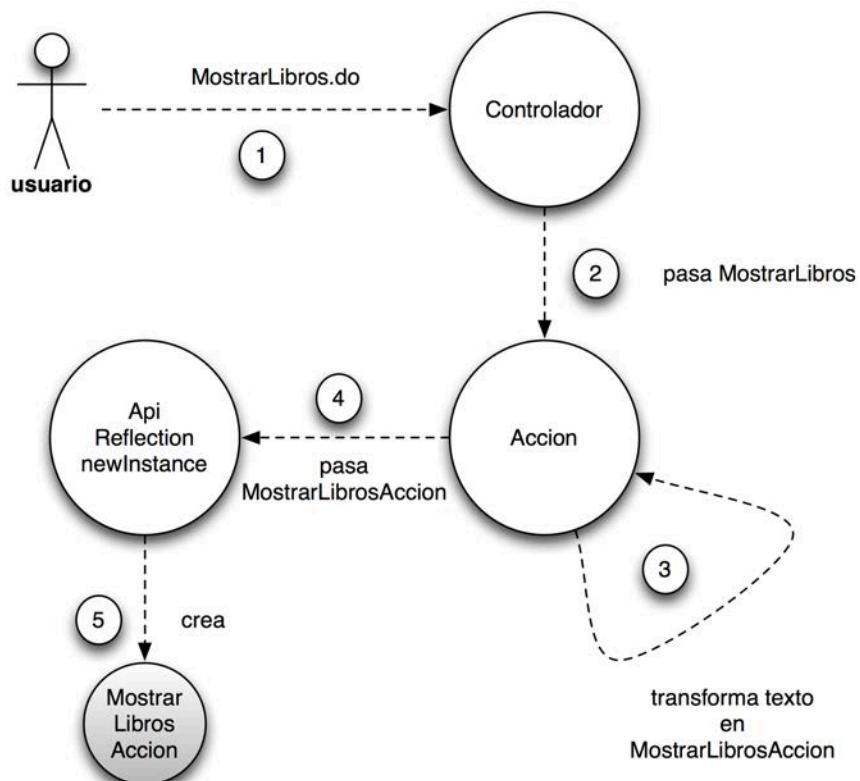
```
FormularioInsertarLibroAccion
```

Obtenido el nombre de la acción, usaremos el api de reflection que ya conocemos de capítulos anteriores y crearemos un objeto basado en el nombre de la clase. Para ello usaremos el siguiente método del api:

Código 9.9:

```
Class.forName.newInstance(String nombreClase)
```

A continuación se muestra un diagrama aclaratorio:

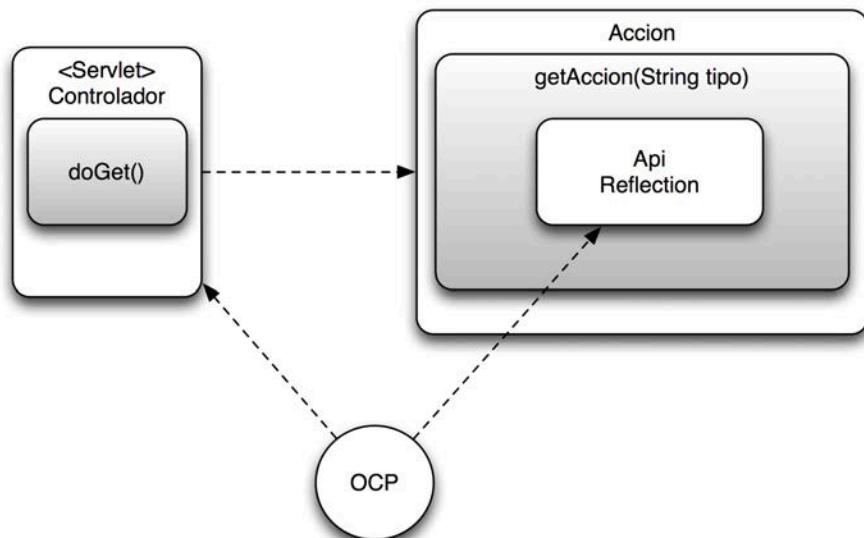


Una vez creado el objeto, invocaremos a su método ejecutar. A continuación se muestra el código de cómo el API de reflection permite ir insertando nuevas acciones sin modificar el código previamente construido.

Código 9.10:

```
public static Accion getAccion(String tipo) {
    Accion accion = null;
    try {
        accion = (Accion) Class.forName(getPackage()
            +"."+tipo+"Accion").newInstance();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return accion;
}
```

Terminado este cambio, podemos ir añadiendo nuevas acciones a nuestra aplicación sin impactar en las anteriormente construidas, cumpliendo con el principio OCP (ver imagen).



Resumen

Hemos terminado de modificar nuestra aplicación que ahora se apoya en el principio OCP para ir añadiendo nueva funcionalidad sin afectar a la anteriormente construida. Este diseño es muy habitual a día de hoy en frameworks web como ha sido el caso de struts o es el caso de JSF y Spring MVC. Al tratarse de una modificación importante del Modelo MVC a este modelo se le suele denominar Modelo MVC 2 o de Controlador Frontal.

10. Hibernate

Acabamos de aplicar el principio OCP y hemos migrado nuestra aplicación a un modelo MVC2. Ésta ha sido la última versión de la aplicación que no hace uso en alguna de sus partes de un framework JEE.

A partir de este capítulo comienza la segunda parte del libro orientada a la utilización de distintos frameworks que nos serán útiles para acelerar el desarrollo. En este capítulo introduciremos Hibernate como framework de persistencia y substituiremos la capa de persistencia actual (DataBaseHelper.java) por Hibernate.

Objetivos :

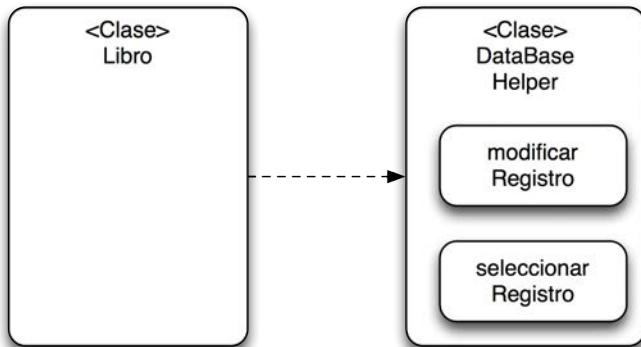
- Introducción a Hibernate
- Utilizar Hibernate en nuestra aplicación

Tareas :

1. Concepto de Framework de persistencia.
2. Instalación de Hibernate.
3. Introducción a Hibernate
4. Configuración de Hibernate
5. Insertar objetos en la base de datos con Hibernate.
6. Selección de objetos de la base de datos con Hibernate.
7. Seleccionar un único objeto con Hibernate
8. Borrar objetos de la base de datos con Hibernate.
9. Filtrar objetos de la base de datos con Hibernate
10. Construcción de la clase Hibernate Helper.
11. Mapeo del fichero de configuración.
12. El principio de Convención sobre configuración

1. Introducción al concepto de framework de persistencia

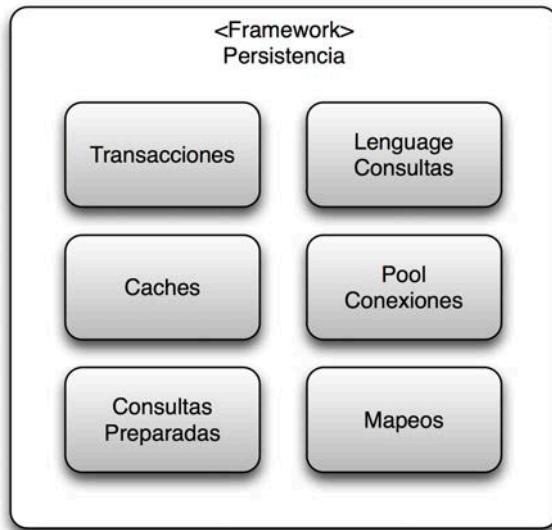
Hasta este momento hemos usado la clase DataBaseHelper para que se encargara de las operaciones ligadas con la base de datos (ver imagen).



Sin embargo la clase aunque funcional, tiene muchas limitaciones tales como:

- No soporta transacciones que agrupen varios métodos ya que la conexión no es compartida y cada método es totalmente independiente.
- No soporta uso de pool de conexiones y cada vez se construye una conexión nueva
- No soporta consultas preparadas y usa consultas SQL estándar con los problemas de inyección de SQL que ésto genera
- No soporta varios tipos de motores de bases de datos y se centra en MySQL

Podríamos continuar con la lista de limitaciones, pero parece claro que si queremos construir una solución Enterprise, la clase que hemos construido no es el mejor camino. Nos ha sido útil hasta el momento y ha ayudado a ir asentando conceptos. Así pues, si queremos evolucionar nuestra solución a algo más serio, deberemos apoyarnos en un framework de persistencia que elimine las limitaciones que nuestra clase actual tiene y aporte características adicionales (ver imagen).



En nuestro caso utilizaremos Hibernate ya que es el framework de persistencia más conocido por la comunidad, la siguiente tarea abordara su instalación.

2. Instalación de Hibernate

Para poder instalar Hibernate lo primero que tendremos que hacer es bajarnos las librerías de las cuales se compone, que están disponibles en la siguiente url:

<http://sourceforge.net/projects/hibernate/files/hibernate3/>

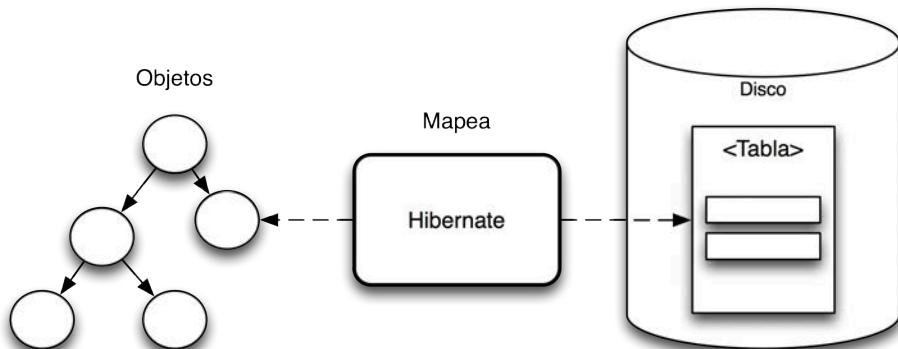
Obtenidas las librerías, pasaremos a instalarlas en un proyecto Java Estándar de Eclipse en el cuál construiremos unos sencillos ejemplos para aprender a manejar el framework. A continuación se muestran las librerías que debemos añadir a nuestro proyecto para que los siguientes ejemplos se puedan ejecutar correctamente.



Tras instalar el framework, es momento de comenzar a trabajar con él.

3. Introducción a Hibernate

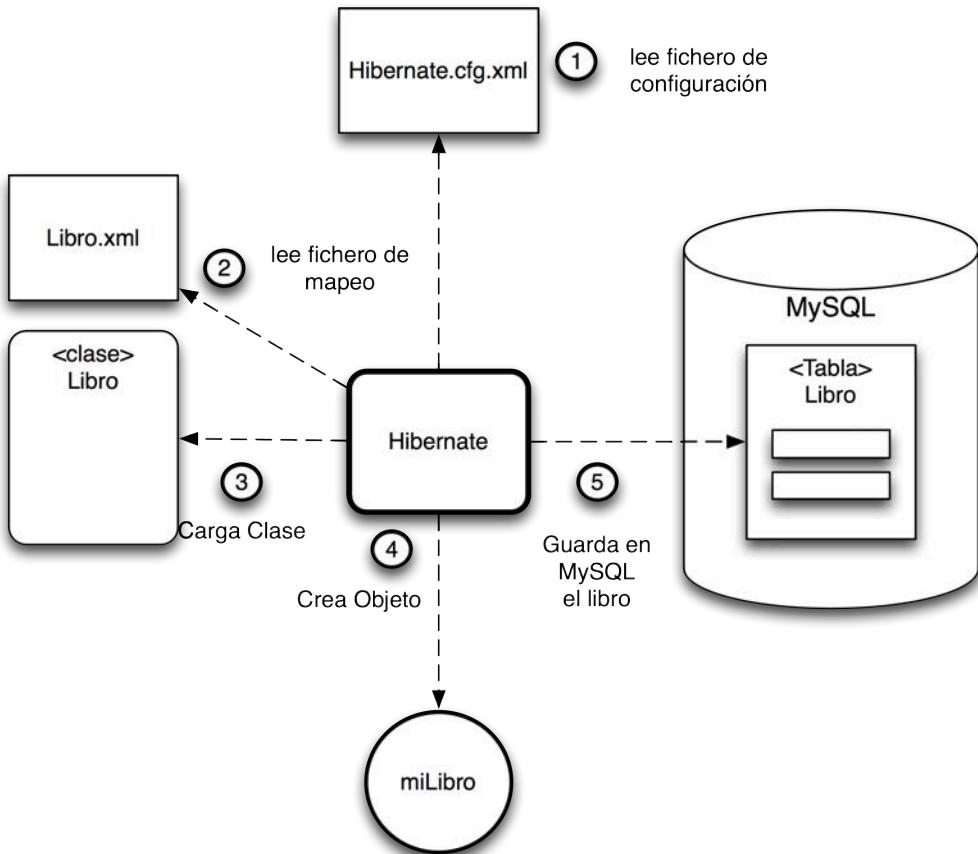
Hibernate es un framework ORM (Object Relational Mapping) cuya tarea es la de permitir a un desarrollador mapear objetos contra registros de una base de datos (ver imagen).



Todos los frameworks de persistencia trabajan de forma similar, pero cada uno tiene sus propios ficheros de configuración. En el caso del framework Hibernate existen una serie de ficheros y conceptos claves a la hora de comenzar a trabajar con él. Los enumeraremos a continuación basándonos en la clase Libro.

- **Hibernate.cfg.xml:** Es el fichero principal de configuración del framework, es donde se configura el driver JDBC de acceso a datos la Ip del servidor de base de datos, el usuario y la password ,así como los ficheros de mapeo que van a utilizar las distintas clases.
- **Libro.xml:** Es el fichero de mapeo que almacena la información relevante referida a cómo un objeto que pertenece a una clase determinada, en este caso un Libro, es mapeado a una fila de la tabla Libros
- **Libro.java :** Clase java que es mapeada

A continuación se muestra un diagrama de alto nivel sobre cuales son los ficheros y pasos generales que realiza el framework Hibernate a la hora de trabajar con los objetos java que se encarga de persistir.



A continuación vamos a explicar cada uno de los pasos del diagrama:

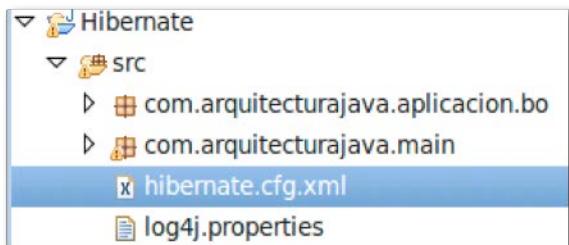
- 1. Leer `Hibernate.cfg.xml`:** Como primera tarea, el framework busca y lee el fichero de configuración donde se encuentra la información de usuario, clave, driver y url de conexión. De esta forma se conecta a la base de datos. Junto a esta información se encuentra también la lista de ficheros de mapeo que leerá posteriormente.
- 2. Leer Fichero de mapeo:** Se encarga de leer todos los ficheros de mapeo de la aplicación para saber como mapear un objeto determinado contra alguna de las tablas existentes en la base de datos.
- 3. Carga la clase :** El programa que estemos construyendo carga las clases en memoria

4. **Crear Objeto** : El programa que estemos construyendo crea varios objetos de la clase
5. **Salvar**: Hibernate se encarga de guardar los datos en la base de datos

Una vez realizada una introducción a cómo Hibernate trabaja en líneas generales, es momento de construir nuestro primer ejemplo.

4. Configuración Hibernate

Una vez creado un proyecto Java estándar y añadidas las librerías necesarias para trabajar con Hibernate, lo primero que necesitaremos será configurar el fichero hibernate.cfg.xml. Este fichero estará ubicado dentro de nuestro classpath, como la imagen muestra:



Clarificada la necesidad de este fichero, vamos a ver un ejemplo con su contenido:

Código 10.1(hibernate.cfg.xml)

```
<?xml version='1.0' encoding='utf-8'?>
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost/arquitecturajava</property>
<property name="connection.username">root</property>
<property name="connection.password">java</property>
<property name="connection.pool_size">5</property>
<property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
<property name="show_sql">true</property>
<mapping resource="com/arquitecturajava/aplicacion/bo/Libro.xml"/>
</session-factory>
</hibernate-configuration>
```

Como antes hemos comentado, el fichero contiene la información de conexión y los ficheros de mapeo a utilizar (mapping-resources). Vamos a comentar detalladamente cada una de las propiedades que este fichero contiene.

Código 10.2(hibérnate.cfg.xml):

```
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
```

Esta línea se encarga de seleccionar el driver nativo de conexión a la base de datos .Para ello necesitaremos añadir el driver tipo 4 de MySQL a nuestra aplicación.

Código 10.3(hibérnate.cfg.xml):

```
<property name="connection.url">jdbc:mysql://localhost/arquitecturajava</property>
```

Esta línea define la URL de acceso al servidor de base de datos y a una de sus bases de datos en concreto.

Código 10.4(hibérnate.cfg.xml):

```
<property name="connection.username">root</property>
<property name="connection.password">java</property>
```

Estas dos líneas configuran el usuario y password de acceso a la base de datos.

Código 10.5(hibérnate.cfg.xml):

```
<property name="connection.pool_size">5</property>
```

Esta línea configura el tamaño del pool de conexiones del framework a 5.

Código 10.6(hibérnate.cfg.xml):

```
<property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>
```

Esta línea selecciona el dialecto que usara Hibernate a la hora de realizar las distintas operaciones contra la base de datos .En este caso el dialecto elegido es el de MySQL aunque Hibernate soporta muchos más como SQLServer, Oracle etc y permite que una

Arquitectura Java

misma aplicación basada en Hibernate pueda cambiar de forma transparente el motor de base de datos utilizado .Esta es una de las ventajas fundamentales de hacer uso de un framework de persistencia.

Código 10.7(hibernate.cfg.xml):

```
<property name="show_sql">true</property>
```

Configura Hibernate para que muestre todas las consultas SQL que genera. Este parámetro suele ser una ayuda importante a la hora de valorar si Hibernate está funcionando de forma correcta o no.

Código 10.8(hibernate.cfg.xml):

```
<mapping resource="com/arquitecturajava/aplicacion/bo/Libro.xml"/>
```

Esta última línea es la que se encarga de dar de alta los distintos ficheros que mapean las clases a las tablas de la base de datos .En nuestro ejemplo introductorio únicamente tenemos un fichero el fichero Libro.xml (ver imagen).



Vamos a mostrar a continuación el contenido de este fichero y analizar su contenido.

Código 10.11(Libro.xml):

```
<?xml version="1.0"?>
<hibernate-mapping package="com.arquitecturajava.aplicacion.bo">
    <class name="Libro" table="Libros">
        <id name="isbn" type="string" />
        <property name="titulo" type="string" column="titulo" />
        <property name="categoria" type="string" column="categoria" />
    </class>
</hibernate-mapping>
```

Como podemos ver el fichero xml no es difícil de entender , ya que simplemente mapea cada una de las propiedades que tiene nuestra clase con una columna de la tabla de la base de datos. En este caso únicamente disponemos de las siguientes propiedades.

Código 10.12(Libro.xml):

```
<property name="titulo" type="string" column="titulo" />
<property name="categoria" type="string" column="categoria" />
```

Para complementar el mapeo Hibernate necesita saber cuál de las propiedades de la clase es definida como clave primaria. Para ello el framework usa la etiqueta `<id>` que identifica una propiedad como clave primaria.

Código 10.13(Libro.xml):

```
<id name="isbn" type="string" />
```

Una vez realizados estos pasos, podemos pasar a construir un programa sencillo que use Hibernate como framework de persistencia.

5. Insertar Libro con Hibernate

Como punto de partida vamos a crear un programa que se encarga de crear un objeto Libro y salvarlo en la base de datos Hibernate, a continuación se muestra el código:

Código 10.14(Principal.java):

```
public static void main(String[] args) {
    Session session=null;
    Transaction transaccion =null;
    try {
        SessionFactory factoria = new Configuration().configure()
            .buildSessionFactory();
        session = factoria.openSession();
        transaccion = session.beginTransaction();
        Libro l= new Libro("1","java","programacion");
        session.saveOrUpdate(l);
        transaccion.commit();
    } catch (HibernateException e) {
        System.out.println(e.getMessage());
        transaccion.rollback();
    } finally {
        session.close();}
```

Arquitectura Java

Vamos a explicar línea a línea el bloque de código construido para que paulatinamente nos vayamos familiarizando con los distintos conceptos.

Código 10.15(Principal.java):

```
SessionFactory factoria =new Configuration().configure().buildSessionFactory();
```

Esta primera línea de código se encarga de leer el fichero hibernate.cfg.xml y configurar el acceso a datos para el framework. Una vez configurado el acceso, nos devuelve un objeto SessionFactory que es un objeto único a nivel del framework y será el encargado de darnos acceso al resto del API de Hibernate. Una vez disponemos del objeto SessionFactory, invocamos el método openSession(), como se muestra a continuación.

Código 10.16(Principal.java):

```
Session session = factoriaSession.openSession();
```

Este método nos devuelve una sesión que se puede entender como una conexión clásica a la base de datos. Una vez construida una sesión, ya disponemos de una conexión que nos permitirá salvar objetos en la base de datos. Para ello nos construiremos un primer objeto.

Código 10.17(Principal.java):

```
Libro libro= new Libro (“1”, “java”, “programacion”);
```

Creado este nuevo objeto, usaremos el objeto sesión para comenzar una nueva transacción.

Código 10.18(Principal.java):

```
transaccion= session.beginTransaction();
```

Una vez que nos encontramos dentro de una transacción, solicitamos al objeto session que salve el objeto usando el método saveOrUpdate.

Código 10.19(Principal.java):

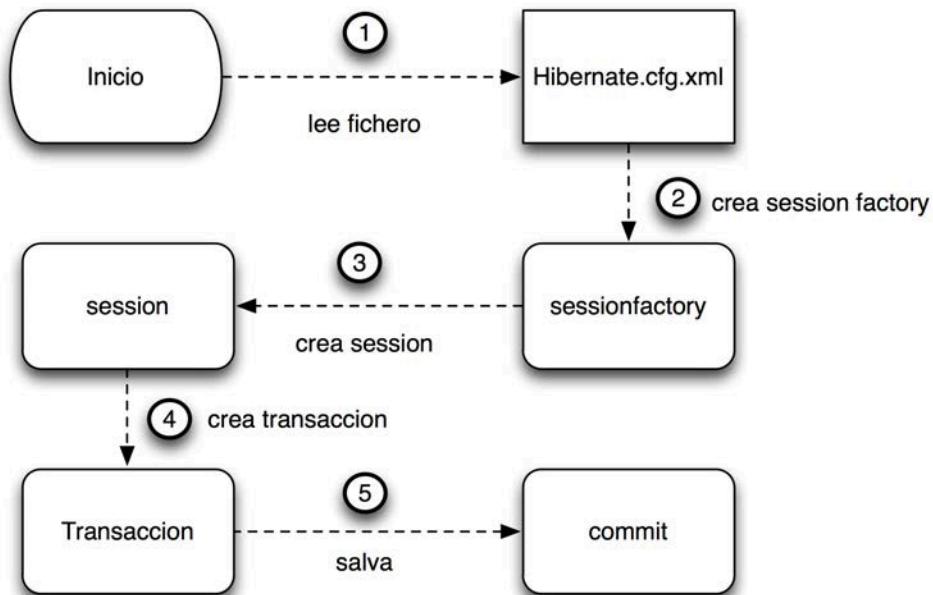
```
session.saveOrUpdate(libro);
```

Para ello el framework Hibernate se apoyará en los ficheros de mapeo construidos (Libro.xml) y que están disponibles a través del objeto Session, encargándose de aplicar el mapeo definido. Una vez realizada la operación, finalizamos la transacción.

Código 10.20(Principal.java):

```
session.getTransaction().commit();
```

Hemos salvado el objeto en la base de datos y hemos visto las clases fundamentales del framework. A continuación el diagrama nos lo resume.



Una vez realiza esta operación de inserción de libros en la base de datos, vamos a ver cómo podemos usar Hibernate para realizar otras operaciones complementarias.

6. Seleccionar Objetos con Hibernate

Vamos a construir un nuevo ejemplo con el framework Hibernate que se encargue de seleccionar todos los libros de la base de datos, para ello construiremos el siguiente programa main.

Código 10.21(Principal.java):

```
public static void main(String[] args) {
    Session session=null;
    try {
        SessionFactory factoria = new Configuration().configure()
            .buildSessionFactory();
        session= factoria.openSession();
        Query consulta= session.createQuery("from Libro libro");
        List<Libro> lista= consulta.list();
        for(Libro l: lista) {
            System.out.println(l.getIsbn());
            System.out.println(l.getTitulo());
            System.out.println(l.getCategoría());
        }
    }catch(HibernateException e) {
        System.out.println(e.getMessage());
    }finally {
        session.close();
    }
}
```

Este ejemplo tiene fuerte similitudes con el ejemplo anterior la única diferencia importante es que aparece un nuevo concepto el de **Query**.

Query : El objeto Query hace referencia en Hibernate a una consulta que lanzamos contra la base de datos .Eso si en vez de ser una consulta SQL se trata de una consulta HQL (**Hibernate Query Language**) lenguaje propietario de Hibernate que nos permite trabajar con el mismo tipo de sentencias sin importarnos el motor de base de datos que utilicemos .Más adelante, esta consulta será transformada en consulta SQL para el motor de base de datos que se haya seleccionado como dialecto en el fichero hibernate.cfg.xml. Una vez tenemos claro cuál es el concepto de Query, vamos a analizar nuestra consulta

Código 10.22(Principal.java):

```
Query consulta= session.createQuery("from Libro libro");
```

En este caso se trata de una consulta sencilla que usa HQL para seleccionar todos los libros de la tabla usando un alias. Una vez hemos creado la consulta, invocamos al método list() del objeto Query que nos devuelve una lista de objetos a recorrer.

Código 10.23(Principal.java):

```
List<Libro> lista= consulta.list();
```

Dicha lista la recorremos con un sencillo bucle for

Código 10.24(Principal.java):

```
for(Libro l: lista) {
    System.out.println(l.getIsbn());
    System.out.println(l.getTitulo());
    System.out.println(l.getCategoría());
}
```

Este bucle for nos presentará la información por la consola.

```
Problems @ Javadoc Declaration Console Servers
<terminated> Principal2 [Java Application] /usr/lib/jvm/java-6-openjdk/
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation.
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder
Hibernate: select libro0_.isbn as isbn0_, libro0_.titulo a
1
Java
Programacion
2
NET
Programacion
```

Hasta este momento hemos visto cómo insertar libros o cómo seleccionar una lista de éstos ; ahora vamos a seguir trabajando con el framework.

7. Seleccionar un único Objeto

El siguiente ejemplo nos mostrará cómo seleccionar un único registro de la tabla usando Hibernate.

Arquitectura Java

Código 10.25(Principal.java):

```
public static void main(String[] args) {  
    Session session=null;  
    try {  
        SessionFactory factoria = new Configuration().configure()  
            .buildSessionFactory();  
        session= factoria.openSession();  
        Libro libro=(Libro) session.get(Libro.class,"1");  
        System.out.println(libro.getTitulo());  
    }catch(HibernateException e) {  
        System.out.println(e.getMessage());  
    }finally {  
        session.close();  
    }  
}
```

En este ejemplo que acabamos de construir lo único destacable es cómo Hibernate se encarga de seleccionar un único objeto a través de la siguiente línea.

Código 10.26(Principal.java):

```
Libro libro=(Libro) session.get(Libro.class,"1");
```

Una vez seleccionado, simplemente imprimimos sus datos por pantalla.

Código 10.27(Principal.java):

```
System.out.println(libro.getTitulo());
```

Quedan por construir únicamente dos operaciones más que nos vayan a ser útiles a la hora de hacer evolucionar nuestra aplicación: las operaciones de borrar y filtrar.

8. Borrar un objeto

En el caso de borrar no vamos a profundizar mucho porque es tan sencillo como invocar el método delete del objeto sesión una vez obtenido el objeto a través de Hibernate; a continuación se muestra el código:

Código 10.28(Principal.java):

```
Session session=null;
try {
    SessionFactory factoria = new Configuration().configure()
        .buildSessionFactory();
    session= factoria.openSession();
    Libro libro=(Libro) session.get(Libro.class,"1");
    session.delete(libro);
}catch(HibernateException e) {
    System.out.println(e.getMessage());
}finally {
    session.close();
}
```

9. Filtrar objetos con Hibernate

Por último, como introducción al framework, vamos a ver cómo se puede lanzar una consulta con HQL que reciba parámetros y filtre la selección de datos.

Código 10.29(Principal.java):

```
public static void main(String[] args) {
    Session session = null;
    try {
        SessionFactory factoria = new Configuration().configure()
            .buildSessionFactory();
        session = factoria.openSession();
        Query consulta = session
            .createQuery(" from Libro libro where
                libro.categoría=:categoria");
        consulta.setString("categoria", "programacion");
        List<Libro> lista = consulta.list();
        for (Libro l : lista) {
            System.out.println(l.getIsbn());
            System.out.println(l.getTitulo());
            System.out.println(l.getCategoría());
        }
    } catch (HibernateException e) {
        System.out.println(e.getMessage());
    } finally {
        session.close();
    }
}
```

Arquitectura Java

Como podemos ver, esta consulta se encarga de localizar todos los libros que están dentro de una categoría determinada. Para ello se construye una consulta en HQL y se le asigna un parámetro: categoría a la cual se le asigna un valor.

Código 10.30(Principal.java):

```
Query consulta = session.createQuery(" from Libro libro where  
libro.categoría=:categoría");  
consulta.setString("categoría", "programación");
```

Una vez hecho esto, invocamos como en casos anteriores al método list() del objeto Query y recorremos la lista.

Código 10.31(Principal.java):

```
List<Libro> lista = consulta.list();  
for (Libro l : lista) {  
    System.out.println(l.getisbn());  
    System.out.println(l.getTitulo());  
    System.out.println(l.getCategoría());  
}
```

Con esta breve introducción al framework Hibernate tendremos suficiente para hacer frente a los cambios de nuestra aplicación, que se abordarán a continuación.

10. Migrar nuestra aplicación a Hibernate

Es momento de ver cómo podemos encajar el framework dentro de nuestra aplicación. En primer lugar vamos a construir una nueva clase que substituya a la clase DataBaseHelper y se encargue de inicializar el framework Hibernate. Esta clase se denominará HibernateHelper y inicializará el framework. A continuación se muestra su código.

Código 10.32 (Principal.java):

```
package com.arquitecturajava.aplicacion;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateHelper {
private static final SessionFactory sessionFactory = buildSessionFactory();
private static SessionFactory buildSessionFactory() {
    try {
        return new Configuration().configure().buildSessionFactory();
    }
    catch (Throwable ex) {
        System.err.println("Error creando una factoria de session." + ex);
        throw new ExceptionInInitializerError(ex);
    }
}
public static SessionFactory getSessionFactory() {
    return sessionFactory;
}
}
```

Esta sencilla clase nos permitirá obtener de forma directa el objeto SessionFactory, que inicializa el framework , como se muestra a continuación.

Código 10.33 (Principal.java):

```
SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
Session session = factoriaSession.openSession();
Libro libro=(Libro) session.get(Libro.class,isbn);
session.close();
```

Una vez construida la clase HibernateHelper, es momento de cambiar nuestra clase Libro para que se apoye completamente en el API de Hibernate. Después de esto, podremos eliminar la clase DataBaseHelper de nuestra aplicación. A continuación se muestra el código fuente de la clase Libro actualizado para que se apoye en Hibernate.

Arquitectura Java

Código 10.34 (Libro.java):

```
package com.arquitecturajava.aplicacion;
//omitimos imports
public class Libro {

    private String isbn;
    private String titulo;
    private String categoria;

    @Override
    public int hashCode() {
        return isbn.hashCode();
    }
    @Override
    public boolean equals (Object o) {
        String isbnLibro= ((Libro)o).getIsbn();
        return isbnLibro.equals(isbn);

    }
    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getCategoría() {
        return categoria;
    }
    public void setCategoría(String categoria) {
        this.categoría = categoria;
    }
    public Libro(String isbn) {
        super();
        this.isbn = isbn;
    }
    public Libro() {
        super();
    }
    public Libro(String isbn, String titulo, String categoria) {
        super();
        this.isbn = isbn;
        this.titulo = titulo;
        this.categoría = categoria;
    }
}
```

```

}

@SuppressWarnings("unchecked")
public static List<Libro> buscarTodasLasCategorias() {

    SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
    Session session = factoriaSession.openSession();
    String consulta="select distinct libro.categoría from Libro libro";
    List<Libro> listaDeCategorias = session.createQuery(consulta).list();
    session.close();
    return listaDeCategorias;
}
public void insertar() {

    SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
    Session session = factoriaSession.openSession();
    session.beginTransaction();
    session.save(this);
    session.getTransaction().commit();
}
public void borrar(){

    SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
    Session session = factoriaSession.openSession();
    session.beginTransaction();
    session.delete(this);
    session.getTransaction().commit();
}
public void salvar() {

    SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
    Session session = factoriaSession.openSession();
    session.beginTransaction();
    session.saveOrUpdate(this);
    session.getTransaction().commit();
}
@SuppressWarnings("unchecked")
public static List<Libro> buscarTodos() {
    SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
    Session session = factoriaSession.openSession();
    List<Libro> listaDeLibros = session.createQuery(" from Libro libro").list();
    session.close();
    return listaDeLibros;
}
public static Libro buscarPorClave(String isbn) {
    SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
    Session session = factoriaSession.openSession();
    Libro libro=(Libro) session.get(Libro.class,isbn);
    session.close();
    return libro;
}
}

```

```
@SuppressWarnings("unchecked")
public static List<Libro> buscarPorCategoria(String categoria) {

    SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
    Session session = factoriaSession.openSession();
        Query consulta=session.createQuery(" from Libro libro where
libro.categoría=:categoría");
    consulta.setString("categoría", categoría);
    List<Libro> listaDeLibros = consulta.list();
    session.close();
    return listaDeLibros;
}

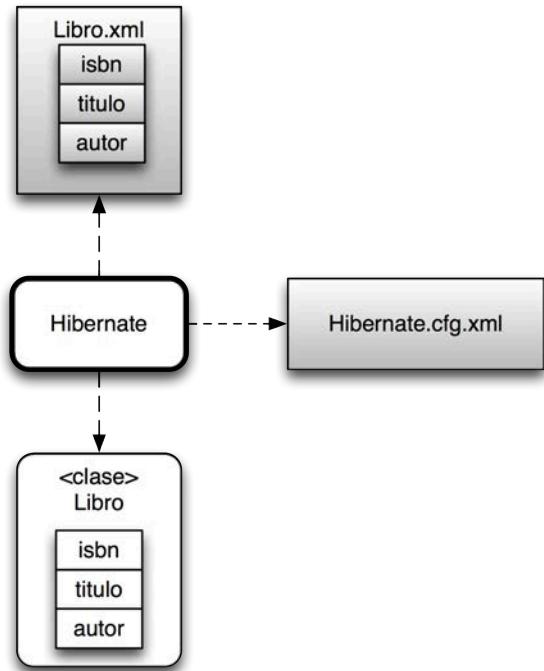
}
```

Una vez realizadas estas modificaciones, nuestra clase Libro se apoya completamente en el framework de persistencia para realizar cualquier operación. La estructura física del proyecto ha añadido los siguientes elementos en un nuevo paquete denominado bo (business objects).



11. Hibernate y Convención sobre Configuración.

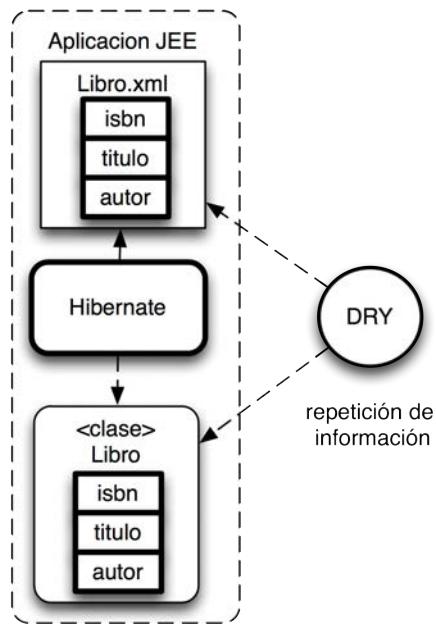
Hemos migrado nuestra aplicación de un sistema de persistencia elemental que usaba una clase java (DataBaseHelper) al uso de Hibernate. Para ello hemos tenido que modificar la implementación de parte de nuestras clases, así como añadir una serie de ficheros de configuración del framework adicionales, como se muestra en la siguiente figura.



Aunque nos pueda parecer que estamos usando Hibernate de la forma adecuada y que prácticamente no tenemos código repetido en nuestra aplicación, vemos que lo que si tenemos repetido es la **información** respecto a los conceptos que estamos manejando a nivel de capa de persistencia .En este caso estamos repitiendo la información sobre el concepto de libro en las siguientes ubicaciones.

- Clase Java
- Fichero de configuración (Libro.xml)
- Tabla Libro (Mysql)

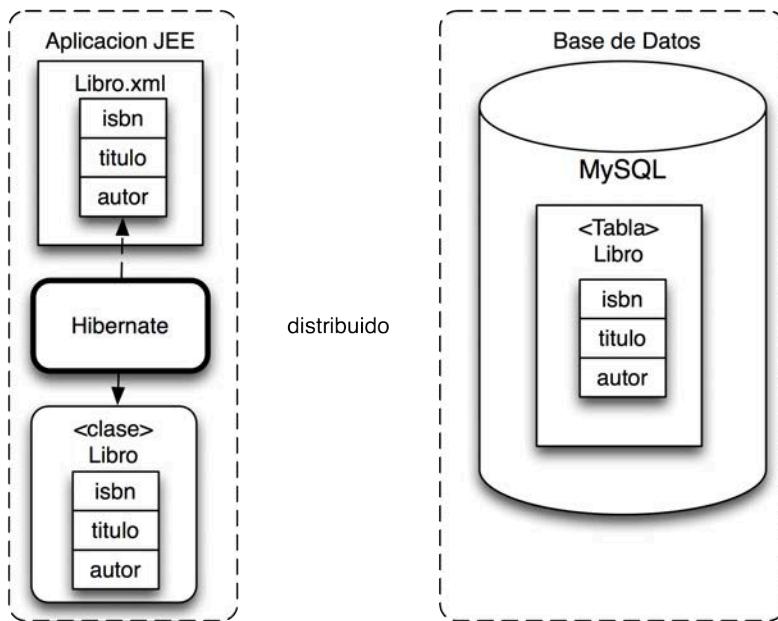
Este problema está ligado con el principio DRY ya que, aunque no se trata de repetición de código, si se trata de repetición de **información** a nivel de nuestra aplicación (ver imagen).



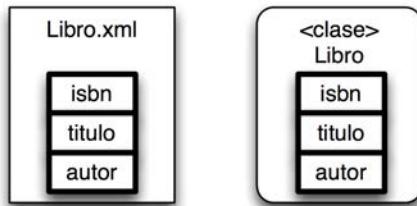
El objetivo principal de la siguiente tarea será eliminar las repeticiones de información de nuestra aplicación en cuanto a la capa de persistencia se refiere.

12. DRY e Hibernate

En principio no podemos aplicar una solución sencilla a la repetición de información que tenemos entre la aplicación JEE y la base de datos, ya que al tratarse de sistemas independientes ambos deben almacenar la información relativa a nuestro negocio (ver imagen).



Sin embargo sí que podemos avanzar en solventar la repetición de información que existe a nivel de nuestra aplicación JEE en la cuál tenemos la misma información almacenada a nivel de las clases Java y de los ficheros xml de configuración del framework (ver imagen).



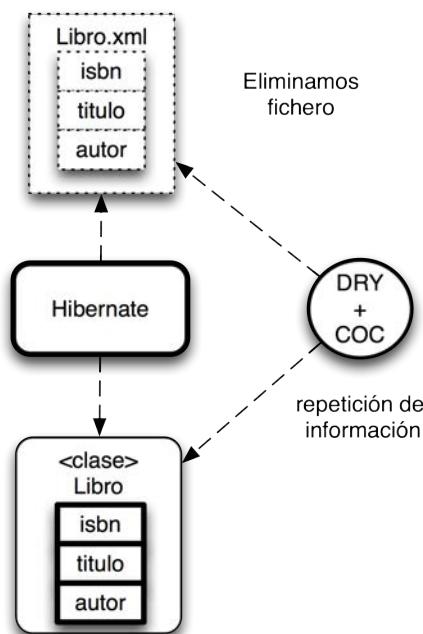
La repetición de código, ya sabemos por capítulos anteriores, conlleva graves problemas de mantenimiento .Lo mismo sucede con la repetición de información. Nos basta por ejemplo con referirnos a las clásicas reglas de normalización de una base de datos para saber los problemas que se generan. Sin embargo aquí no podemos aplicar estas reglas de normalización, ya que nos encontramos no con una base de datos sino con código Java. Así pues es momento de introducir otro principio de ingeniería de software que apoya al principio DRY. El principio de Convención sobre Configuración o COC (Convention Over Configuration).

COC :El principio COC define que, antes de abordar el desarrollo, un programador puede declarar una serie de convenciones que le permiten asumir una configuración por defecto del sistema.

Quizá esta definición sea algo abstracta y difícil de entender en un principio . Un ejemplo es la forma de percibirlo más claramente. Vamos a usar el principio COC en nuestra aplicación definiendo la siguiente convención.

Los campos de nuestras clases java de negocio (Libro) siempre serán idénticos a los campos de la tabla de la base de datos donde se guarden los objetos de ésta.

El hecho de aplicar este concepto a nuestra aplicación implicará que el fichero XML que construimos por cada clase de persistencia no será a partir de ahora necesario, ya que al aplicar esta convención se sobreentiende que los campos y propiedades coinciden. Consecuentemente, como la imagen muestra, este fichero ya no es necesario.



Sin embargo recordemos que este fichero no se encargaba únicamente de mapear los campos, sino que además se encargaba de definir cuál de todas las propiedades de la clase se identificada como clave primaria . A continuación se muestra el fichero en forma de recordatorio.

Código 10.35 (Libro.xml):

```
<class name="Libro" table="Libros">
    <id name="isbn" type="string" />
    <property name="titulo" type="string" column="titulo" />
    <property name="categoria" type="string" column="categoria" />
</class>
```

Así pues no podemos eliminar este fichero hasta que reubiquemos la información sobre las claves primarias en algún otro lugar. Para ello haremos uso del sistema de anotaciones que soporta java y que permite añadir información adicional a nuestras clases.

Un ejemplo claro de este tipo de información es la anotación @Override, que nos permite definir cómo un método ha de ser sobrecargado . A continuación se muestra un ejemplo de código.

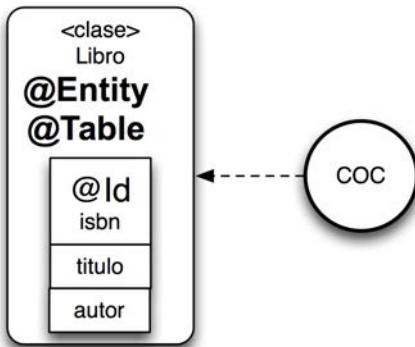
Código 10.36:

```
@Override
public void metodo() {
}
```

Apoyándonos en esta idea, podemos volver a Hibernate. Hibernate soporta otro conjunto de anotaciones orientadas a facilitar la persistencia de nuestros objetos. Vamos a hacer uso de estas anotaciones.

- **@Entity** : Identifica a una clase como clase a persistir
- **@Id**: Identifica la propiedad de la clase como clave primaria de la tabla
- **@Table**: Identifica cuál es el nombre de la tabla contra la que se mapea la clase.

El uso de este conjunto de anotaciones nos permitirá eliminar los ficheros xml de mapeo que hemos usado hasta ahora .



A continuación se muestra el código fuente de nuestra clase.

Código 10.37 (Libro.xml):

```
@Entity
@Table(name="Libros")
public class Libro {
    @Id
    private String isbn;
    private String titulo;
    private String categoria;
    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getCategoria() {
        return categoria;
    }
    //resto de metodos
```

Como podemos ver, ya no necesitamos el fichero de mapeo, es suficiente con el uso de anotaciones. El modelo de anotaciones nos obliga también a realizar algunas modificaciones a nivel del fichero Hibernate.cfg.xml , vamos a verlo.

Código 10.38 (Hibernate.cfg.xml):

```
<hibernate-configuration><session-factory>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">
        jdbc:mysql://localhost/arquitecturaJavaORM
    </property>
    <property name="connection.username">root</property>
    <property name="connection.password">java</property>
    <property name="connection.pool_size">5</property>
    <property name="dialect">
        org.hibernate.dialect.MySQL5Dialect
    </property>
    <mapping class="com.arquitecturajava.aplicacion.bo.Libro" />
</session-factory>
</hibernate-configuration>
```

Resumen

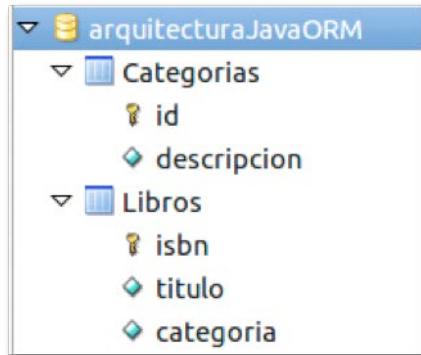
Este capítulo ha servido de introducción al framework Hibernate . Hemos aplicado el principio COC para configurar de una forma mas cómoda el framework a la hora de usarlo en nuestra aplicación.

11.Hibernate y Relaciones

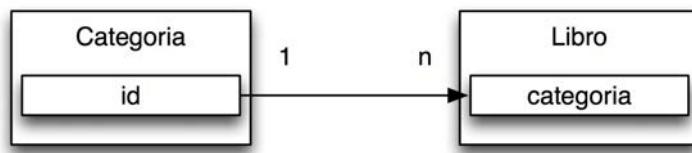
Hemos modificado nuestra aplicación para que se apoye en el framework Hibernate a la hora de gestionar la capa de persistencia. Sin embargo aún nos quedan bastantes puntos a la hora de trabajar con un framework de persistencia .Quizás lo mas destacable del uso de un framework de persistencia es que nos permite construir modelos más complejos y flexibles donde existan varias clases y se construyan relaciones entre las mismas. En estos momentos, nuestro caso parte de un modelo elemental donde únicamente tenemos una tabla en la base de datos , tal como se muestra en la siguiente imagen.



Es momento de construir un modelo más sólido entidad-relación en el cuál el framework Hibernate se pueda apoyar. Así pues modificaremos la estructura de la base de datos para que aparezca una nueva tabla, la tabla Categorías (ver imagen).



De esta forma, ambos conceptos están relacionados a través del id de la tabla categoría y del campo categoría de la tabla Libros. Una vez construidas ambas tablas, éstas quedarán relacionadas a través de una clave externa (ver imagen).



Una vez modificado el modelo de tablas, debemos encargarnos de realizar las modificaciones oportunas en nuestra aplicación para que soporte un modelo más complejo donde las entidades se relacionan, éste será el objetivo principal del capítulo. A continuación se listan los objetivos y tareas del mismo.

Objetivos:

Evolucionar la aplicación JEE para que soporte un modelo de objetos complejo a nivel de capa de persistencia.

Tareas:

1. Construir la clase Categoría y mapearla
2. Modificar la capa de presentación para soportar la clase Categoría
3. Creación de relaciones entre clases
4. Relaciones y persistencia con Hibernate
5. Relaciones y capa de presentación

1. Construir la clase categoría y mapearla

Nos encontraremos que, igual que existen dos tablas en la base de datos, deberemos tener también en principio dos clases en nuestro modelo de objetos de negocio (ver imagen)



Así pues debemos crear una nueva clase en nuestra aplicación que se encargará de gestionar el concepto de Categoría. Vamos a ver su código.

Código 11.1: (Categoria.java)

```

package com.arquitecturajava.aplicacion.bo;
//omitimos imports
@Entity
@Table(name = "Categorias")
public class Categoria {
    @Id
    private String id;
    private String descripcion;
    public int hashCode() {
        return id.hashCode();
    }
    @Override
    public boolean equals (Object o) {
        String categoriald= ((Categoria)o).getId();
        return id.equals(categoriald);
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getDescripcion() {
        return descripcion;
    }
    public void setDescripcion(String descripcion) {
        this.descripcion = descripcion;
    }
    public static List<Categoria> buscarTodos() {
        SessionFactory factoriaSession =
            HibernateHelper.getSessionFactory();
        Session session = factoriaSession.openSession();
        List<Categoria> listaDeCategorias = session.createQuery(
            " from Categoria categoria").list();
        session.close();
        return listaDeCategorias;
    }
}

```

Una vez creada esta clase, observaremos que comparte el mismo sistema de anotaciones que tiene la clase libro y deberemos añadirla a la zona de clases mapeadas que tiene el fichero hibernate.cfg.xml para que sea incluida como clase de persistencia. Hemos además sobrecargado el método equals y hashCode, lo cuál evitara problemas a la hora de comparar objetos de la misma clase. A continuación se muestra el código que debemos añadir a nuestro fichero de configuración para dar de alta esta nueva clase.

Código 11.2: (hibernate.cfg.xml)

```
<hibernate-configuration>
<session-factory>
    <!--resto de propiedades-->
    <mapping class="com.arquitecturajava.aplicacion.bo.Libro" />
    <mapping class="com.arquitecturajava.aplicacion.bo.Categoría" />
</session-factory>
</hibernate-configuration>
```

2. Modificar la Capa de presentación y soportar la clase Categoría

Una vez hecho esto, es necesario modificar algunas de nuestras acciones para que se hagan eco de los cambios recién introducidos, ya que ahora el método de buscar todas las categorías se encuentra ubicado en la clase Categoría. Vamos a ver como este cambio afecta a la clase FormularioInsertarLibroAccion.

Código 11.3: (formularioInsertarLibroAccion.xml)

```
public class FormularioInsertarLibroAccion extends Accion {
    @Override
    public String ejecutar(HttpServletRequest request,
                          HttpServletResponse response) {
        List<Categoría> listaDeCategorias = null;
        listaDeCategorias = Categoría.buscarTodos();
        request.setAttribute("listaDeCategorias", listaDeCategorias);
        return "FormularioInsertarLibro.jsp";
    }
}
```

Como podemos ver, la clase de acción hace uso de la nueva clase Categoría a la hora de buscar todas las categorías de la aplicación. Este cambio afectará de igual modo a otras acciones .Una vez modifiquemos todas las acciones afectadas, deberemos también modificar las distintas páginas JSP que hacen uso de la lista de categorías. A continuación se muestra cómo varía el código de la página FormularioInsertarLibro.jsp.

Código 11.4: (MostrarLibros.jsp)

```
<form id="formularioInsercion" action="InsertarLibro.do">
<fieldset><legend>FormularioaltaLibro</legend>
<p><label for="isbn">ISBN:</label><input type="text" id="isbn"
   name="isbn" /></p>
<p><label for="titulo">Título:</label><input type="text"
   id="titulo" name="titulo" /></p>
<p><label for="categoria">Categoria :</label>
<select name="categoria">
    <c:forEach var="categoria" items="${listaDeCategorias}">
        <option value="${categoria.id}">
            ${categoria.descripcion}
        </option>
    </c:forEach>
</select><br />
</p>
<p><input type="submit" value="Insertar" /></p>
</fieldset>
</form>
```

Este cambio afectará tanto a los formularios de edición como a la página MostrarLibros.jsp .Ahora bien, en esta página tendremos algunos problemas añadidos con el siguiente bloque de código.

Código 11.5: (MostrarLibros.jsp)

```
<c:forEach var="libro" items="${listaDeLibros}">
    ${libro.isbn}
    ${libro.titulo}
    ${libro.categoría}
    <a href="BorrarLibro.do?isbn=${libro.isbn}">borrar</a>
    <a href="FormularioEditarLibro.do?isbn=${libro.isbn}">editar</a>
    <br />
</c:forEach>
```

Aunque en un principio nos parezca que no existe ningún cambio a este nivel, si ejecutamos la página, podremos percatarnos de que el resultado es distinto a los ejemplos del capítulo anterior, ya que no aparece el nombre de la categoría sino su id tal como se ve en la siguiente imagen.



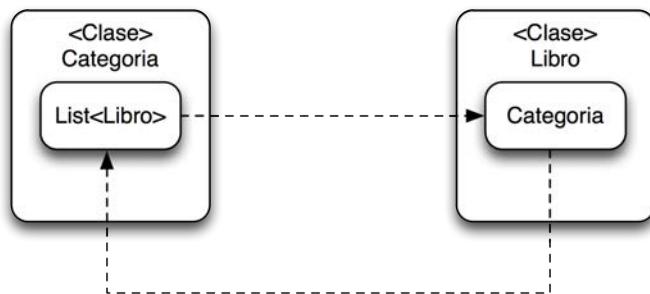
Es aquí donde podemos observar que, aunque hemos construido un modelo más sólido, quedan pasos por realizar. El modelo tiene sus limitaciones y tendremos que avanzar un paso más para superarlas. Para ello, la siguiente tarea se encargará de definir relaciones entre las distintas clases.

3. Creación de relaciones entre clases

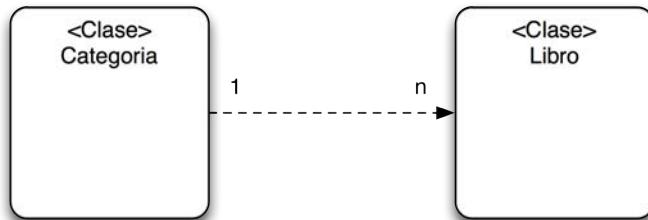
En estos momentos nuestras dos clases Categoría y Libro no están relacionadas entre sí. Es momento de modificar nuestro código fuente y crear las siguientes relaciones entre ambas clases:

- **Una Categoría contiene varios Libros:** Relación de 1 a n en la que una categoría contiene un conjunto de libros
- **Un Libro pertenece a una categoría:** Relación de n a 1 en la que cada Libro pertenece a una categoría concreta

El siguiente diagrama muestra cómo se relacionan ambas clases.



Para que esta relación exista a nivel de código Java, la clase Categoría debe incluir un `List<Libro>` de Libros y la clase Libro debe incluir una variable de tipo Categoría, como el siguiente diagrama muestra.



Una vez tenemos claras estas ideas, es momento de pasar a mostrar el código fuente de ambas clases para ver como estas dos relaciones se implementan.

Código 11.6: (Categoria.java)

```
@Entity
@Table(name = "Categorias")
public class Categoria {
    @Id
    private int id;
    private String descripcion;
    private List<Libro> listaDeLibros;
    public List<Libro> getListaDeLibros() {
        return listaDeLibros;
    }
    public void setListaDeLibros(List<Libro> listaDeLibros) {
        this.listaDeLibros = listaDeLibros;
    }
    // resto de codigo
```

Código 11.7: (Libro.java)

```
@Entity
@Table(name="Libros")
public class Libro {
    @Id
    private String isbn;
    private String titulo;
    private Categoria categoria;

    public Categoria getCategoria() {
        return categoria;
    }
    public void setCategoria(Categoria categoria) {
        this.categoria = categoria;
    }
//resto de codigo
```

Este es un primer paso a la hora de crear relaciones entre nuestras clases. Sin embargo esta primera parte no crea relaciones a nivel de persistencia, sino únicamente a nivel de memoria. La siguiente tarea se apoyará en el uso de anotaciones para que las relaciones que hayamos construido tengan sentido también a este otro nivel.

4. Relaciones y persistencia con Hibernate

Hibernate nos provee de un conjunto nuevo de anotaciones orientadas a crear relaciones entre las distintas clases java y asegurar su persistencia. Vamos a enumerar a continuación algunas de las anotaciones más importantes:

- **@OneToMany** : Anotación que se encarga de crear una relación de 1 a n entre dos clases predeterminadas (Categoría y Libro en nuestro ejemplo).
- **@ManyToOne** :Anotación que se encarga de crear una relación de muchos a uno entre dos clases predeterminadas (Libro y Categoría en nuestro ejemplo).
- **@JoinColumn**: Anotación que sirve a para discernir qué columna de la tabla de la base de datos se usa como clave externa o foránea a la hora de aplicar la relación.

Aclarado este punto , vamos a ver como se aplican las anotaciones a nivel de código.

Código 11.8: (Libro.java)

```
@Entity
@Table(name="Libros")
public class Libro {
    @Id
    private String isbn;
    private String titulo;
    @ManyToOne
    @JoinColumn (name="categoria")
    private Categoria categoria;
}
```

Código 11.9: (Categoria.java)

```
@Entity
@Table(name="Categorias")
public class Categoria {
    @Id
    private int id;
    private String descripcion;
    @OneToMany
    @JoinColumn(name = "categoria")
    private List<Libro> listaDeLibros;
```

Tras construir las relaciones a nivel de persistencia usando las anotaciones de Hibernate, vamos a construir un ejemplo sencillo que muestra cómo utilizarlas.

Código 11.10: (Principal.java)

```
SessionFactory factoriaSession=HibernateHelper.getSessionFactory();
Session session = factoriaSession.openSession();
List<Libro> listaDeLibros = session.createQuery("from Libro libro").list();
for(Libro l :listaDeLibros) {
    System.out.println(l.getTitulo());
    //accedemos a la categoria a traves de la relacion.
    System.out.println(l.getCategoria().getDescripcion());
}
session.close();
```

Como podemos ver, en un primer momento usamos el framework Hibernate y el lenguaje HQL para seleccionar todos los libros de los disponemos.

Arquitectura Java

Código 11.11: (Principal.java)

```
List<Libro> listaDeLibros = session.createQuery("from Libro libro").list();
```

Una vez obtenidos los libros, recorremos la lista con un bucle y vamos mostrando los distintos contenidos.

Código 11.12: (Principal.java)

```
for(Libro l :listaDeLibros) {  
    System.out.println(l.getIsbn());  
}
```

Ahora bien si revisamos la siguiente línea.

Código 11.13: (Principal.java)

```
System.out.println(l.getCategoría().getDescripción());
```

Nos podemos dar cuenta de que nos obliga a acceder a la información de descripción de nuestra categoría, información que previamente no hemos seleccionado con la consulta HQL .Recordemos que la consulta únicamente selecciona el concepto de Libro.

Código 11.14: (Principal.java)

```
"from Libro libro"
```

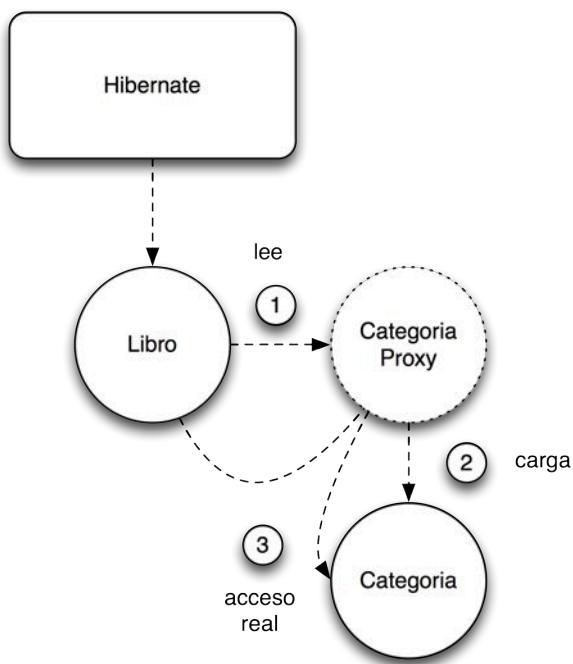
Para cumplir con nuestra petición Hibernate usará la información relativa a las relaciones que acabamos de construir y obtendrá los datos necesarios de la base de datos, como se muestra en la imagen al ejecutar el código.

```

SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details
Hibernate: select libro0_.isbn as isbn0_, libro0_.categoria as categoria0_, l...
Hibernate: select categoria0_.id as id1_0_, categoria0_.descripcion as descri...
Hibernate: select categoria0_.id as id1_0_, categoria0_.descripcion as descri...
1
Java
Web
2
.NET
Programacion

```

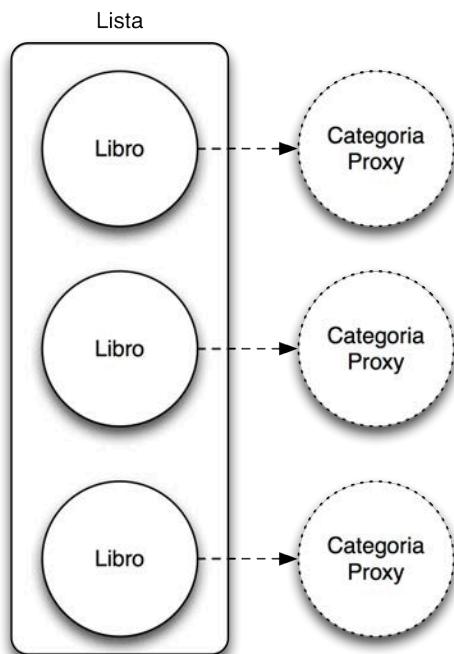
Aunque la aplicación funciona correctamente, no es sencillo entender cómo Hibernate consigue realizar esta operación. Vamos a comentar a grosso modo como funciona este framework de persistencia a la hora de realizar esta operación: Hibernate es capaz de realizar esta operación a través del concepto de Proxy que crean las anotaciones relativas a las relaciones. Un proxy es un objeto falso que simula ser un objeto real. En nuestro caso, Hibernate lee los datos de la clase Categoría y crea un objeto Categoría Proxy que simula ser el objeto original, como se muestra en el siguiente diagrama.



Vamos a comentar a continuación el diagrama

1. Hibernate accede al objeto Categoría, la cuál es un proxy
2. Hibernate crea un objeto Categoría real accediendo a la base de datos
3. Hibernate accede a los datos el objeto real y se los devuelve al Libro

Una vez tenemos claro como funciona el concepto de CategoriaProxy y que no se trata de un objeto real ,vamos a mostrar cuál es la estructura básica de memoria que se construye cuando ejecutamos la consulta “ select libro from libro” .A continuación se muestra una imagen.

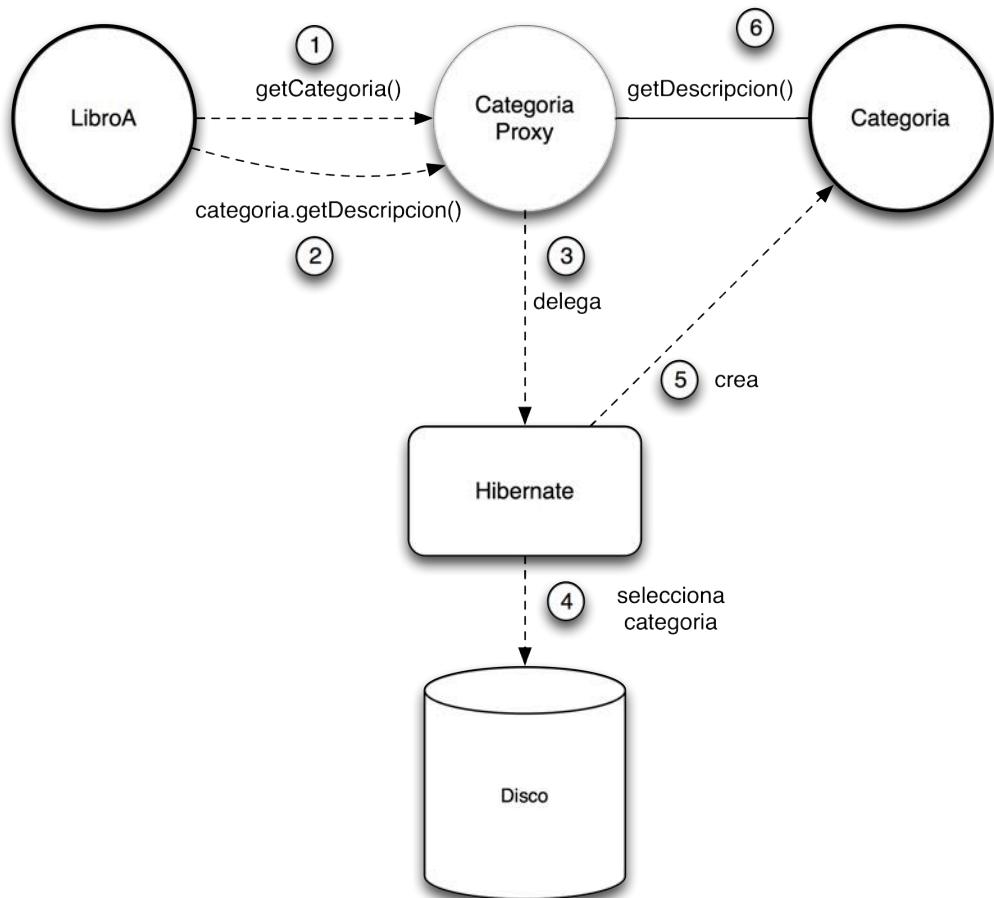


Es visible que se carga en memoria una lista de Libros ,sin embargo esta lista no contiene ningún objeto Categoría Real sino simplemente un conjunto de ProxiesCategoria que dan la sensación de ser los objetos reales. Ahora bien, cuando ejecutemos la siguiente línea de código

Código 11.15: (Principal.java)

```
System.out.println(l.getCategoria().getDescripcion());
```

Hibernate responderá a la llamada al método `getCategoria().getDescripcion()`, como se muestra en la figura.



Vamos a explicar detalladamente cada uno de los pasos

1. En el primer paso, el objeto `Libro` invoca al método `getCategoria()`, el cuál en principio nos devuelve un `ProxyCategoria`
2. Nuestro objeto libro invoca al método `getDescripcion()` del `ProxyCategoria`.

3. Al no estar ligado a un objeto Categoría real sino a un ProxyCategoria, éste recibe la petición y delega en Hibernate.
4. El framework Hibernate revisa las relaciones construidas y lanza una consulta de selección que selecciona la categoría a la que el libro pertenece
5. Una vez hecho esto, construye un objeto real Categoría con los datos obtenidos de la base de datos y lo vincula al ProxyCategoria
6. Por ultimo, el ProxyCategoria delega la invocación original del objeto Libro en la Categoría que se acaba de construir en memoria, devolviendo la descripción al Libro que la solicitó.

Esta forma de trabajar de Hibernate se conoce comúnmente como “carga vaga” o lazyload, ya que únicamente accede a la base de datos para obtener información sobre las relaciones entre los objetos cuando la aplicación cliente lo solicita y nunca antes.

5. Relaciones y capa de presentación

Una vez hecho esto modificaremos el código de la capa de presentación para que se apoye en el uso de relaciones como se muestra a continuación.

Código 11.16: (Principal.java)

```
<c:forEach var="libro" items="${listaDeLibros}">
    ${libro.isbn}${libro.titulo}${libro.categoría.descripcion}
    <a href="BorrarLibro.do?isbn=${libro.isbn}">borrar</a>
    <a href="FormularioEditarLibro.do?isbn=${libro.isbn}">editar</a>
    <br />
</c:forEach>
```

Como podemos ver, ahora usamos JSTL para acceder a la relación **\${libro.categoría.descripcion}**. Una vez realizada esta operación, los datos de libros y categorías se mostrarán correctamente.



Ahora bien, si revisamos las consultas SQL que Hibernate ha tenido que ejecutar, veremos que pueden ser mejorables .A continuación se muestra una imagen con las consultas ejecutadas.

```

Markers Properties Data Source Explorer Servers Snippets Console
Tomcat v7.0 Server at localhost [Apache Tomcat] /usr/lib/jvm/java-6-openjdk/bin/java (Oct 29, 2011)
Hibernate: select libro0_.isbn as isbn0_, libro0_.categoria as categoria0_
Hibernate: select categoria0_.id as id1_0_, categoria0_.descripcion as descr...
Hibernate: select categoria0_.id as id1_0_, categoria0_.descripcion as descr...
Hibernate: select categoria0_.id as id1_, categoria0_.descripcion as descr...

```

Como podemos ver, se realiza una consulta adicional por cada categoría que debe obtener de la base de datos. Así pues si tuviéramos cien categorías se ejecutarían cien consultas. En este caso hubiera sido mejor ejecutar una única consulta de tipo join que obtenga los datos de las dos tablas simultáneamente. Para ello vamos a usar la sintaxis de HQL para modificar la consulta y dejarla con el siguiente código.

Código 11.17: (Principal.java)

```
List<Libro> listaDeLibros = session.createQuery("from Libro libro right join fetch libro.categoría").list();
```

Esta cláusula rigth join fetch obliga a Hibernate a ejecutar una consulta de joins entre las dos tablas, cargando todos los datos en una única consulta (ver imagen).

```

Tomcat v7.0 Server at localhost [Apache Tomcat] /usr/lib/jvm/java-6-openjdk/bin/java (Oct 29, 2011)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details
Hibernate: select libro0_.isbn as isbn0_0_, categoria1_.id as id1_1_, libro0_.cat...
Hibernate: select categoria0_.id as id1_, categoria0_.descripcion as descripc2_1_

```

Arquitectura Java

Una vez optimizada esta consulta, es momento de volver a mirar hacia el código de nuestra aplicación y modificar el método que busca todos los libros para que cargue a través de un join todos los datos en una única consulta. Vamos a verlo.

Código 11.18: (Libro.java)

```
public static List<Libro> buscarTodos() {  
    SessionFactory factoriaSession = HibernateHelper.getSessionFactory();  
    Session session = factoriaSession.openSession();  
    List<Libro> listaDeLibros = session.createQuery("from Libro libro right join fetch  
libro.categoría").list();  
    session.close();  
    return listaDeLibros;  
}
```

Una vez creada esta consulta HQL, modificaremos la página MostrarLibros.jsp para que se encargue de mostrar las descripciones de las categorías apoyándose en las relaciones creadas con Hibernate y en la consulta optimizada que acabamos de definir. A continuación se muestra como queda el nuevo bloque de código.

Código 11.19: (MostrarLibros.jsp)

```
<c:forEach var="libro" items="#{listaDeLibros}">  
    ${libro.isbn}  
    ${libro.titulo}  
    ${libro.categoría.descripcion}  
    <a href="BorrarLibro.do?isbn=${libro.isbn}">borrar</a>  
    <a href="FormularioEditarLibro.do?isbn=${libro.isbn}">editar</a>  
    <br />  
</c:forEach>
```

A continuación se muestra el resultado.

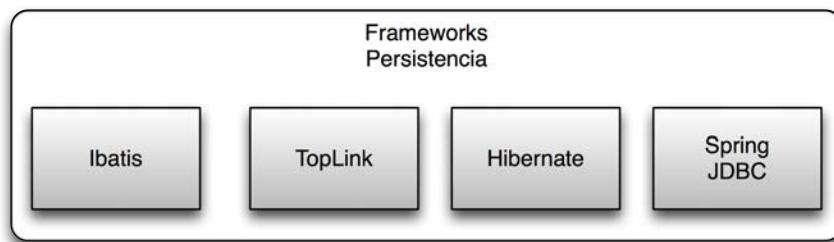


Resumen

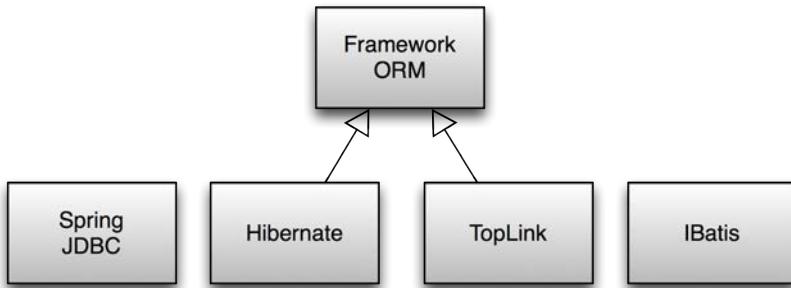
En este capítulo hemos mostrado como construir un modelo de persistencia más flexible construyendo relaciones entre los distintos objetos. Aunque hemos aumentado la flexibilidad, también hemos incrementado la complejidad a la hora de abordar un desarrollo, introduciendo características avanzadas de los frameworks de persistencia.

12. Java Persistence API

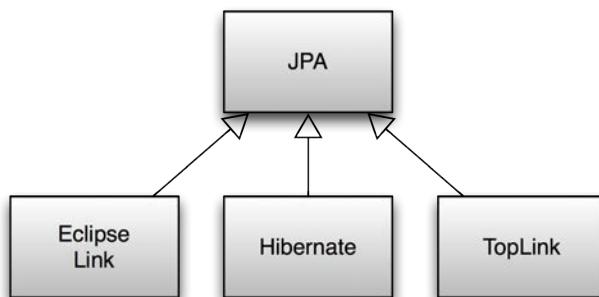
Durante muchos años en la plataforma J2EE han existido distintos frameworks de persistencia como Ibatis, Hibernate , TopLink etc (ver imagen).



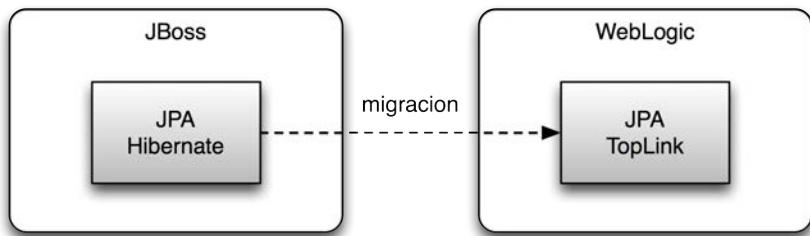
Cada uno de estos frameworks de persistencia ha utilizado una filosofía propia a la hora de decidir de qué forma se almacena la información en la base de datos. De este amplio conjunto de frameworks hay un subconjunto que son frameworks ORM (Object Relational Mapping) como TopLink y Hibernate. Algunos otros frameworks no pueden ser incluidos dentro de esta categoría, caso de IBatis o Spring JDBC, que no realizan un mapeo propiamente dicho (ver imagen).



A día de hoy la plataforma JEE se ha encargado de estandarizar la capa de persistencia a través de la creación del estándar JPA (**Java Persistence API**). Este estándar se ha basado fundamentalmente en las experiencias que los distintos fabricantes han tenido con los frameworks de ORM, dejando fuera a aquellos que no lo son. Por esa razón prácticamente la totalidad de los frameworks de ORM soportan el uso del api de JPA (ver imagen).



Así pues el siguiente paso que abordaremos en nuestra aplicación será la migración de la capa de persistencia del uso del API de Hibernate al uso del API de JPA sobre Hibernate. Así será prácticamente transparente migrar una aplicación de un framework de persistencia a otro como por ejemplo de Hibernate a TopLink. Ésto nos aportará flexibilidad a la hora de elegir fabricante para nuestro servidor de aplicaciones. Podemos empezar con una aplicación instalada en JBoss con Hibernate y migrarla de una forma relativamente sencilla a WebLogic con TopLink (ver imagen).



Seguidamente se detallan los objetivos y tareas de este capítulo.

Objetivos :

- Migrar la capa de persistencia de nuestra aplicación a JPA

Tareas:

1. Introducción al API de JPA
2. Migración de la aplicación a JPA.
3. Manejo de Excepciones y JPA

1. Introducción al API de JPA

Esta tarea se centrará en hacer evolucionar nuestra capa de persistencia creada con Hibernate a una nueva capa de persistencia que delega en Hibernate pero se apoya en el API de JPA para trabajar. El API de JPA es muy similar al API de Hibernate, ya que el estándar de JPA se apoyó en el éxito de Hibernate como framework para grandes partes de su diseño. A continuación se explican cuáles son las clases fundamentales que usa el API de JPA.

- **Persistence:** Hace referencia a la configuración de JPA y es similar a la clase Configurator de Hibernate.
- **EntityManagerFactory:** Factoría que se encarga de inicializar el framework, su rol es similar al SessionFactory de Hibernate que hemos utilizado en capítulos anteriores.
- **EntityManager:** Clase que se encarga de gestionar la persistencia de un conjunto de entidades similar al objeto Session del API de Hibernate.
- **EntityTransaction:** Clase que se encarga de definir el concepto de transacción, similar a Transaction en Hibernate.

- **TypedQuery:** Identifica una consulta definida con JPQL (Java Persistence Query Language) similar a la clase Query de Hibernate con la única salvedad que soporta el uso de genéricos.
- **Persistence.xml:** Fichero que reemplaza al fichero clásico de Hibernate “Hibernate.cfg.xml” y que contiene todos los parámetros necesarios para conectarnos contra un servidor de base de datos.

Una vez explicadas las clases principales, vamos a ver un ejemplo sencillo de cómo el api de JPA se encarga de buscar todos los libros de nuestra aplicación a través de un programa de consola.

Código 12.1: (Principal.java)

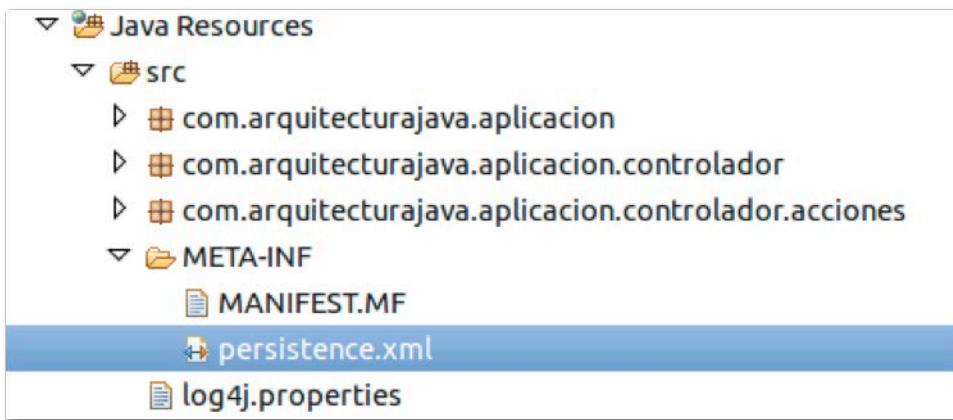
```
public static void main (String args[]) {
    EntityManagerFactory emf=
    Persistence.createEntityManagerFactory("arquitecturaJava");
    EntityManager em=emf.createEntityManager();
    TypedQuery<Libro> consulta= em.
        createQuery("Select l from Libro l",Libro.class);
    List<Libro> lista= consulta.getResultList();
    for(Libro l: lista) {
        System.out.println(l.getTitulo());
    }
}
```

Como podemos ver, el código es muy similar al usado en Hibernate. Vamos a comentarlo línea a línea.

Código 12.2: (Principal.java)

```
EntityManagerFactory
emf=Persistence.createEntityManagerFactory("arquitecturaJava");
```

Esta parte del código se encarga de crear un EntityManagerFactory, que es el objeto que inicializa todo lo necesario para que el estándar de JPA pueda trabajar. Para ello se apoyará en el fichero persistence.xml, que se muestra a continuación. Este fichero se encuentra ubicado en el directorio META-INF de nuestra aplicación (ver imagen).



A continuación se muestra el contenido de este fichero.

Código 12.3: (persistence.xml)

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
  <persistence-unit name="arquitecturaJava">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.arquitecturajava.aplicacion.bo.Libro</class>
    <class>com.arquitecturajava.aplicacion.bo.Categoría</class>
    <properties>
      <property name="hibernate.show_sql" value="true"/>
      <property name="javax.persistence.transactionType"
                value="RESOURCE_LOCAL" />
      <property name="javax.persistence.jdbc.driver"
                value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://localhost/arquitecturaJavaORM" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="java" />
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.MySQL5Dialect" />
    </properties>
  </persistence-unit>
</persistence>
```

Como podemos ver el fichero es muy similar al fichero clásico de configuración del framework Hibernate .Una vez configurado, ya podemos comenzar a usar JPA. Es momento hablar del el objeto EntityManager (ver código).

Código 12.4: (Principal.java)

```
EntityManager em=emf.createEntityManager();
```

Este objeto es el encargado de gestionar el ciclo de vida de las entidades y es capaz tanto de guardar objetos en la base de datos como de seleccionarlos a través de la creación de consultas JPQL (Java Persistence Query Language), como se muestra en la siguiente línea.

Código 12.5: (Principal.java)

```
TypedQuery<Libro> consulta= em.  
        createQuery("Select l from Libro l",Libro.class);  
List<Libro> lista= consulta.getResultList();
```

Una vez creada la consulta invocados al método getResultList() que nos devuelve una lista de Libros genérica. Por ultimo simplemente recorremos la lista y presentamos por consola el titulo de cada libro.

Código 12.6: (Principal.java)

```
for(Libro l: lista) {  
    System.out.println(l.getTitulo());  
}
```

Como complemento al ejemplo anterior, vamos a ver un ejemplo en el cual usamos JPA para persistir un objeto en la base de datos.

Código 12.7: (Principal.java)

```
public static void main(String[] args) {  
    EntityManagerFactory emf=  
        Persistence.createEntityManagerFactory("arquitecturaJava");  
    EntityManager manager = emf.createEntityManager();  
    Libro libro= new Libro ("2","java","programacion");  
    EntityTransaction tx = null;  
    tx=manager.getTransaction();  
    tx.begin();  
    manager.merge(libro);  
    tx.commit();  
    manager.close();  
}  
}
```

Como podemos ver, aparece un nuevo tipo de Objeto EntityTransaction que se encarga de ejecutar transacciones sobre JPA .La única diferencia con el ejemplo anterior es que usamos el metodo merge () de la clase EntityManager para guardar el objeto en la base de datos , método que es similar al saveOrUpdate que en capítulos anteriores usamos para Hibernate.

2. Migración de Aplicación a JPA

Vista una introducción al api de JPA, vamos a migrar nuestra aplicación al uso de JPA. Para ello construiremos, como hicimos en el caso de Hibernate, una clase Helper que nos ayude a trabajar con el API de JPA , a continuación se muestra su código fuente.

Código 12.8: (JPALevelHelper.java)

```
package com.arquitecturajava.aplicacion;  
//omitimos imports  
public class JPALevelHelper {  
    private static final EntityManagerFactory emf = buildEntityManagerFactory();  
    private static EntityManagerFactory buildEntityManagerFactory() {  
        try {  
            return Persistence.createEntityManagerFactory("arquitecturaJava");  
        } catch (Throwable ex) {  
            throw new RuntimeException("Error al crear la factoria de JPA");  
        }  
    }  
    public static EntityManagerFactory getJPAFactory() {  
        return emf;  
    }  
}
```

Una vez construida la clase helper y definido el fichero persistence.xml, podemos modificar nuestra aplicación y que nuestras clases de negocio utilicen el API de JPA de forma sencilla. A continuación se muestra el código fuente de la clase Libro una vez migrada a JPA ,el código de la clase Categoría es muy similar.

Código 12.9: (Categoria.java)

```
package com.arquitecturajava.aplicacion;
import java.util.List;
// imports omitidos
@Entity
@Table(name = "Libros")
public class Libro {
// omitimos propiedades

    public void insertar() {

        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        EntityTransaction tx = null;
        tx = manager.getTransaction();
        tx.begin();
        manager.persist(this);
        tx.commit();

    }
    public void borrar() {

        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        EntityTransaction tx = null;
        tx = manager.getTransaction();
        tx.begin();
        manager.remove(manager.merge(this));
        tx.commit();
        manager.close();
    }
    public void salvar() {

        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        EntityTransaction tx = null;
        tx = manager.getTransaction();
        tx.begin();
        manager.merge(this);
        tx.commit();
        manager.close();
    }
}
```

```
public static List<Libro> buscarTodos() {  
  
    EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();  
    EntityManager manager = factoriaSession.createEntityManager();  
    TypedQuery<Libro> consulta = manager.createQuery(  
        "SELECT l FROM Libro l JOIN FETCH l.categoría",  
        Libro.class);  
    List<Libro> listaDeLibros = null;  
    listaDeLibros = consulta.getResultList();  
    manager.close();  
    return listaDeLibros;  
}  
  
public static Libro buscarPorClave(String isbn) {  
  
    EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();  
    EntityManager manager = factoriaSession.createEntityManager();  
    TypedQuery<Libro> consulta = manager.createQuery(  
        "Select l from Libro l JOIN FETCH l.categoría where l.isbn=?1",  
        Libro.class);  
    consulta.setParameter(1, isbn);  
    Libro libro = null;  
    libro = consulta.getSingleResult();  
    manager.close();  
    return libro;  
}  
  
public static List<Libro> buscarPorCategoria(Categoría categoria) {  
    EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();  
    EntityManager manager = factoriaSession.createEntityManager();  
    TypedQuery<Libro> consulta = manager.createQuery(  
        "Select l from Libro l where l.categoría=?1",  
        Libro.class);  
    consulta.setParameter(1, categoria);  
    List<Libro> listaDeLibros = null;  
    listaDeLibros = consulta.getResultList();  
    manager.close();  
    return listaDeLibros;  
}  
}
```

Acabamos de migrar los métodos de persistencia de la clase Libro al standard JPA. La siguiente tarea se tratará de abordar otro tema que teníamos pendiente: la gestión de excepciones a nivel de capa de persistencia.

3. Manejo de Excepciones

En estos momentos si en alguna ocasión una consulta produce una excepción, no se garantiza que las conexiones, sentencias etc. se cierren correctamente y por lo tanto podemos acabar teniendo problemas. En esta tarea vamos a encargarnos de modificar los métodos de la capa de persistencia para que se encarguen de una correcta gestión de excepciones. Para ello comenzaremos modificando el método borrar de la clase libro añadiéndole las cláusulas try/catch/finally necesarias. Seguidamente se muestra su código.

Código 12.10: (Libro.java)

```
public void borrar() {
    EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
    EntityManager manager = factoriaSession.createEntityManager();
    EntityTransaction tx = null;
    try {
        tx=manager.getTransaction();
        tx.begin();
        manager.remove(manager.merge(this));
        tx.commit();
    } catch (PersistenceException e) {
        manager.getTransaction().rollback();
        throw e;
    } finally {
        manager.close();
    }
}
```

Como podemos ver, el manejo de excepciones es muy similar al código que anteriormente se albergaba a nivel de JDBC y no vamos a comentarlo con más detalle. A continuación se muestra como complementario el método buscaTodos() como método de selección .

Arquitectura Java

Código 12.11: (Libro.java)

```
public static List<Libro> buscarTodos() {  
    EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();  
    EntityManager manager = factoriaSession.createEntityManager();  
    TypedQuery<Libro> consulta = manager.createQuery(  
        "SELECT l FROM Libro l JOIN FETCH l.categoría",  
        Libro.class);  
  
    List<Libro> listaDeLibros = null;  
    try {  
        listaDeLibros = consulta.getResultList();  
    } finally {  
        manager.close();  
    }  
    return listaDeLibros;  
}
```

Así, el código es similar, con la única diferencia que no utilizamos cláusula catch ya que no es necesario realizar un rollback de ninguna excepción. El mismo código hubiera sido necesario en Hibernate, pero hemos preferido obviarlo y centrarnos en JPA.

Resumen

En este capítulo no hemos realizado grandes cambios a nivel de la arquitectura de la aplicación pero sí en cuanto a cómo se implementa la capa de persistencia utilizando JPA como standard.

13. El principio ISP y el patrón DAO

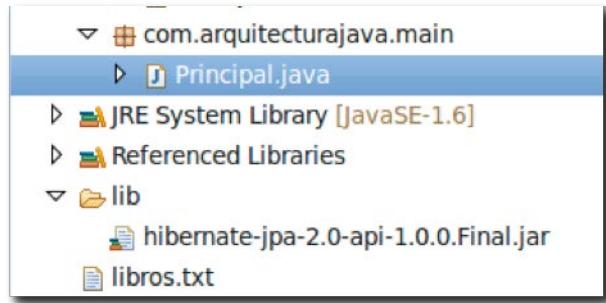
En el capítulo anterior hemos hecho evolucionar nuestra aplicación para que soporte el estándar de JPA. Es momento de seguir progresando en su diseño. Para esto, es necesario salir momentáneamente del entorno web a fin de poder explicar algunos nuevos conceptos. Vamos a suponer que nuestra aplicación necesita de un nuevo módulo que se encargue de leer una lista de libros almacenada en un fichero de texto. La estructura del fichero es la siguiente:

1,Java,1,Programación
2,JSP ,2, Web

En principio construir un programa para leer estos datos e imprimir la lista de libros por consola es sencillo, tenemos además construidas las clases de negocio que se adaptan a estos datos.

- Libro
- Categoría

Así pues vamos a construir un nuevo proyecto Java que denominaremos LeerLibros usando eclipse (proyecto java standard) que use nuestras clases de negocio . A continuación se muestra una imagen de los elementos requeridos.



Una vez importadas estas clases, nos percatamos de que no es posible trabajar con ellas de forma directa sin incluir la clase JPAHelper, así pues añadimos esta clase a nuestro proyecto. Después estaremos también obligados a añadir la librería de Hibernate y JPA, ya que nuestras clases hacen uso de anotaciones de este tipo. Así pues nuestras clases Libro y categoría tienen las siguientes dependencias.

- JPAHelper (Clase)
- Hibernate-jpa.2.0 (Libreria)

Vamos a ver el código fuente del programa que se encarga de leer la lista de libros:

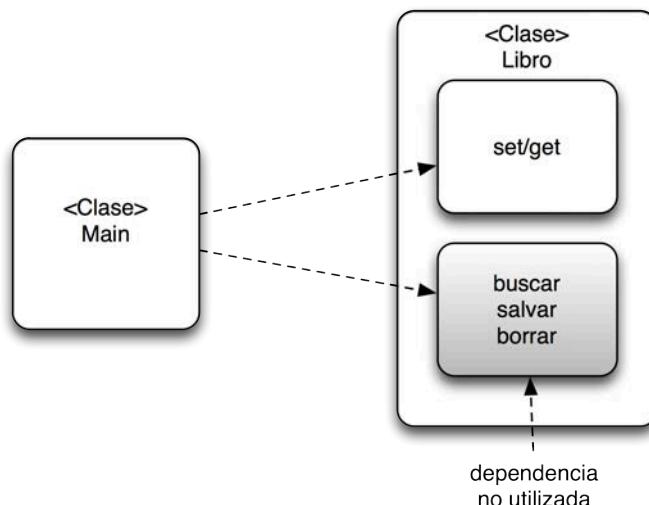
Código 13.1: (Principal.java)

```
public class Principal {
    public static void main(String[] args) throws IOException {
        String texto="";
        Reader l= new FileReader("libros.txt");
        BufferedReader lector= new BufferedReader(l);
        List<Libro> lista= new ArrayList<Libro>();
        Libro libro=null;
        Categoria categoria=null;
        while ((texto=lector.readLine())!=null ){
            String[] datos= texto.split(",");
            categoria= new Categoria(Integer.parseInt(datos[2]),datos[3]);
            libro = new Libro(datos[0],datos[1],categoria);
            lista.add(libro);
        }
        for(Libro l1 :lista) {
            System.out.println(l1.getTitulo());
        }
    }
}
```

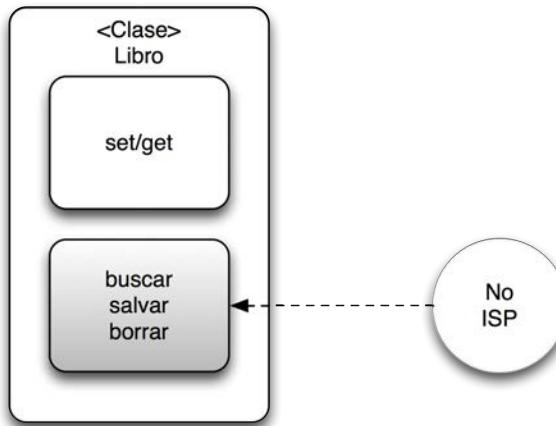
El programa es muy sencillo: lee el fichero, lo recorre línea a línea, crea una nueva lista de libros la cuál finalmente recorre e imprime por pantalla. Ahora bien: van a aparecer algunos problemas. En primer lugar nuestro programa no necesita para nada ninguno de los métodos de persistencia de la clase Libro o de la clase Categoría, ni mucho menos hace uso de la clase JPAHelper y de las librerías de JPA. Sin embargo, no tenemos más remedio que importarlas a nuestro proyecto. Esto claramente no tiene ningún sentido .Vamos a centrar el objetivo de este capítulo en solventar este tipo de problemas .Para ello vamos a introducir otro principio denominado principio ISP o Interface Segregation Principle cuya definición se muestra seguidamente:

- **ISP :** Una clase cliente A que tiene una dependencia con la clase B no debe verse forzada a depender de métodos de la clase B que no vaya a usar jamás.

En nuestro caso parece claro que tenemos un problema relativo a este principio, ya que nuestra clase cliente (clase Principal) depende de métodos que nunca va a utilizar. Métodos que están en la clase libro, concretamente los relativos al uso de persistencia, como se muestra en la figura.



Así, nos encontramos con un problema de diseño en el cuál no hemos aplicado el principio ISP, como se muestra en la siguiente figura.



He aquí los objetivos del presente capítulo:

Objetivos :

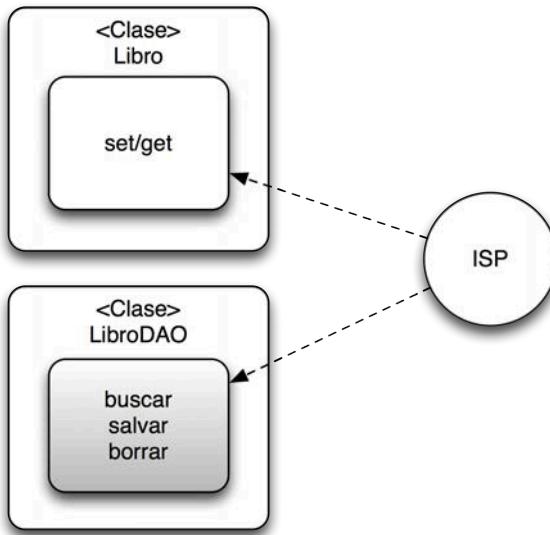
- Aplicar el principio ISP a la capa de Persistencia.

Tareas:

1. Crear las clases LibroDAO y PersonaDAO que cumplan con el principio ISP
2. Uso de interfaces a nivel DAO para permitir varias implementaciones
3. Creación de DAOs Genéricos
4. Rendimiento

1. Crear las clases LibroDAO y CategoriaDAO.

Vamos a usar a continuación el principio ISP para eliminar dependencias entre la clase Main (cliente) y los métodos de la clase Libro. Para ello la clase Libro externalizará todos los métodos de persistencia a una nueva clase denominada LibroDAO. DAO(Data Access Object) hace referencia a un patrón de diseño de la capa de persistencia que se encarga de cumplir con el principio ISP y separar las responsabilidades de negocio y persistencia. Así pues, la clase Libro se encargará de los métodos set/get (Negocio) y la clase LibroDAO de todos los métodos de persistencia (insertar/borrar etc.). A continuación se muestra una figura aclaratoria:

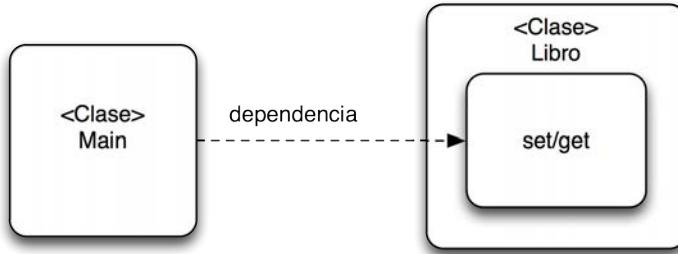


Una vez creadas las nuevas clases, se ubicaran en un nuevo paquete de la aplicación denominado DAO (ver imagen).



Acabamos de separar los métodos de nuestra clase `Libro` en dos clases independientes, cada una de las cuales se encarga de una determinada funcionalidad. Hemos aplicado el principio ISP, ya que era el que de forma más directa encajaba. Igualmente podríamos haber aplicado el principio SRP (Simple Responsibility Principle) ya que también estamos separando las distintas responsabilidades de nuestra clase. El hecho de que dos principios encajen con el refactoring que deseamos aplicar a nuestro código nos ayuda a fortalecer la idea de que podemos obtener una solución mejor. Frecuentemente el ver que dos principios de ingeniería nos orientan hacia la misma

solución refuerza el concepto de que los refactorings van en la dirección adecuada. Ahora nuestro programa Principal únicamente dependerá de la clase Libro, como se muestra en la figura y no tendrá necesidad de usar nada de la capa de persistencia.



Después de tener claro cómo dividir las clases, vamos a ver detalladamente cómo queda el código fuente de la clase Libro así como el de la clase LibroDAO.

Código 13.2: (Libro.java)

```

@Entity
@Table(name = "Libros")
public class Libro {
    @Id
    private String isbn;
    private String titulo;
    @ManyToOne
    @JoinColumn(name = "categoria")
    private Categoria categoria;
    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public Categoria getCategoria() {
        return categoria;
    }
    public void setCategoria(Categoria categoria) {
        this.categoria = categoria;
    } //omitimos constructores
}
  
```

A continuación se muestra el código de la clase LibroDAO:

Código 13.3: (LibroDAO.java)

```
public class LibroDAO {  
    public void insertar(Libro libro) {  
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();  
        EntityManager manager = factoriaSession.createEntityManager();  
        EntityTransaction tx = null;  
        try {  
            tx=manager.getTransaction();  
            tx.begin();  
            manager.merge(libro);  
            tx.commit();  
        } catch (PersistenceException e) {  
            manager.getTransaction().rollback();  
            throw e;  
        } finally {  
            manager.close();  
        }  
    }  
    public void borrar(Libro libro) {  
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();  
        EntityManager manager = factoriaSession.createEntityManager();  
        EntityTransaction tx = null;  
        try {  
            tx=manager.getTransaction();  
            tx.begin();  
            manager.remove(manager.merge(libro));  
            tx.commit();  
        } catch (PersistenceException e) {  
            manager.getTransaction().rollback();  
            throw e;  
        } finally {  
            manager.close();  
        }  
    }  
    public void salvar(Libro libro) {  
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();  
        EntityManager manager = factoriaSession.createEntityManager();  
        EntityTransaction tx = null;  
        try {  
            tx=manager.getTransaction();  
            tx.begin();  
            manager.merge(libro);  
            tx.commit();  
        } catch (PersistenceException e) {  
            manager.getTransaction().rollback();  
            throw e;  
        }  
    }  
}
```

```

        } finally {
            manager.close();
        }
    }
    public List<Libro> buscarTodos() {
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        TypedQuery<Libro> consulta = manager.createQuery(
            "Select l from Libro l", Libro.class);
        List<Libro> listaDeLibros = null;
        try {
            listaDeLibros = consulta.getResultList();
        } finally {
            manager.close();
        }
        return listaDeLibros;
    }
    public Libro buscarPorClave(String isbn) {

        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        TypedQuery<Libro> consulta = manager.createQuery(
            "Select l from Libro l where l.isbn=?1", Libro.class);
        consulta.setParameter(1, isbn);
        Libro libro = null;
        try {
            libro = consulta.getSingleResult();
        } finally {
            manager.close();
        }
        return libro;
    }
    public List<Libro> buscarPorCategoria(Categoría categoria) {
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        TypedQuery<Libro> consulta = manager.createQuery(
            "Select l from Libro l where l.categoría=?1", Libro.class);
        consulta.setParameter(1, categoria);
        List<Libro> listaDeLibros = null;
        try {
            listaDeLibros = consulta.getResultList();
        } finally {
            manager.close();
        }
        return listaDeLibros;
    }
}
}

```

El código fuente de ambas clases es prácticamente idéntico a lo que teníamos antes de separarlas, exceptuando que los métodos de la clase LibroDAO reciben ahora como parámetros Libros y han dejado de ser estáticos. Una vez visto este refactoring, vamos a ver como quedaría el código en nuestras acciones; a continuación mostramos una de ellas.

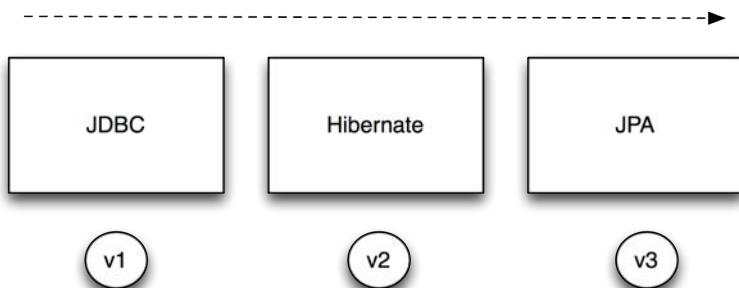
Código 13.4: MostrarLibroAccion.java)

```
@Override  
    public String ejecutar(HttpServletRequest request,  
                           HttpServletResponse response) {  
        LibroDAO libroDAO = new LibroDAO();  
        CategoriaDAO categoriaDAO = new CategoriaDAO();  
        List<Libro> listaDeLibros = libroDAO.buscarTodos();  
        List<Categoria> listaDeCategorias =  
            categoriaDAO.buscarTodos();  
        request.setAttribute("listaDeLibros", listaDeLibros);  
        request.setAttribute("listaDeCategorias", listaDeCategorias);  
        return "MostrarLibros.jsp";  
    }
```

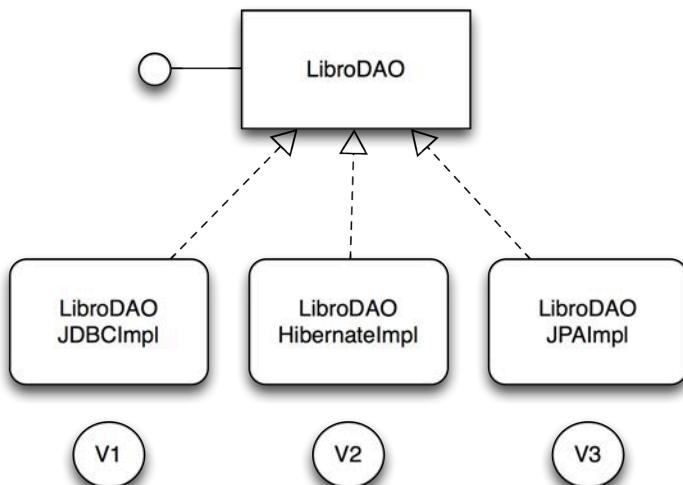
Hemos terminado de aplicar el principio ISP a nuestra aplicación. Seguidamente avanzaremos en el diseño de la capa DAO para aportar flexibilidad.

2. Uso de interfaces en componentes DAO

Acabamos de separar de forma clara la capa de persistencia de la capa de negocio en nuestra aplicación. En estos momentos la capa de persistencia esta construida con JPA pero en ejemplos anteriores ha estado construida con Hibernate o con JDBC . Así pues hemos tenido una evolución en cuanto a cómo construimos esta capa de persistencia (ver imagen).



Muchas aplicaciones enterprise tienen un tiempo de vida amplio. Es habitual que la capa de persistencia evolucione con el paso del tiempo y de las versiones de la aplicación. Así pues, es interesante diseñar esta capa de forma que, en caso de migrar de JDBC a Hibernate o de Hibernate a JPA, los cambios a realizar en el código sean los mínimos posibles. Así pues vamos a construir un interface LibroDAO y un Interface CategoríaDAO de los cuáles se puedan generar varias implementaciones (ver imagen).



Para poder generar estos interfaces hay que rediseñar el conjunto de paquetes que estamos utilizando .A partir de ahora usaremos los siguientes en cuanto a capa de persistencia:

- **com.arquitecturajava.dao** : Paquete que contiene únicamente los interfaces DAO
- **com.arquitecturajava.dao.jpa**: Paquete que contiene la implementación para JPA.

Arquitectura Java

Vamos a ver a continuación una imagen que clarifica la estructura.



Una vez clara la estructura de la capa de persistencia, vamos a ver el código fuente de uno de estos interfaces, concretamente el interface LibroDAO, el caso del interface CategoríaDAO es muy similar.

Código 13.5: (LibroDAO.java)

```
package com.arquitecturajava.aplicacion.dao;
import java.util.List;
import com.arquitecturajava.aplicacion.bo.Categoría;
import com.arquitecturajava.aplicacion.bo.Libro;

public interface LibroDAO {

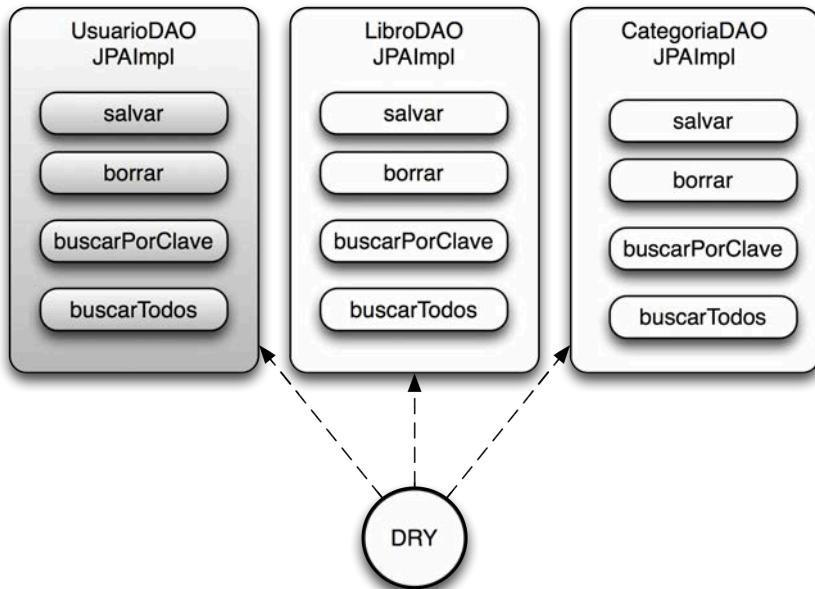
    public abstract void borrar(Libro libro);
    public abstract void salvar(Libro libro);
    public abstract List<Libro> buscarTodos();
    public abstract Libro buscarPorClave(String isbn);
    public abstract List<Libro> buscarPorCategoría(Categoría categoria);
}
```

Una vez creado el interface podemos hacer que nuestra clase LibroDAOJPAImpl implemente el interface.

Código 13.6: (LibroDAOJPAImpl.java)

```
public class LibroDAOJPAImpl implements LibroDAO {
// el resto del código no varia
}
```

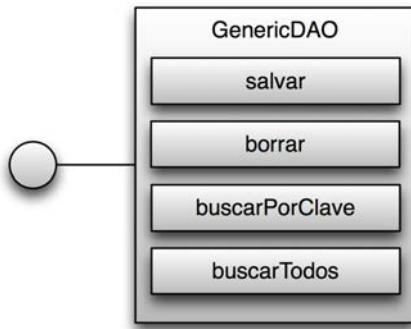
Con esta tarea hemos ganado en flexibilidad a la hora de poder evolucionar nuestra aplicación de una forma sencilla entre un sistema de persistencia y otro. Es momento de revisar la estructura de nuestras clases DAO referentes a JPA (`LibroDAOJPImpl`,`CategoríaDAOJPImpl`) para que nos demos cuenta de que todas ellas comparten la misma estructura incluso si añadimos nuevas clases (ver imagen).



Este es un claro síntoma de que quizás nuestras clases de persistencia no cumplen del todo con el principio DRY y mucha funcionalidad está repetida. La siguiente tarea se encargará de abordar este problema.

3. El principio DRY y el patrón GenericDAO

En primer lugar vamos a diseñar un nuevo interface que incluya las funciones que serán compartidas por todas las clases de forma idéntica (`borrar`, `salvar`, `buscarPorClave`, `buscarTodos`). Para ello vamos a construir un interface distinto a los anteriores, ya que se tratará de un interface de tipo Genérico de tal forma que sean luego más adelante cada una de las clases las que definan a qué tipo hace referencia .Vamos a ver un diagrama aclaratorio:

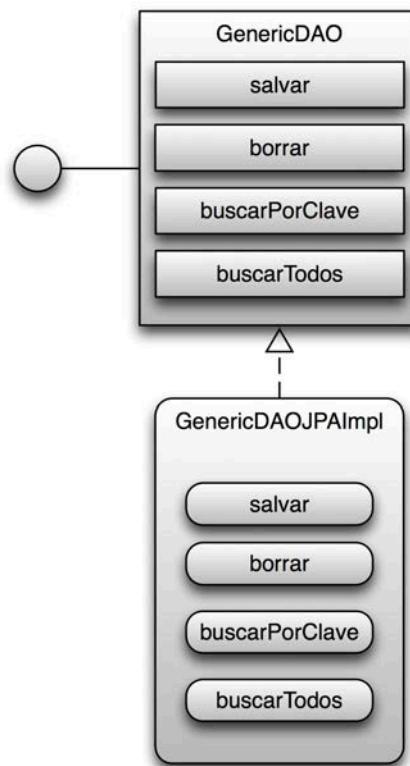


Una vez tenemos claro cuál es el interface definido como genérico, vamos a ver su código.

Código 13.7: (GenericDAO.java)

```
package com.arquitecturajava.aplicacion.dao;  
  
import java.io.Serializable;  
import java.util.List;  
public interface GenericDAO<T,Id extends Serializable> {  
    T buscarPorClave (Id id);  
    List<T>buscarTodos();  
    void salvar(T objeto);  
    void borrar(T objeto);  
}
```

Creada este interface, construiremos una clase Genérica que implemente el interface implementando los métodos de forma genérica (ver imagen).



Al hacer uso de genéricos podremos eliminar una gran parte de la repetición de código que tenemos en nuestras clases DAO de JPA .A continuación se muestra la clase completa con todos sus métodos.

Arquitectura Java

Código 13.8: (GenericDAOJPImpl)

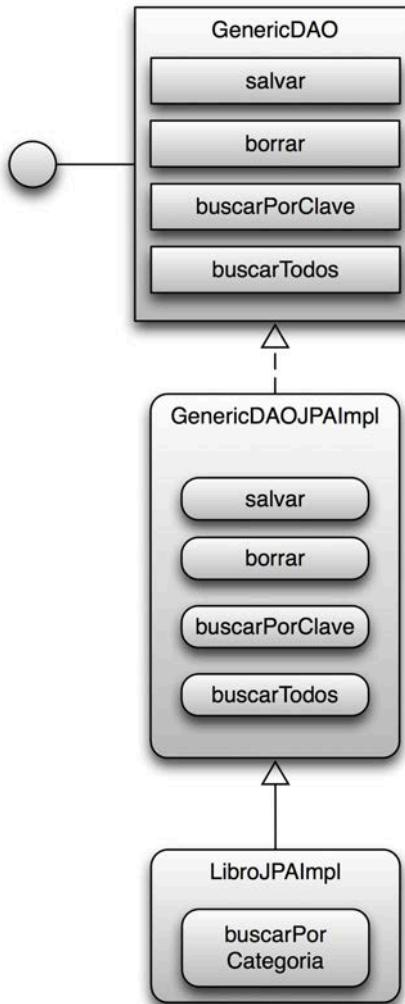
```
//omitimos imports etc
public abstract class GenericDAOJPImpl<T, Id extends Serializable> implements
    GenericDAO<T, Id> {
    private Class<T> claseDePersistencia;
    @SuppressWarnings("unchecked")
    public GenericDAOJPImpl() {
        this.claseDePersistencia = (Class<T>) ((ParameterizedType)
            getClass().getGenericSuperclass()).getActualTypeArguments()[0];
    }
    @Override
    public T buscarPorClave(Id id) {
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        T objeto = null;
        try {
            objeto = (T) manager.find(claseDePersistencia, id);
            return objeto;
        } finally {
            manager.close();
        }
    }
    @Override
    public List<T> buscarTodos() {
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        List<T> listaDeObjetos = null;
        try {
            TypedQuery<T> consulta = manager.createQuery("select o from "
                + claseDePersistencia.getSimpleName() + " o",
                claseDePersistencia);
            listaDeObjetos = consulta.getResultList();
            return listaDeObjetos;
        } finally {
            manager.close();
        }
    }
    public void borrar(T objeto) {
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        EntityTransaction tx = null;
        try {
            tx = manager.getTransaction();
            tx.begin();
            manager.remove(manager.merge(objeto));
        }
```

```
        tx.commit();
    } catch (PersistenceException e) {
        tx.rollback();
        throw e;
    } finally {
        manager.close();
    }
}

public void salvar(T objeto) {
    EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
    EntityManager manager = factoriaSession.createEntityManager();
    EntityTransaction tx = null;
    try {
        tx = manager.getTransaction();
        tx.begin();
        manager.merge(objeto);
        tx.commit();
    } catch (PersistenceException e) {
        tx.rollback();
        throw e;
    } finally {
        manager.close();
    }
}

public void insertar(T objeto) {
    EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
    EntityManager manager = factoriaSession.createEntityManager();
    EntityTransaction tx = null;
    try {
        tx = manager.getTransaction();
        tx.begin();
        manager.persist(objeto);
        tx.commit();
    } catch (PersistenceException e) {
        tx.rollback();
        throw e;
    } finally {
        manager.close();
    }
}
```

El código de esta clase genera las operaciones de persistencia elementales para todas las clases DAO. Por lo tanto no es necesario implementar estos métodos en las clases hijas.,ya que si estas clases extienden de la clase GenericDAOJPImpl es suficiente. A continuación se muestra una imagen aclaratoria.



Una vez que tenemos claro como la clase **GenericDAOJPAImpl** nos ayuda en el desarrollo de nuestra aplicación, es momento de ver el código fuente de la clase **LibroJPAImpl** y de la Clase **CategoríaJPAImpl** para ver como quedan simplificadas.

Código 13.9: (LibroDAOJPAImpl.java)

```

package com.arquitecturajava.aplicacion.dao.jpa;

public class LibroDAOJPAImpl extends GenericDAOJPAImpl<Libro, String>
implements LibroDAO {

    public List<Libro> buscarPorCategoria(Categoría categoria) {
        EntityManagerFactory factoriaSession = JPAHelper.getJPAFactory();
        EntityManager manager = factoriaSession.createEntityManager();
        CategoríaDAO cdao= new CategoríaDAOJPAImpl();

        TypedQuery<Libro> consulta = manager.createQuery(
                "Select l from Libro l where l.categoría=?1",
                Libro.class);
        consulta.setParameter(1, categoria);
        List<Libro> listaDeLibros = null;
        try {
            listaDeLibros = consulta.getResultList();
        } finally {
            manager.close();
        }
        return listaDeLibros;
    }
}

```

Código 13.10: (CategoriaDAOJPAImpl.java)

```

package com.arquitecturajava.aplicacion.dao.jpa;

import com.arquitecturajava.aplicacion.bo.Categoría;
import com.arquitecturajava.aplicacion.dao.CategoríaDAO;

public class CategoríaDAOJPAImpl extends GenericDAOJPAImpl<Categoría,
Integer> implements CategoríaDAO {
//no hay nada que añadir todo lo aporta la clase GenericDAO de la cual extendemos
}

```

Como podemos ver, el caso de la clase CategoríaDAOImpl es el más exagerado de todos ya que la clase genérica consigue eliminar todos los métodos que tenía e incluso añadir algunos que nos podrán ser útiles. Tras construir estas clases, podemos ver como queda la estructura de clases de persistencia en nuestro proyecto.

```
▽ com.arquitecturajava.aplicacion.dao.jpa
  ▷ CategoríaDAOJPImpl.java
  ▷ GenericDAOJPImpl.java
  ▷ JPAHelper.java
  ▷ LibroDAOJPImpl.java
```

4. Rendimiento

En muchos casos el uso básico de Hibernate acaba en soluciones cuya escalabilidad es reducida, no por el uso del framework sino por la falta de conocimiento por parte del desarrollador . Frecuentemente nuestro modelo de clases deberá sobreescribir métodos para mejorar el rendimiento del framework ORM. A continuación se muestra un ejemplo sencillo donde se sobrecarga el método buscarTodos para obtener un mejor rendimiento.

Código 13.10: (LibroDAOJPImpl.java)

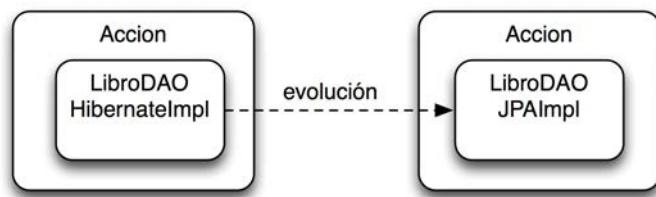
```
public List<Libro> buscarTodos() {
    TypedQuery<Libro> consulta = getManager().createQuery("SELECT l FROM Libro l
        JOIN FETCH l.categoría",Libro.class);
    return consulta.getResultList();
}
```

Resumen

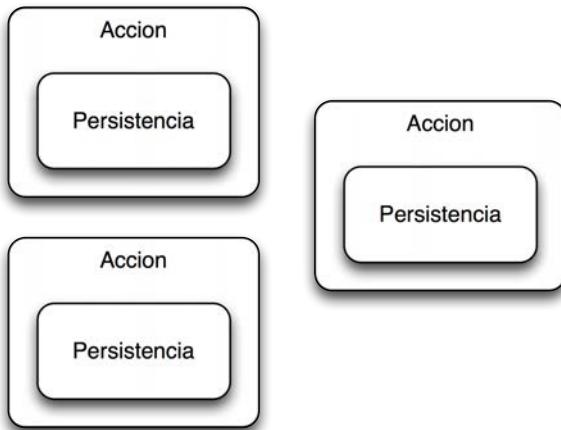
En este capítulo hemos visto cómo el principio ISP y el principio SRP nos orientan hacia el diseño de una capa de persistencia más flexible mientras que el principio DRY y el uso de Genéricos nos permite eliminar mucha funcionalidad repetida entre las distintas clases que definen la capa DAO. Por último, hemos revisado brevemente temas de rendimiento.

14. El principio de inversión de control y patrón factory

En el capítulo anterior hemos aplicado el principio ISP y con él hemos construido una capa de persistencia basada en el patrón DAO. Esto nos ha permitido separar las responsabilidades de negocio y persistencia. En este capítulo vamos a modificar la capa de persistencia para que no solo admita varios sistemas de persistencia (jdbc, hibernate, jpa) a través del uso de interfaces, sino para que además sea sencillo o transparente cambiar unas implementaciones por otras. En estos momentos disponemos de dos posibles implementaciones de capa de persistencia: una basada en Hibernate y otra basada en Hibernate sobre JPA. A continuación se muestra cómo hemos hecho evolucionar el código en el capítulo anterior.



Ahora bien, para realizar estos cambios no nos ha quedado más remedio que modificar el código fuente de nuestras Acciones. Si en algún momento quisiéramos realizar el proceso inverso, tendríamos que volver a cambiar estas. Esto se debe a que la responsabilidad sobre el tipo de objeto de persistencia para construir recae en el programador y en el código que él construye en las diferentes acciones (ver imagen).



Si en algún momento queremos cambiar de tipo de persistencia, será el programador quien deberá realizar los cambios. Este diseño no cumple con el principio OCP ya que si queremos cambiar la capa de persistencia, nos veremos obligados a modificar el código que previamente hemos construido. En este capítulo vamos a introducir un nuevo principio de ingeniería: **el principio de Inversión de Control** que nos ayudará a solventar este problema y permitirá que nuestra capa de persistencia pueda evolucionar cumpliendo con OCP. Vamos a explicar este principio.

- **Inversión de Control (Inversion of Control o IOC):** El principio de inversión de control consiste en que el control de la construcción de los objetos no recae directamente en el desarrollador a través del uso del operador **new**, sino que es otra clase o conjunto de clases las que se encargan de construir los objetos que necesitamos. Aunque la definición es de entrada algo confusa, durante el resto del capítulo la iremos clarificando a través de la creación de ejemplos.

Objetivos :

- Aplicar el principio de inversión de control IOC a la capa de persistencia de nuestra aplicación.

Tareas :

1. Crear factorías e implementar el principio de IOC.
2. El principio DRY y el patrón Abstract Factory.
3. El patrón Abstract Factory y el uso de interfaces.

1. Crear Factorías e implementar el principio de IOC,

Para implementar el principio de IOC en nuestra aplicación debemos añadirle nuevas clases que se encarguen de construir los distintos objetos de la capa de persistencia . Como punto de partida vamos a revisar el código fuente de una de nuestras acciones para ver cómo construye los distintos objetos de la capa DAO.

- LibroDAOJPAImpl
- CategoriaDAOJPAImpl.

A continuación se muestra el código fuente de la clase MostrarLibrosAccion.

Código 14.1: (MostrarLibrosAccion.java)

```
public String ejecutar(HttpServletRequest request,
                      HttpServletResponse response) {

    LibroDAO libroDAO = new LibroDAOJPAImpl();
    CategoriaDAO categoriaDAO = new CategoriaDAOJPAImpl();
    List<Libro> listaDeLibros = libroDAO.buscarTodos();
    List<Categoria> listaDeCategorias = categoriaDAO.buscarTodos();
    request.setAttribute("listaDeLibros", listaDeLibros);
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    return "MostrarLibros.jsp";

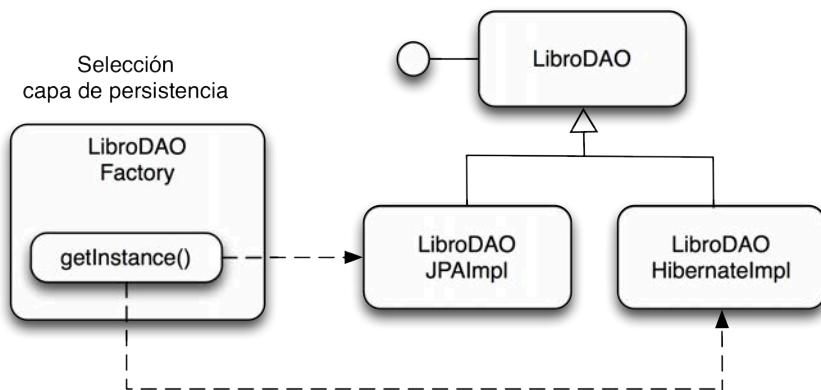
}
```

Como podemos ver, es responsabilidad del programador que desarrolla la aplicación el crear cada uno de los distintos objetos de la capa DAO. Según la definición del principio de inversión de control, no será a partir de ahora el programador el que construya los distintos de objetos de la capa de persistencia, sino que se encargará otra clase. Vamos a ver el código fuente de la clase LibroDAOFactory que sería la encargada de crear los objetos de tipo LibroDAO.

Código 14.2: (LibroDAO.java)

```
public class LibroDAOFactory {
    public static LibroDAO getInstance() {
        String tipo = "JPA";
        if (tipo.equals("Hibernate")) {
            return LibroDAOHibernateImpl();
        } else {
            return new LibroDAOJPImpl();
        }
    }
}
```

Como el código muestra, esta clase tiene la responsabilidad de construir las distintas implementaciones que existan del interface LibroDAO (ver imagen).



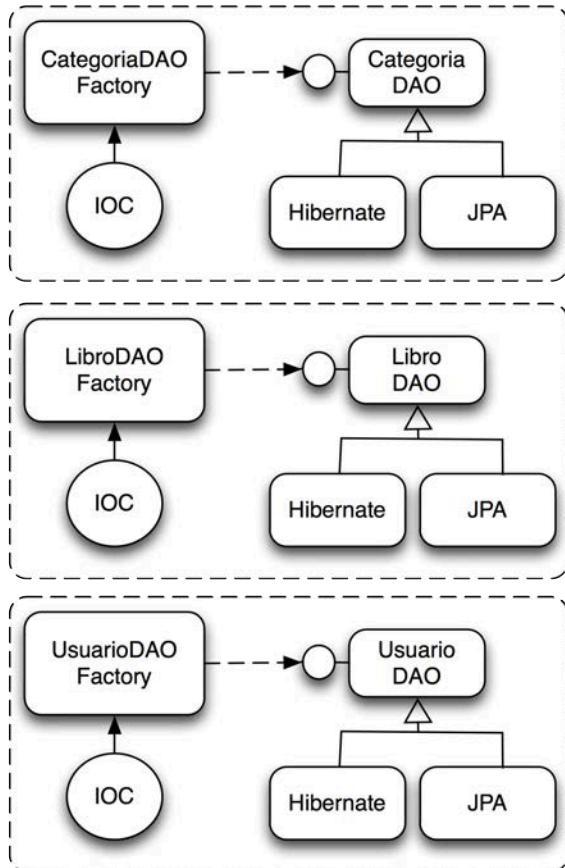
De esta manera y dependiendo del tipo de persistencia definamos, la factoría nos devuelve una implementación u otra. Habitualmente las factorías se apoyan en un fichero de properties para leer el tipo de persistencia que estamos utilizando. En este caso nosotros devolveremos siempre la implementación de JPA por simplicidad .Una vez que hemos construido esta clase factoría, vamos a ver cómo queda el código fuente de cada una de nuestras acciones .Para ello vamos a mostrar el código de la acción MostrarLibros, ya que usa tanto la clase de persistencia CategoríaDAO como la clase LibroDAO.

Arquitectura Java

Código 14.3: (MostrarLibrosAccion.java)

```
public String ejecutar(HttpServletRequest request,
                      HttpServletResponse response)
    CategoriaDAO categoriaDAO= CategoriaDAOFactory.getInstance();
    LibroDAO libroDAO= LibroDAOFactory.getInstance();
    List<Libro> listaDeLibros = libroDAO.buscarTodos();
    List<Categoria> listaDeCategorias = categoriaDAO.buscarTodos();
    request.setAttribute("listaDeLibros", listaDeLibros);
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    return "MostrarLibros.jsp";
}
```

El código no es complicado de entender y podemos ver como ya el programador no necesita hacer uso del operador **new** sino que se apoyará en dos factorías que hemos construido, una para los Libros y otra para las Categorías. De esta forma simplemente cambiando la implementación que devuelve cada una las factorías podremos intercambiar una capa de persistencia por otra (obligando por ejemplo a las factorías a leer la implementación de un fichero de propiedades). Sin embargo, para conseguir este nivel de transparencia a la hora de poder intercambiar las capas de persistencia y ganar en flexibilidad, debemos pagar un precio bastante alto: por cada clase de negocio que tengamos aparecerá una nueva factoría (ver imagen).

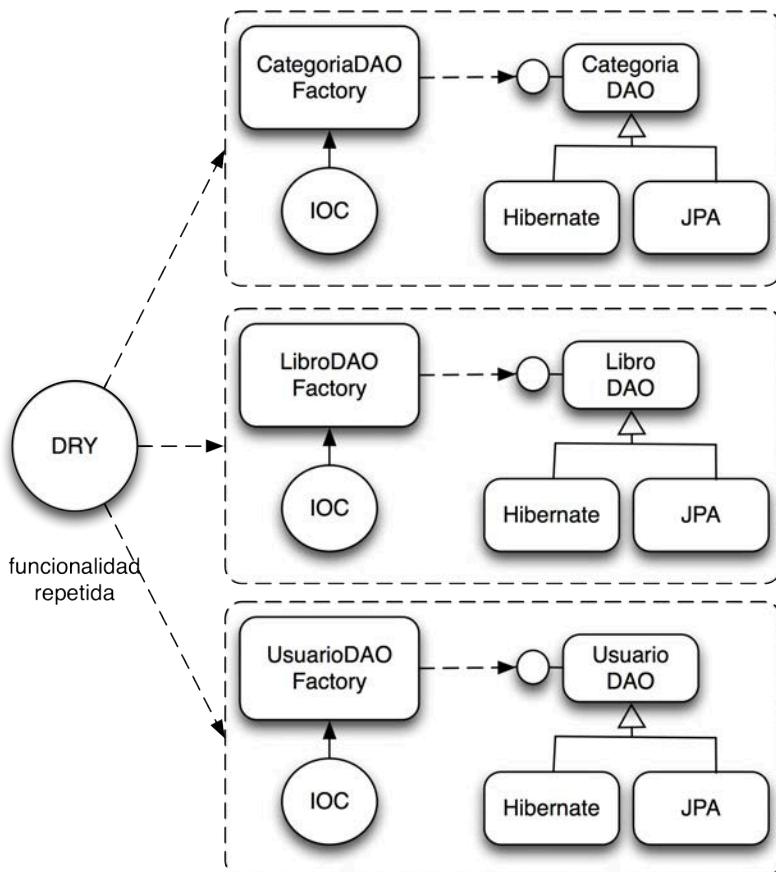


Esto es un problema importante ya que nos encontraremos con una violación clara del principio DRY pues prácticamente todas las factorías que construimos son idénticas, ya que todas cubren la misma funcionalidad al elegir una implementación u otra de la capa de persistencia. A continuación se muestra el código fuente de la factoría CategoríaDAOFactor , podemos advertir que es prácticamente idéntico al código de la factoría LibroDAOFactor.

Código 14.4: (CategoriaDAOFactory.java)

```
public class CategoriaDAOFactory {  
    public static CategoriaDAO getInstance() {  
        String tipo = "JPA";  
        if (tipo.equals("Hibernate")) {  
            return new CategoriaDAOHibernateImpl();  
        } else {  
            return new CategoriaDAOJPImpl();  
        }  
    }  
}
```

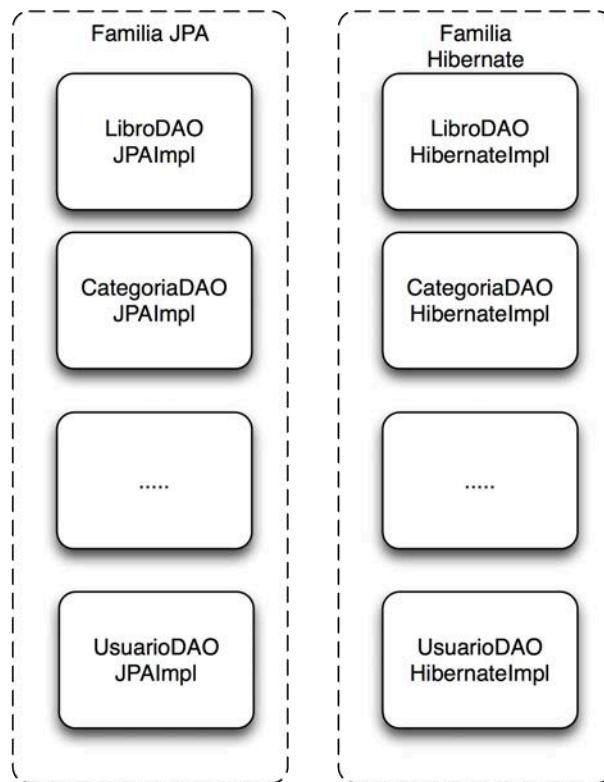
Así pues tenemos un problema de repetición de funcionalidad (ver imagen).



Por lo tanto debemos refactorizar nuestro código para evitar tantas repeticiones de funcionalidad y a la vez mantener la capacidad que tiene nuestra aplicación de cambiar de implementación de capa de persistencia de forma transparente (Hibernate, JPA etc) . Para ello la siguiente tarea se encargara de solventar este problema.

2. El principio DRY y el patrón Abstract Factory

La estructura de clases de capa de persistencia que tenemos tiene una peculiaridad : pueden ser agrupadas por familias. Una familia sería la implementación de JPA y otra familia sería la implementación de Hibernate (ver imagen).



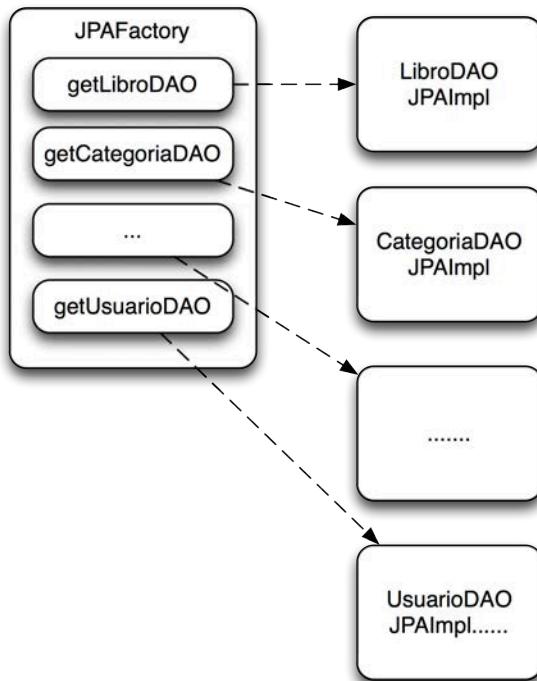
Cuando tenemos esta casuística tan especial existe un patrón de diseño que nos puede ayudar a cumplir con el principio IOC y mantener nuestro compromiso con el principio DRY, permitiéndonos por lo tanto cambiar de forma transparente entre una implementación y otra sin tener que pagar un alto precio de repetición de código en nuestra aplicación. Este patrón de diseño se denomina Abstract Factory , vamos a introducirlo a continuación.

Arquitectura Java

El patrón abstract factory es un patrón de diseño que se apoya en la construcción de un grupo reducido de factorías. Concretamente una factoría por cada familia de clases que tengamos y otra factoría que agrupa a las factorías encargadas de crear las familias. Así, en nuestro ejemplo tendremos la necesidad de construir tres factorías.

1. Crear una factoría para la familia de Hibernate
2. Crear una factoría para la familia de JPA
3. Crear una factoría para las dos factorías que hemos creado

Así pues vamos a empezar con la primera tarea construir una factoría que sea capaz de crear todos los objetos de una de las familias, para ello elegiremos la familia de JPA. Nuestra factoría tendrá la peculiaridad de tener un método por cada una de las clases JPA que construyamos. A continuación se muestra un diagrama aclaratorio de cómo está construida esta factoría.



Una vez creada la factoría de JPA, vamos a ver su código fuente para que nos ayude a asentar ideas.

Código 14.5: (DAOJPAFactory.java)

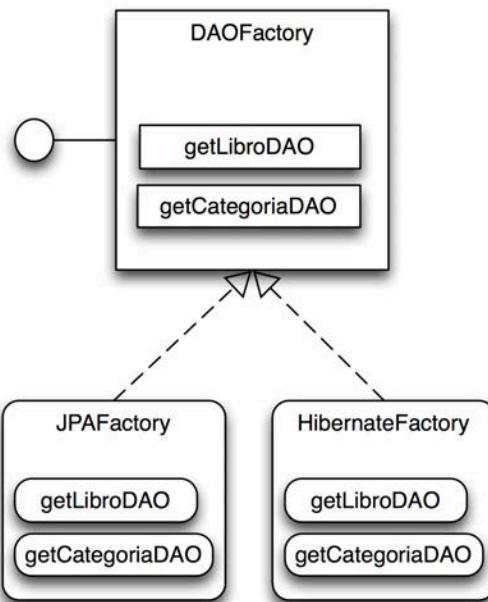
```
public class DAOJPAFactory {  
    public CategoriaDAO getCategoriaDAO() {  
        return new CategoriaDAOJPAImpl();  
    }  
    public LibroDAO getLibroDAO() {  
        return new LibroDAOJPAImpl();  
    }  
}
```

La factoría nos devuelve los objetos que pertenecen a la familia de JPA LibroDAOJPAImpl y CategoriaDAOJPAImpl. Vamos a ver a continuación el código de la factoría que se encarga de la otra familia la familia de Hibernate.

Código 14.6: (DAOHibernateFactory.java)

```
public class DAOHibernateFactory {  
    public CategoriaDAO getCategoriaDAO() {  
        return new CategoriaDAOHibernateImpl();  
    }  
    public LibroDAO getLibroDAO() {  
        return new LibroDAOHibernateImpl();  
    }  
}
```

Una vez creadas ambas clases , es fácil identificar que comparten el mismo conjunto de métodos y por lo tanto se puede definir un interface común para ambas, como el que se muestra en el siguiente diagrama.



A continuación se muestra el código fuente del interface para el caso sencillo de nuestra aplicación.

Código 14.7: (DAOFactory.java)

```
package com.arquitecturajava.aplicacion.dao;

public interface DAOFactory {
    public CategoriaDAO getCategoryDAO();
    public LibroDAO getLibroDAO();
}
```

Una vez definido el interface, obligaremos a cada una de nuestras factorías a implementarlo

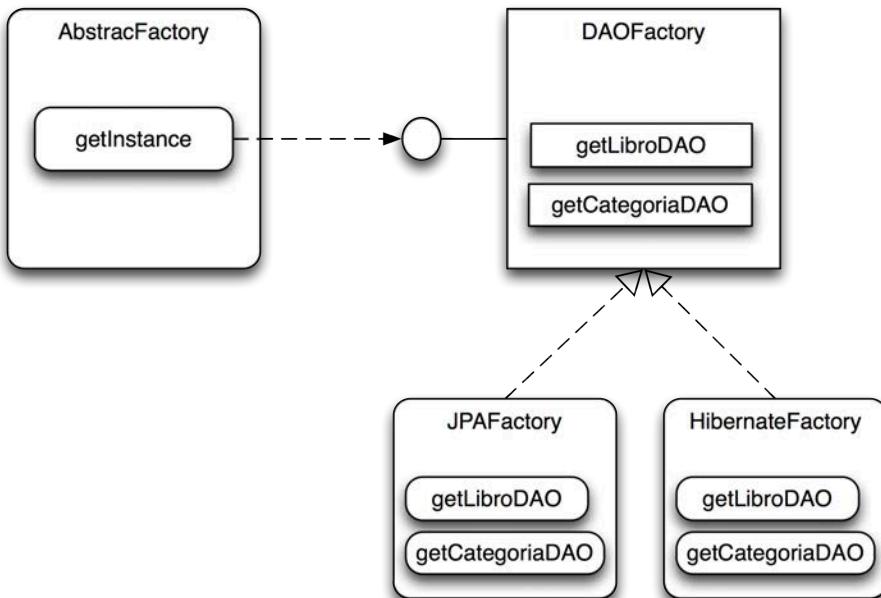
Código 14.8: (DAOJPFactory.java)

```
public class DAOJPFactory implements DAOFactory {
    // resto del código
}
```

Código 14.9: (DAOJPFactory.java)

```
public class DAOHibernateFactory implements DAOFactory {
// resto del código
}
```

Definido el interface común para ambas factorías, nos encontramos con que podemos definir una factoría que nos devuelva una implementación de este interface para JPA (DAOJPFactory) o una implementación para Hibernate (DAOHibernateFactory), algo muy similar a lo que hacíamos antes. A esta factoría, dado que crea objetos que son de tipo factoría, se la denomina AbstractFactory. La siguiente figura muestra la relación entre las distintas clases.



A continuación se muestra su código fuente.

Código 14.10: (DAOAbstractFactory.java)

```
package com.arquitecturajava.aplicacion.dao;

public abstract class DAOAbstractFactory {
    public static DAOFactory getInstance() {
        String tipo="JPA";
        if (tipo.equals("Hibernate")) {
            new DAOHibernateFactory();
        } else {
            return new DAOJPAFactory();}
    }
}
```

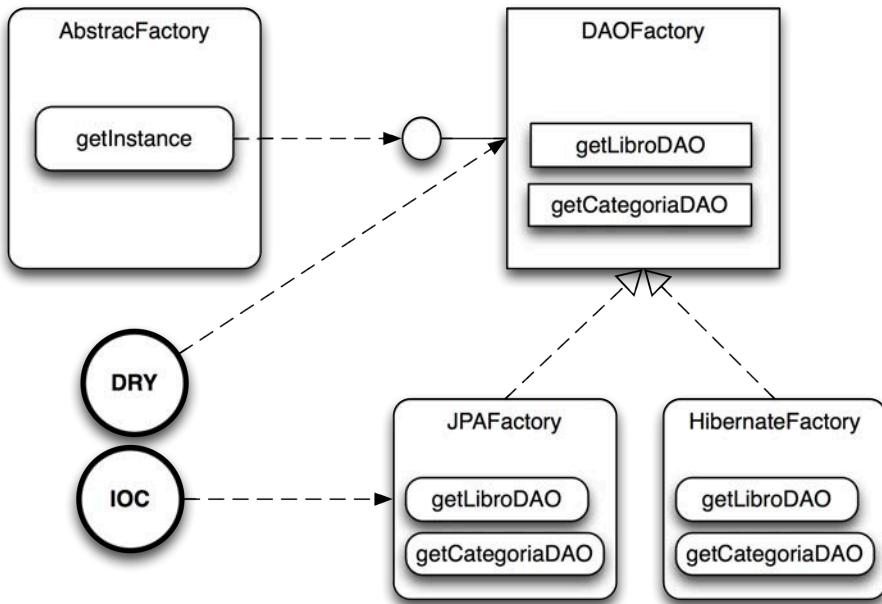
Esta clase crea objetos bien de un tipo de factoría bien de otro . Más tarde cada una de las factorías creadas serán las encargadas de devolvernos los distintos objetos DAO para una implementación concreta. A continuación se muestra cómo se usa a nivel de las acciones el concepto de AbstractFactory y de Factory, poniendo como ejemplo el método ejecutar de MostrarLibrosAccion.

Código 14.11: (MostrarLibrosAccion.java)

```
public String ejecutar(HttpServletRequest request,
                      HttpServletResponse response) {
    DAOFactory factoria= DAOAbstractFactory.getInstance();
    CategoriaDAO categoriaDAO= factoria.getCategoriaDAO();
    LibroDAO libroDAO=factoria.getLibroDAO();
    List<Libro> listaDeLibros = libroDAO.buscarTodos();
    List<Categoria> listaDeCategorias = categoriaDAO.buscarTodos();
    request.setAttribute("listaDeLibros", listaDeLibros);
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    return "MostrarLibros.jsp";
}
```

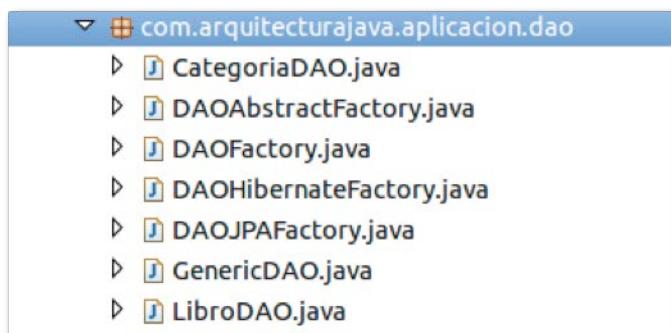
A partir de este momento el programador creará una instancia de la clase AbstractFactory la cuál se apoyara habitualmente en un fichero de propiedades para elegir qué familia de objetos de persistencia construir. Una vez hecho esto, nos encontramos con que el nuevo diseño de la aplicación cumple con el principio IOC y con

el principio DRY a nivel de capa de persistencia, permitiéndonos variar entre una implementación y otra de forma transparente (ver imagen).



Resumen

Una vez rediseñada la capa de persistencia podremos de una manera sencilla cambiar una capa por otra sin tener necesidad de tocar el código fuente, cumpliendo con el principio OCP. Para ello habremos añadido las siguientes clases a la capa DAO:

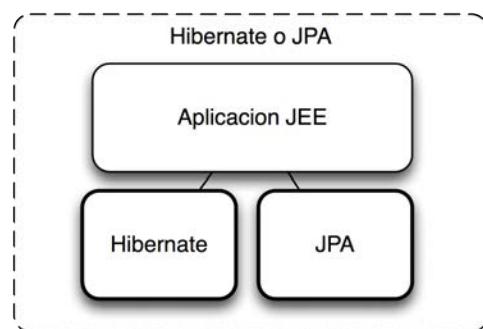


15. El Principio DRY y el patrón servicio

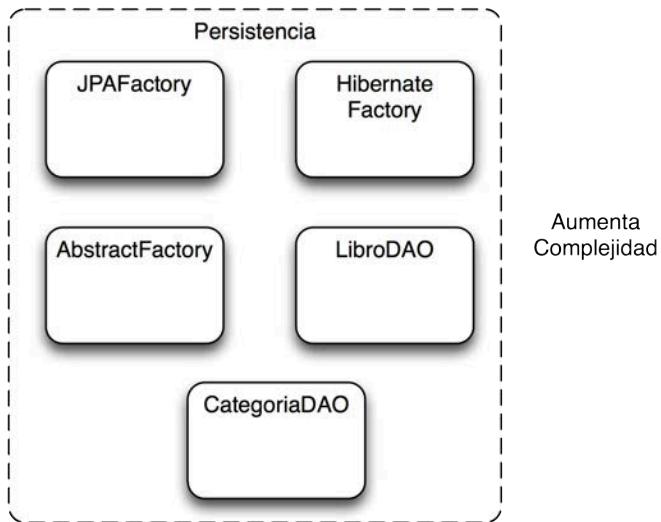
En el capítulo anterior hemos añadido versatilidad a la capa de persistencia utilizando los siguientes patrones de diseño.

- Factory
- Abstract Factory

permitiéndonos intercambiar las distintas capas de persistencia de forma prácticamente transparente como muestra la figura.



Sin embargo, para poder disponer de esta funcionalidad, hemos tenido que sacrificar algo :hemos incrementado la complejidad del manejo de la capa de persistencia por parte de los desarrolladores y se necesita un conjunto de clases mayor que antes para realizar las mismas operaciones (ver imagen).



Es momento de abordar este problema .El objetivo principal de este capítulo será simplificar el trabajo de los desarrolladores con la capa de persistencia al construir la capa de presentación e interactuar con la misma. Enumeremos los objetivos y tareas planteados:

Objetivos:

- Simplificar el acceso a la capa de persistencia

Tareas:

1. Uso del principio DRY en el acceso a la capa de persistencia
2. Creación de una clase de servicio que simplifique el acceso.

1. El principio DRY y el acceso a la capa de persistencia

Vamos a revisar el código fuente de la clase MostrarLibrosAccion del capítulo anterior una vez hemos hecho uso de factorías para añadir flexibilidad a esta capa.

Código 15.1: (MostrarLibrosAccion.java)

```
public String ejecutar(HttpServletRequest request,
                      HttpServletResponse response) {

    DAOFactory factoria= DAOAbstractFactory.getInstance();
    CategoriaDAO categoriaDAO= factoria.getCategoriaDAO();
    LibroDAO libroDAO=factoria.getLibroDAO();
    List<Libro> listaDeLibros = libroDAO.buscarTodos();
    List<Categoria> listaDeCategorias = categoriaDAO.buscarTodos();
    request.setAttribute("listaDeLibros", listaDeLibros);
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    return "MostrarLibros.jsp";
}
```

Como podemos ver se ha hecho uso de las factorías y del abstract factory para generar los objetos que pertenecen a la capa de persistencia. Ahora bien si revisamos alguna de nuestras otras acciones (como por ejemplo la acción de FormularioEditarLibro cuyo código se muestra a continuación),

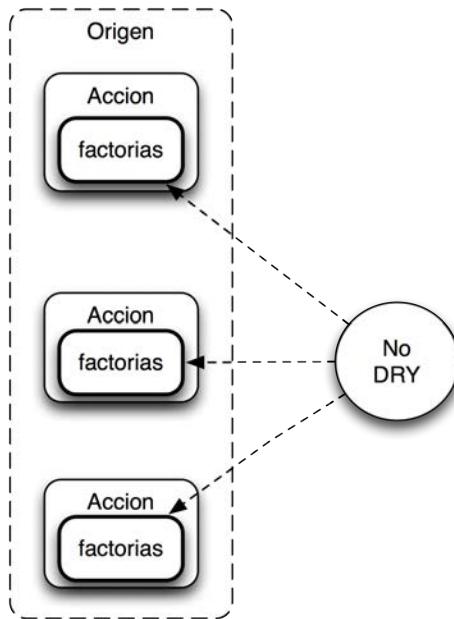
Código 15.2: (FormularioEditarLibro.java)

```
public String ejecutar(HttpServletRequest request,
                      HttpServletResponse response) {

    DAOFactory factoria= DAOAbstractFactory.getInstance();
    CategoriaDAO categoriaDAO= factoria.getCategoriaDAO();
    LibroDAO libroDAO=factoria.getLibroDAO();

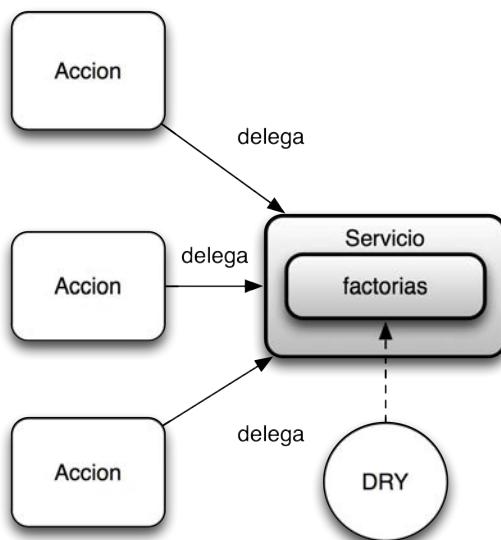
    String isbn = request.getParameter("isbn");
    List<Categoria> listaDeCategorias = categoriaDAO.buscarTodos();
    Libro libro = libroDAO
                  .buscarPorClave(request.getParameter("isbn"));
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    request.setAttribute("libro", libro);
    return "FormularioEditarLibro.jsp";
}
```

nos podremos dar cuenta de que comparten el mismo código fuente de inicialización de las factorías a la hora de utilizar la capa de persistencia .Este es un problema claro de repetición de código en el cuál no se está utilizando el principio DRY (ver imagen).

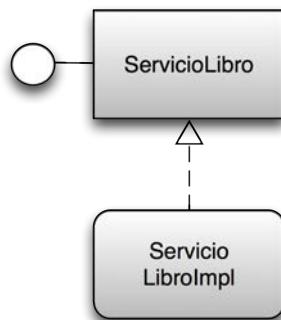


2. Creación de una clase de Servicio

Si queremos eliminar las repeticiones de código que tenemos en cada una de nuestras acciones, debemos crear una nueva clase que se encargue de centralizar la creación de los objetos de la capa DAO para un subconjunto de clases relacionadas a nivel de negocio. A estas clases se las denomina habitualmente clases de servicio y como podemos ver en la siguiente figura, un grupo de acciones delega en ellas.



Para que esta nueva capa que aparece en la aplicación pueda disponer posteriormente de una flexibilidad similar a la que dispone la capa de persistencia, la construiremos apoyándonos en el uso de interfaces. En nuestro caso construiremos un servicio denominado servicioLibros que agrupa la funcionalidad asociada a la gestión de los libros y sus categorías. A continuación se muestra un diagrama aclaratorio de la estructura.



Tenemos claro ya el interfaces y el servicio. Vamos a pasar a mostrar el código fuente del interface de servicio.

Código 15.3: (ServicioLibros.java)

```
package com.arquitecturajava.aplicacion.servicios;
import java.util.List;
import com.arquitecturajava.aplicacion.bo.Categoría;
import com.arquitecturajava.aplicacion.bo.Libro;
public interface ServicioLibros {
    public void salvarLibro(Libro libro);
    public void borrarLibro(Libro libro);
    public List<Libro> buscarTodosLosLibros();
    public List<Categoría> buscarCategoríasLibros();
    public Libro buscarLibroPorClave(String isbn);
    public Categoría buscarCategoríaPorClave(int id);
    public List<Libro> buscarLibrosPorCategoría(int categoría);
}
```

Definido el interface ,vamos a ver su implementación y cómo ésta delega en las factorías que hemos creado anteriormente para realizar su funcionalidad.

Código 15.4: (ServicioLibros.java)

```
package com.arquitecturajava.aplicacion.servicios.impl;

// omitidos imports
public class ServicioLibrosImpl implements ServicioLibros {

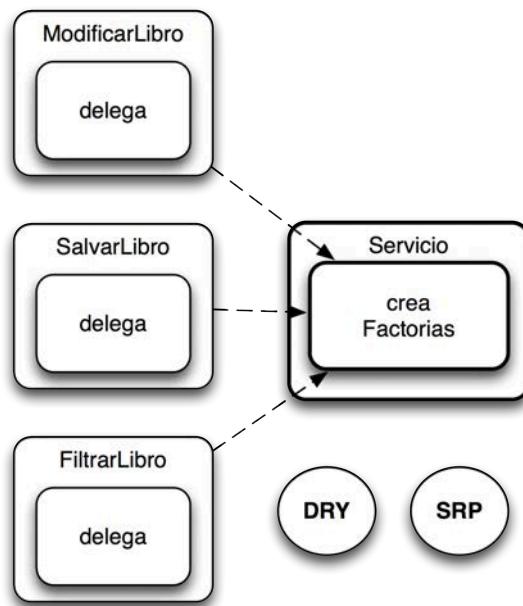
    private LibroDAO libroDAO=null;
    private CategoriaDAO categoriaDAO=null;
    public ServicioLibrosImpl() {
        DAOFactory factoria= DAOAbstractFactory.getInstance();
        libroDAO= factoria.getLibroDAO();
        categoriaDAO=factoria.getCategoríaDAO();
    }

    public void salvarLibro(Libro libro) {
        libroDAO.salvar(libro);
    }
    public void borrarLibro(Libro libro) {
        libroDAO.borrar(libro);
    }
    public List<Libro> buscarTodosLosLibros() {
        return libroDAO.buscarTodos();
    }
    public List<Categoria> buscarCategoriasLibros() {
        return categoriaDAO.buscarTodos();
    }

    public Libro buscarLibroPorClave(String isbn) {
        return libroDAO.buscarPorClave(isbn);
    }
    public Categoria buscarCategoriaPorClave(int id) {
        return categoriaDAO.buscarPorClave(id);
    }
    public List<Libro> buscarLibrosPorCategoria(int id) {
        Categoria categoria= categoriaDAO.buscarPorClave(id);
        return libroDAO.buscarPorCategoria(categoria);
    }
}
```

Arquitectura Java

Una vez construida esta clase, podemos ver como su constructor se encarga de poner a nuestra disposición las distintas factorías necesarias. La siguiente imagen muestra la relación entre la capa de servicios, la capa de persistencia y la capa de presentación y cómo el construir esta capa reduce la repetición de código.

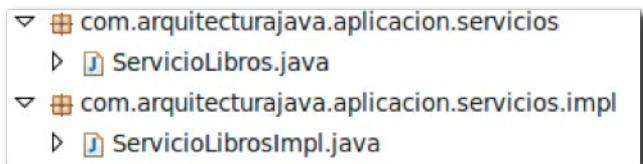


Construida la clase de servicio, vamos a ver por último cómo el código fuente de la clases de acción queda simplificado. Seguidamente se muestra el nuevo código de la clase MostrarLibrosAccion.

Código 15.5: (MostrarLibrosAccion.java)

```
public String ejecutar(HttpServletRequest request,
                      HttpServletResponse response) {
    ServicioLibros servicioLibros= new ServicioLibrosImpl();
    List<Libro> listaDeLibros = servicioLibros.buscarTodosLosLibros();
    List<Categoria> listaDeCategorias =
        servicioLibros.buscarCategoriasLibros();
    request.setAttribute("listaDeLibros", listaDeLibros);
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    return "MostrarLibros.jsp";
}
```

Una vez realizados estos cambios podemos mostrar los nuevos ficheros y paquetes que aparecen en nuestra aplicación.

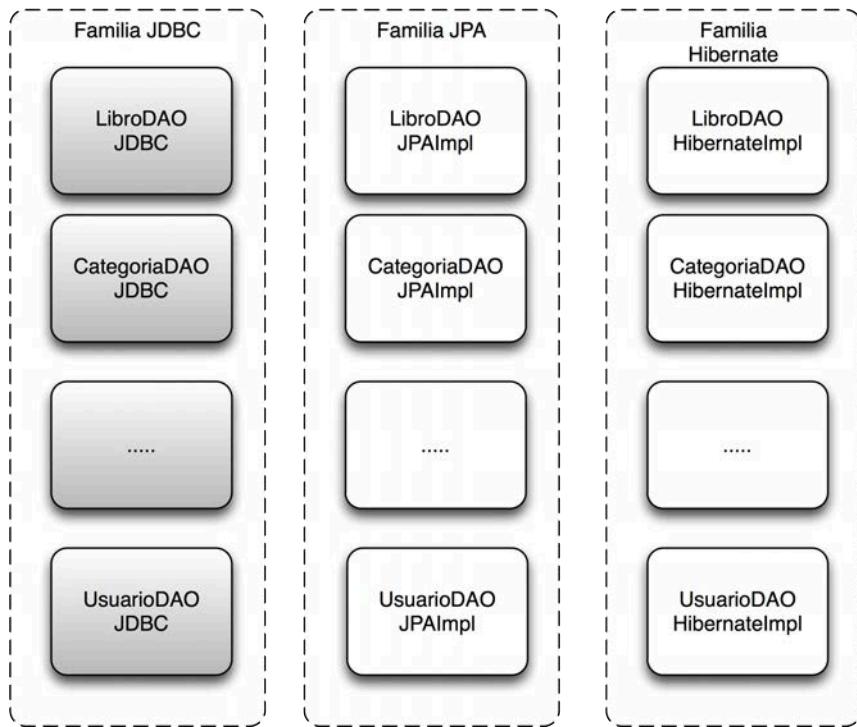


Resumen

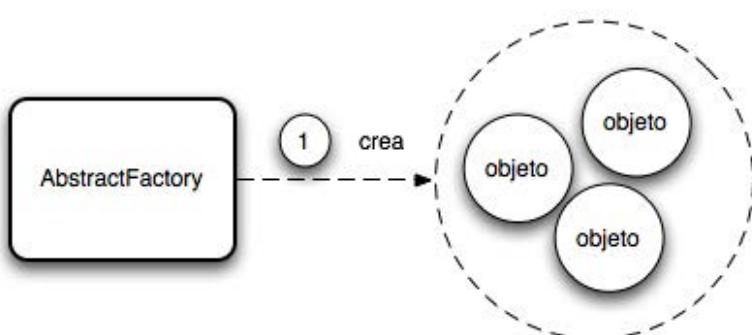
En este capítulo nos hemos centrado en simplificar el acceso a la capa de persistencia apoyándonos para ello en la construcción de una nueva capa: la capa de servicios clásica.

16. El principio IOC y el framework Spring

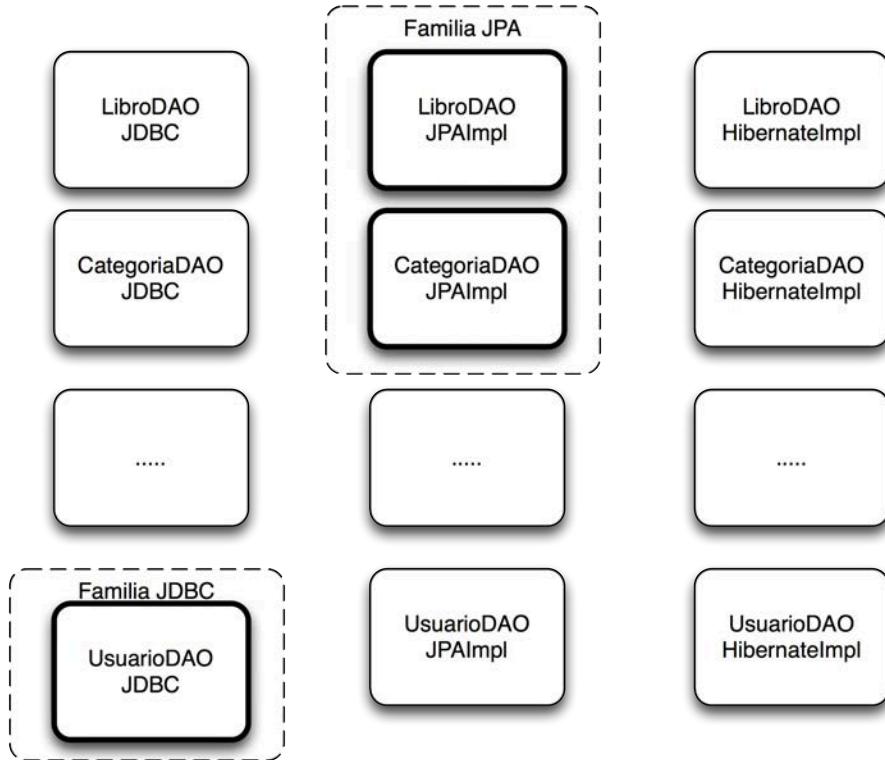
En el capítulo anterior hemos construido una capa de servicios que nos ha permitido eliminar repeticiones innecesarias de código así como aislar completamente la capa de presentación de la capa de persistencia. En este capítulo vamos a añadir flexibilidad a la capa de persistencia construida. Ahora nuestra capa de persistencia soporta nuevas implementaciones a través del uso del patrón AbstractFactory que define un conjunto de familias, pudiendo añadir nuevas familias mas adelante . Por ejemplo puede ser que en algún momento necesitemos una familia de XML o una familia JDBC como muestra en la siguiente figura.



Como acabamos de comentar, es la clase `AbstractFactory` la que se encarga de la gestión de las familias y la responsable de crear los distintos objetos (ver imagen).



Ahora bien el patrón `Abstract Factory` aunque aporta ventajas, también tiene limitaciones. Si nos apoyamos en él, no nos será posible disponer de una capa de persistencia híbrida en la que una parte de las clases DAO sean por ejemplo JDBC y otra JPA (ver imagen).



Puede parecer extraño el que una aplicación necesite un modelo híbrido pero pueden existir muchas situaciones en las cuales esto sea útil, a continuación se enumeran varias.

- **Aplicación de legado:** Puede ser que nuestra aplicación tenga alguna parte de legado y necesitemos por ejemplo acceder a esa parte vía JDBC.
- **Modulo no relacional:** Situación en la que la capa de persistencia necesita puntualmente acceder a una información que no está almacenado en una base de datos relacional, por ejemplo ficheros XML y necesitamos que parte de la implementación de la capa de persistencia acceda a ficheros xml.
- **Objeto Mock :** Necesitamos construir una clase que simule una funcionalidad concreta de la capa de persistencia ya que en estos momentos no ha sido todavía construida o en nuestro entorno de desarrollo no tenemos acceso a ella pero queremos poder simularla.

- **Problemas de rendimiento:** Partes de la aplicación tienen problemas con la capa de persistencia habitual que utilizamos y necesitan usar otro tipo de implementación que dé mejor rendimiento.

Estas son algunas de las situaciones por las cuáles soportar un modelo híbrido es interesante pero existen muchas más. El objetivo principal de este capítulo será el permitir variar la implementación de parte de nuestra capa de persistencia sin tener que estar ligados al uso concreto de una familia A o familia B de tal forma que nuestra aplicación sea más flexible. Para ello introduciremos el framework Spring y sus capacidades de inversión de control.

Objetivos:

- Introducción al framework Spring.
- Permitir a la aplicación cambiar las implementaciones de clases concretas de forma totalmente transparente al desarrollador, aportando flexibilidad a la aplicación usando Spring framework

Tareas:

1. Creación de ejemplo elemental de factorías
2. Instalación de Spring Framework
3. Configuración de Spring en nuestra aplicación

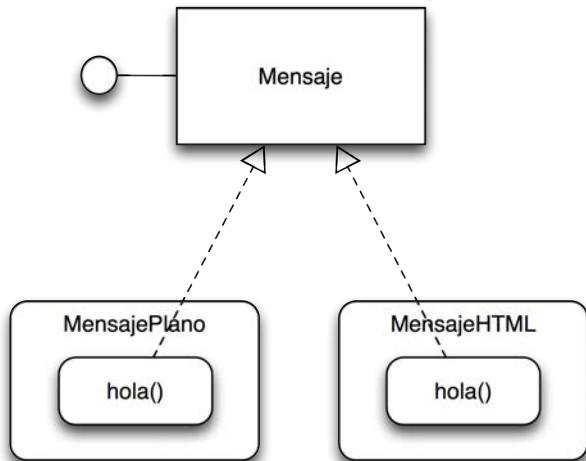
1. Creación de ejemplo de factorías

El framework Spring es un framework de Inversión de Control que nos permite gestionar nuestras necesidades de inversión de control agrupada y sencillamente sin tener que recurrir al uso de patrones de diseño clásicos (como Factory y Abstract Factory). Para comprender cómo funciona este framework, vamos a realizar un ejemplo sencillo antes de retomar nuestra aplicación y aplicar las ventajas aportadas. Para ello construiremos un sencillo interface Mensaje que dispone de dos implementaciones. A continuación se desglosan los conceptos a construir:

- **Interface Mensaje:** Interface con un único método denominado `hola()`.
- **Clase MensajeHTML :**Clase que implementa el interface e imprime por pantalla un mensaje en formato HTML.
- **Clase MensajePlano:** Clase que implementa el interface e imprime por pantalla un mensaje.
- **Clase FactoriaMensajes:** Clase que se encarga de aplicar el principio de inversión de control.

Arquitectura Java

A continuación se muestra un diagrama con la estructura básica.



Clarificado el diagrama, vemos el código fuente de cada una de las clases e interfaces:

Código 16.1: (Mensaje.java)

```
public interface Mensaje {
    public void hola();
}
```

Código 16.2: (MensajeHTML.java)

```
public class MensajeHTML implements Mensaje {
    @Override
    public void hola() {
        System.out.println("<html>hola</html>");
    }
}
```

Código 16.3: (MensajePlano.java)

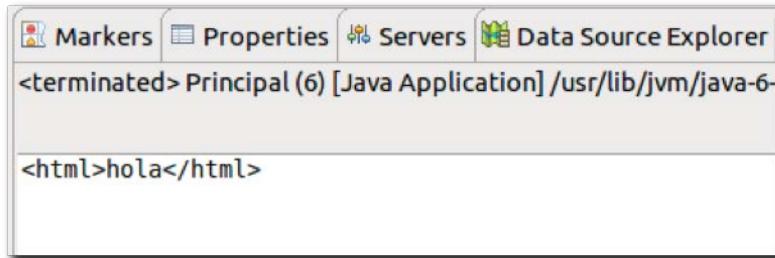
```
public class MensajePlano implements Mensaje {
    @Override
    public void hola() {
        System.out.println("hola");
    }
}
```

Ya tenemos el código de cada una de nuestras clases. Ahora vamos a construir un sencillo programa que haga uso de la implementación de HTML .El código se muestra a continuación.

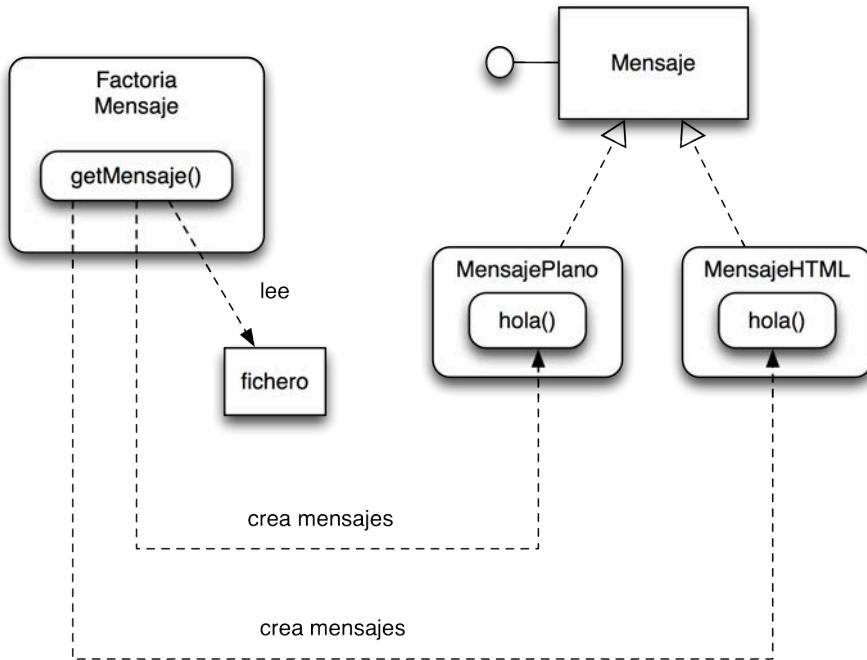
Código 16.4: (Principal.java)

```
package com.arquitecturajava;
public class Principal {
    public static void main(String[] args) {
        Mensaje mensaje= newMensajeHTML();
        mensaje.hola();
    }
}
```

Construido el código, podemos ejecutar el programa y ver qué resultados genera. Véase una imagen una vez ejecutado el programa:



Una vez que hemos construido las clases, vamos a usar el principio de inversión de control y construir una sencilla factoría apoyada en un fichero de propiedades para crear un tipo de mensaje u otro dependiendo del contenido del fichero (ver imagen).



Una vez que tenemos claro el diagrama, vamos a ver el contenido del fichero de propiedades que vamos a utilizar así como el código fuente de nuestra factoría.

Código 16.5: (mensaje.properties)

<code>tipo=html</code>

Código 16.5: (MensajeFactory.java)

```

package com.arquitecturajava;

import java.io.FileInputStream;
import java.util.Properties;

public class MensajeFactory {

    public static Mensaje getMensaje(){
        Properties propiedades= new Properties();
        Mensaje mensaje=null;
        try {
            propiedades.load(new FileInputStream("mensaje.properties"));
            String tipo=propiedades.getProperty("tipo");
            if (tipo.equals("html")) {
                mensaje=new MensajeHTML();
            }else {
                mensaje= new MensajePlano();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return mensaje;
    }
}

```

Acabamos de aplicar el principio de inversión de control a través del uso de una factoría. Ahora podremos construir el programa principal de la siguiente forma y el resultado será idéntico al anterior:

Código 16.6: (Principal.java)

```

public static void main(String[] args) {

    Mensaje mensaje= MensajeFactory.getMensaje();
    mensaje.hola();
}

```

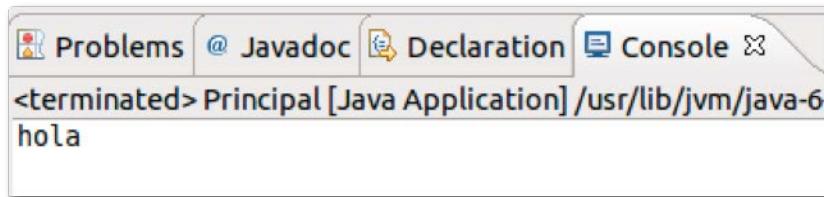
De igual manera podremos modificar el fichero de propiedades que acabamos de construir para que la implementación y el funcionamiento de nuestro programa cambie. En este caso cambiamos en el fichero el tipo (ver código).

Arquitectura Java

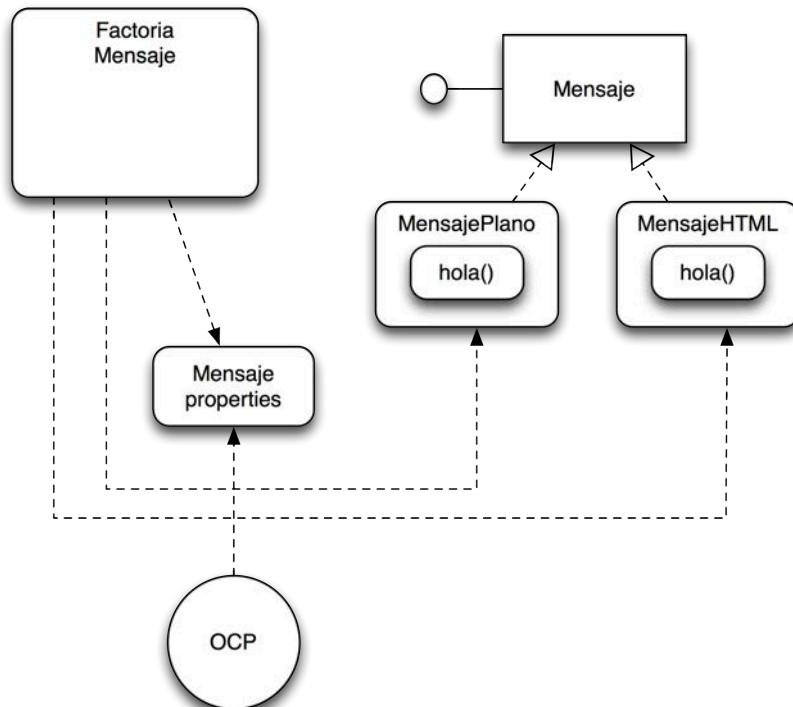
Código 16.7: (fichero)

```
tipo=plano
```

El resultado será el siguiente:



Una vez hemos visto cómo utilizar de forma sencilla el principio de inversión, podremos darnos cuenta de cómo al apoyarnos en un fichero de propiedades, conseguimos cambiar de una implementación de Mensaje a otra sin tocar el código fuente de nuestro programa .De esta forma cumplimos además con el principio OCP (ver imagen).



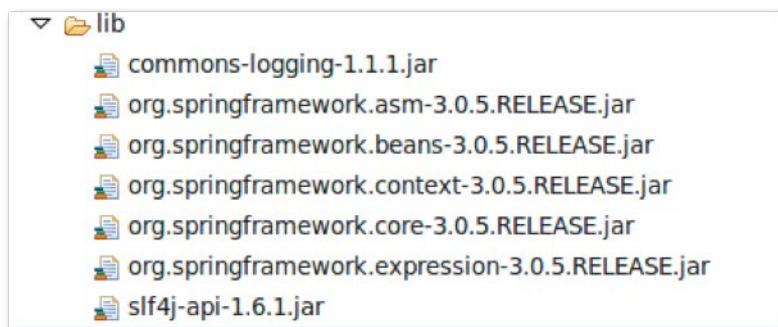
Es momento de dar por finalizada esta revisión del principio IOC y de cómo puede ligarse al principio OCP para introducir en la siguiente tarea el framework Spring y ver qué ventajas adicionales aporta a nuestra solución actual.

2. Instalación de Spring

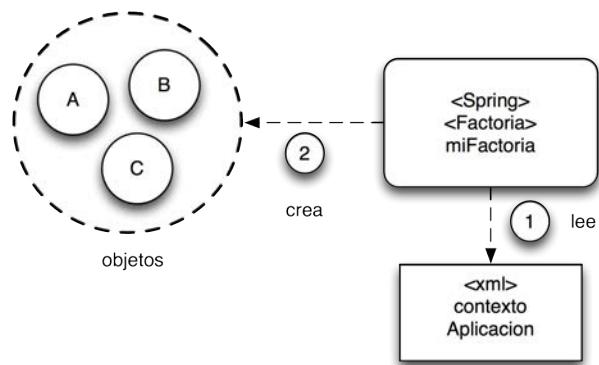
Una vez tenemos el ejemplo más elemental de inversión de control construido, lo haremos evolucionar para que haga uso de Spring framework para lo cuál el primer paso será obtener el framework de la siguiente url.

<http://www.springsource.com/download/community>

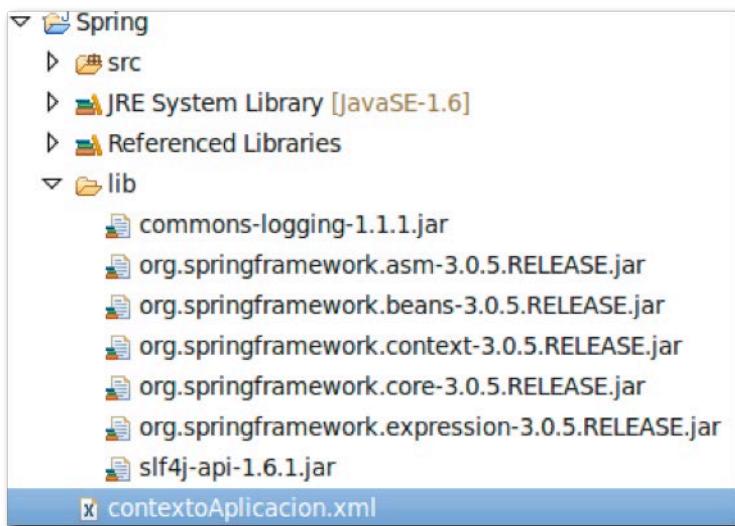
Una vez obtenido el framework, pasaremos a instalar en nuestros ejemplos los ficheros jar necesarios para realizar las operaciones básicas. A continuación se muestra una imagen con los jars.



Una vez añadida la carpeta lib y los jars necesarios, podemos comenzar a usar el framework Spring. Este framework trabaja como inversor de control y hace las tareas de una factoría. En este caso se trata de una factoría capaz de crear objetos apoyándose en un fichero de configuración XML (contextoAplicacion.xml) (ver imagen).



Una vez que tenemos claro que Spring funciona como si de una factoría se tratase y usa un fichero de configuración, vamos a ver una imagen que muestra el proyecto de Spring, sus librerías y la ubicación del fichero de configuración.



Vamos a ver cual es el contenido que este fichero tendrá para poder realizar una funcionalidad similar a la que nosotros realizábamos con nuestra factoría.

Código 16.8: (contextoAplicacion.xml)

```
<?xmlversion="1.0"encoding="UTF-8"?>
<beansxmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<bean id="mensajeHTML" class="com.arquitecturajava.MensajeHTML">
</bean>
</beans>
```

Se puede identificar claramente que el fichero se compone de una etiqueta <beans> principal la cual contiene un grupo de etiquetas <bean> anidadas. Estas etiquetas <bean> serán las encargadas de definir los distintos objetos a instanciar por parte del framework. Para ello cada objeto usará dos atributos

- id : Identifica el nombre del objeto
- class: identifica a partir de qué clase se debe construir el objeto

A continuación se muestra el bloque de código que define un objeto de tipo MensajeHTML.

Código 16.9: (contextoAplicacion.xml)

```
<bean id="mensajeHTML" class="com.arquitecturajava.MensajeHTML">
</bean>
```

Una vez tenemos claro esto, vamos a ver cómo modificar nuestro programa original para hacer uso de Spring framework a la hora de instanciar nuestros objetos y no tener que construirnos nosotros nuestras propias factorías.

Código 16.10: (Principal.java)

```
public static void main(String[] args) {
    ApplicationContextfactoria = new
    FileSystemXmlApplicationContext("contextoAplicacion.xml");
    Mensaje mimensaje= (Mensaje)factoria.getBean("mensajeHTML");
    mimensaje.hola();
}
```

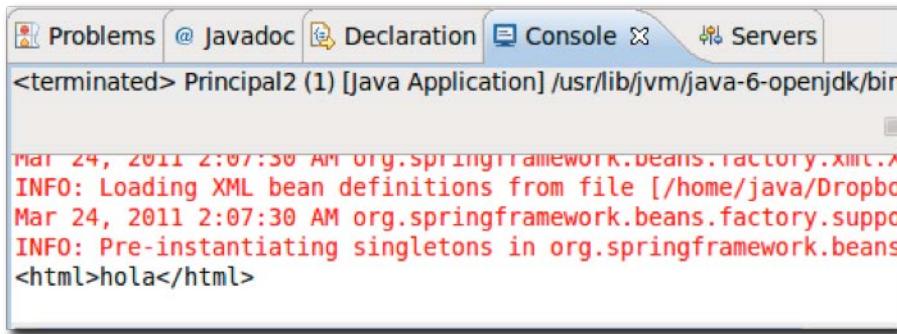
Visiblemente el código varía un poco ya que hacemos uso no ya de nuestra factoría sino de una factoría propia del framework Spring.

FileSystemXmlApplicationContext

la cuál recibe como parámetro el fichero de configuración de Spring

contextoAplicacion.xml

Si ejecutamos nuestro programa, el resultado es idéntico al programa inicial (ver imagen)



The screenshot shows the Eclipse IDE's Console view. The title bar indicates it is for a Java Application named 'Principal2'. The console output shows the following log entries:

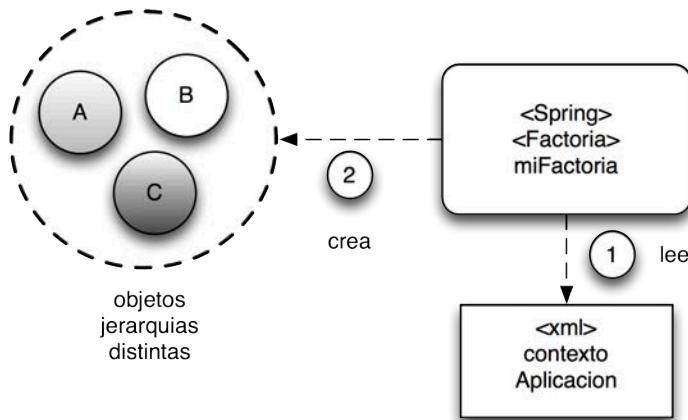
```
Mar 24, 2011 2:07:30 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [/home/java/Dropbox/Java/ArquitecturaJava/ArquitecturaJava/Principal2/src/main/resources/applicationContext.xml]
Mar 24, 2011 2:07:30 AM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory
<html>holá</html>
```

La ventaja de hacer uso de Spring es que permite añadir todas las clases que deseemos a nivel del fichero xml (ver código).

Código 16.11: (aplicacionContexto.xml)

```
<bean id="mensajeHTML" class="com.arquitecturajava.MensajeHTML" />
<bean id="mensajeBasico" class="com.arquitecturajava.MensajePlano" />
```

Así se añade una mayor flexibilidad a la hora de crear objetos y se elimina las limitaciones habituales que tiene una factoría que únicamente construya objetos de una jerarquía de clases concreta.

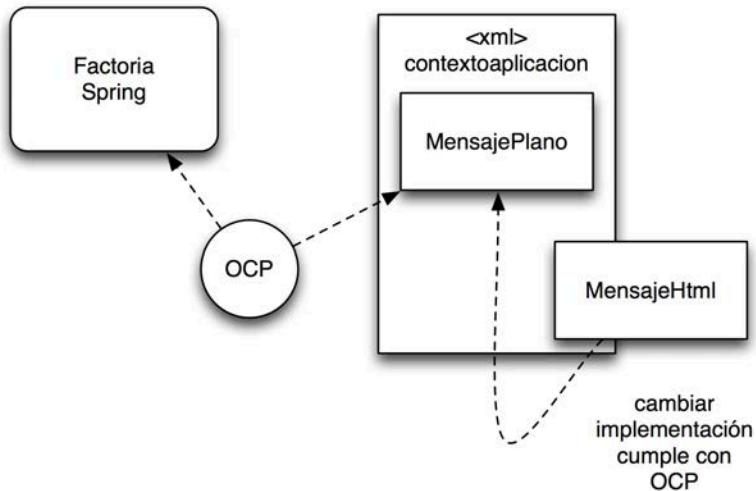


Por último, vamos a ver el código fuente de la clase principal al trabajar con dos implementaciones distintas a través del uso de Spring.

Código 16.12: (Principal.java)

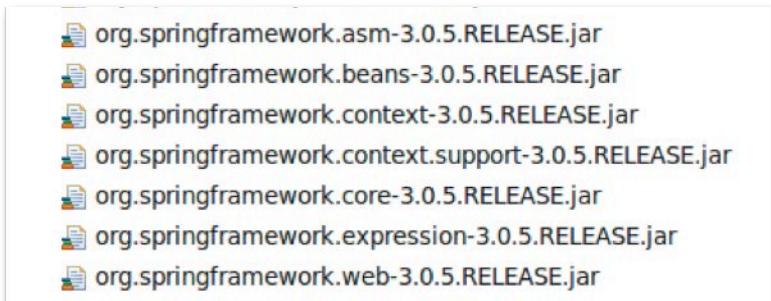
```
public static void main(String[] args) {
    ApplicationContextfactoria = new
    FileSystemXmlApplicationContext("contextoAplicacion.xml");
    Mensaje hola= (Mensaje)factoria.getBean("mensajeHTML");
    hola.hola();
    Mensaje hola= (Mensaje)factoria.getBean("mensajePlano");
    hola.hola();
}
```

Ahora tenemos claro que podemos añadir varias clases al fichero de configuración contextoAplicacion.xml .Podemos usar el framework Spring como factoría principal y se encargará de crear todos los objetos que le solicitemos. Además podremos añadir nuevas clases a instanciar o cambiar una implementación por otra sin necesidad de cambiar nuestro código fuente, cumpliendo con el principio OCP (ver imagen).



3. Instalación de Spring en nuestra aplicación.

Visto como funcionan de una manera elemental las capacidades de inversión de control del framework Spring, vamos a ver cómo podemos aplicarlas a la aplicación construida. Lo primero que vamos a hacer es instalar las librerías necesarias del framework Spring en nuestra aplicación web (ver imagen).



Una vez realizada esta operación y añadidas las librerías, hay que configurar el framework Spring para que funcione correctamente en una aplicación web. Para ello modificaremos el fichero `web.xml` y añadiremos las siguientes etiquetas.

Código 16.13: (web.xml)

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:contextoAplicacion.xml</param-value>
</context-param>
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
```

El parámetro de contexto define cuál va a ser el nombre del fichero de configuración que Spring debe cargar. Por otro lado el listener de la aplicación permite que el framework Spring se cargue y lea el fichero de configuración antes de que la aplicación web entre en funcionamiento.

Una vez configurado el framework, podemos definir el fichero XML con las clases que queremos sean instanciadas por la factoría de spring. Para comenzar poco a poco únicamente solicitaremos al framework que instancie las clases DAO. A continuación se muestra el código del fichero contextoAplicacion.xml.

Código 16.14: (contextoAplicacion.java)

```
<?xmlversion="1.0"encoding="UTF-8"?>
<beansxmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="libroDAO"
class="com.arquitecturajava.aplicacion.dao.jpa.LibroDAOJPAImpl">
</bean>

<bean id="categoriaDAO"
class="com.arquitecturajava.aplicacion.dao.jpa.CategoríaDAOJPAImpl">
</bean>
</beans>
```

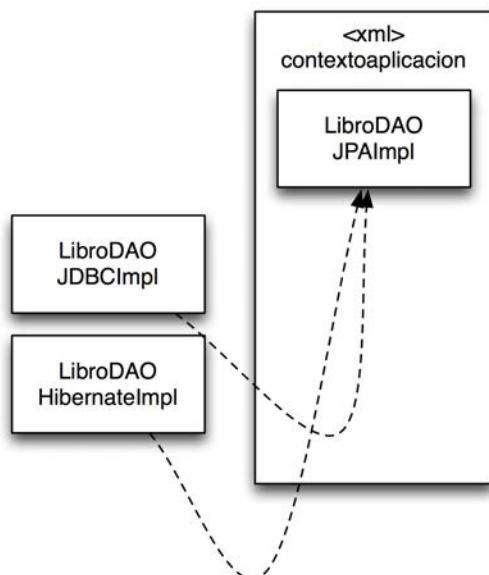
Configurado el fichero contextoAplicacion.xml , es necesario usar la capa de servicio para crear los distintos elementos de la capa de persistencia. Ahora bien, en lugar de usar factorías y abstract factories, en este caso usaremos el framework Spring ,como se muestra a continuación en el siguiente bloque de código.

Arquitectura Java

Código 16.15: (ServicioLibrosImpl.java)

```
public class ServicioLibrosImpl implements ServicioLibros {  
  
    private LibroDAO libroDAO=null;  
    private CategoriaDAO categoriaDAO=null;  
  
    public ServicioLibrosImpl() {  
        ClassPathXmlApplicationContext factoria =  
            new ClassPathXmlApplicationContext("contextoAplicacion.xml");  
        libroDAO= (LibroDAO)factoria.getBean("libroDAO");  
        categoriaDAO=(CategoriaDAO)factoria.getBean("categoriaDAO");  
    }  
}
```

Está claro que se utiliza otra factoría :ClassPathXmlApplicationContext, factoría de Spring que busca un fichero dentro del classpath . Así hemos usado un framework de inversión de control para poder cambiar la implementación de la capa de persistencia de una forma flexible, permitiendo tanto usar JPA, Hibernate, JDBC como otras . Añadir nuevas implementaciones únicamente implica añadir o modificar el fichero xml de configuración. No hay que tocar para nada la aplicación (ver imagen).



Una vez que hemos realizado los primeros cambios a nivel de Spring framework, podremos eliminar todas las clases de factoría que habíamos creado para la capa de persistencia y simplificar el modelo de nuestra aplicación en el que ya no necesitaremos el grupo de factorias (ver imagen).

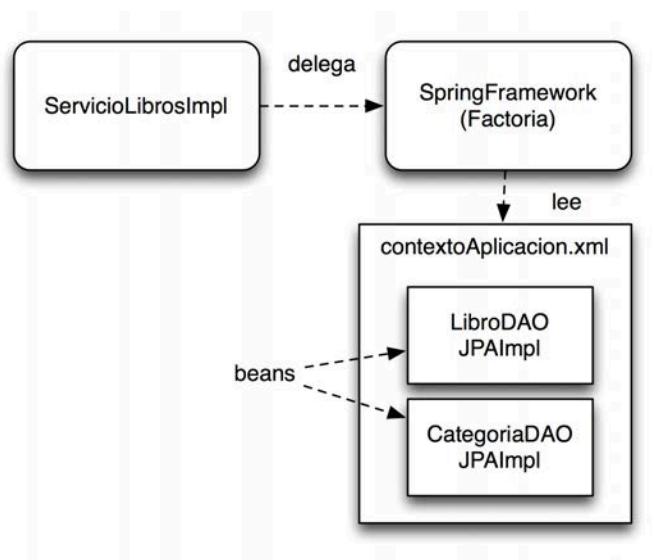


Resumen

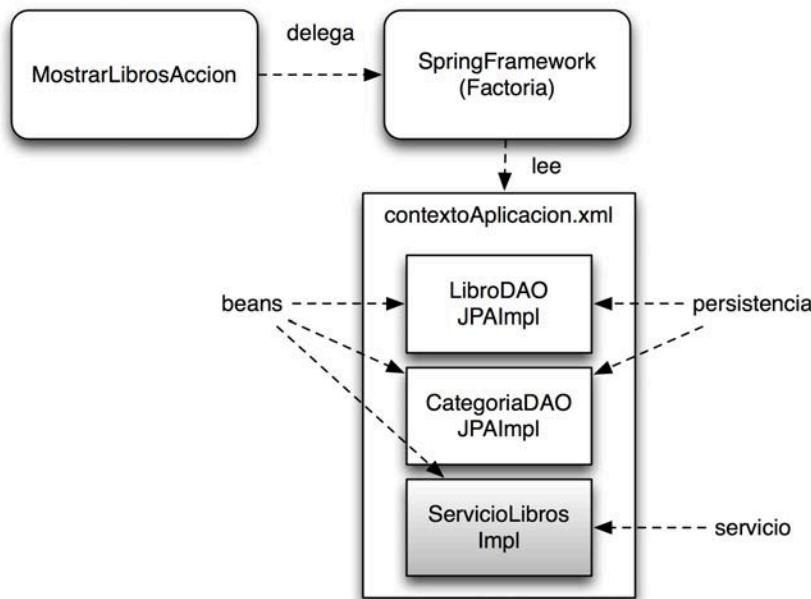
En este capítulo hemos visto una introducción al framework Spring y cómo este framework permite eliminar el sistema de factorías previamente construido, añadiendo mayor flexibilidad a la aplicación en proceso de construcción, utilizando el principio de inversión de control.

17. Inyección de Dependencia y Spring framework

En el capítulo anterior hemos comenzado a usar el principio de inversión de control para que la clase ServicioLibrosImpl se encargue de crear los distintos objetos de la capa de persistencia a través del framework Spring (ver imagen).



En este capítulo vamos a avanzar en el uso del principio de inversión de control y vamos a utilizar el framework Spring para que se encargue de inicializar no sólo las clases que pertenecen a la capa de persistencia sino también las clases de servicio, en nuestro caso ServicioLibrosImpl. De esta forma nuestras acciones delegarán ahora en el framework Spring para cargar las clases de servicio (ver imagen).

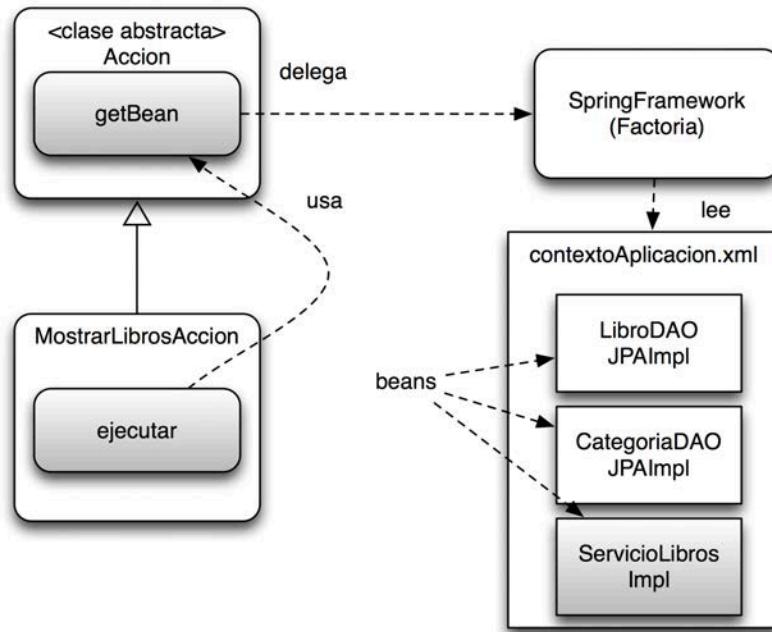


Para ello nuestro primer paso será modificar el fichero contextoAplicacion.xml y añadir nuestra clase de servicios (ver código).

Código 17.1: (contextoAplicacion.xml)

```
<bean id="servicioLibros"
      class="com.arquitecturajava.aplicacion.servicios.impl.ServicioLibrosImpl">
</bean>
```

Una vez realizada esta operación, necesitamos modificar la clase abstracta Acción de tal forma que el resto de las acciones se puedan apoyar en Spring al solicitar el servicio de libros. Para ello crearemos un nuevo método en esta clase denominado **getBean(String nombreBean)**, así todas las demás acciones al heredarlo pueden delegar en él para construir la clase de servicios necesarios (ver imagen).



Una vez entendido este concepto, vemos cómo implementar esta funcionalidad a nivel de la clase Acción. Ahora se muestra el código de nuestro nuevo método.

Código 17.2: (Accion.java)

```

public Object getBean(String nombre) {
    ClassPathXmlApplicationContextfactoria = new
    ClassPathXmlApplicationContext("contextoAplicacion.xml");
    return factoria.getBean(nombre);
}
    
```

Terminada esta operación, las acciones crearán los servicios de una forma muy sencilla como se muestra en el siguiente código de la clase **MostrarLibrosAccion**:

Código 17.3: (MostrarLibrosAccion.java)

```
public class MostrarLibrosAccion extends Accion {

    @Override
    public String ejecutar(HttpServletRequest request,
                           HttpServletResponse response) {

        ServicioLibros servicio =
            (ServicioLibros) getBean("servicioLibros");

        List<Libro> listaDeLibros = servicio.buscarTodosLosLibros();
        List<Categoria> listaDeCategorias =
            servicio.buscarTodasLasCategorias();
        request.setAttribute("listaDeLibros", listaDeLibros);
        request.setAttribute("listaDeCategorias", listaDeCategorias);
        return "MostrarLibros.jsp";
    }

}
```

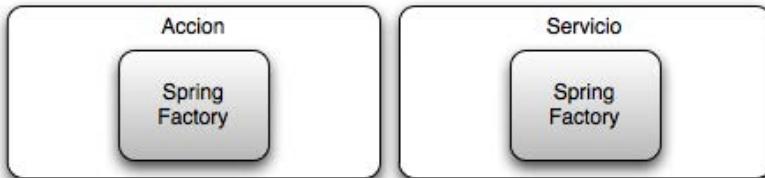
Tras modificar las acciones para que deleguen en el framework Spring al crear la capa de servicios, no olvidemos que también hemos usado en el capítulo anterior el framework Spring para crear objetos de la capa de persistencia necesarios para la capa de servicio, como muestra el siguiente bloque de código.

Código 17.4: (ServicioLibrosImpl.java)

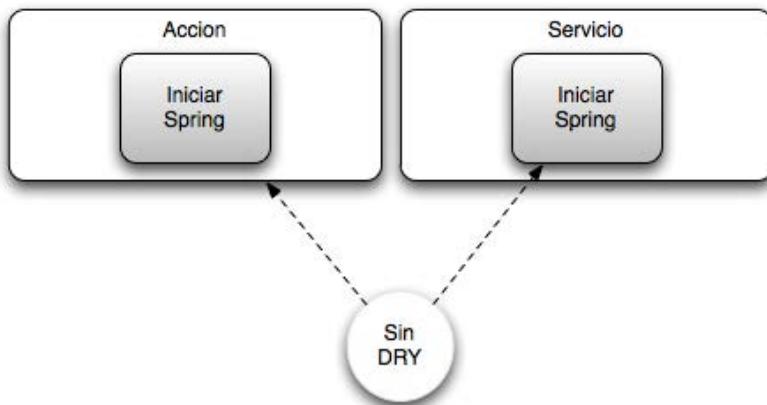
```
public ServicioLibrosImpl() {

    ClassPathXmlApplicationContext factoria =
        new ClassPathXmlApplicationContext("applicationContext.xml");
    libroDAO= (LibroDAO)factoria.getBean("libroDAO");
    categoriaDAO=(CategoriaDAO)factoria.getBean("categoriaDAO");
}
```

Así pues, en estos momentos hemos usado el framework Spring en dos partes de nuestra aplicación, primero a nivel de las acciones y segundo a nivel de los servicios (ver imagen).



Según vayamos progresando en el diseño de la aplicación nos daremos cuenta de que la responsabilidad al inicializar el framework se encuentra implementada en varias clases y por lo tanto tenemos un problema de repetición de código (ver imagen).



Visto el problema, es momento de definir los objetivos y tareas de este capítulo: nos centraremos en eliminar las repeticiones de código a la hora de inicializar Spring y nos encargaremos de centralizar la responsabilidad de cómo se construyen cada uno de los distintos objetos. Para ello nos apoyaremos en un patrón denominado Inyección de dependencia (DI) que será abordado más adelante.

Objetivos:

- Eliminar repeticiones de código en cuanto a la inicialización de Spring.
- Centralizar la responsabilidad de crear los distintos objetos.

Tareas:

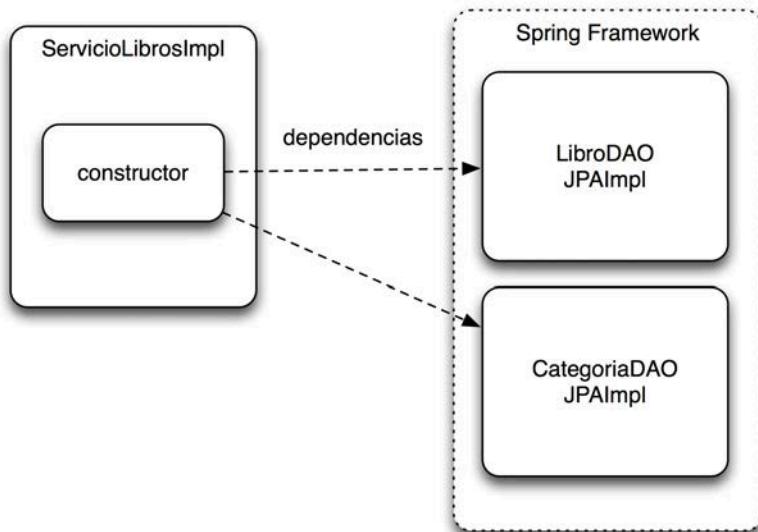
1. Introducción al patrón de Inyección de Dependencia
2. Spring como framework de Inyección de dependencia.
3. Spring y factoría web.
4. Spring inyección de dependencia y Capa DAO.

1. Introducción al principio de Inyección de Dependencia

En esta tarea vamos a introducir brevemente el patrón de Inyección de dependencia que consiste en lo siguiente:

Inyección de Dependencia: Las dependencias que una clase tiene no deben ser asignadas por ella misma sino por un agente externo.

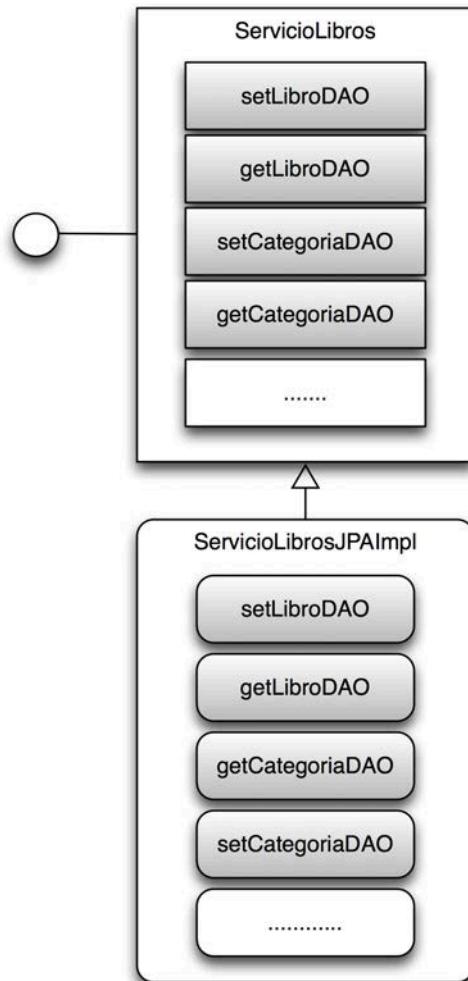
Quizá la definición no sea sencilla en un primer momento, para clarificarla vamos a aplicar el patrón de inyección de dependencia a las clases de servicio y persistencia que tenemos construidas. Antes de abordar esta tarea, vamos a mostrar cuál es la relación existente entre estas clases y cuáles han usado Spring para inicializar las distintas dependencias relativas a la clase ServiciosLibrosImpl.



Como podemos ver en el diagrama, la clase ServicioLibrosImpl ha hecho uso de Spring para inicializar sus dependencias. Vamos a utilizar el principio de inyección de dependencia en la clase ServicioLibroImpl de tal forma que, a partir de este momento no sea ella la encargada de inicializar sus dependencias sino un agente externo. Para ello pasaremos a añadirle los siguientes métodos a su implementación e interface.

- **setLibroDAO(LibroDAOlibroDAO)** :Asigna un objeto de tipo LibroDAO a la clase de servicio
- **getLibroDAO()** : Devuelve un objeto de tipo LibroDAO
- **setCategoriaDAO (CategoriaDAOcategoriaDAO)**: Asigna un objeto de tipo categoriaDAO a la clase de servicio.
- **getCategoriaDAO()** :Devuelve un objeto de tipo CategoriaDAO

Una vez claro qué métodos vamos a añadir, vamos a ver un diagrama que muestra como quedan el interface y la clase.



Visto el diagrama, vamos a ver el código fuente de ambos

```
public interface ServicioLibros {  
    public LibroDAO getLibroDAO() {  
        public void setLibroDAO(LibroDAO libroDAO) {  
            public CategoriaDAO getCategoriaDAO()  
            public void setCategoríaDAO(CategoríaDAO categoriaDAO)  
        //resto de métodos  
    }  
}
```

Código 17.5: (ServicioLibrosImpl.java)

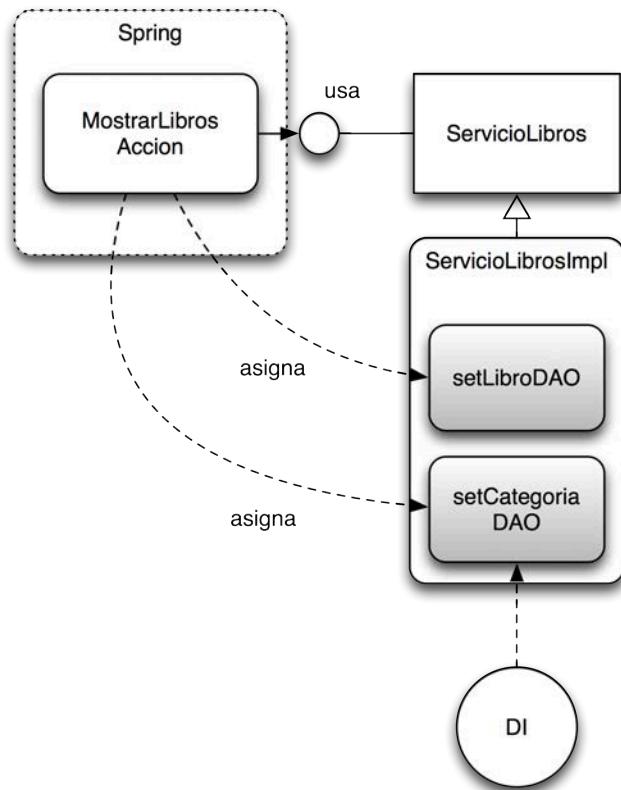
```
public class ServicioLibrosImpl implements ServicioLibros {  
  
    private LibroDAOlibroDAO=null;  
    private CategoriaDAOcategoriaDAO=null;  
    public LibroDAO getLibroDAO() {  
        return libroDAO;  
    }  
  
    public void setLibroDAO(LibroDAO libroDAO) {  
        this.libroDAO = libroDAO;  
    }  
  
    public CategoriaDAO getCategoriaDAO() {  
        return categoriaDAO;  
    }  
  
    public void setCategoriaDAO(CategoriaDAO categoriaDAO) {  
        this.categoriaDAO = categoriaDAO;  
    }  
    //resto de codigo
```

Una vez realizada esta operación, la clase ServicioLibroJPImpl ya no necesita que se le asignen las dependencias en su constructor ServicioLibroJPImpl() .Sino que estas dependencias podrán ser asignadas a través de los nuevos métodos set/get .Estos métodos serán invocados en nuestro caso por una clase de tipo Accion (ver código).

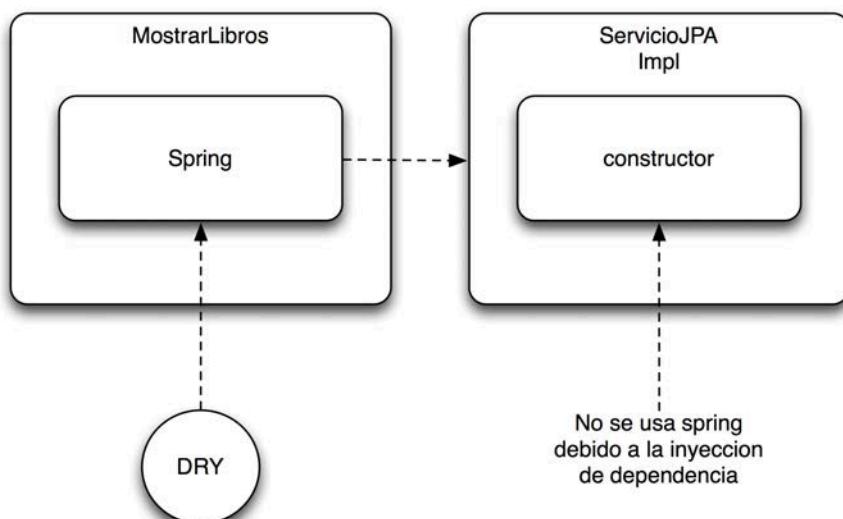
Código 17.6: (MostrarLibrosAccion.java)

```
public String ejecutar(HttpServletRequest request,  
                      HttpServletResponse response) {  
    ServicioLibro servicio= getBean("ServicioLibro");  
    CategoriaDAOcategoriaDAO= getBean("categoriaDAO");  
    LibroDAOlibroDAO= getBean("libroDAO");  
    servicio.setLibroDAO(libroDAO);  
    servicio.setCategoriaDAO(categoriaDAO)  
}
```

De esta manera habremos utilizado el patrón de Inyección de Dependencia para asignar las dependencias a nuestra clase de servicio. Ya no es necesario usar Spring a nivel de capa de servicios en el constructor del servicio sino que son las propias acciones las que asignan las dependencias necesarias (ver imagen).



De esta manera habremos dejado de inicializar varias veces el framework Spring y cumpliremos con el principio DRY como se ve en el diagrama.



2. Spring e inyección de dependencia.

Hemos visto cómo el principio de inyección de dependencia permite centralizar la creación de los distintos objetos y agrupar la responsabilidad de crearlos en las acciones. Ahora bien, aunque hemos usado el principio DRY y eliminado el código de inicialización de las clases de Servicio (ver código).

Código 17.7: (ServicioLibrosImpl.java)

```
public ServicioLibrosImpl() {  
    //ya no existe código de Spring  
}
```

Aun así todavía nos queda un problema por resolver :todas las acciones comparten el uso de la factoría de Spring y la inyección de dependencias entre la clase de servicios y las clases DAO como muestran los siguientes bloques de código que son prácticamente idénticos..

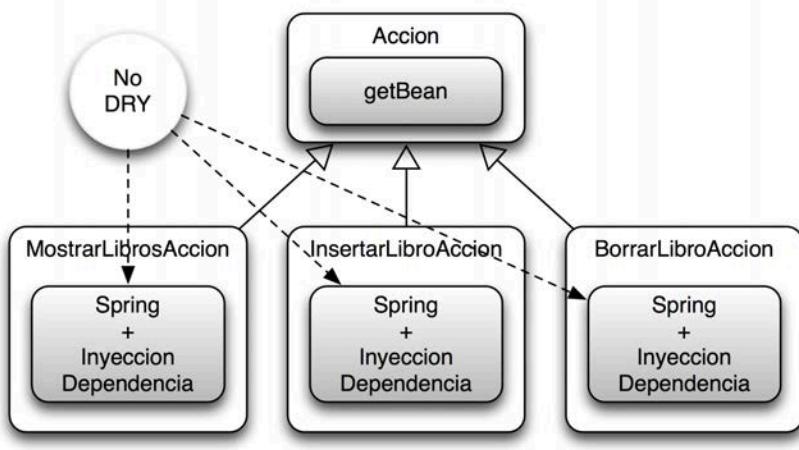
Código 17.8: (MostrarLibrosAccion.java)

```
public String ejecutar(HttpServletRequest request,  
                      HttpServletResponse response) {  
    ServicioLibro servicio=(ServicioLibro) getBean("ServicioLibro");  
    CategoriaDAO categoriaDAO= (CategoriaDAO) getBean("categoriaDAO");  
    LibroDAO libroDAO= (LibroDAO) getBean("libroDAO");  
    servicio.setLibroDAO(libroDAO);  
    servicio.setCategoriaDAO(categoriaDAO)  
    //resto de código  
}
```

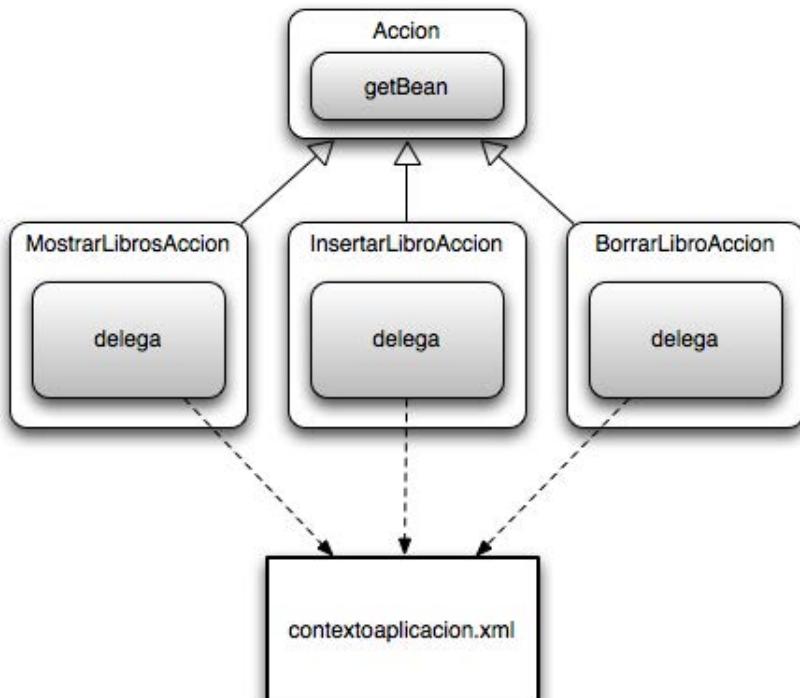
Código 17.9: (FiltroLibrosPorCategoriaAccion.java)

```
publicString ejecutar(HttpServletRequest request,  
                      HttpServletResponse response) {  
    ServicioLibro servicio=(ServicioLibro) getBean("ServicioLibro");  
    CategoriaDAO categoriaDAO= (CategoriaDAO) getBean("categoriaDAO");  
    LibroDAO libroDAO= (LibroDAO) getBean("libroDAO");  
    servicio.setLibroDAO(libroDAO);  
    servicio.setCategoriaDAO(categoriaDAO)  
}
```

No es difícil darse cuenta de que tenemos código repetido en las dos acciones : volvemos a tener un problema respecto al principio DRY (ver imagen).



Para solventar este problema vamos a usar Spring **como framework de Inyección de dependencia**. Los frameworks de inyección de dependencia permiten asignar las dependencias fuera de nuestro propio código fuente apoyándonos normalmente en ficheros de configuración xml. Este es el caso de Spring, que a través de su fichero xml es capaz de injectar las distintas dependencias a los distintos objetos que construyamos .Por lo tanto ahora todo el código de inyección de dependencia en cada una de nuestras clases de acción quedará relegado a un fichero xml (ver imagen).



Vamos a mostrar a continuación el código fuente del fichero contextoaplicación.xml para ver qué sintaxis usa el framework spring a la hora de inyectar las distintas dependencias y así clarificar los conceptos explicados.

Código 17.10: (contextoAplicacion.xml)

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="servicioLibros"
          class="com.arquitecturajava.aplicacion.servicios.impl.ServicioLibrosImpl">
        <propertyname="libroDAO" ref="libroDAO"></property>
        <propertyname="categoriaDAO" ref="categoriaDAO"></property>
    </bean>
    <bean id="libroDAO"
          class="com.arquitecturajava.aplicacion.dao.jpa.LibroDAOJPAImpl">
    </bean>
    <bean id="categoriaDAO"
          class="com.arquitecturajava.aplicacion.dao.jpa.CategoriaDAOJPAImpl">
    </bean>
</beans>
```

Como podemos ver, utilizamos la etiqueta **<property>** para usar el principio de inyección de dependencia e injectar desde el fichero XML las dependencias que nuestro servicio necesita. De esta forma nuestro código fuente a nivel de cada una de las acciones quedará simplificado pues únicamente necesitaremos obtener el servicio a través del método getBean y Spring a través de su fichero xml se encargará de injectar las distintas dependencias, como se muestra en el siguiente bloque de código que corresponde a la clase MostrarLibroAccion.

Código 17.11: (MostrarLibrosAccion.java)

```
public String ejecutar(HttpServletRequest request,
                       HttpServletResponse response) {
    ServicioLibros servicio = (ServicioLibros)
        getBean("servicioLibros");
    List<Libro> listaDeLibros = servicio.buscarTodosLosLibros();
    List<Categoria> listaDeCategorias = servicio.buscarTodasLasCategorias();
    request.setAttribute("listaDeLibros", listaDeLibros);
    request.setAttribute("listaDeCategorias", listaDeCategorias);
    return "MostrarLibros.jsp";
}
```

3. Spring y factoría para aplicaciones web

Hasta este momento hemos estado utilizando la factoría clásica de Spring (ver código).

Código 17.12: (Accion.java)

```
public Object getBean(String nombre) {
    ClassPathXmlApplicationContextfactoria = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    returnfactoria.getBean(nombre);
}
```

Es momento de configurar Spring para que use una factoría específica a nivel de aplicación web. Por eso es necesario modificar el método `getBean` de nuestra clase `Accion` y substituir la factoría habitual por el siguiente bloque de código.

Código 17.13: (MostrarLibrosAccion.java)

```
public Object getBean(String nombre,HttpServletRequest request) {
    WebApplicationContextfactoria =WebApplicationContextUtils.
    getRequiredWebApplicationContext(request.getSession().
    getServletContext());
    return factoria.getBean(nombre);
}
```

De esta forma nos aseguraremos de que Spring sólo carga el fichero de configuración una única vez al arrancar la aplicación web. Realizado este cambio, vamos a seguir aplicando el principio de inyección de dependencia a otras clases de nuestra aplicación.

4. Spring inyección de dependencia y Capas DAO

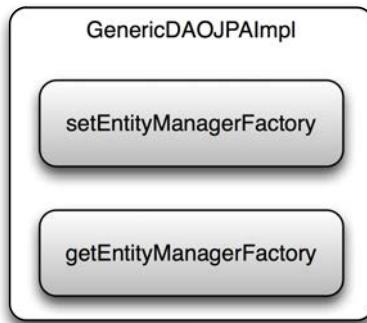
Hemos trabajado con el framework Spring como inyector de dependencia a la hora de crear los servicios y las capas DAO. Es momento de aplicar el principio de inyección de dependencia a las propias clases DAO que tenemos construidas. Si revisamos los distintos métodos de la clase `GenericDAOJPImpl` como por ejemplo el siguiente.

Arquitectura Java

Código 17.14: (GenericDAOJPImpl.java)

```
public void borrar(T objeto) {  
  
    EntityManager manager = getEntityManagerFactory().createEntityManager();  
    EntityTransaction tx = null;  
    try {  
        tx = manager.getTransaction();  
        tx.begin();  
        manager.remove(manager.merge(objeto));  
        tx.commit();  
    } catch (PersistenceException e) {  
        tx.rollback();  
        throw e;  
    } finally {  
        manager.close();  
    }  
}
```

Nos daremos cuenta de que no solo este método sino todos los demás dependen de un objeto EntityManagerFactory. Así pues vamos a refactorizar el código de la clase GenericDAOJPImpl así como el de su interface para inyectar a través de Spring dicho objeto (ver imagen).



Una vez que tenemos claro qué operación vamos a realizar, es momento de ver el código fuente de nuestra nueva clase.

Código 17.15: (GenericDAOJPImpl.java)

```
public abstract class GenericDAOJPImpl<T, Id extends Serializable> implements
GenericDAO<T, Id> {
    .....
    public EntityManagerFactory getEntityManagerFactory() {
        return entityManagerFactory;
    }
    public void setEntityManagerFactory(EntityManagerFactory
        entityManagerFactory) {
        this.entityManagerFactory = entityManagerFactory;
    }
}
```

Una vez definidos los nuevos métodos que nos permitirán inyectar la dependencia de EntityManagerFactory a nuestras clases DAO, es momento de ver qué nuevos bean han de ser añadidos a nivel de fichero xml .A continuación se enumeran:

- **DriverManagerDataSource**: Bean encargado de crear un pool de conexiones contra la base de datos que seleccionemos pasando usuario, password , url, driver.
- **EntityManagerFactory**: Hace uso del DataSource creado y asigna el tipo de persistencia y algunos parámetros adicionales, como es el dialecto de JPA que se utilizará .

A continuación se muestra la nueva versión del fichero.

Código 17.16: (contextoAplicacion.java)

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="fuenteDeDatos"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost/arquitecturaJavaORM" />
    <property name="username" value="root" />
    <property name="password" value="java" />
</bean>
```

```
<bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

    <property name="persistenceUnitName" value="arquitecturaJava" />
    <property name="dataSource" ref="fuenteDeDatos" />
    <property name="jpaVendorAdapter">
      <bean
        class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        </bean>
        <property name="databasePlatform"
          value="org.hibernate.dialect.MySQL5Dialect" />
        <property name="showSql" value="true" />
      </bean>
    </property>
  </bean>
<bean id="servicioLibros"
  class="com.arquitecturajava.aplicacion.servicios.impl.ServicioLibrosImpl">
    <property name="libroDAO" ref="libroDAO"/>
    <property name="categoriaDAO" ref="categoriaDAO"/>
  </bean>
<bean id="libroDAO"
  class="com.arquitecturajava.aplicacion.dao.jpa.LibroDAOJPImpl">
    <property name="entityManagerFactory"
      ref="entityManagerFactory"/>
  </bean>
<bean id="categoriaDAO"
  class="com.arquitecturajava.aplicacion.dao.jpa.CategoriaDAOJPImpl">
    <property name="entityManagerFactory"
      ref="entityManagerFactory"/>
  </bean>
</beans>
```

Es evidente que ahora también las capas DAO se apoyan en Spring para inyectar la dependencia que necesitan de EntityManagerFactory. Esta dependencia (EntityManagerFactory) necesita a su vez la inyección de varias propiedades y de una fuente de datos (pool de conexiones a base de datos). Es lo que hemos configurado en el fichero. Veamos como estos cambios afectan a nuestros métodos:

Código 17.16: (GenericDAOJPALimpl.java)

```
public void salvar(T objeto) {
    EntityManager manager =
        getEntityManagerFactory().createEntityManager();

    EntityTransaction tx = null;
    try {
        tx = manager.getTransaction();
        tx.begin();
        manager.merge(objeto);
        tx.commit();

    } catch (PersistenceException e) {
        tx.rollback();
        throw e;
    } finally {
        manager.close();
    }
}
```

Resumen

En este capítulo nos hemos centrado en explicar el patrón de Inyección de dependencia y cómo se configura a través del framework Spring y nos hemos apoyado en él para redefinir la configuración de las clases de servicio y capas DAO, simplificando el código de nuestras clases y externalizando la configuración en gran medida.

18. El principio DRY y Spring Templates

En el capítulo anterior hemos visto cómo usar el principio de inyección de dependencia que nos permite centralizar la responsabilidad de crear objetos en el framework Spring y su fichero de configuración. Este capítulo volverá a centrarse en la evolución de la capa de persistencia. Para ello vamos a revisar dos métodos de la capa de persistencia. En concreto dos que pertenecen a la clase GenericDAOJPALimpl: salvar y borrar. A continuación mostramos el código de los mismos:

Código 18.1: (GenericDAOJPALimpl.java)

```
public void salvar(T objeto) {
    EntityManager manager =
        getEntityManagerFactory().createEntityManager();

    EntityTransaction tx = null;
    try {
        tx = manager.getTransaction();
        tx.begin();
        manager.merge(objeto);
        tx.commit();

    } catch (PersistenceException e) {
        tx.rollback();
        throw e;
    } finally {
        manager.close();
    }
}
```

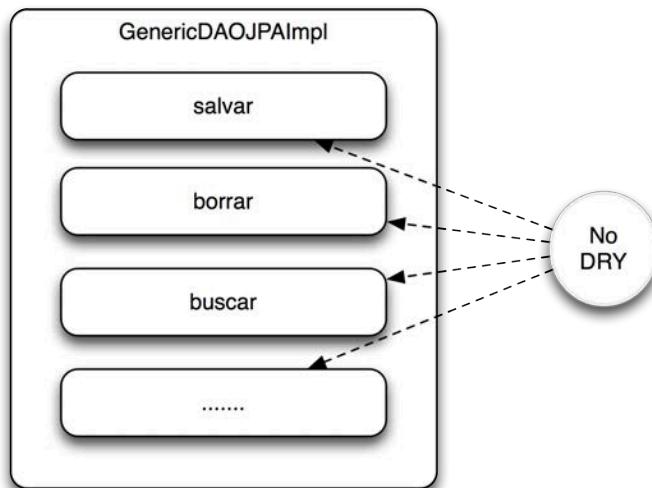
Código 18.2: (GenericDAOJPALimpl.java)

```

public void borrar(T objeto) {
    EntityManager manager = getEntityManagerFactory().createEntityManager();
    EntityTransaction tx = null;
    try {
        tx = manager.getTransaction();
        tx.begin();
        manager.remove(manager.merge(objeto));
        tx.commit();
    } catch (PersistenceException e) {
        tx.rollback();
        throw e;
    } finally {
        manager.close();
    }
}

```

Como podemos ver, el código de la capa de persistencia de ambos métodos, aunque no es idéntico sí que se trata de código muy similar (hay bloques idénticos). Por lo tanto, nos encontramos otra vez en una situación de violación del principio DRY, ya que tenemos grandes bloques de código repetidos en métodos distintos (ver imagen).



Una vez que tenemos clara la repetición de código, podemos identificar que ambos métodos comparten los mismos pasos que a continuación se enumeran.

- Obtener EntityManager
- Abrir nueva Transaccion
- Ejecutar/Abortar Transaccion
- Cerrar Recursos

El objetivo principal de este capítulo será eliminar dichas repeticiones de código de tal forma que las clases de persistencia queden mucho más sencillas. Para ello introduciremos un nuevo patrón: el patrón Template o plantilla.

Objetivos:

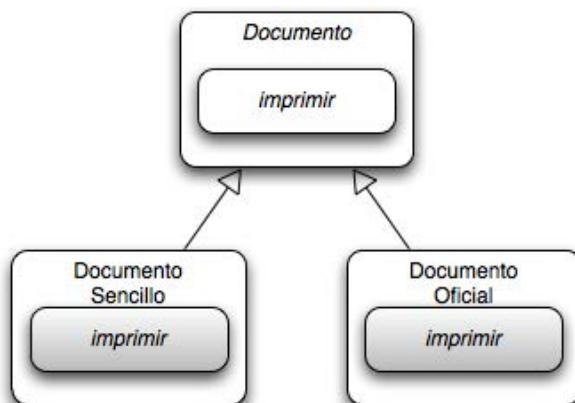
- Eliminar repeticiones de código en la capa DAO

Tareas

1. Introducir el patrón de diseño Template
2. Spring y el patrón Template
3. Spring Plantillas y JPADAOSupport

1. El patrón Template

El patrón Template o plantilla es un patrón de diseño que se encarga de definir una serie de métodos a nivel de clase que funcionen en forma de plantilla . Así que clarificaremos el funcionamiento de este patrón a través de un ejemplo, para ello vamos a crear la clase Documento y dos clases hijas: Documento Sencillo y Documento Oficial. La clase documento nos aportará el método imprimir que se encarga de imprimir por pantalla el texto del documento .En la siguiente imagen se muestra la relación de clases y métodos soportados.



Vamos a ver a continuación el código fuente de cada una de ellas y cómo funciona el programa cuando imprime un documento.

Código 18.3: (Documento.java)

```
package com.arquitecturajava;
public abstract class Documento {
    public abstract void imprimir(String mensaje);
}
```

Código 18.4: (DocumentoOficial.java)

```
package com.arquitecturajava;
public class DocumentoOficial extends Documento{
    public void imprimir(String mensaje) {
        System.out.println("<oficial>cabecera documento oficial</oficial>");
        System.out.println("<oficial>" + mensaje + "</oficial>");
        System.out.println("<oficial>pie documento oficial</oficial>");
    }
}
```

Código 18.5: (Documento.java)

```
package com.arquitecturajava;
public class DocumentoPlano extends Documento {
    public void imprimir(String mensaje) {
        System.out.println("cabecera documento sencillo");
        System.out.println(mensaje);
        System.out.println("pie documento sencillo");
    }
}
```

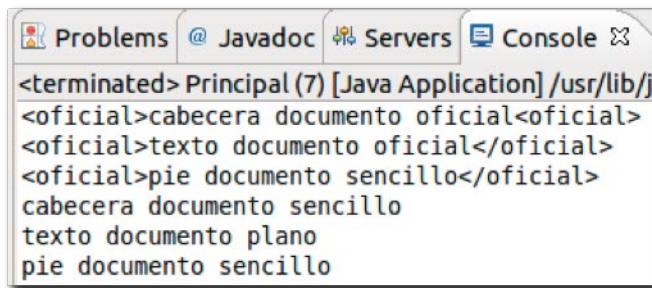
Tras construir el código de cada una de nuestras claves, vamos a ver cómo queda el código del programa main o principal que se encarga de imprimir ambos documentos por pantalla.

Arquitectura Java

Código 18.6: (Principal.java)

```
package com.arquitecturajava;
public class Principal {
    public static void main(String[] args) {
        Documento d= new DocumentoOficial();
        d.imprimir("texto documento oficial");
        d= new DocumentoPlano();
        d.imprimir("texto documento plano");
    }
}
```

El resultado de imprimir estos documentos será el siguiente:



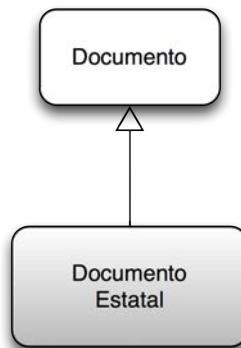
The screenshot shows a Java application running in an IDE. The title bar says '<terminated> Principal (7) [Java Application] /usr/lib/j'. The console tab is active, displaying the following text:
<oficial>cabecera documento oficial</oficial>
<oficial>texto documento oficial</oficial>
<oficial>pie documento sencillo</oficial>
cabecera documento sencillo
texto documento plano
pie documento sencillo

Como podemos ver todo se imprime correctamente y parece que el programa construido no tiene fisuras. Ahora bien, imaginémonos que tenemos otro tipo de documento prácticamente idéntico al oficial pero que al ser a nivel del estado, la cabecera cambia de forma puntual y tiene el siguiente texto.

Código 18.7: (Principal.java)

```
<oficial> cabecera documento ESTATAL</oficial>
```

¿Cómo podemos abordar este cambio de funcionalidad con el código actual? En este caso es sencillo: simplemente debemos crear una nueva clase que extienda de la clase documento y se denomine DocumentoEstatal. Véase una imagen aclaratoria:



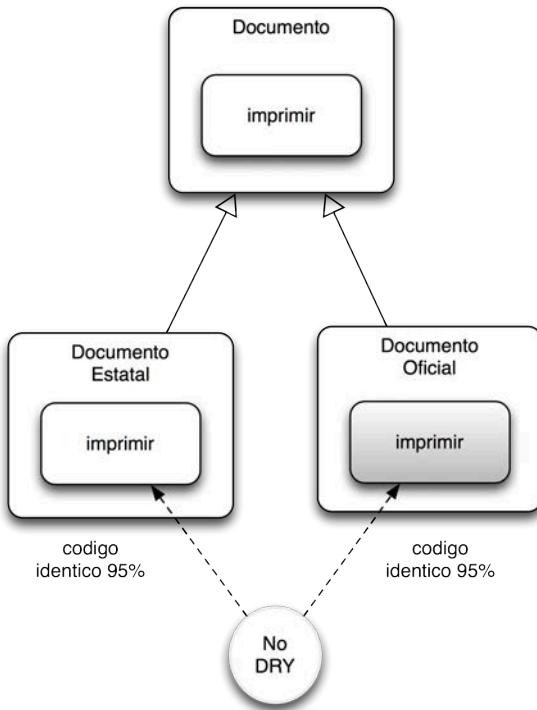
A continuación se muestra el código fuente de la nueva clase que define el documento estatal

Código 18.8: (DocumentoEstatal.java)

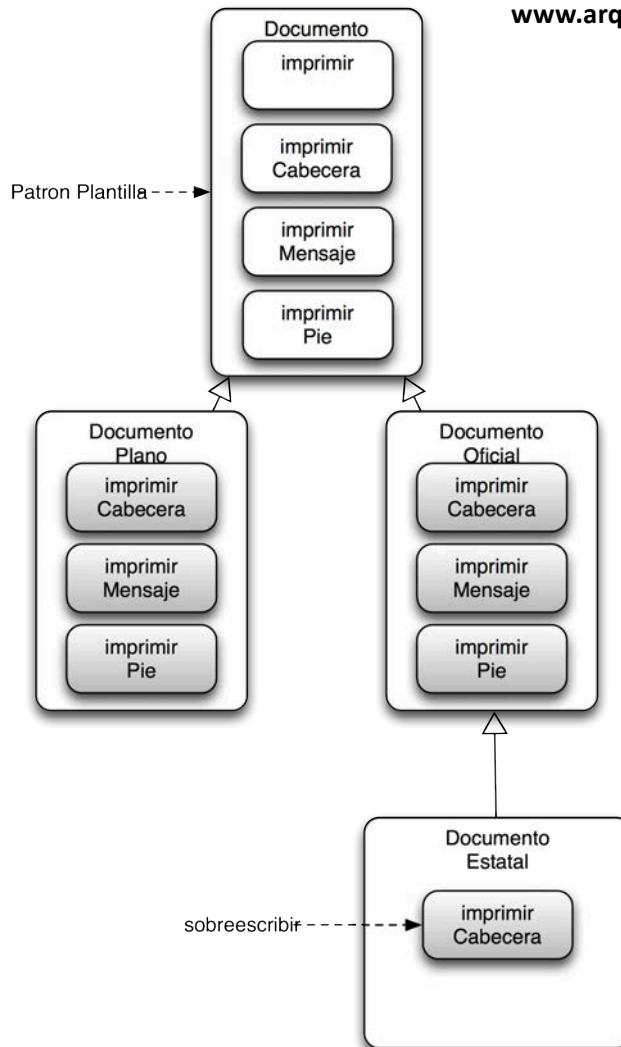
```

package com.arquitecturajava;
public class DocumentoEstatal extends Documento{
    public void imprimir(String mensaje) {
        System.out.println("<oficial>cabecera documento ESTATAL</oficial>");
        System.out.println("<oficial>" +mensaje+ "</oficial>");
        System.out.println("<oficial>pie documento oficial</oficial>");
    }
}
  
```

A través de la herencia hemos solventado en principio el problema del nuevo tipo de documento .Ahora bien, si nos fijamos más, percibiremos un problema de código a nivel del principio DRY, ya que la clase DocumentoOficial y DocumentoEstatal comparten código al 95% (ver imagen).



Una vez tenemos claro cuál es nuestro problema, vamos a refactorizar nuestro código usando el patrón Template o plantilla. Éste se encarga de dividir la funcionalidad de un método en un conjunto de métodos que implementan la misma funcionalidad y que forman una estructura de plantilla, ya que cada uno de estos métodos puede ser sobrescrito por las clases hijas dependiendo de nuestras necesidades. Vamos a ver una imagen aclaratoria de la estructura de este patrón:



Una vez realizada esta operación, podremos crear nuevas clases que se encarguen de sobreescribir aquellas partes que nos interesen. Vamos a ver a continuación el código de este nuevo diagrama de clases:

Arquitectura Java

Código 18.9: (DocumentoEstatal.java)

```
package com.arquitecturajava.plantillas;
public abstract class Documento {
    public void imprimir(String mensaje) {
        imprimirCabecera();
        imprimirMensaje(mensaje);
        imprimirPie();
    }
    protected abstract void imprimirCabecera();
    protected abstract void imprimirMensaje(String mensaje);
    protected abstract void imprimirPie();
}
```

Código 18.10: (DocumentoOficial.java)

```
package com.arquitecturajava.plantillas;

public class DocumentoOficial extends Documento{
    @Override
    public void imprimirCabecera() {
        System.out.println("<oficial>cabecera documento oficial </oficial>");
    }
    @Override
    public void imprimirMensaje(String mensaje) {
        System.out.println("<oficial>" + mensaje + "</oficial>");
    }
    @Override
    public void imprimirPie() {
        System.out.println("<oficial>pie documento sencillo</oficial>");
    }
}
```

Visiblemente el código es bastante distinto, pero si ejecutamos otra vez nuestro programa, el resultado será idéntico. Ahora bien, cuando deseemos construir la clase DocumentoEstatal ,podremos hacer uso del patrón plantilla y únicamente sobreescribir la parte que realmente cambia.

Código 18.11: (DocumentoEstatal.java)

```
package com.arquitecturajava.plantillas;
public class DocumentoEstatal extends DocumentoOficial{
    public void imprimirCabecera() {
        System.out.println("<oficial>cabecera documento ESTATAL</oficial>");
    }
}
```

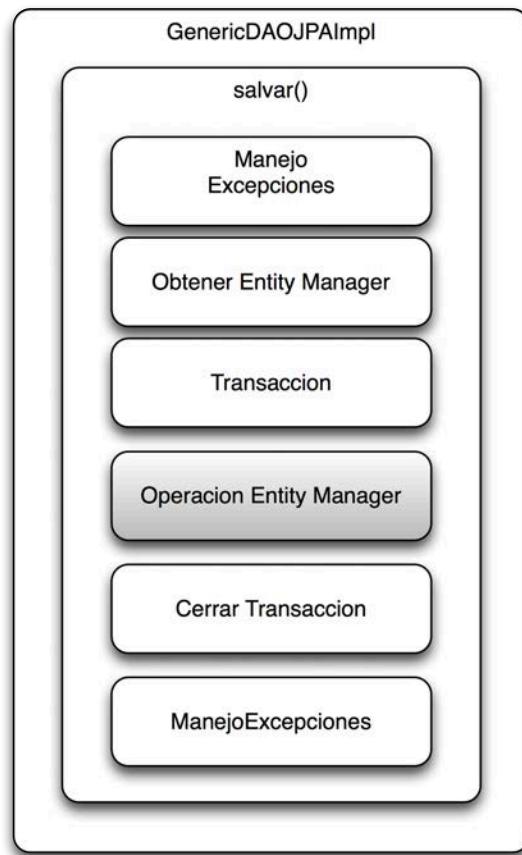
Para construir el documento estatal únicamente hemos tenido que sobrescribir un método y hemos podido eliminar las repeticiones de código (ver imagen).



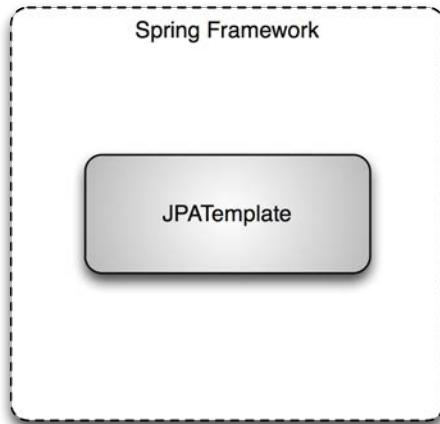
A continuación vamos a usar el patrón Template junto al framework Spring para eliminar repeticiones en la capa de persistencia.

2. Spring y plantillas.

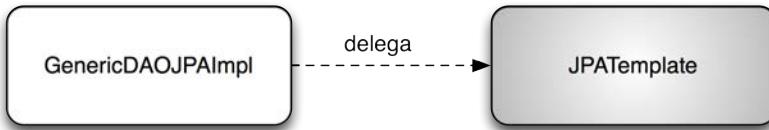
Si revisamos cualquiera de los métodos que en estos momentos tenemos en la capa de persistencia, nos daremos cuenta de que se comportan de una forma muy similar al patrón Template que acabamos de comentar, ya que cada método de persistencia puede dividirse en varias operaciones (ver imagen).



Prácticamente toda la funcionalidad que construimos en los distintos métodos de persistencia es idéntica, salvo la operación concreta que el EntityManager ejecuta. Así pues toda esta funcionalidad puede ser rediseñada para que se apoye en el patrón Template. Para ello el framework Spring nos provee de una clase que implementa toda la funcionalidad de Manejo de Excepciones, Gestión de Entity Manager y Transacciones. A continuación se muestra un diagrama con la clase.



Una vez que sabemos que el framework Spring nos puede ayudar a gestionar la capa de persistencia a través de una clase plantilla, vamos a apoyarnos en esta clase para refactorizar el código de las clases DAO. Para ello refactorizaremos nuestra clase GenericDAOJPALimpl para que delegue en la clase JPATemplate del framework Spring al realizar todas las operaciones de persistencia.



Vamos a ver a continuación cómo queda refactorizado el código de nuestra clase y cómo se apoya en la plantilla.

Arquitectura Java

Código 18.12: (GenericDAOJPAImpl.java)

```
package com.arquitecturajava.aplicacion.dao.jpa;
//omitimos imports
public abstract class GenericDAOJPAImpl<T, Id extends Serializable>implements
    GenericDAO<T, Id> {

    private Class<T> claseDePersistencia;
    private JpaTemplate plantillaJPA;

    public JpaTemplate getPlantillaJPA() {
        return plantillaJPA;
    }
    public void setPlantillaJPA(JpaTemplate plantillaJPA) {
        this.plantillaJPA = plantillaJPA;
    }
    @SuppressWarnings("unchecked")
    public GenericDAOJPAImpl() {

        this.claseDePersistencia = (Class<T>) ((ParameterizedType)getClass()
            .getGenericSuperclass()).getActualTypeArguments()[0];
    }
    @Override
    public T buscarPorClave(Id id) {
        return plantillaJPA.find(claseDePersistencia, id);
    }
    @SuppressWarnings("unchecked")
    public List<T> buscarTodos() {
        return plantillaJPA.find("select o from "
            + claseDePersistencia.getSimpleName() + " o");
    }
    public void borrar(T objeto) {
        plantillaJPA.remove(plantillaJPA.merge(objeto));
    }
    public void salvar(T objeto) {
        plantillaJPA.merge(objeto);
    }
    public void insertar(T objeto) {
        plantillaJPA.persist(objeto);
    }
}
```

Como podemos ver el framework Spring nos aporta una clase JPATemplate que ya implementa toda la funcionalidad necesaria de JPA y que es inyectada (ver código).

Código 18.13: (GenericDAOJPImpl.java)

```

private JpaTemplate plantillaJPA;
public JpaTemplate getPlantillaJPA() {
    return plantillaJPA;
}
public void setPlantillaJPA(JpaTemplate plantillaJPA) {
    this.plantillaJPA = plantillaJPA;
}

```

Al realizar esta operación todos los métodos de la clase GenericDAOJPImpl quedarán claramente simplificados. Es evidente que la clase JPATemplate actúa como una plantilla y únicamente necesitamos ejecutar un método concreto ya que el resto de funcionalidad viene implementada por defecto.

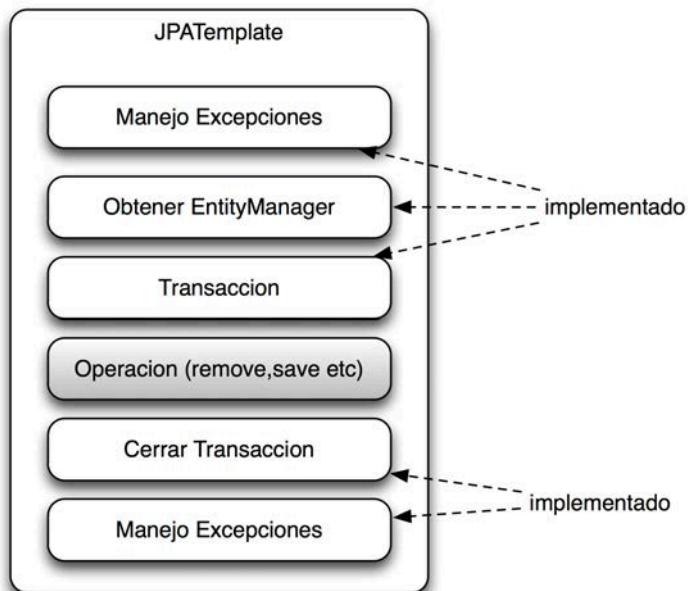
Código 18.14: (GenericDAOJPImpl.java)

```

public void borrar(T objeto) {
    plantillaJPA.remove(plantillaJPA.merge(objeto));
}

```

La siguiente imagen clarifica cómo funciona la clase JPATemplate en cuanto a los métodos que maneja y qué partes de la funcionalidad vienen implementadas ya por defecto, de tal forma que no debemos preocuparnos por ello.



Arquitectura Java

Una vez realizada esta operación, deberemos usar Spring como Inyector de dependencia y configurar la plantilla de JPA a nivel del fichero XML al ser la plantilla la encargada de la gestión de la capa de persistencia .Será esta clase a la que inyectemos el entityManagerFactory (ver imagen).

Código 18.15: (contextoAplicacion.xml)

```
<bean id="plantillaJPA" class="org.springframework.orm.jpa.JpaTemplate">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
<bean id="servicioLibros"
    class="com.arquitecturajava.aplicacion.servicios.impl.ServicioLibrosImpl">
    <property name="libroDAO" ref="libroDAO"></property>
    <property name="categoriaDAO" ref="categoriaDAO"></property>
</bean>
<bean id="libroDAO"
    class="com.arquitecturajava.aplicacion.dao.jpa.LibroDAOJPImpl">
    <property name="plantillaJPA" ref="plantillaJPA" />
</bean>
<bean id="categoriaDAO"
    class="com.arquitecturajava.aplicacion.dao.jpa.CategoriaDAOJPImpl">
    <property name="plantillaJPA" ref="plantillaJPA" />
</bean>
```

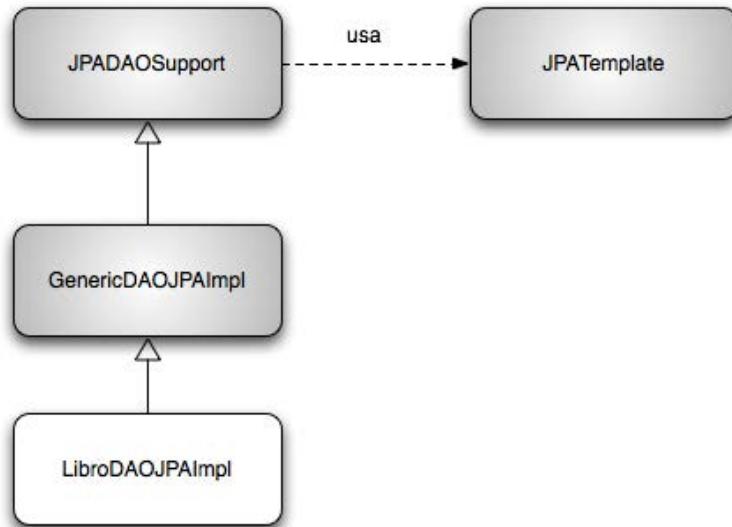
Ejecutada esta operación, la aplicación seguirá funcionando de la misma forma con la ventaja de haber simplificado sobremanera el código que existía en nuestras clases DAO, cumpliendo con el principio DRY.

3. Spring Herencia Plantillas y JPADAOsupport

Otra opción más practica es usar una clase que ya aporte la plantilla que necesitamos y que nuestra clase genérica extienda de ella. El Framework Spring nos aporta esta clase a nivel de JPA y se denomina JPADAOsupport (ver imagen).



Una vez que esta clase hace uso de JPATemplate, podemos hacer que nuestra clase genérica extienda de ella y así todas nuestras clases DAO implementarán la funcionalidad (ver imagen).



A continuación se muestra el código con la nueva versión de la clase:

Código 18.16: (GenericDAOJPAImpl.java)

```

package com.arquitecturajava.aplicacion.dao.jpa;
//omitimos imports
public abstract class GenericDAOJPAImpl<T, Id extends Serializable>extends
JpaDaoSupport implements
    GenericDAO<T, Id> {

    private Class<T> claseDePersistencia;
    @SuppressWarnings("unchecked")
    public GenericDAOJPAImpl() {
        this.claseDePersistencia = (Class<T>) ((ParameterizedType) getClass()
            .getGenericSuperclass()).getActualTypeArguments()[0];
    }
    @Override
    public T buscarPorClave(Id id) {
        return getJpaTemplate().find(claseDePersistencia, id);
    }
    // resto de métodos de persistencia no cambian
}
  
```

Arquitectura Java

Como podemos ver la clase ha quedado muy simplificada y la capa de persistencia muy sencilla, pero debemos modificar de nuevo nuestro fichero de configuración de Spring ya que nuestras clases ahora se apoyaran en el método setEntityManagerFactory que implementa la clase JPADAOSupport (ver código).

Código 18.17: (contextoAplicacion.xml)

```
<bean id="servicioLibros"
      class="com.arquitecturajava.aplicacion.servicios.impl.ServicioLibrosImpl">
    <property name="libroDAO" ref="libroDAO"></property>
    <property name="categoriaDAO" ref="categoriaDAO"></property>
</bean>
<bean id="libroDAO"
      class="com.arquitecturajava.aplicacion.dao.jpa.LibroDAOJPAImpl">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
<bean id="categoriaDAO"
      class="com.arquitecturajava.aplicacion.dao.jpa.CategoríaDAOJPAImpl">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Resumen

Hemos refactorizado nuestra capa DAO a través del uso del patrón plantilla (template) y hemos eliminado mucho código redundante. En estos momentos la capa de persistencia ha quedado muy simplificada, el principio DRY ha vuelto a ser una de las piezas claves para poder seguir avanzando con el diseño.

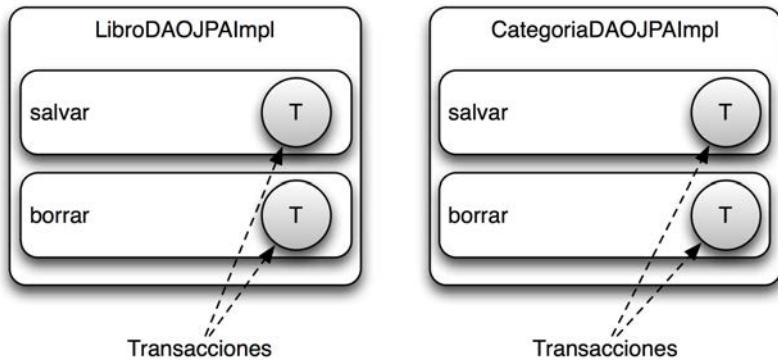
19. Programación Orientada a Aspecto (AOP)

En el capítulo anterior hemos utilizado el principio DRY para simplificar el código de la capa de persistencia y reducirlo a la mínima expresión, como muestra el siguiente método.

Código 19.1: (GenericDAOJPALimpl.java)

```
public void borrar(T objeto) {  
    getJpaTemplate().remove(getJpaTemplate().merge(objeto));  
}
```

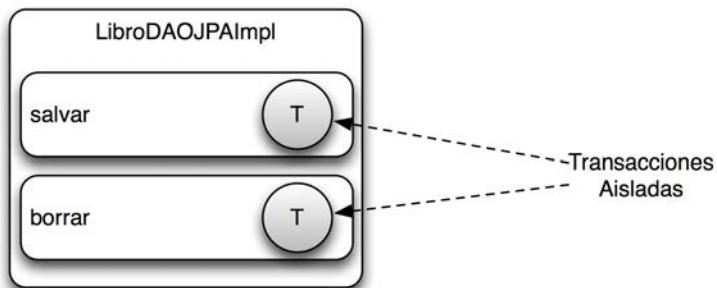
Parece que es imposible simplificar más el código que tenemos y estamos en lo cierto en cuanto al principio DRY se refiere. Aun así, existen algunas mejoras que se pueden realizar en la capa de persistencia aunque no son evidentes. Si revisamos el código que acabamos de mostrar, nos podremos dar cuenta de que no solo se encarga de borrar un objeto de la base de datos, sino que además ejecuta esa operación dentro de una transacción. Igual que este método ejecuta una transacción, el resto de métodos de modificación también realizan esta operación, como se muestra en la imagen.



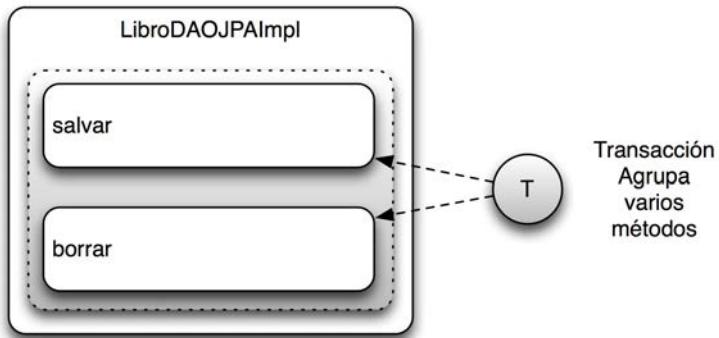
En principio esto no parece acarrear ningún problema ya que al habernos apoyado en las plantillas en el capítulo anterior, no tenemos código repetido en estas clases. Lo que sucede es que la responsabilidad sobre la gestión de transacciones está distribuida y cada uno de los métodos es responsable de gestionar sus propias transacciones. Ahora bien, supongamos que deseamos realizar la siguiente transacción:

- insertar un nuevo libro
- borrar un libro

Ambas operaciones se tienen que realizar de una forma atómica e indivisible es decir: o se realizan ambas operaciones o ninguna. Es en este momento cuando nos encontraremos con el siguiente problema: cada una de estas operaciones es una transacción completamente independiente de la otra (ver imagen).



No es posible ,tal como tenemos construida la solución, definir una transacción que agrupe operaciones de varios métodos de forma conjunta (ver imagen).



Así pues nuestro diseño actual tiene un problema importante ya que la necesidad de que varios métodos se ejecuten de forma conjunta (transaccional) es algo muy necesario para las aplicaciones. Para solventar este problema, necesitamos centralizar la gestión de las transacciones y no tenerla dispersa por todo el código. Este capítulo se centrará en solventar esta cuestión. Para ello introduciremos el concepto de programación orientada a aspecto o AOP (AspectOrientedProgramming).

Objetivos:

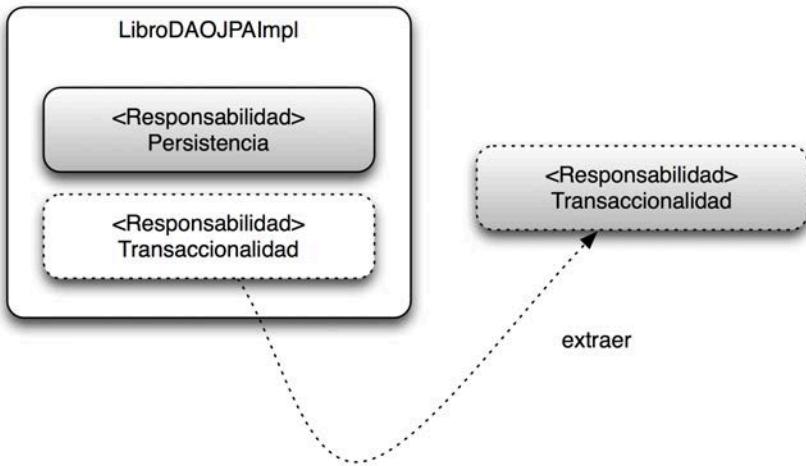
- Centralizar la gestión de transacciones.

Tareas:

1. Introducción a AOP.
2. Spring y Proxies.
3. Configuración de proxies con spring

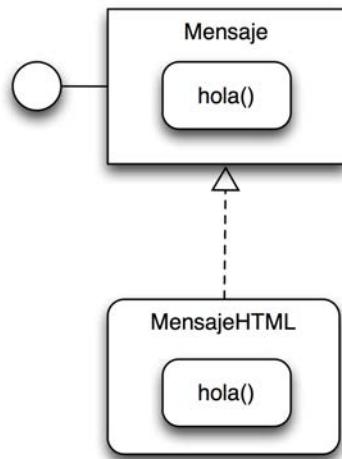
1. Introducción a AOP

En estos momentos cada uno de los métodos de nuestras clases se encarga de gestionar de forma independiente las transacciones .Si queremos unificar la gestión de transacciones, deberemos extraer la responsabilidad de ejecutar las transacciones de nuestras clases (ver imagen).



No parece sencillo realizar esta operación ya que si extraemos la responsabilidad de nuestras clases, no podremos ejecutar los métodos de forma transaccional ya que no dispondrán de esa capacidad. Para solventar esto, vamos a volver a retomar el ejemplo de la clase Mensaje y vamos a usar esta clase para introducir algunos conceptos fundamentales de la programación orientada a aspecto. En este caso, un patrón de diseño fundamental para este tipo de programación el patrón proxy.

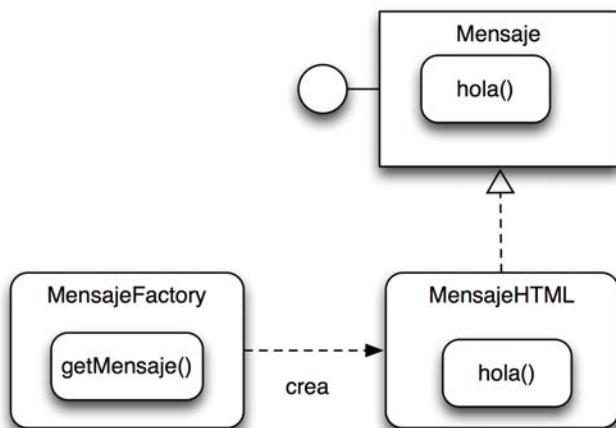
Patrón Proxy : Este patrón hace las tareas de intermediario entre un objeto y su cliente permitiendo controlar el acceso a él añadiendo o modificando la funcionalidad existente de forma transparente. Para aclarar el concepto, vamos a construir un ejemplo sencillo que nos facilite entender cómo se construye un Proxy, a tal fin vamos a partir del interface Mensaje y de su implementación MensajeHTML, como se muestra en la siguiente figura y bloque de código.



Código 19.2: (MensajeHTML.java)

```
public class MensajeHTML implements Mensaje {  
    public void hola() {  
        System.out.println("<html>hola</html>");  
    }  
}
```

Para construir objetos de la clase MensajeHTML vamos a hacer uso de una sencilla factoría (ver imagen).



El código de la factoria es el siguiente.

Código 19.3: (Principal.java)

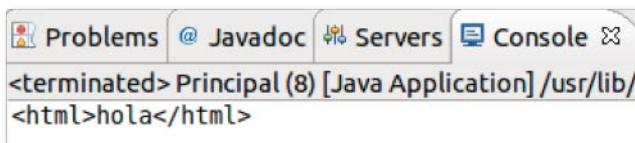
```
public class MensajeFactory {  
    public static Mensaje getMensaje() {  
        return new MensajeHTML();  
    }  
}
```

Realizada esta operación, vamos a crear un objeto de nuestra clase a través de la factoría.

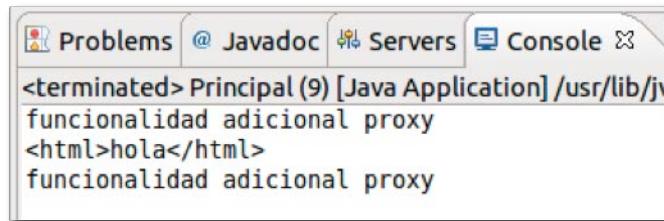
Código 19.4: (Principal.java)

```
MensajeFactormifactoria= new MensajeFactory()  
Mensaje mensaje= mifactoria.getMensaje();  
mensaje.hola ()
```

A continuación podemos ver el resultado por la consola.



Así pues la clase factoria se encarga simplemente de crear el objeto y luego nosotros simplemente invocamos al método hola() y el código funciona perfectamente. Ahora bien, si quisieramos añadir una nueva responsabilidad a la clase de tal forma que al invocar el método hola se imprima por pantalla lo siguiente.

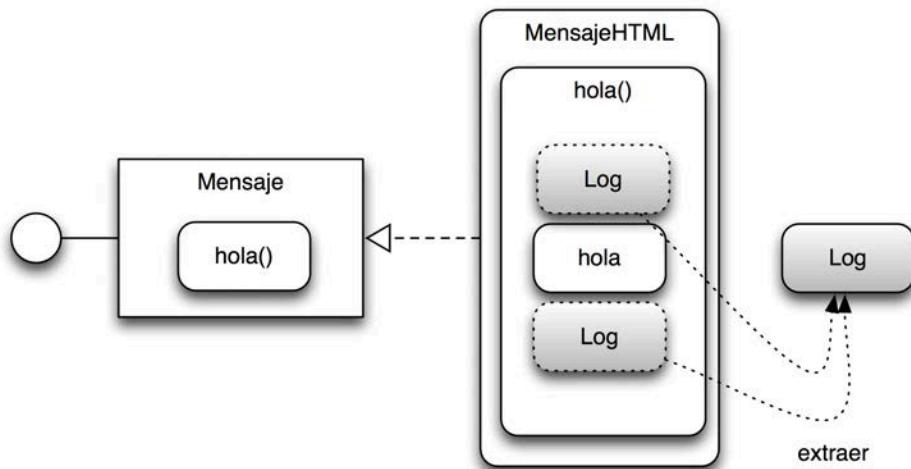


No nos quedaría más remedio que añadir una nueva funcionalidad al método mensaje (ver imagen).

Código 19.5: (Mensaje.java)

```
public void mensaje(String ) {  
    System.out.println("funcionalidad adicional proxy");  
    System.out.println("<html>"+mensaje+"</html>");  
    System.out.println("funcionalidad adicional proxy");  
}
```

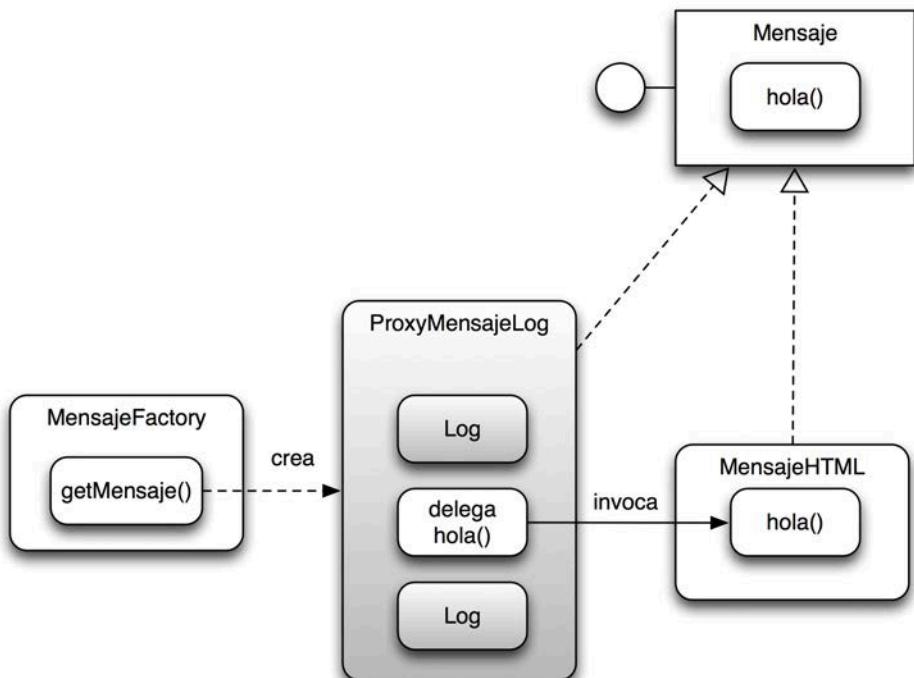
De esta manera la funcionalidad que añadimos queda ligada a nuestro código de forma totalmente estática y definitiva. A partir de ahora cuando invoquemos al método mensaje siempre se imprimirá la funcionalidad de log. Si queremos que la funcionalidad añadida sea dinámica y podamos ejecutarla en un momento determinado o no, hay que extraerla de la clase en la cuál la acabamos de ubicar (ver imagen).



Para extraer la responsabilidad de log crearemos una clase que se denominé ProxyMensajeLog. Esta clase hará de intermediaria entre la factoría y la clase mensaje, añadiendo la funcionalidad extra sin modificar la clase original. A continuación el código fuente de la nueva clase así como un diagrama que ayuda a clarificar.

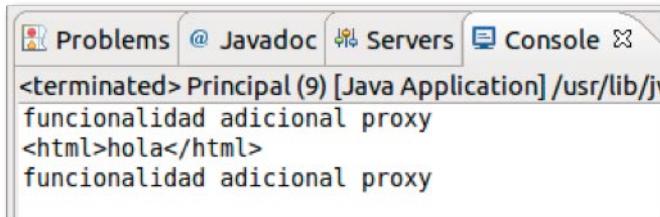
Código 19.6: (MensajeProxy.java)

```
public class MensajeProxy implements Mensaje {
    private Mensaje mensaje;
    public Mensaje getMensaje() {
        return mensaje;
    }
    public void hola() {
        System.out.println("funcionalidad adicional proxy");
        mensaje.hola();
        System.out.println("funcionalidad adicional proxy");
    }
    public MensajeProxy() {
        this.mensaje = new MensajeHTML();
    }
}
```



Arquitectura Java

La imagen muestra con claridad cómo la clase MensajeHTML no ha sido modificada. Podemos volver a ejecutar nuestro programa y veremos cómo hemos añadido nueva funcionalidad al concepto de mensaje (ver imagen).



Sin embargo la clase MensajeFactory si se habrá modificado para devolver un objeto de tipo MensajeProxy como se muestra a continuación.

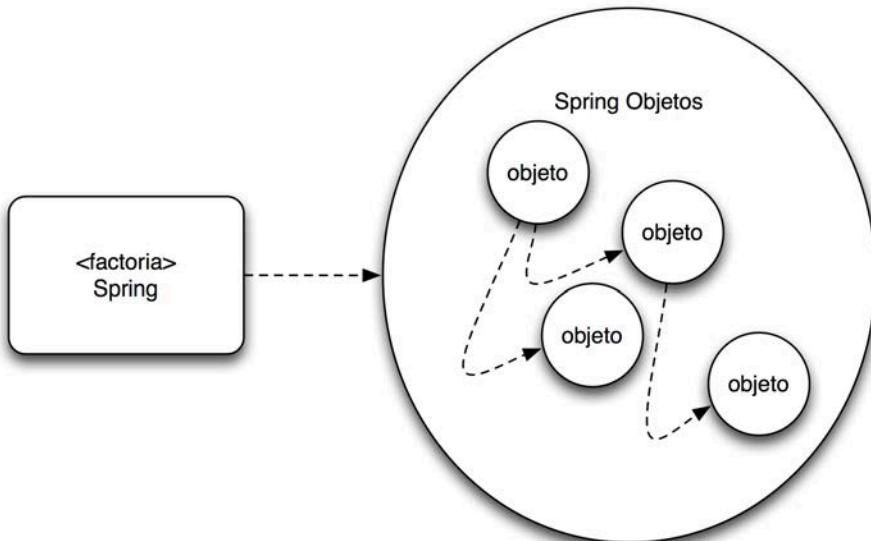
Código 19.5: (MensajeFactory.java)

```
public class MensajeFactory
    public static Mensaje getMensaje() {
        return new MensajeProxy();
    }
```

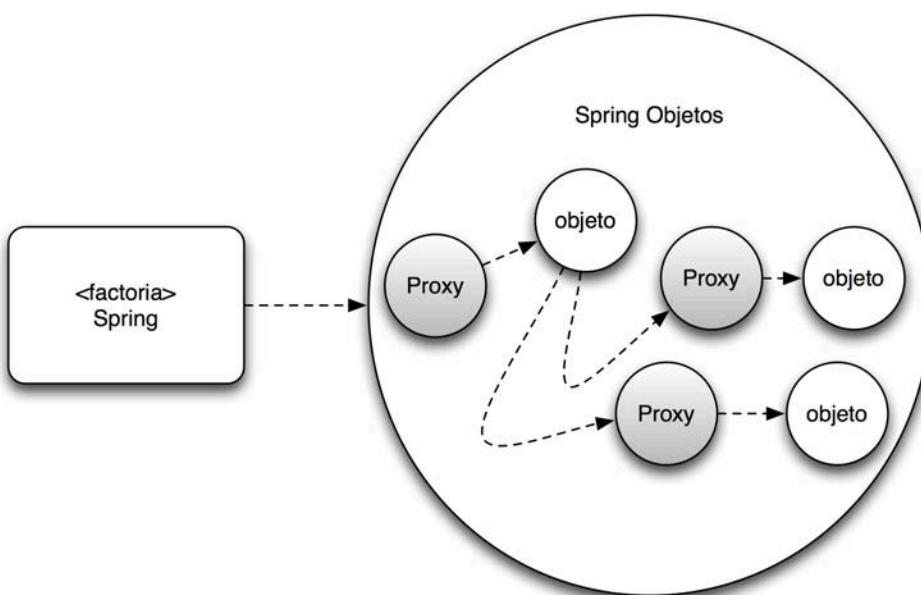
Para el cliente que usa la factoría ni siquiera es visible que existe un proxy intermedio realizando las operaciones adicionales ,esto es totalmente transparente para el programador ya que el código del programa principal no varía. Nosotros podríamos configurar la factoría para que use la clase proxy o la clase original según nuestras necesidades a través por ejemplo de un fichero de properties.

2. Usando Proxies con Spring

Ahora bien si volvemos a nuestra aplicación e intentamos realizar un símil, nos daremos cuenta de que toda la capa de servicios y DAO es creada a través del Framework Spring el cuál tiene la función de factoría (ver imagen).

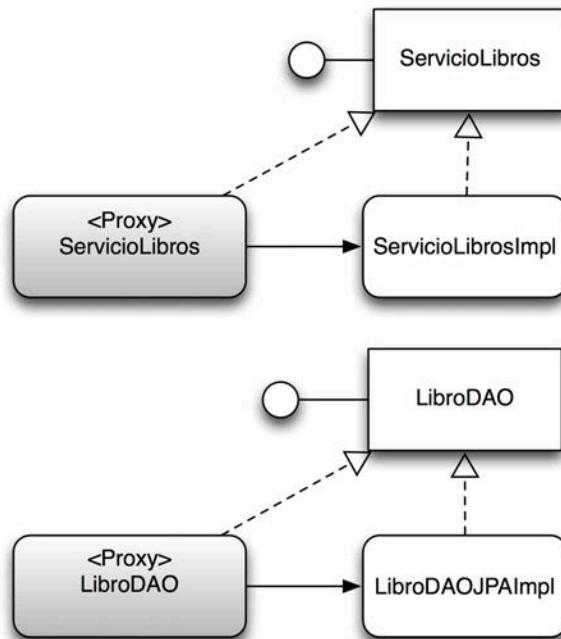


Por lo tanto al ser Spring una factoría, podría ser el encargado de construir un proxy para cada uno de los objetos creados de tal forma que sean estos proxies los encargados de aglutinar la responsabilidad relativa a las transacciones (funcionalidad adicional), como se muestra en la siguiente figura en la que todos los objetos disponen de un proxy a través del cual se accede a los mismos .

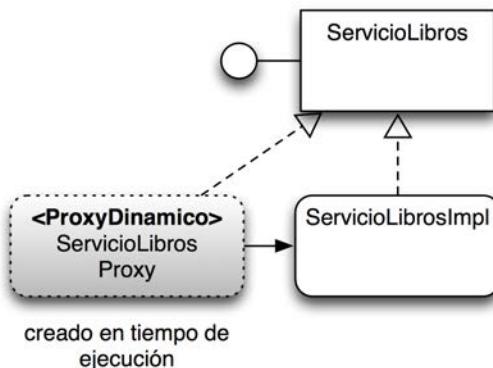


Ahora bien, aunque podamos usar el framework Spring para que cada uno de nuestros objetos disponga de un proxy con el cuál gestionar las transacciones, tenemos el

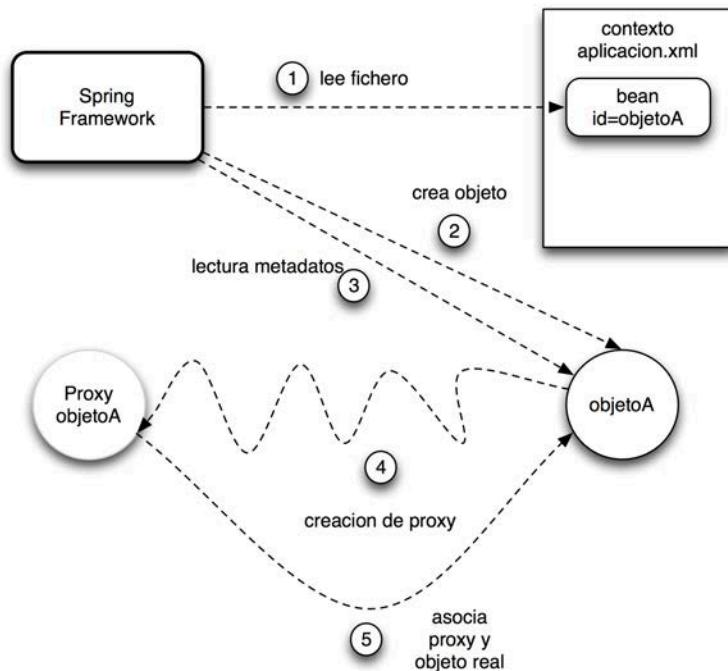
problema de crear una clase proxy por cada una de las clases que dé servicio y DAO que tenemos en estos momentos (ver imagen).



Esto parece una tarea titánica ya que en una aplicación podemos tener cientos de clases de servicio o DAO. Sin embargo Spring posee la capacidad de generar dinámicamente proxies basándose en las clases ya existentes sin tener que escribir una línea de código. Estos proxies son especiales y se les denomina **proxies dinámicos** ya que son creados en *tiempo* de ejecución. Véase una imagen del proxy:



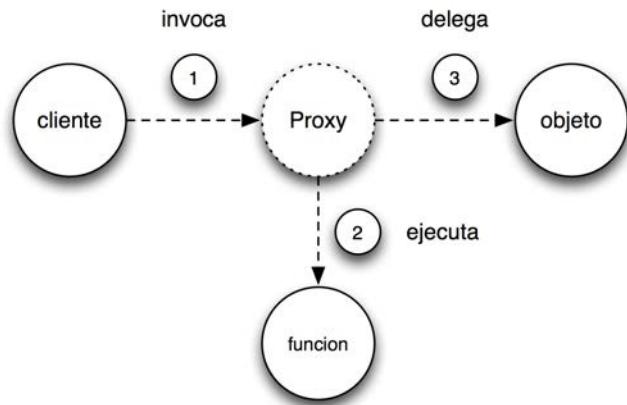
Estos tipos de proxies quedan asociados a los objetos reales .El proceso de construcción automática de un proxy es complejo (ver imagen).



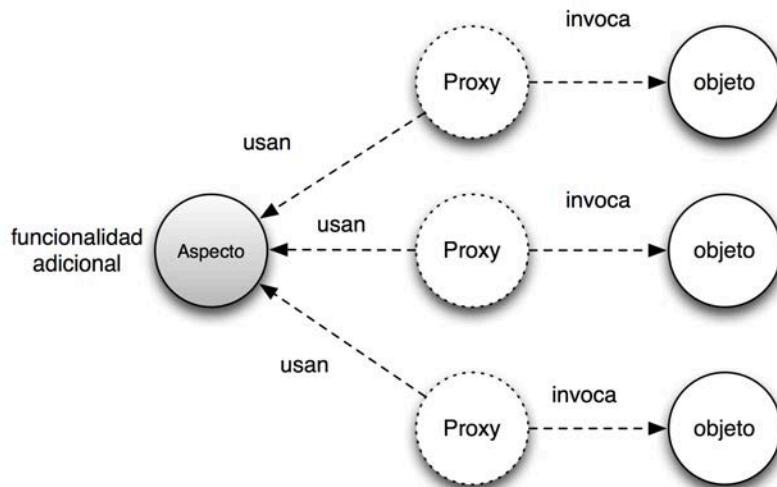
Vamos a explicarlo a continuación con mayor profundidad :

1. Spring lee el fichero de configuración con los distintos tipos de beans existentes
2. Spring instancia los objetos requeridos por el fichero
3. Spring lee los metadatos de un objeto determinado que acabamos de instanciar (clase, métodos, interfaces, etc) a través del API de reflection.
4. Spring crea un proxy por cada objeto instanciado que lo necesite
5. Spring asocia el proxy al objeto

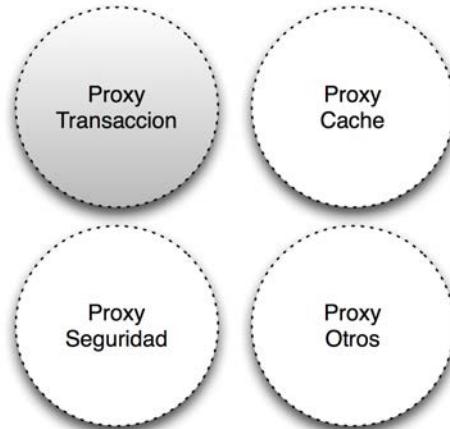
Por lo tanto con Spring podemos crear el conjunto de objetos que necesitemos y asignarles a todos un proxy de forma automática . Una vez hecho esto, cada vez que una aplicación cliente desee acceder a un objeto determinado, el proxy actuará de intermediario y añadirá la funcionalidad adicional que deseemos (ver imagen).



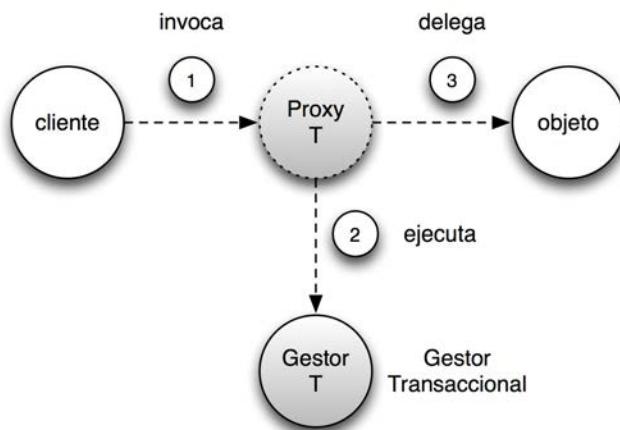
Esta funcionalidad adicional que añadimos a través del proxy es comúnmente conocida como **Aspecto** y es compartida habitualmente por varios proxies (ver imagen).



Una vez que tenemos claro que el framework Spring puede generar estos proxies, debemos decidir qué tipo de proxy queremos crear pues Spring soporta varios (ver imagen).



En nuestro caso vamos a definir proxies que nos ayuden a gestionar las transacciones entre las distintas invocaciones a los métodos de la capa de persistencia. Por ello elegiremos los proxies de transacción, que serán los encargados de ejecutar la funcionalidad relativa a las transacciones (ver imagen).



3. Configuración de proxies y transacciones.

Para poder trabajar con proxies que gestionen transacciones en nuestra aplicación deberemos añadir un gestor transaccional a la configuración del framework Spring

Arquitectura Java

como la figura anterior muestra .En este caso utilizaremos un gestor transaccional orientado a JPA, que se define a través de un bean de Spring.

Código 19.7: (contextoAplicacion.xml)

```
<bean id="transactionManager"
  class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

En segundo lugar, al fichero de configuración debemos añadir una nueva etiqueta que será la encargada de crear los distintos proxies transaccionales para todas nuestras clases .

Código 19.8: (contextoAplicacion.java)

```
<tx:annotation-driven />
```

Esta etiqueta, a nivel de fichero xml, permitirá al framework Spring dar de alta proxies para todas aquellas clases que dispongan de métodos marcados como transaccionales. Para marcar los métodos como transaccionales debemos añadir un nuevo conjunto de anotaciones a nuestra aplicación (ver código).

Código 19.9: (GenericDAOJPImpl.java)

```
@Transactional
```

Estas anotaciones son aportadas por el framework Spring y se encargan de definir la transaccionalidad para cada uno de los métodos de nuestras clases. A continuación se muestra una lista con los distintos tipos de transacciones soportadas por Spring:

1. **Required** :Se requiere de una transacción, si existe una transacción en curso, el método se ejecutara dentro de ella si no, el método se encargará de iniciar una nueva.
2. **Required_New** :Requiere ejecutarse de forma transaccional , si existe una transacción creada , suspenderá la que esta en curso. En el caso de no existir una transacción, creará una.
3. **Supports** : Si existe una transacción, se ejecutará dentro de ella. Si no existe, no iniciará ninguna
4. **Not_Supported** : Se ejecutará fuera de una transacción. Si esta transacción existe, será suspendida.

5. **Mandatory:** Debe ejecutarse obligatoriamente de forma transaccional. Si no lo hace, lanzará una excepción.
6. **Never :**Debe ejecutarse siempre de forma no transaccional. Si existe una transacción, lanzará una excepción.
7. **Nested :**Creará una nueva transacción dentro de la transacción actual.

Una vez declarados los distintos atributos se usarán así a nivel de clase:

Código 19.10: (LibroJPADAOImpl.java)

```
@Transactional(propagation=Propagation.Required)
public void insertarLibro() {
// codigo
}
```

De esta manera el método se ejecutará de forma transaccional. El valor Propagation.Required se aplica por defecto en el caso de que no se haya especificado nada. Por lo tanto, el método también podría haberse anotado de la siguiente forma:

Código 19.10: (LibroJPDADAOImpl.java)

```
@Transactional
public void insertarLibro() {
// codigo
}
```

Así estaremos señalando que el método se ejecuta dentro de una transaccion.Vamos a ver como afectan estas anotaciones a nuestras clases de la capa DAO y de la capa de Servicios.

Arquitectura Java

Código 19.11: (GenericDAOJPImpl.java)

```
public abstract class GenericDAOJPImpl<T, Id extends Serializable> extends JpaDaoSupport implements GenericDAO<T, Id> {
    private Class<T> claseDePersistencia;

    @SuppressWarnings("unchecked")
    public GenericDAOJPImpl() {
        this.claseDePersistencia = (Class<T>) ((ParameterizedType) getClass()
                .getGenericSuperclass()).getActualTypeArguments()[0];
    }

    @Override
    public T buscarPorClave(Id id) {
        return getJpaTemplate().find(claseDePersistencia, id);
    }

    @SuppressWarnings("unchecked")
    @Transactional(readOnly=true)
    public List<T> buscarTodos() {

        return getJpaTemplate().find("select o from "
                + claseDePersistencia.getSimpleName() + " o");
    }

    @Transactional
    public void borrar(T objeto) {
        getJpaTemplate().remove(getJpaTemplate().merge(objeto));
    }

    @Transactional
    public void salvar(T objeto) {

        getJpaTemplate().merge(objeto);
    }

    @Transactional
    public void insertar(T objeto) {
        getJpaTemplate().persist(objeto);
    }
}
```

Código 19.12: (ServicioLibrosImpl.java)

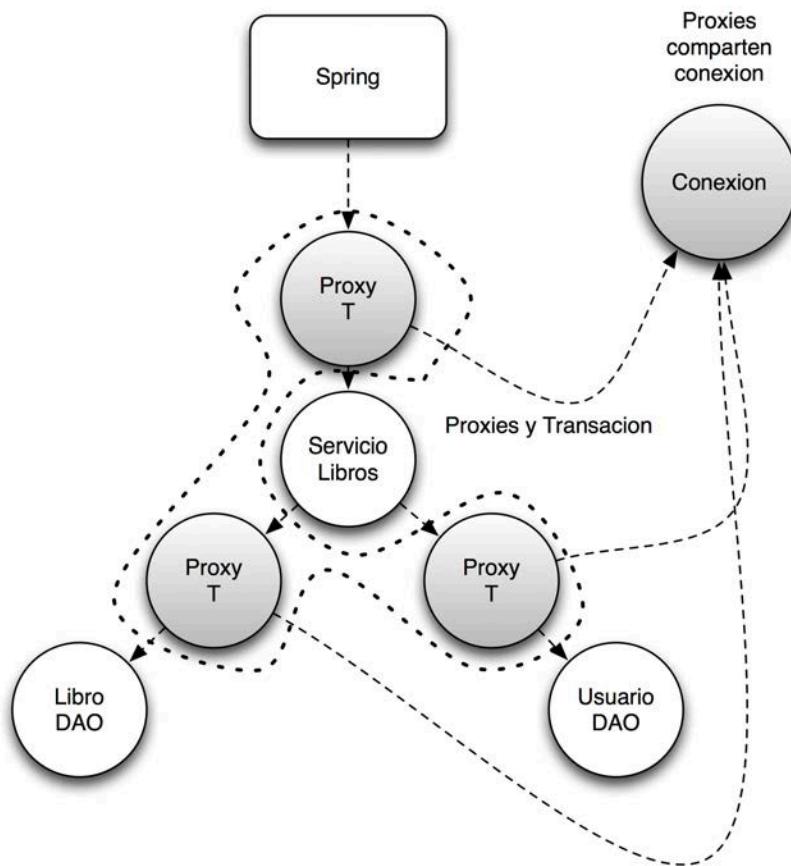
```
package com.arquitecturajava.aplicacion.servicios.impl;

//omitimos imports
public class ServicioLibrosImpl implements ServicioLibros {

    private LibroDAO libroDAO=null;
    private CategoriaDAO categoriaDAO=null;
    //omitimos set/get
    @Transactional
    public void salvarLibro(Libro libro) {
        libroDAO.salvar(libro);
    }
    @Transactional
    public void borrarLibro(Libro libro) {
        libroDAO.borrar(libro);
    }
    @Transactional
    public List<Libro> buscarTodosLosLibros() {
        return libroDAO.buscarTodos();
    }
    @Transactional
    public void insertarLibro(Libro libro) {
        libroDAO.insertar(libro);
    }
    @Transactional
    public List<Libro> buscarLibrosPorCategoria(Categoría categoria) {
        return libroDAO.buscarPorCategoria(categoria);
    }
    @Transactional
    public Libro buscarLibroPorISBN(String isbn) {
        return libroDAO.buscarPorClave(isbn);
    }
    @Transactional
    public Categoría buscarCategoria(int id) {
        // TODO Auto-generated method stub
        return categoriaDAO.buscarPorClave(id);
    }
    @Transactional
    public List<Categoría> buscarTodasLasCategorías() {
        // TODO Auto-generated method stub
        return categoriaDAO.buscarTodos();
    }
}
```

Arquitectura Java

Una vez dispuestas las anotaciones que marcan los métodos que soportan transacciones, es momento para mostrar como Spring se encarga de ejecutar una transacción atómica entre varios métodos a través del uso de un conjunto de proxies que comparten el mismo objeto conexión. A continuación se muestra un diagrama aclaratorio:



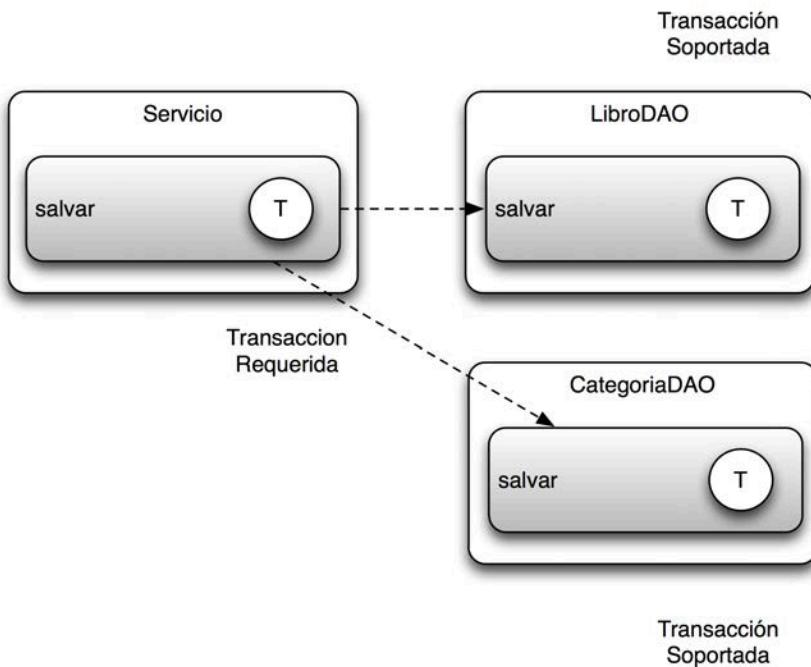
Vamos a ver cuál es la estructura final del fichero de configuración de Spring con las nuevas etiquetas.

Código 19.13: (contextoAplicacion.xml.java)

```

<beans>
    <tx:annotation-driven />
    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <propertynames="driverClassName" value="com.mysql.jdbc.Driver" />
        <propertynames="url"
        value="jdbc:mysql://localhost/arquitecturaJavaORM" />
        <propertynames="username" value="root" />
        <propertynames="password" value="java" />
    </bean>
    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <propertynames="persistenceUnitName" value="arquitecturaJava" />
        <propertynames="dataSource" ref="dataSource" />
        <propertynames="jpaVendorAdapter">
            <bean
                class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <propertynames="databasePlatform"
                value="org.hibernate.dialect.MySQL5Dialect" />
                <propertynames="showSql" value="true" />
            </bean>
        </property>
    </bean>
    <bean id="servicioLibros"
        class="com.arquitecturajava.aplicacion.servicios.impl.ServicioLibrosImpl">
        <propertynames="libroDAO" ref="libroDAO"></property>
        <propertynames="categoriaDAO" ref="categoriaDAO"></property>
    </bean>
    <bean id="libroDAO"
        class="com.arquitecturajava.aplicacion.dao.jpa.LibroDAOJPAImpl">
        <propertynames="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
    <bean id="categoriaDAO"
        class="com.arquitecturajava.aplicacion.dao.jpa.CategoriaDAOJPAImpl">
        <propertynames="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
</beans>
```

Realizada esta operación, serán los proxies los que controlen cómo se ejecuta una transacción y a cuántos métodos afecta de forma simultánea (ver imagen).



Resumen

En este capítulo hemos añadido una gestión transaccional transparente ,distribuida entre los distintos métodos de la capa de servicio y cada DAO, usando programación orientada a objeto y un gestor transacional. Así simplificamos sobremanera la gestión transaccional que habría sido necesario desarrollar en una programación clásica.

20.Uso de anotaciones y COC

Hemos simplificado la capa de persistencia y la capa de servicios a través del uso de Spring tanto como framework de Inversión de Control e Inyección de dependencia como usando sus capacidades de programación orientada a aspecto a la hora de diseñar la gestión de transacciones. Sin embargo, aunque primeramente parece que no hemos pagado ningún precio por ello, la realidad es distinta. Según pasan los capítulos el fichero de configuración (contextoAplicacion.xml) que tenemos es cada vez más grande y con más elementos. Por lo tanto cada vez será más difícil de configurar y de entender . Volvemos a la situación de los capítulos de Hibernate donde podíamos acabar con un número muy elevado de ficheros xml de mapeo. Es momento de hacer uso del principio de convención sobre configuración (COC) y, a través del uso de anotaciones, simplificar el fichero xml con el que trabajamos. Ese será el objetivo del presente capítulo.

Objetivos:

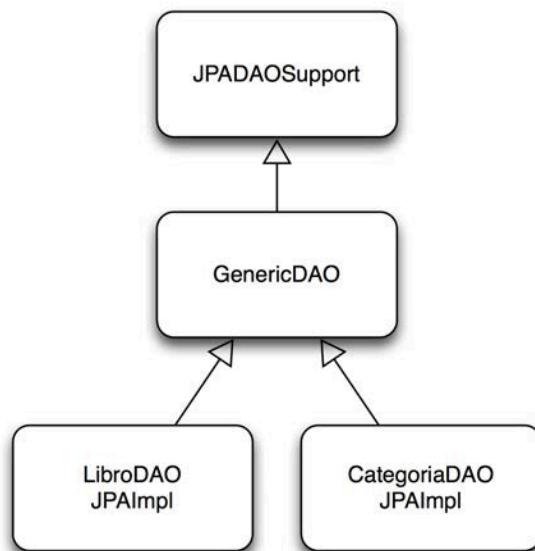
- Simplificar el fichero de configuración de Spring apoyándonos en anotaciones y en el principio COC.

Tareas:

1. @PersistenceContext y Entity manager
2. @Repository y manejo de excepciones
3. @Service y capas de Servicio
4. @AutoWired e inyección de dependencia

1. @PersistenceContext y EntityManager

En estos momentos tenemos diseñada nuestra capa DAO a través de JPA y utilizando la clase JPADAOsupport como apoyo a la hora de crear cualquiera de nuestras clases DAO (ver código)



Al depender todas nuestras clases de la clase JPADAOsupport, todas soportarán la asignación de un entityManagerFactory a través del fichero contextoAplicacion.xml de Spring (ver figura).

Arquitectura Java

Código 20.1: (contextoAplicacion.xml)

```
<bean id="libroDAO"
      class="com.arquitecturajava.aplicacion.dao.jpa.LibroDAOJPAImpl">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
<bean id="categoriaDAO"
      class="com.arquitecturajava.aplicacion.dao.jpa.CategoríaDAOJPAImpl">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

Como anteriormente hemos comentado, es momento de intentar poco a poco reducir este fichero de configuración y eliminar etiquetas .A tal fin, haremos uso de una anotación soportada por el estándar de JPA: la anotación @PersistenceContext .Esta anotación nos inyectará de forma directa un entityManager a cada una de nuestras clases DAO de forma transparente, sin tener que usar etiquetas <property> a nivel del fichero contextoAplicacion.xml . Para ello debemos añadir esta nueva línea:

Código 20.2: (contextoAplicacion.xml)

```
<context:annotation-config />
```

Esta línea se encarga de inyectar de forma automática el entity manager a todas las clases que usen la anotación @PersistenceContext. Por lo tanto, a partir de este momento, ninguna de las clases de la capa de persistencia tendrá asignada propiedades de entity manager (ver figura).

Código 20.3: (contextoAplicacion.xml)

```
<bean id="libroDAO"
      class="com.arquitecturajava.aplicacion.dao.jpa.LibroDAOJPAImpl">
</bean>
<bean id="categoriaDAO"
      class="com.arquitecturajava.aplicacion.dao.jpa.CategoríaDAOJPAImpl">
</bean>
```

Seguidamente se muestra la modificación que hemos tenido que realizar a nivel de la clase GenericDAOJPAImpl.

Código 20.4: (GenericDAOJPImpl.java)

```

public abstract class GenericDAOJPImpl<T, Id extends Serializable> implements
GenericDAO<T, Id> {
    private Class<T> claseDePersistencia;
    @PersistenceContext
    private EntityManager manager;

    public EntityManager getManager() {
        return manager;
    }
    public void setManager(EntityManager manager) {
        this.manager = manager;
    }
    @SuppressWarnings("unchecked")
    public GenericDAOJPImpl() {

        this.claseDePersistencia = (Class<T>) ((ParameterizedType) getClass()
                .getGenericSuperclass()).getActualTypeArguments()[0];
    }
    public T buscarPorClave(Id id) {
        return getManager().find(claseDePersistencia, id);
    }

    @Transactional(readOnly=true)
    public List<T> buscarTodos() {
        List<T> listaDeObjetos = null;
        TypedQuery<T> consulta = manager.createQuery("select o from "
                + claseDePersistencia.getSimpleName() + " o",
                claseDePersistencia);
        listaDeObjetos = consulta.getResultList();
        return listaDeObjetos;
    }
    @Transactional
    public void borrar(T objeto) {
        getManager().remove(getManager().merge(objeto));
    }
    @Transactional
    public void salvar(T objeto) {
        getManager().merge(objeto);
    }
    @Transactional
    public void insertar(T objeto) {

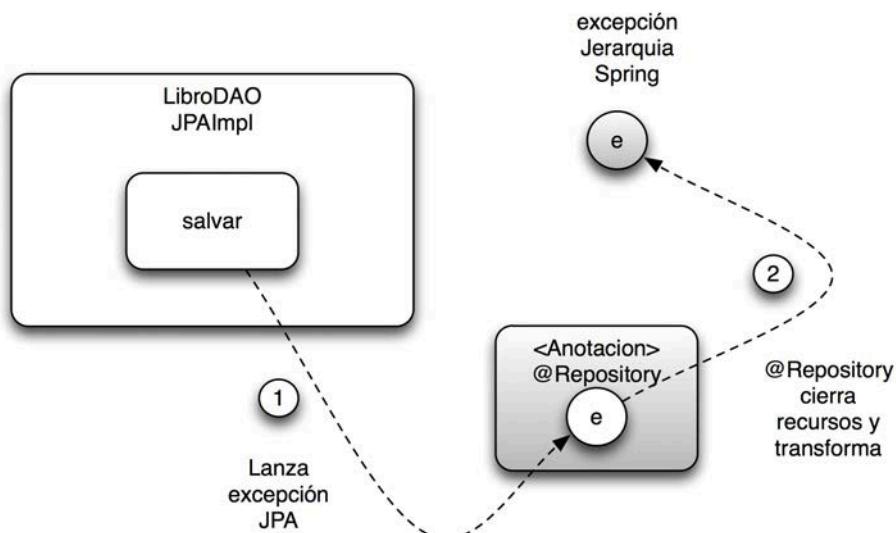
        getManager().persist(objeto);
    }
}

```

Como podemos ver hemos usado directamente la clase manager para invocar los métodos de persistencia y no hemos usado para nada el método getTemplate() de la clase JPDAOsupport. Esta clase se encargaba de obtener el manager de Spring y de gestionar las excepciones. Ya no heredamos de ella puesto que no la necesitamos para injectar el objeto EntityManager. Sin embargo si tenemos que gestionar excepciones, una vez que hemos dejado de heredar de JPDAOsupport, deberemos añadir algo más para que todo funcione como antes. La siguiente tarea se encargara de esto.

2. @Repository y manejo de excepciones

Spring puede trabajar con varios frameworks de persistencia desde JDBC a Hibernate pasando por JPA o iBatis y para cada una de estas soluciones genera un conjunto de excepciones distinto. Por lo tanto, para que nuestra aplicación se comporte de la misma forma sin importar cuál es el framework de persistencia usado o cuáles son las excepciones que se lanzan, deberemos marcar nuestras clases de capa DAO con la anotación @Repository, la cual se encargará de traducir las excepciones que se hayan construido, así como de cerrar recursos (ver figura).



Vamos a ver cómo nuestras clases hacen uso de la anotación . A continuación se muestra el código fuente de una de ellas. Todas las clases DAO incorporan a partir de ahora esta anotación.

Código 20.5: (contextoAplicacion.xml)

```
@Repository
public abstract class GenericDAOJPImpl<T, Id extends Serializable>implements
    GenericDAO<T, Id>{
    private Class<T> claseDePersistencia;
    @PersistenceContext
    private EntityManager manager;
    public EntityManager getManager() {
        return manager;
    }
}
```

Al marcar nuestras clases con la anotación `@Repository`, conseguimos que Spring gestione el manejo de excepciones de una forma más integrada, encargándose de cerrar los recursos de conexiones abiertas etc . Ahora bien, al marcar la clase con esta anotación, ya no será necesario registrar esta clase a nivel de contextoAplicacion.xml pues al anotarla queda registrada de forma automática. A continuación se muestra cómo queda nuestro fichero.

Código 20.6: (contextoAplicacion.xml)

```
<bean id="servicioLibros"
      class="com.arquitecturajava.aplicacion.servicios.impl.ServicioLibrosImpl">
</bean>
<!--eliminamos las clases dao-->
```

Eliminadas las clases DAO del fichero xml, pasaremos a revisar las clases de servicio para realizar la misma operación

3. Anotación `@Service`

Vamos a añadir una nueva anotación a nuestra clase de servicios denominada `@Service` que nos permite eliminar también estas clases del fichero de configuración xml donde las tenemos ubicadas . A continuación se muestra el código fuente de nuestra clase una vez añadida la anotación:

Código 20.7: (contextoAplicacion.xml)

```
@Service(value="servicioLibros")
public class ServicioLibrosImpl implements ServicioLibros {
// resto de código
}
```

Arquitectura Java

Una vez realizada esta operación, podremos eliminar las clases de servicio del fichero xml y simplificarlo (ver código).

Código 20.8: (contextoAplicacion.xml)

```
<bean id="transactionManager"
  class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
<!--eliminamos las clases de servicio-->
```

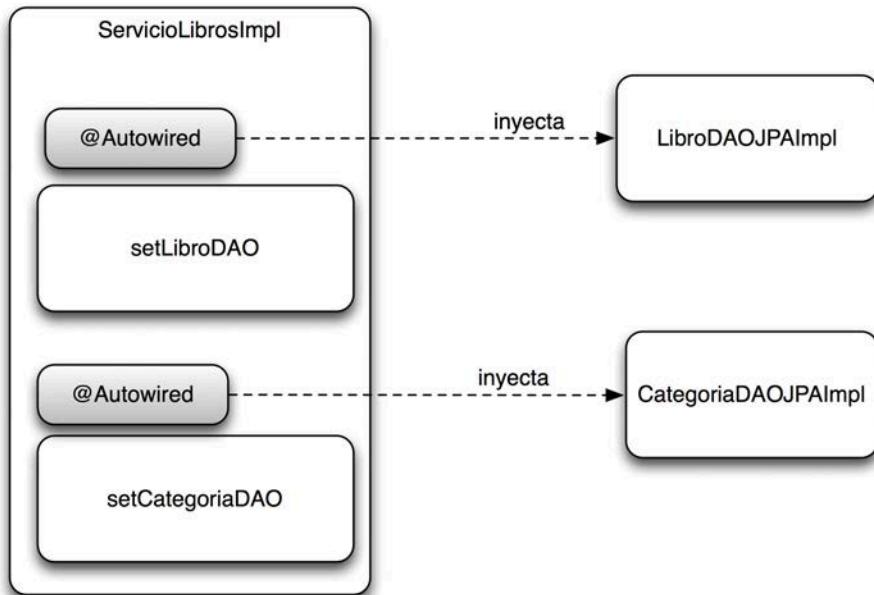
Hemos eliminado todas las clases de servicio y clases DAO de nuestro fichero xml ya que a través de las anotaciones se registran como componentes de forma automática. Para ello, únicamente deberemos añadir la siguiente línea de código a nuestro fichero:

```
<context:component-scan base-package="com.arquitecturajava.aplicacion.*" />
```

Esta línea se encargará de buscar en los distintos paquetes clases anotadas y registrarlas sin necesidad de que aparezcan en el fichero. Hecho esto, ya tenemos todas las clases registradas. Lamentablemente el fichero xml no sólo se encargaba de registrarlas sino también de relacionarlas a través de la etiqueta `<property>` y esta funcionalidad todavía no la hemos cubierto. La siguiente anotación nos ayudará a cumplir con esta tarea.

4. Anotaciones @AutoWired

La anotación `@AutoWired` está basada en el principio de convención sobre configuración para simplificar la inyección de dependencias entre las distintas clases. Se encarga de buscar las relaciones existentes entre los distintos objetos registrados e inyecta de forma automática las dependencias(ver imagen).



Una vez tenemos claro cómo funciona esta anotación, vamos a ver cómo queda el código de la clase de servicio.

Código 20.6: (ServicioLibrosImpl.xml)

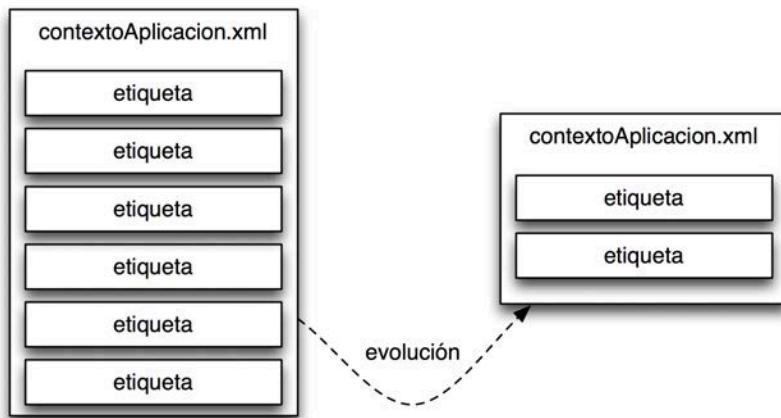
```

@Service(value="servicioLibros")
public class ServicioLibrosImpl implements ServicioLibros {
    private LibroDAO libroDAO=null;
    private CategoriaDAO categoriaDAO=null;
    @Autowired
    public void setLibroDAO(LibroDAO libroDAO) {
        this.libroDAO = libroDAO;
    }
    @Autowired
    public void setCategoríaDAO(CategoríaDAO categoriaDAO) {
        this.categoríaDAO = categoriaDAO;
    }
    //resto de código
}

```

De esta manera hemos pasado de una configuración basada exclusivamente en ficheros xml a una configuración en la que las anotaciones tienen mas peso y el fichero se reduce significativamente (ver imagen).

Arquitectura Java



Las anotaciones y el principio de convención sobre configuración nos han permitido eliminar información redundante a nivel de la configuración. Seguidamente se muestra cómo queda el fichero contextoAplicacion.xml , podemos ver cómo claramente ha sido simplificado sin tener que registrar ninguna clase.

```
<tx:annotation-driven />
    <context:annotation-config />
    <context:component-scan base-package="com.arquitecturajava.aplicacion.*" />
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url"
value="jdbc:mysql://localhost/arquitecturaJavaORM" />
        <property name="username" value="root" />
        <property name="password" value="java" />
</bean>
<bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" >
        <property name="persistenceUnitName" value="arquitecturaJava" />
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <property name="databasePlatform"
value="org.hibernate.dialect.MySQL5Dialect" />
                    <property name="showSql" value="true" />
                </bean>
            </property>
        </bean>
        <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
            <property name="entityManagerFactory" ref="entityManagerFactory" />
        </bean>
```

Resumen

En este capítulo hemos conseguido simplificar sobremanera el fichero de configuración de Spring apoyandonos en el conceptos de convención sobre configuración así como en la programación orientada a aspecto.

21.Java Server faces

En los últimos capítulos nos hemos centrado en el uso de Spring para mejorar y simplificar la construcción de la capa de persistencia y de servicios de nuestra aplicación. Es momento de volver a mirar hacia la capa de presentación que hemos construido apoyándonos en el standard de JSTL y hacerla evolucionar hacia estándares más actuales, en concreto hacia Java Server Faces o JSF . JSF es el framework standard a nivel de capa de presentación en arquitecturas JEE. Este capítulo servirá de introducción a JSF .Se crearán algunos ejemplos útiles a la hora de migrar la aplicación a esta tecnología en próximos capítulos

Objetivos:

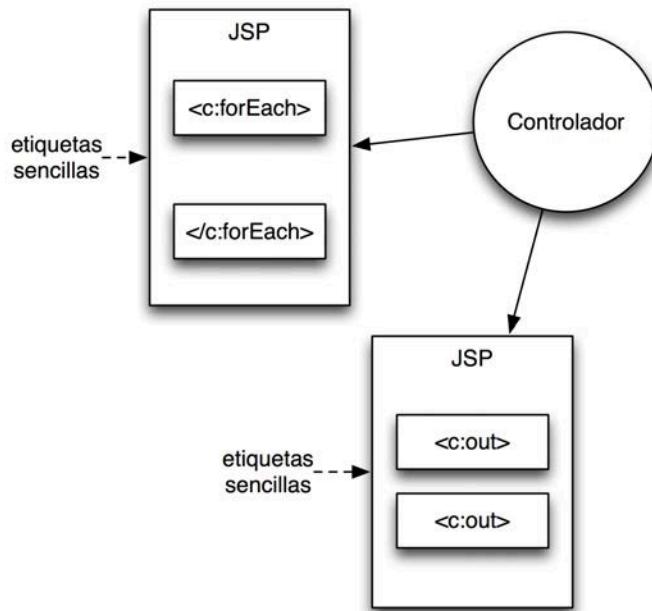
- Introducir JSF y desarrollar unos ejemplos básicos que nos sean útiles cuando migremos nuestra aplicación

Tareas:

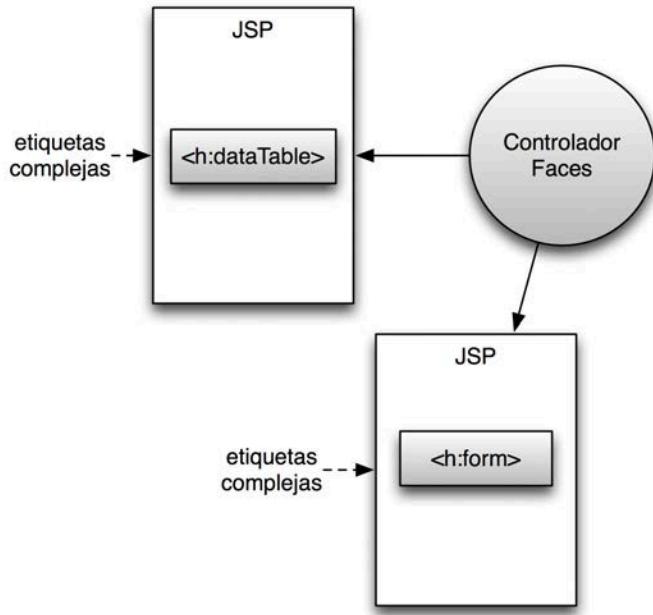
1. Introducción a JSF.
2. Instalar JSF 2.0.
3. HolaMundo con JSF
4. Construcción de un combo con JSF
5. Creación de tabla con JSF

1. Introducción a JSF

Hasta este momento hemos usado JSTL y un modelo MVC2 para gestionar la capa de presentación. Ahora bien, la capa de presentación construida hasta este momento se basa en el uso de un conjunto de etiquetas sencillas como son las etiquetas `<c:forEach>` y `<c:out>` que se muestran en la siguiente figura y que están ligadas al modelo MVC2 y su controlador.



Es momento de avanzar y modificar el conjunto de etiquetas que tenemos para poder usar etiquetas más complejas que aporten mayor funcionalidad. Son etiquetas aportadas por el framework de JSF y en lugar de definir bloques sencillos o sentencias if están orientadas a definir controles complejos , de tal forma que el desarrollador pueda utilizar estos controles para desarrollar aplicaciones de mayor complejidad más rápidamente . A continuación se muestra una imagen aclaratoria.



Como podemos ver, en este caso las etiquetas definen un formulario y una tabla que muestra una lista de datos así como un nuevo controlador. Es evidente que el nivel de abstracción se ha incrementado. Una vez explicado el concepto a nivel general, vamos a instalar el framework de JSF y crear unos ejemplos sencillos.

2. Instalación de JSF

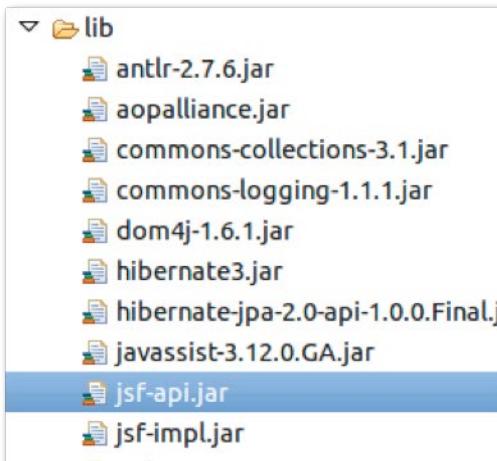
Para instalar el framework de Java Server Faces en nuestra aplicación debemos acceder a la siguiente url y obtener las librerías necesarias. En este caso las que correspondan a la versión 2.x.

- <http://javaserverfaces.java.net/>

Una vez obtenido el framework, es necesario extraer los siguientes ficheros jar del zip que acabamos de bajar

- jsf-api.jar.
- jsf-impl.jar.

Finalmente extraemos los ficheros al directorio lib de nuestra aplicación.



Realizada esta operación, vamos a pasar a configurar el propio framework .A diferencia de la situación actual en la que nosotros tenemos construido con controlador para nuestra aplicación ,el framework de JSF aporta un controlador pre construido que se denomina FacesServlet. Así , para poder trabajar con JSF hay que dar de alta un nuevo controlador a nivel de web.xml y eliminar el existente. A continuación se muestra cómo dar de alta el nuevo controlador en el web.xml.

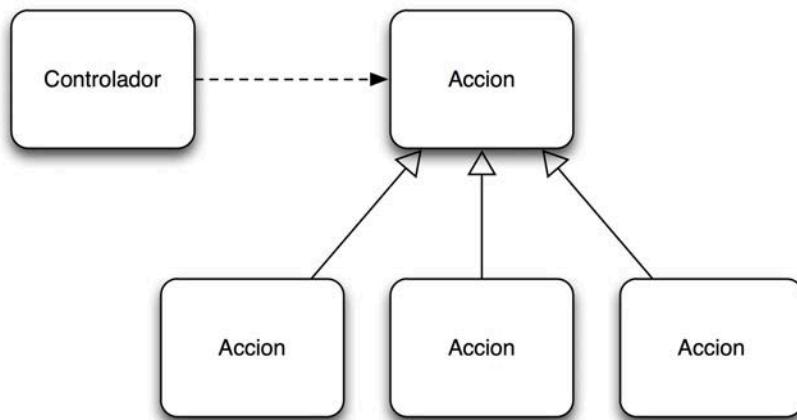
Código 21.1: (web.xml)

```
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

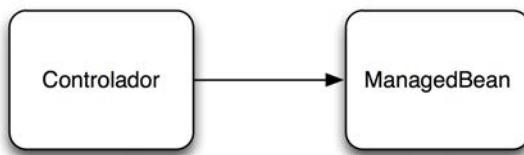
Antes de entrar a modificar nuestra aplicación, vamos a construir una serie de ejemplos sencillos que nos permitan conocer la tecnología y manejar este nuevo controlador.

3. Hola Mundo con JSF

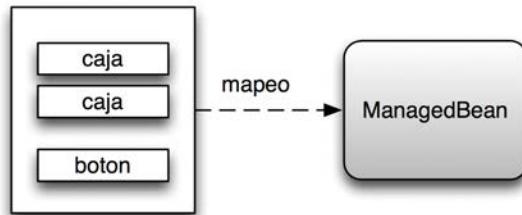
Hasta este momento nuestra capa de presentación se basaba fundamentalmente en un controlador y un conjunto de acciones asociados a éste (ver imagen).



Al usar el framework de JSF no vamos a tener la necesidad de apoyarnos en un conjunto de acciones como tal, sino que el framework se apoya en un nuevo concepto denominado **ManagedBean**. Por lo tanto a partir de ahora nuestro nuevo controlador FaceServlet se apoyará en un conjunto de ManagedBeans para realizar las operaciones (ver imagen).



Un managedBean es una clase que realiza tareas de intermediario entre la capa de presentación y la capa de servicios . Por lo tanto todas las páginas JSF se apoyan en ellas para enviar o recibir información (ver imagen).



Los ManagedBeans desempeñan el mismo rol que tenían hasta ahora los comandos o acciones pero a diferencia de éstos, son capaces de tratar con varios eventos y no con uno solo, como ocurre con las acciones construidas hasta ahora. Vamos a comenzar a tomar contacto con JSF como tecnología de capa de presentación. Para ello nos vamos a construir nuestra pagina de HolaMundo usando JSF. En este caso vamos a partir de un formulario elemental que nos solicita un nombre (ver imagen).



Una vez que tenemos claro cuál es el aspecto del formulario, vamos a ver el código fuente a nivel de JSF.

Código 21.2: (formularioNombre.xhtml)

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
<f:viewContentType="text/html"/>
<h:head>
<title>HolaMundo</title>
</h:head>
<h:body>
<h:form id="holaMundo">
<h:inputText id="nombreUsuario" value="#{usuarioBean.nombre}" />
<h:commandButton id="submit" action="resultado" value="Aceptar" />
</h:form>
</h:body>
</html>
  
```

Arquitectura Java

Podemos observar que las etiquetas de JSF tienen cierta similitud con las etiquetas JSTL y la diferencia más importante entre ellas es que las JSF están claramente orientadas a la definición de controles, como es el caso de la etiqueta <h:inputText> que define una caja de texto.

Código 21.3: (formularioNombre.xhtml)

```
<h:inputText id="nombreUsuario" value="#{usuarioBean.nombre}">
```

Una vez definida la caja de texto, define el siguiente atributo

Código 21.4: (formularioNombre.xhtml)

```
value="#{usuarioBean.nombre}"
```

Este atributo define que el valor que se escriba en la caja de texto está asociado a un ManagedBean , concretamente al campo nombre de este managedBean. Vamos pues a ver cómo se construye un ManagedBean que esté asociado a este formulario.

Código 21.5: (formularioNombre.xhtml)

```
package com.arquitectura.beans;
import javax.faces.bean.ManagedBean;

@ManagedBean
public class UsuarioBean {

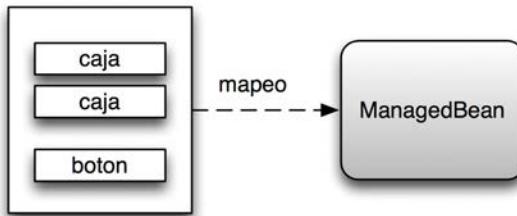
    private String nombre;
    Public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Visiblemente se trata de una sencilla clase java a la que se le asigna una nueva anotación @ManagedBean. Una vez asignada esta anotación al formulario, JSF y el managedBean se relacionan a través de la etiqueta.

Código 21.6: (formularioNombre.xhtml)

```
<h:inputText id="nombreUsuario" value="#{usuarioBean.nombre}" />
```

Como la siguiente imagen muestra.



Una vez que tenemos claro esta relación, vamos a explicar el otro control que existe a nivel de la página y que consiste en un botón.

Código 21.7: (formularioNombre.xhtml)

```
<h:commandButton id="submit" action="resultado" value="Aceptar" />
```

Este botón simplemente se encarga de invocar a otra pagina denominada resultado.xhtml haciendo uso del atributo action. Una vez pulsado el botón accederemos a la hora pagina que mostrará la siguiente estructura:

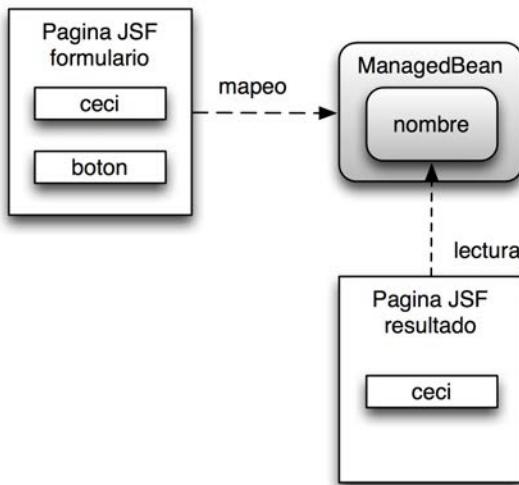


Visto el resultado, vamos a comprobar cuál es la estructura de la página resultado.xhtml.

Código 21.8: (formularioNombre.xhtml)

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
<f:viewContentType="text/html; charset=iso-8859-1"/>
<head><title>Respuesta</title></head>
<body>
<h:form id="formularioRespuesta">
<h2>Hola, #{usuarioBean.nombre}</h2>
<h:commandButton id="volver" value="volver" action="formularioNombre" />
</h:form>
</body>
</html>
```

De esta manera quedan ligadas las dos páginas al managedBean. La primera envía datos al managedBean, que los almacena. La segunda página lee datos del ManagedBean. La siguiente figura aclara los conceptos:



4. Construir un combo con JSF

Acabamos de ver el ejemplo más sencillo de JSF .Sin embargo en nuestra aplicación existen dos situaciones algo mas complejas. La primera es la carga del desplegable de categorías y la segunda la presentación de datos en una tabla . Vamos a cubrir cada una de estas situaciones usando JSF con ejemplos básicos para luego integrarlos en nuestra

aplicación. A tal fin comenzaremos usando JSF para cargar una lista de elementos en un formulario sencillo (ver imagen).



Una vez que tenemos claro que debemos construir vamos a verlo con detalle en el siguiente bloque de código.

Código 21.9: (formularioSeleccion.xhtml)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html">
<f:viewcontentType="text/html"/>
<h:head>
<title>HolaLista</title>
</h:head>
<h:body>
<h:form id="holaMundo">
    <h:selectOneMenu value="#{usuarioBeanLista.nombre}">
        <f:selectItems value="#{usuarioBeanLista.listaNombres}" />
    </h:selectOneMenu>
<h:commandButton id="submit" action="resultadoLista" value="Aceptar"/>
</h:form>
</h:body>
</html>
```

Es evidente que nuestro código fuente hace uso de un nuevo control como se muestra a continuación.

Arquitectura Java

Código 21.10: (formularioSeleccion.xhtml)

```
<h:selectOneMenu value="#{usuarioBeanLista.nombre}">
```

Este control es el encargado de definir un combo desplegable .Ahora bien, recordemos que estos controles realizan las dos siguientes funciones:

- Mostrar una lista de elementos
- Permitir seleccionar uno

Para mostrar la lista de elementos nuestro control hace uso de la siguiente etiqueta

Código 21.11: (formularioSeleccion.xhtml)

```
<f:selectItems value="#{usuarioBeanLista.listaNombres}" />
```

apoyándose en un managedBean denominado usuarioBeanLista, que carga una lista de elementos .Por otro lado, para elegir un elemento y almacenarlo, hace uso del mismo bean y de la propiedad nombre.

Código 21.12: (formularioSeleccion.xhtml)

```
<h:selectOneMenu value="#{usuarioBeanLista.nombre}">
```

Una vez claros cuáles son los elementos que ha de tener nuestro ManagedBean, vamos a mostrar su código fuente.

Código 20.12: (formularioSeleccion.xhtml)

```

package com.arquitectura.beans;
//omitimos import
@ManagedBean
public class UsuarioBeanLista {
    privateList<SelectItem>listaNombres;

    private String nombre;

    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public List<SelectItem>getListaNombres() {

        listaNombress= new ArrayList<SelectItem>();
        listaNombres.add(new SelectItem("1","nombre1"));
        listaNombres.add(new SelectItem("2","nombre2"));
        listaNombres.add(new SelectItem("2","nombre3"));
        returnlistaNombres;
    }
}

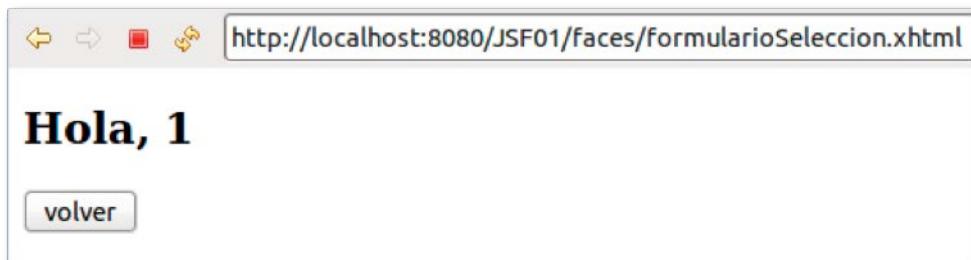
```

Construidos todos los elementos, podemos cargar la página y seleccionar un elemento de la lista. Despues, pulsamos el boton que dispone de un atributo action.

Código 20.13: (formularioSeleccion.xhtml)

```
<h:commandButton id="submit" action="resultadoLista" value="Aceptar"/>
```

Este atributo se encarga de redirigirnos a la página de destino denominada resultadoLista.xhtml que se muestra a continuación:



Arquitectura Java

Podemos ver cómo se muestra el valor del elemento que seleccionamos previamente. Seguidamente se muestra el código de la pagina de destino.

Código 20.14: (formularioSeleccion.xhtml)

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
<f:view contentType="text/html; charset=iso-8859-1"/>
<head><title>RespuestaLista</title></head>
<body>
<h:form id="formularioRespuesta">
<h2>Hola, #{usuarioBeanLista.nombre}</h2>
<h:commandButton id="volver" value="volver" action="formularioSeleccion" />
</h:form>
</body>
</html>
```

Hemos terminado de ver como construir un desplegable con JSF y cargarlo con datos, concretamente con elementos del tipo SelectItem. Es momento de avanzar un poco más y ver cómo construir una tabla : el último elemento que necesitamos.

5. Crear Tabla con JSF

Vamos a construir una tabla y cargarla de datos usando la tecnología JSF. Para ello vamos a apoyarnos en el concepto de usuario (nombre, edad) . La tabla que aparecerá será algo similar a lo siguiente:



Una vez que tenemos claro como ha de ser la tabla, vamos a ver el código fuente de la página TablaUsuarios.xhtml.

Código 20.15: (TablaUsuarios.xhtml)

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
<f:viewContentType="text/html">
<body>
<h:form>
<h:dataTable id="tabla" value="#{tablaUsuarioBean.usuarios}" var="usuario">
<h:column>
<h:outputText value="#{usuario.nombre}" />,
<h:outputText value="#{usuario.edad}" />
</h:column>
</h:dataTable>
</h:form>
</body>
</html>

```

Elaborada la pagina JSF, vamos a construir los dos elementos que necesitamos para cargar la tabla de datos: una clase usuario y un ManagedBean que nos devuelva una lista de ellos .Vamos a ver el código fuente de cada uno:

Código 20.16: (TablaUsuarios.xhtml)

```

package com.arquitectura.beans;
public class Usuario {
    private String nombre;
    private int edad;
    public Usuario(String nombre, int edad) {
        super();
        this.nombre = nombre;
        this.edad = edad;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
}

```

Arquitectura Java

Código 20.17: (TablaUsuarioBean)

```
//omitimos imports
@ManagedBean
public class TablaUsuarioBean {

    private List<Usuario> usuarios;

    public void setUsuarios(List<Usuario> usuarios) {
        this.usuarios=usuarios;
    }

    public List<Usuario> getUsuarios() {
        return usuarios;
    }

    @PostConstruct
    public void iniciar() {
        Usuario u= new Usuario("pedro",25);
        Usuario u1= new Usuario("crhistina",30);
        usuarios = new ArrayList<Usuario>();
        usuario.add(u);
        usuario.add(u1);

    }
}
```

Como podemos ver el código del método iniciar() se encarga de rellenar la lista de usuarios que necesitaremos en la tabla haciendo uso de la anotación @PostConstruct que define que método ejecutar una vez creado el ManagedBean. Visto cómo construir tres de los elementos básicos necesarios para nuestra aplicación, podremos en el siguiente capítulo abordar la migración de nuestra aplicación a JSF.

Resumen

En este capítulo no hemos avanzado en el diseño de la arquitectura de la aplicación pero hemos presentado una tecnología que pertenece a los estándares JEE y que usaremos en el próximo capítulo para simplificar la aplicación .

21. Migración a Java Server Faces

En el capítulo anterior hemos realizado una breve introducción a JSF ya que se trata de un framework de capa de presentación que tiene cierta complejidad .En este capítulo vamos a hacer migrar nuestra aplicación a JSF. Para ello necesitaremos eliminar algunas clases construidas anteriormente y agregar otras nuevas.

Objetivos:

- Usar JSF como framework de presentación en nuestra aplicación.

Tareas:

1. Añadir controlador de JSF y eliminar el existente con sus acciones.
2. Crear un ManagedBean
3. Crear la pagina MostrarLibros.xhtml
4. Borrar Libro
5. Insertar Libro
6. Editar Libro
7. Filtrar lista de Libros
8. Mejora Expression Language
9. Integración Spring

1. Añadir controlador de JSF y eliminar el existente

Hasta este momento hemos usado un controlador construido por nosotros mismos para gestionar todo el flujo de la capa de presentación. A partir de estos momentos no nos será necesario ya que como hemos visto en el capítulo anterior, JSF nos provee de un controlador propio. Así pues nuestro primer paso a la hora de migrar la aplicación a JSF es eliminar el código del controlador existente mostrado a continuación.

Código 22.1 (web.xml)

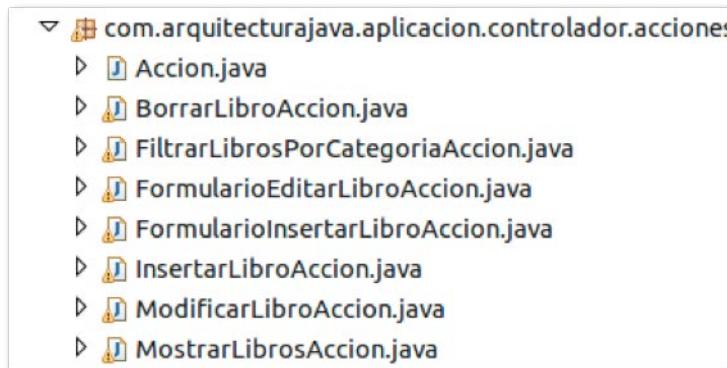
```
<servlet>
<description></description>
<display-name>ControladorLibros</display-name>
<servlet-name>ControladorLibros</servlet-name>
<servlet-class>com.arquitecturajava.aplicacion.controlador.ControladorLibros</servlet-
class>
</servlet>
<servlet-mapping>
<servlet-name>ControladorLibros</servlet-name>
<url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Eliminado este controlador, instalaremos como en el capítulo anterior las librerías de JSF en la carpeta lib y añadiremos el controlador de JSF a nuestro web.xml (ver imagen)

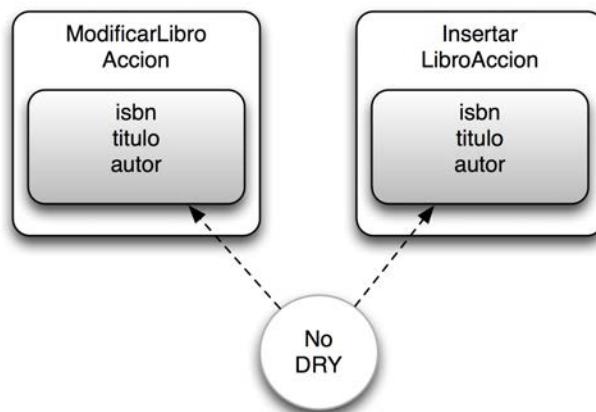
Código 22.2: (web.xml)

```
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

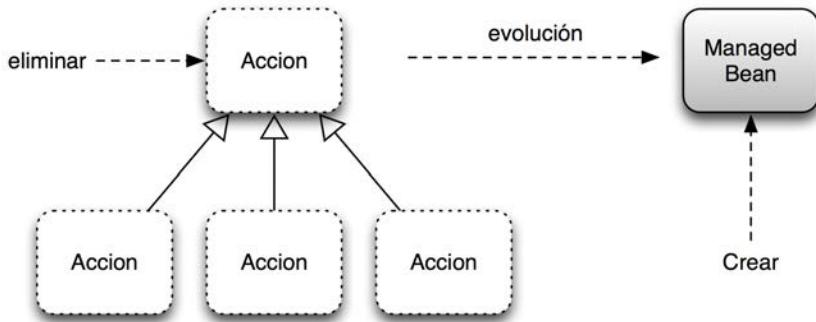
Una vez eliminado el controlador antiguo y creado uno nuevo, se presenta otro problema: a nivel de nuestro código fuente el controlador antiguo gestionaba un conjunto de acciones (ver imagen).



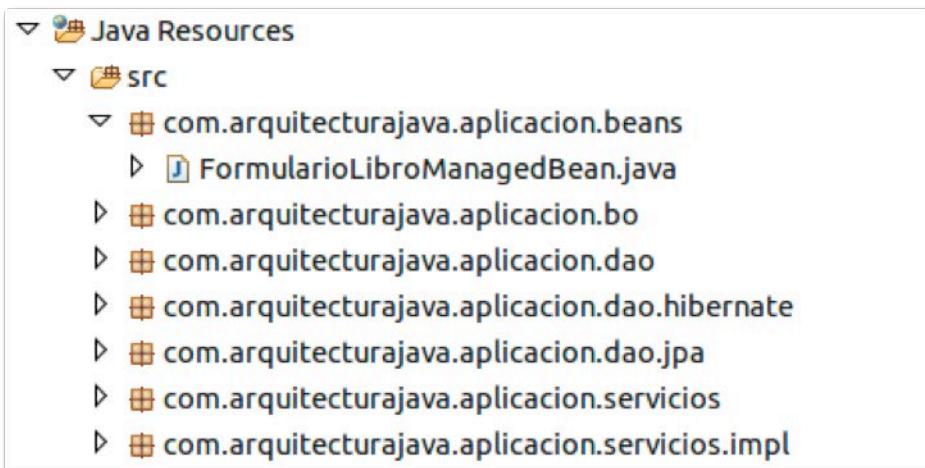
Es en estas acciones donde se encuentra ubicada la lógica que liga la capa de presentación con la capa de negocio y persistencia. Si revisamos estas acciones en profundidad, observaremos que, para las pocas que hemos diseñado, muchas comparten código, como es el caso de ModificarLibroAccion y InsertarLibroAccion, que gestionan las mismas variables y no cumplen con el principio DRY, ya que gestionan variables idénticas de la misma forma (ver imagen)



De la misma forma que hemos substituido un controlador por otro, es momento de substituir un conjunto de acciones por un ManagedBean (ver imagen)



Así pues una vez eliminado el controlador y las acciones de nuestra aplicación, la estructura de paquetes a nivel de capa de presentación será la siguiente.



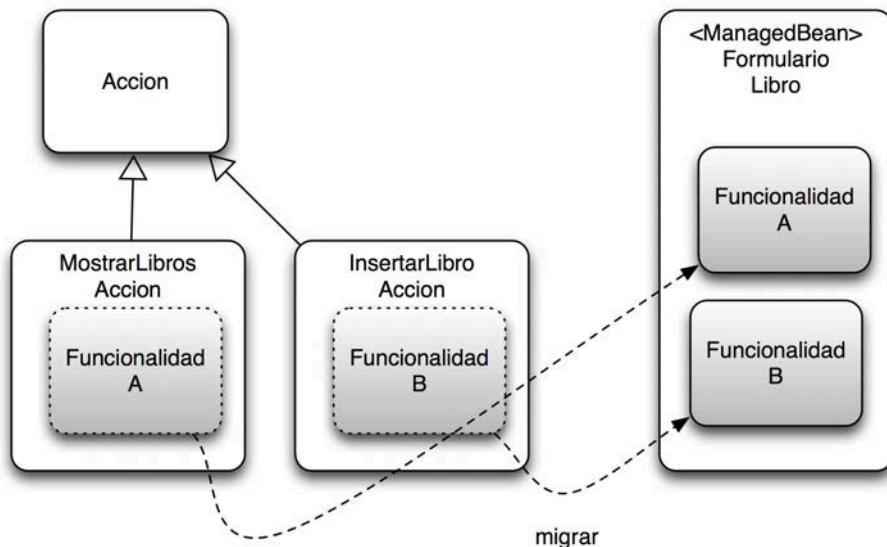
Es evidente que han sido eliminados los paquetes relacionados con el controlador y las acciones y únicamente se ha añadido un nuevo paquete denominado beans, que contiene un único Managedbean debido a reducido tamaño de nuestra aplicación.

Antes de crear nuestro ManagedBean debemos recordar que esta clase es la que hace las funciones de nexo con la capa de servicios y por lo tanto con toda la infraestructura que hemos diseñado con Spring framework. Así, el primer paso que debemos dar es ver cómo podemos configurar el framework de JSF para que pueda hacer uso de Spring. En principio no hace falta realizar ninguna operación especial y lo configurado anteriormente valdrá. Ahora bien, cuando construyamos el ManagedBean debemos

tener en cuenta que hacemos uso de Spring a la hora de ligar la capa de servicios con él de una forma orientada a JSF.

2. Creación ManagedBean

Como vimos en el capítulo anterior, construir un managedBean no es algo complicado. Simplemente deberemos usar la anotación @ManagedBean .No obstante, el ManagedBean que nosotros vamos a construir almacenará toda la funcionalidad ubicada en las distintas acciones en una sola clase (ver imagen).



A continuación se muestra el código fuente de esta clase, que tiene un ámbito de sesión que en nuestro caso es lo más cómodo debido al tamaño de la aplicación (pequeña), así estará accesible siempre. Aunque tenemos que tener claro que abusar de este tipo de ámbito puede producir problemas importantes de escalabilidad . En el manejo de JSF es clave trabajar de forma correcta con los distintos ámbitos.

Código 22.3: (FormularioLibroManagedBean.java)

```
//omitimos imports
@ManagedBean
@SessionScoped
public class FormularioLibroManagedBean {

    private String isbn;
    private String titulo;
    private String categoria;
    private List<SelectItem> listaDeCategorias;
    private List<Libro> listaDeLibros;

    public List<Libro> getListaDeLibros() {
        return listaDeLibros;
    }

    @PostConstruct
    public void iniciar() {
        listaDeLibros = getServicioLibros().buscarTodosLosLibros();

        List<Categoria> categorias = getServicioLibros()
            .buscarTodasLasCategorias();
        listaDeCategorias = new ArrayList<SelectItem>();
        for (Categoria categoria : categorias) {
            listaDeCategorias.add(new SelectItem(categoria.getId(),
                categoria.getDescripcion()));
        }
    }

    public void setListaDeLibros(List<Libro> listaDeLibros) {
        this.listaDeLibros = listaDeLibros;
    }

    public List<SelectItem> getListaDeCategorias() {
        return listaDeCategorias;
    }

    public void setListaDeCategorias(List<SelectItem> listaDeCategorias) {
        this.listaDeCategorias = listaDeCategorias;
    }

    public String getIsbn() {
        return isbn;
    }
}
```

```
}

public void setIsbn(String isbn) {
    this.isbn = isbn;
}

public String getTitulo() {
    return titulo;
}

public void setTitulo(String titulo) {
    this.titulo = titulo;
}

public String getCategoría() {
    return categoria;
}

public void setCategoría(String categoria) {
    this.categoría = categoria;
}

public void insertar(ActionEvent evento) {

    getServicioLibros().insertarLibro(
        new Libro(isbn, titulo, new Categoría(Integer
            .parseInt(categoría))));
    setListaDeLibros(getServicioLibros().buscarTodosLosLibros());
    categoria="0";
}

public void borrar(ActionEvent evento) {

    UIComponent componente = (UIComponent) evento.getComponent();
    String isbn = componente.getAttributes().get("isbn").toString();
    getServicioLibros().borrarLibro(new Libro(isbn));
    setListaDeLibros(getServicioLibros().buscarTodosLosLibros());
}

public void filtrar(ValueChangeEvent evento) {

    int idCategoría = Integer.parseInt(evento.getComponent()
        .getAttributes().get("value").toString());

    if(idCategoría!=0) {

        setListaDeLibros(getServicioLibros()).
```

```

        buscarLibrosPorCategoria(new Categoria(idCategoria)));
    }else {
        setListaDeLibros(getServicioLibros());
        buscarTodosLosLibros();
    }
}

public void editar(ActionEvent evento) {
    UIComponent componente = (UIComponent) evento.getComponent();
    Libro libro = getServicioLibros().buscarLibroPorISBN(
        componente.getAttributes().get("isbn").toString());
    isbn = libro.getIsbn();
    titulo = libro.getTitulo();
}
public void formularioInsertar(ActionEvent evento) {
    isbn = "";
    titulo = "";
}
public void salvar(ActionEvent evento) {
    getServicioLibros().salvarLibro(
        new Libro(isbn, titulo, new Categoria(Integer
            .parseInt(categoría))));
    setListaDeLibros(getServicioLibros().buscarTodosLosLibros());
    categoría="0";
}
public ServicioLibros getServicioLibros() {
    ApplicationContext contexto = FacesContextUtils
        .getWebApplicationContext(FacesContext.getCurrentInstance());
    return (ServicioLibros) contexto.getBean("servicioLibros");
}
}
}

```

Como podemos ver, el ManagedBean que hemos construido tiene bastante código mucho del cuál todavía no hemos explicado . Vamos a ir comentando los distintos bloques según los vayamos necesitando. En primer lugar vamos a hablar del método `getServicioLibros()` cuyo código mostramos a continuación.

Código 22.4: (FormularioLibroManagedBean.java)

```
public ServicioLibros getServicioLibros() {  
    ApplicationContext contexto = FacesContextUtils  
        .getWebApplicationContext(FacesContext.getCurrentInstance());  
    return (ServicioLibros) contexto.getBean("servicioLibros");  
}
```

Este método es el que se encarga de cargar todos los beans a nivel de framework Spring y ponerlos a disposición de JSF. Una vez configurado el framework Spring, podemos invocar a la clase ServicioLibros y ejecutar los distintos métodos, como por ejemplo `getLibros()` y `getCategorias()`. Esta operación la realizaremos a través del método iniciar, que se encargará de que la propiedad `listaDeLibros` y `listaDeCategorias` se rellenen de forma correcta . A continuación se muestra el código fuente de este método:

Código 22.5: (FormularioLibroManagedBean.java)

```
@PostConstruct  
public void iniciar() {  
    listaDeLibros = getServicioLibros().buscarTodosLosLibros();  
    List<Categoria> categorias = getServicioLibros()  
        .buscarTodasLasCategorias();  
    listaDeCategorias = new ArrayList<SelectItem>();  
    for (Categoria categoria : categorias) {  
        listaDeCategorias.add(new SelectItem(categoria.getId(),  
            categoria.getDescripcion()));  
    }  
}
```

Vamos a crear seguidamente la página MostrarLibros.xhtml y ver cómo se apoya en las variables que hemos asignado en el constructor.

3. Crear MostrarLibro.xhtml

En el capítulo anterior hemos visto cómo podemos construir una pagina JSF que sea capaz de mostrar una lista de personas. En cuanto a nuestra aplicación, se trata de un ejemplo similar en el cuál debemos mostrar una lista de libros . Para ello el código que construiremos será el siguiente, que se apoya completamente en JSF.

Código 22.6: (MostrarLibro.xhtml.)

```

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
<f:view>
    <body>
        <h:form>
            <h:selectOneMenu id="combo"
                valueChangeListener="#{formularioLibroManagedBean.filtrar}"
                value="#{formularioLibroManagedBean.categoría}" onchange="submit()">
                <f:selectItem itemValue="0" itemLabel="seleccionar" />
                <f:selectItems
                    value="#{formularioLibroManagedBean.listaDeCategorias}" />
            </h:selectOneMenu>
            <h:dataTable id="tabla"
                value="#{formularioLibroManagedBean.listaDeLibros}"
                var="libro">
                <h:column>
                    #{libro.isbn}
                </h:column>
                <h:column>
                    #{libro.titulo}
                </h:column>
                <h:column>
                    #{libro.categoría.descripcion}
                </h:column>
                <h:column>
                    <h:commandLink value="borrar"
                        actionListener="#{formularioLibroManagedBean.borrar}">
                        <f:attribute name="isbn" value="#{libro.isbn}" />
                    </h:commandLink>
                </h:column>
                <h:column>
                    <h:commandLink value="editar"
                        actionListener="#{formularioLibroManagedBean.editar}">
                        action="FormularioEditarLibro">
                            <f:attribute name="isbn" value="#{libro.isbn}" />
                        </h:commandLink>
                    </h:column>
                </h:columnTable>
                <h:commandButton
                    actionListener="#{formularioLibroManagedBean.formularioInsertar}"
                    action="FormularioInsertarLibro" value="insertar"/>
            </h:form>
        </body>
    </f:view>
</html>

```

Arquitectura Java

Una vez tenemos el código de la pagina MostrarLibros.xhtml vamos a comentar algunas de las líneas como la siguiente.

Código 22.6: (MostrarLibro.xhtml.)

```
<f:selectItems value="#{formularioLibroManagedBean.listaDeCategorias}" />
```

Esta linea de encarga de cargar el combo de categorias con la lista de categorías que nos devuelve nuestro managedBean. Algo similar hace la siguiente linea

Código 22.7: (MostrarLibro.xhtml)

```
<h:dataTable id="tabla"
    value="#{formularioLibroManagedBean.listaDeLibros}"
    var="libro">
```

Que se encarga de cargar la lista de libros a través del managedBean. Una vez que tenemos ambas colecciones cargadas JSF nos mostrara la siguiente página al solicitar MostrarLibros.xhtml.



Por ahora únicamente hemos comentado la funcionalidad que se encarga de cargar correctamente los datos en la pagina .Sin embargo no hemos cubierto todavía el botón de insertar ni los links de editar y borrar. Vamos a ir avanzando con estos elementos poco a poco.

4. Borrar Libro

Si revisamos el código de la pagina MostrarLibros.xhtml nos encontraremos con lo siguiente.

Código 22.7: (MostrarLibro.xhtml)

```
<h:commandLink value="borrar"
    actionListener="#{formularioLibroManagedBean.borrar}"
    <f:attribute name="isbn" value="#{libro.isbn}" />
</h:commandLink>
```

Este control de JSF es un control de tipo CommandLink o enlace se encargará de invocar a nuestro managedBean y ejecutara el método borrar. Una vez ejecutado, volverá a cargar la misma página. Vamos a mostrar a continuación el código de la función borrar de nuestro managedBean.

Código 22.8: (FormularioLibroManagedBean.xhtml)

```
public void borrar(ActionEvent evento) {
    UIComponent componente = (UIComponent) evento.getComponent();
    String isbn = componente.getAttributes().get("isbn").toString();
    getServicioLibros().borrarLibro(new Libro(isbn));
    setListaDeLibros(getServicioLibros().buscarTodosLosLibros());
}
```

Este método se encarga primero de obtener el componente que lo invocó, en este caso es un CommandLink ,componente que extiende de UIComponent. Una vez que tenemos el componente accedemos a su colección de atributos y obtenemos el isbn del Libro. Hecho esto disponemos del isbn e invocamos al método borrar de la clase ServicioLibros y borramos el registro de la base de datos .Por ultimo volvemos a cargar la nueva lista de libros.

Acabamos de ver como nuestra aplicación usa JSF para mostrar y eliminar registros es momento de seguir cubriendo funcionalidad y ver como implementar la lógica de inserción de libros a través de JSF.

5. Inserción Libro

Para realizar la tarea de inserción debemos apoyarnos en el código que hemos construido en la pagina MostrarLibros.xhtml. En esta página disponemos de un botón de inserción del cuál a continuación mostramos su código.

Código 22.9: (MostrarLibros.xhtml)

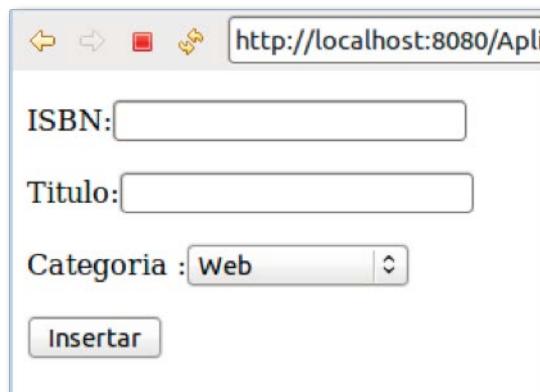
```
<h:commandButton actionListener="#{formularioLibroManagedBean.formularioInsertar}"  
action="FormularioInsertarLibro" value="insertar"/>
```

Como podemos ver, el funcionamiento del botón es muy sencillo: simplemente usa el atributo action para redirigirnos a otra página JSF, en concreto la página FormularioInsertarLibro.xhtml, ejecutando previamente el evento formularioInsertar. Este sencillo evento únicamente se encarga de vaciar las variables isbn y titulo como se muestra a continuación.

Código 22.10: (MostrarLibros.xhtml)

```
public void formularioInsertar(ActionEvent evento) {  
    isbn = "" ; titulo = "" ;  
}
```

Una vez realizada esta operación al pulsar el botón pasaremos a la página que muestra el formulario para insertar un nuevo libro. A continuación mostramos tanto la página como su código fuente.



Código 22.11: (FormularioInsertarLibro.xhtml)

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
<f:view contentType="text/html" />
<h:head>
    <title>FormularioInsertarLibro</title>
</h:head>
<h:body bgcolor="white">
    <h:form>
        <p>
            <label for="isbn">ISBN:</label>
<h:inputText id="isbn" value="#{formularioLibroManagedBean.isbn}" />
</p>
        <p>
            <label for="titulo">Titulo:</label>
<h:inputText id="titulo" value="#{formularioLibroManagedBean.titulo}" />
</p>
        <p><label for="categoria">Categoria :</label>
            <h:selectOneMenu id="categoria"
value="#{formularioLibroManagedBean.categoria}">
                <f:selectItems value="#{formularioLibroManagedBean.listaDeCategorias}" />
            </h:selectOneMenu>
            </p>
<h:commandButton id="submit"
actionListener="#{formularioLibroManagedBean.insertar}" action="MostrarLibros"
value="Insertar" />
</h:form>
</h:body>
</html>

```

Como podemos ver, la página de FormularioInsertarLibro comparte el mismo combo con la página de MostrarLibros.xhtml y por lo tanto invoca la misma función de nuestro managedBean. Una vez cargada esta página, vamos comentar cómo funciona la lógica del botón de insertar que se muestra a continuación.

Arquitectura Java

Código 22.12: (FormularioInsertarLibro.xhtml)

```
<h:commandButton  
id="submit"actionListener="#{formularioLibroManagedBean.insertar}"  
action="MostrarLibros"value="Insertar" />
```

Este botón invocará la función insertar de nuestro managedBean, insertando un nuevo registro en la tabla y volviendo a seleccionar la lista de libros. Hecho esto, el action “MostrarLibros” nos redirigirá otra vez a la lista. Vamos a ver el código del método insertar

Código 22.13: (FormularioInsertarLibro.xhtml)

```
public void insertar(ActionEvent evento) {  
    getServicioLibros().insertarLibro(  
        new Libro(isbn, titulo, new Categoria(Integer  
            .parseInt(categoría))));  
    setListaDeLibros(getServicioLibros().buscarTodosLosLibros());  
    categoría="0";}
```

Nuestro método se apoya en las capacidades que tiene JSF de mapeo de etiquetas a propiedades de los ManagedBeans y se encarga de insertar el registro en la base datos. Una vez hecho esto, vuelve a seleccionar la nueva lista de libros y nos redirige a la pagina MostrarLibros.xhtml, mostrándonos la lista con el nuevo libro insertado.

6. Funcionalidad de Edición

Vamos a ver en la página MostrarLibros.xhtml cuál es el código fuente que nos enlaza con la funcionalidad de edición.

Código 22.14: (FormularioInsertarLibro.xhtml)

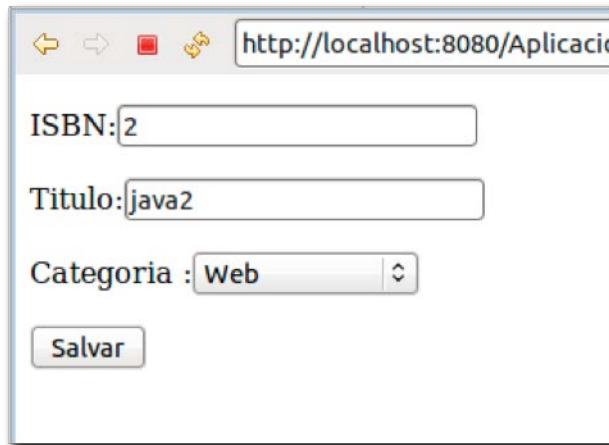
```
<h:column>  
    <h:commandLinkvalue="editar"  
        actionListener="#{formularioLibroManagedBean.editar}"  
        action="FormularioEditarLibro">  
        <f:attributename="isbn" value="#{libro.isbn}" />  
    </h:commandLink>  
</h:column>
```

Ya conocemos algunos atributos a nivel de commandLink, concretamente el atributo action que nos redirigirá a la página FormularioEditarLibro.xhtml . Ahora bien, antes de redirigirnos se ejecuta el evento actionListener que invoca la función de editar. Vamos a ver el código fuente de este método:

Código 22.15: (FormularioLibroManagedBean.java)

```
public void editar(ActionEvent evento) {
    UIComponent componente = (UIComponent) evento.getComponent();
    Libro libro = getServicioLibros().buscarLibroPorClave(
        componente.getAttributes().get("isbn").toString());
    isbn = libro.getIsbn();
    titulo = libro.getTitulo();
}
```

Es evidente que este método carga en el managedBean los datos relativos al Libro que vamos a editar, para ello busca por clave el libro apoyándose en el atributo isbn del commandLink. Una vez realizado esto, se muestra el formulario de edición con los datos cargados (ver imagen).



Vamos a ver el código fuente de este fichero

Arquitectura Java

Código 22.16: (FormularioEditarLibro.xhtml)

```
<!—omitimos doctype→
<f:view contentType="text/html" />
<h:head>
    <title>Hello World!</title>
</h:head>
<h:body bgcolor="white">
    <h:form >
        <p>
            <label for="isbn">ISBN:</label>
<h:inputText id="isbn" value="#{formularioLibroManagedBean.isbn}" />
</p>
        <p>
            <label for="titulo">Titulo:</label>
<h:inputText id="titulo" value="#{formularioLibroManagedBean.titulo}" />
</p>
        <p><label for="categoria">Categoria :</label>
            <h:selectOneMenu id="categoria"
                value="#{formularioLibroManagedBean.categoria}">
                <f:selectItems
                    value="#{formularioLibroManagedBean.listaDeCategorias}" />
            </h:selectOneMenu>
        </p>
        <h:commandButton id="submit"
            actionListener="#{formularioLibroManagedBean.salvar}"
            .action="MostrarLibros" value="Salvar" />
    </h:form>
</h:body>
</html>
```

Visto esto, únicamente nos queda por clarificar cómo se encarga esta página de guardar la información .Para ello, el commandButton de la página se encarga de invocar al método salvar.

Código 22.17: (FormularioInsertarLibro.xhtml)

```
<h:commandButton id="submit" actionListener="#{formularioLibroManagedBean.salvar}"
action="MostrarLibros" value="Salvar" />
```

A continuación se muestra el código fuente de este método:

Código 22.18: (FormularioLibroManagedBean.java)

```
public void salvar(ActionEvent evento) {
    getServicioLibros().salvarLibro(
        new Libro(isbn, titulo, new Categoria(Integer
            .parseInt(categoría))));
    setListaDeLibros(getServicioLibros().buscarTodosLosLibros());
    categoría="0";
}
```

Este método localiza la categoría del Libro y se la asigna a un nuevo libro que es el que salvamos en la base de datos. Por ultimo nos queda implementar de forma correcta la funcionalidad de Filtrar.

7. Filtrar por categorías

Vamos a modificar el combo que tenemos construido con JSF para que permita realizar operaciones de filtrado. Véase el nuevo código fuente de este combo:

Código 22.19: (MostrarLibros.xhtml)

```
<h:selectOneMenu id="combo"
valueChangeListener="#{formularioLibroManagedBean.filtrar}"
value="#{formularioLibroManagedBean.categoría}" onchange="submit()">
    <f:selectItem itemValue="0" itemLabel="seleccionar" />
    <f:selectItems
        value="#{formularioLibroManagedBean.listaDeCategorías}" />
</h:selectOneMenu>
```

Hemos visto que se han aplicado varios cambios .En primer lugar se ha añadido un nuevo item denominado Todas (ver código)

Código 22.20: (MostrarLibros.xhtml)

```
<f:selectItem itemValue="0" itemLabel="Todas"/>
```

Este elemento permite que en el combo se puedan seleccionar “todas” las categorías . Realizada esta operación, se ha añadido el atributo valueChangeListener que define qué función se ejecutará cuando el valor del combo cambie. En este caso la función se denomina filtrar y está a nivel de nuestro ManagedBean.

Por último, usaremos javascript para enviar los datos del formulario al servidor a través del método onchange(). A continuación se muestra una imagen del filtro en ejecución.



Una vez hechos estos cambios, tenemos completamente implementada la funcionalidad de JSF en nuestra aplicación .Es momento de afrontar algunas tareas más orientadas al refactoring.

8. Mejoras Expression Language 2.0

El primer refactoring que vamos a aplicar es el hacer uso de expresión language en nuestra pagina JSF 2.0 de MostrarLibros.xhtml. Para ello hay que registrar un listener especial en el fichero faces-config.xml que hasta este momento no hemos utilizado.Vamos a ver el código de éste:

Código 22.23: (MostrarLibros.xhtml)

```
<faces-config version="2.0" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
    <application>
        <el-resolver>
            org.springframework.web.jsf.el.SpringBeanFacesELResolver
        </el-resolver>
    </application>
</faces-config>
```

Como hemos visto en ejemplos anteriores hay bloques de código que son difíciles de entender, concretamente los relacionados con los links. Vamos a verlo:

Código 22.24: (MostrarLibros.xhtml)

```
<h:column>
    <h:commandLink value="borrar"
        actionListener="#{formularioLibroManagedBean.borrar}" >
        <f:attribute name="isbn" value="#{libro.isbn}" />
    </h:commandLink>
</h:column>
<h:column>
    <h:commandLink value="editar"
        actionListener="#{formularioLibroManagedBean.editar}"
        action="FormularioLibro">
        <f:attribute name="isbn" value="#{libro.isbn}" />
    </h:commandLink>
</h:column>
```

Podemos simplificar cada uno de estos elementos apoyándonos en expression language de tal forma que, a partir de ahora, estos enlaces quedarían definidos de una forma más sencilla ya que el expresión language permite a partir de la versión 2.0 invocar métodos que reciban parámetros. Véase el nuevo bloque de código:

Código 22.25: (MostrarLibros.xhtml)

```
<h:column>
    <h:commandLink value="editar"
        actionListener="#{formularioLibroManagedBean.editar(libro.isbn)}"
        action="FormularioLibro">
    </h:commandLink>
</h:column>
<h:column>
    <h:commandLink value="borrar"
        actionListener="#{formularioLibroManagedBean.borrar(libro.isbn)}"
        action="MostrarLibro">
    </h:commandLink>
</h:column>
```

Una vez modificado el código, realizamos una invocación directa desde la página MostrarLibros.xhtml a los métodos del ManagedBean. Véamos cómo quedan éstos:

Código 22.26: (FormularioLibrosManagedBean.java)

```
public void borrar(String isbn) {  
    getServicioLibros().borrarLibro(new Libro(isbn));  
    setListaDeLibros(getServicioLibros().buscarTodosLosLibros());  
}  
public void editar(String isbn) {  
    Libro libro = getServicioLibros().buscarLibroPorISBN(isbn);  
    isbn = libro.getIsbn();  
    titulo = libro.getTitulo();  
}
```

Es evidente que es mucho más sencillo trabajar de esta forma.

9. Integración de Spring

Quizá el único elemento que no acaba de encajar en la integración con el framework Spring y JSF es el siguiente bloque de código que tenemos a nivel de nuestro managedBean y que se encarga de enlazar JSF y Spring.

Código 22.27: (FormularioLibrosManagedBean.java)

```
public ServicioLibros getServicioLibros() {  
    ApplicationContext contexto = FacesContextUtils  
        .getWebApplicationContext(FacesContext.getCurrentInstance());  
    return (ServicioLibros) contexto.getBean("servicioLibros");  
}
```

Para simplificar este bloque de código, es necesario usar la anotación `@ManagedProperty` e injectar el servicio de forma directa como a continuación se muestra.

Código 22.28 (FormularioLibrosManagedBean.java)

```
@ManagedProperty("#{servicioLibros}")  
private ServicioLibros servicioLibros;
```

Éste es el último paso en cuando a las modificaciones de JSF se refiere y su integración con Spring Framework.

10. JSF 2 y Business Objects

Una de las pocas cosas que no termina de encajar en el ejemplo de JSF es este bloque de código.

Código 22.29 (FormularioLibrosManagedBean.java)

```
@PostConstruct
public void iniciar() {
    listaDeLibros = getServicioLibros().buscarTodosLosLibros();
    List<Categoria> categorias = getServicioLibros()
        .buscarTodasLasCategorias();
    listaDeCategorias = new ArrayList<SelectItem>();
    for (Categoria categoria : categorias) {
        listaDeCategorias.add(new SelectItem(categoria.getId(),
            categoria.getDescripcion()));
    }
}
```

Es evidente que sería mucho mas comodo cargar en las paginas JSF objetos de negocio que una lista de SelectItems. Esto es posible a partir de la version de JSF 2.0 que permite definir los elementos selectOneMenu de la siguiente forma.

Código 22.30 (MostrarLibros.xhtml)

```
<h:selectOneMenu value="#{formularioLibroManagedBean.categoría}">
<f:selectItem itemValue="0" itemLabel="seleccionar" />
<f:selectItems value="#{formularioLibroManagedBean.categorias}" var="categoria"
itemLabel="#{categoria.descripcion}" itemValue="#{categoria.id}" />
</h:selectOneMenu>
```

Resumen

En este capítulo nos hemos centrado en integrar JSF como tecnología en nuestra aplicación, eliminando la jerarquía de acciones previa y creando un único ManagedBean .Evidentemente una aplicación habitual tendrá varios ManagedBean pero siempre serán muchos menos que el grupo de acciones antes usado. Por otra parte, hemos abordado la integración con el framework Spring. Aún así han quedado muchas cosas fuera como el sistema de navegación o las anotaciones CDI que debido a la extensión del capítulo no serán abordadas.

23. Servicios Web y JAX-WS

En el capítulo anterior hemos hecho migrar nuestra aplicación a JSF, en estos momentos la aplicación hace uso de JSF, JPA y Spring utilizando un amplio conjunto de anotaciones. Es momento de cubrir algunos temas complementarios como es el uso de servicios web. Hoy en día la mayor parte de las aplicaciones web que construimos necesitan acceder a información remota de la que otra aplicación nos provee , o bien necesita publicar información para que otras aplicaciones accedan a ella de forma remota .Una de las formas más habituales de publicar este tipo de información es a través del uso de servicios web. En este capítulo nos centraremos en integrar el uso de esta tecnología en nuestra aplicación.

Objetivos:

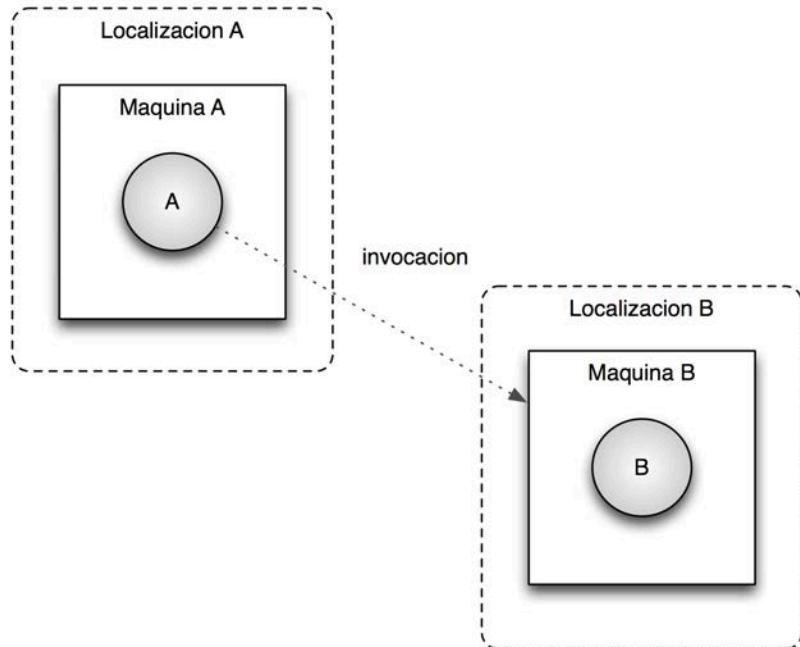
- Publicar a través de un servicio web información de nuestros libros para que aplicaciones de terceros puedan acceder a ella.

Tareas:

1. Introducción a la Programación Distribuida.
2. Introducción al concepto de Servicio Web.
3. Creación de un Servicio Web Básico usando anotaciones.
4. Publicación de Servicio Web
5. Instalación de Apache CXF.
6. Configuración de Apache CXF.
7. Acceso al servicio web.

1. Introducción a programación distribuida

La programación distribuida viene a aportar una solución a un problema clásico de la programación orientada a objeto: cómo comunicar objetos que se encuentran en ubicaciones físicas distintas . A continuación se muestra un diagrama del problema:



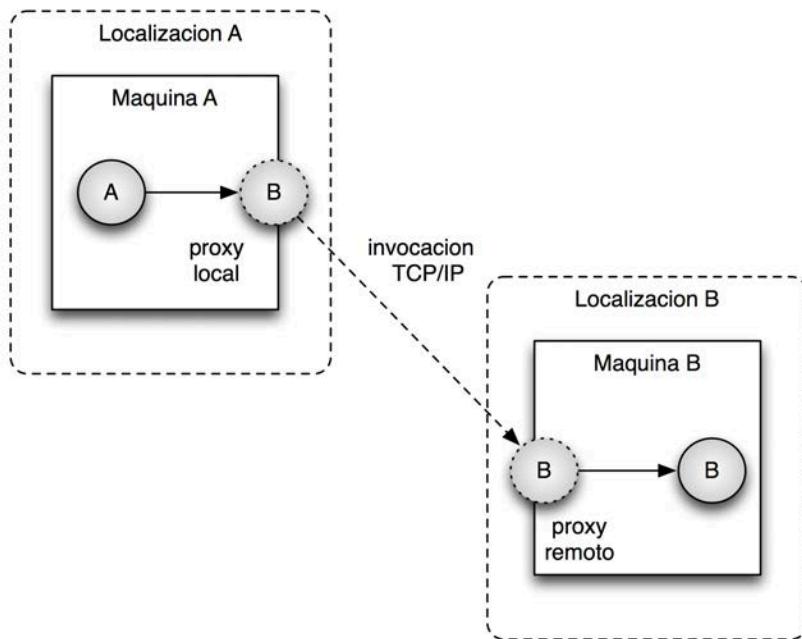
Como podemos ver, tenemos dos objetos A y B que necesitan comunicarse entre ellos. Sin embargo de entrada esto no es posible ya que cada uno de ellos se encuentra ubicado en una máquina distinta y por lo tanto únicamente pueden acceder a objetos ubicados en su misma máquina y proceso. Para conseguir que estos dos objetos puedan intercambiar información necesitamos un diseño más complejo. Este diseño se construye a través del uso de proxies, patrón de diseño que ya hemos visto anteriormente .En este caso se necesitan dos proxies : el proxy local y el proxy remoto que se encargan de facilitar la comunicación. A continuación se explican detalladamente ambos conceptos.

Proxy Local: Objeto que da la sensación de ser el objeto real al que queremos acceder de forma remota y que está ubicado en la máquina local .Se encarga de gestionar la comunicación entre el objeto local y el servidor remoto.

Proxy Remoto: Objeto que da la sensación de ser idéntico al objeto remoto y se ubica en la máquina remota, gestionando la comunicación entre el proxy local y el objeto remoto.

Arquitectura Java

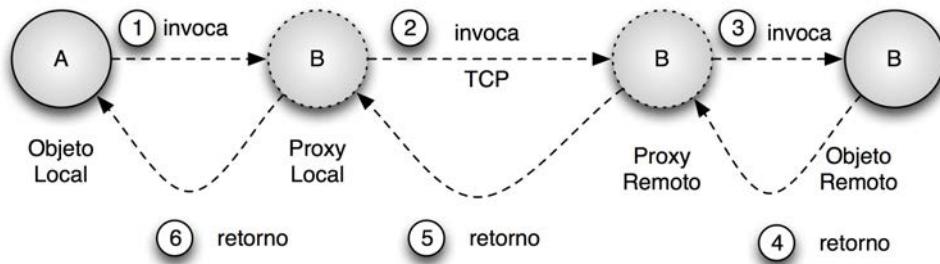
Como estos conceptos pueden ser complejos en un primer momento, vamos a mostrar un diagrama que incluye los objetos reales y los proxies.



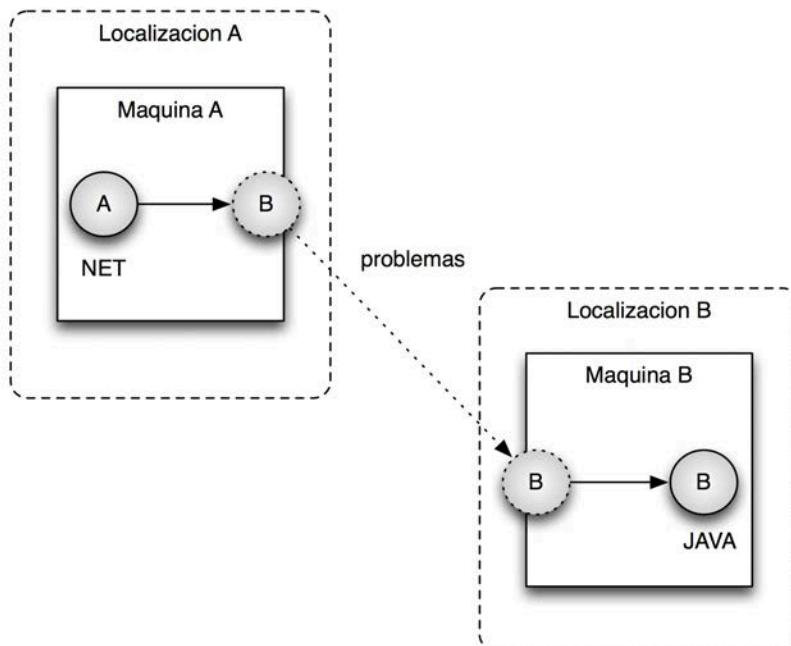
De esta forma la funcionalidad es implementada dando los siguientes pasos

1. Un objeto A invoca un método del proxy local B que se encuentra en su misma máquina.
2. Una vez este objeto recibe la invocación, implementa la funcionalidad necesaria para convertir esa invocación en una llamada por red via TCP/IP hacia el proxy remoto.
3. Una vez el proxy remoto recibe el mensaje, lo convierte en una petición normal e invoca al objeto remoto que es el que tiene la lógica de negocio a la cual queríamos acceder.
4. Este objeto remoto nos devolverá un resultado que el proxy remoto convertirá en una petición TCP y lo enviará al proxy local
5. El proxy local devolverá esta información al objeto local
6. El objeto local podrá hacer uso de esta información

A continuación se muestra un diagrama aclaratorio:



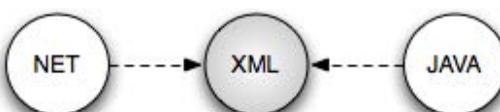
Esta forma de comunicación está soportada por varias plataformas a día de hoy, entre las cuales destacaríamos JAVA y .NET. Éstas son capaces de crear automáticamente los proxies necesarios de una forma transparente. En java la comunicación distribuida se realiza a través de RMI y en .NET a través de .NET Remoting. Ahora bien, este tipo de comunicación sólo es válida cuando ambas maquinas albergan la misma tecnología (Java o NET). En el caso de que una maquina use Java y la otra .NET, la comunicación no es posible ya que los protocolos son distintos (ver imagen).



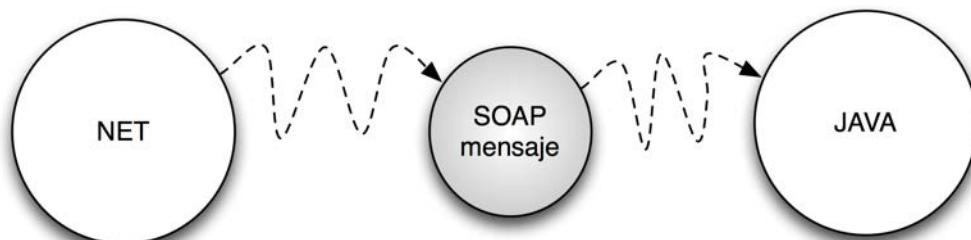
Este tipo de problema es el que los servicios web nos ayudarán a solventar. A continuación se introducen sus conceptos.

2. Introducción a Servicios Web

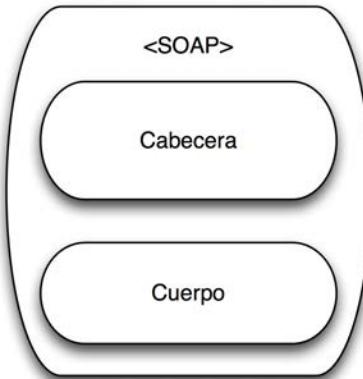
Como acabamos de ver, la comunicación entre .NET y Java no puede ser directa. Para conseguir que dos plataformas heterogéneas se puedan comunicar, necesitamos usar un estándar que ambas entiendan. Esta estándar es XML ya que tanto .NET como Java son capaces de trabajar con esta tecnología (ver imagen).



Una vez que hemos decidido usar XML como tecnología, tendremos que definir cuál es el protocolo de comunicación entre los dos objetos. Este protocolo ya existe y se denomina SOAP (Simple Object Access Protocol). SOAP permitirá comunicar dos tecnologías diferentes usando XML como standard a la hora de transferir información entre los dos sistemas. A continuación se muestra una figura aclaratoria.



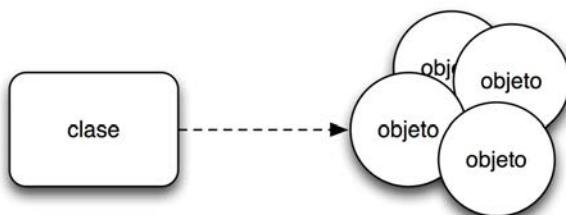
Una vez que entendemos que SOAP es el protocolo de comunicación basado en XML que permite la transferencia de información entre dos objetos creados en tecnologías distintas, es momento de hablar un poco más en profundidad de la estructura de un mensaje SOAP. Un mensaje SOAP está construido con etiquetas XML y se compone de dos estructuras principales: cabecera y cuerpo (ver imagen).



Vamos a comentar a grossó modo cuál es la responsabilidad de cada una de estas partes:

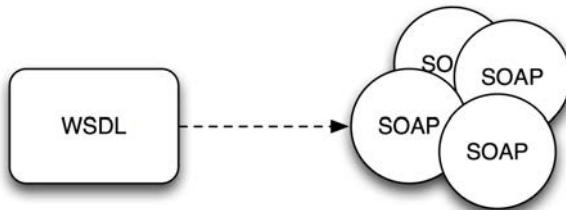
- **Cabecera:** Se encarga de definir conceptos transversales al mensaje, como por ejemplo tipo de autenticación.
- **Cuerpo:** Se encarga de definir los métodos que se van a invocar de forma remota así como los tipos de datos asociados a estos métodos.

Evidentemente a través de SOAP podemos realizar cientos de peticiones hacia el mismo objeto y mismo método. Envaremos un mensaje SOAP y nos será devuelto otro. Ahora bien, a la hora de construir el mensaje, debemos usar otra tecnología que se encarga de definir cuál es la estructura de estos mensajes. En programación orientada a objeto cuando queremos construir un objeto, tenemos que definir primero una clase (ver imagen)



Arquitectura Java

Igualmente que una clase define cómo se construyen los distintos objetos de ella, los servicios web disponen de un standard que define cómo han de crearse los distintos mensajes SOAP para un servicio web en concreto. Este standard se denomina WSDL(Web ServiceDescriptionLanguage). A continuación se muestra una imagen aclaratoria.



Así pues a nivel de servicios web, el fichero WSDL hace las funciones de clase y los mensajes SOAP hacen la función de objetos que pasan entre los dos sistemas distribuidos definidos a nivel de XML. Una vez claros estos conceptos, es momento de comenzar a construir un servicio web.

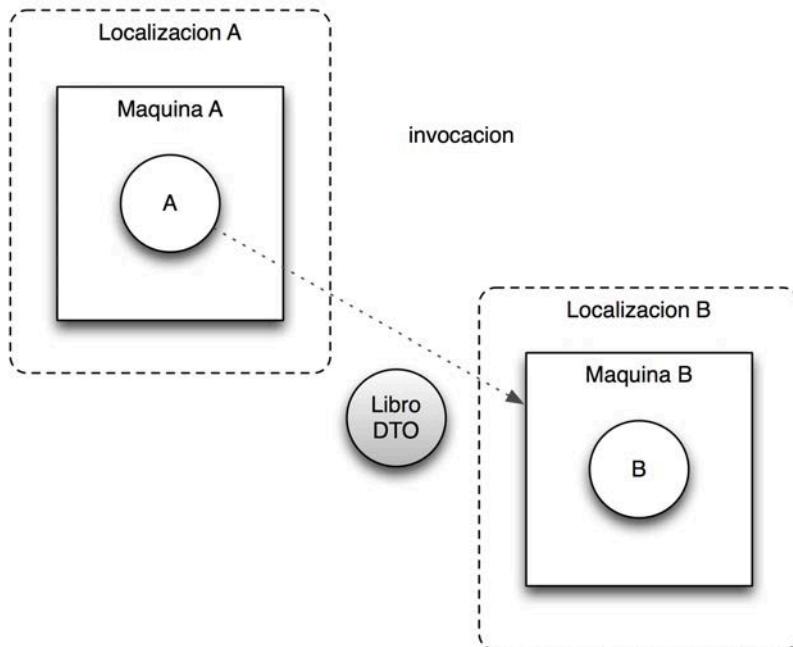
3. Servicio Web

Una de las ventajas de usar tecnología Java a la hora de construir Servicios Web es que permite construir un servicio web a partir de una sencilla clase Java a través del uso de anotaciones. Estas anotaciones están definidas en el estándar JAX-WS de JEE. Una vez anotemos la clase ,JAX-WS se encargará de construir los proxies y el fichero WSDL de forma automática. Es momento de definir qué información queremos publicar vía servicio web .En nuestro caso el servicio web a construir nos devolverá una lista con todos los libros incluyendo la siguiente información:

- isbn
- título
- descripción de la categoría

Si nos fijamos, ninguna de nuestras dos clases de negocio que en este momento tenemos en nuestra aplicación Libro y Categoría incorpora la información necesaria. Así pues, vamos a construir una nueva clase que realice las funciones de contenedor de información y encaje con los campos solicitados. Estas clases se denominan habitualmente DTO (Data Transfer Objects) ya que son clases construidas específicamente para la transmisión de información entre dos sistemas.

Los DTO o Data Transfer Objects son patrones de diseño orientados a programación distribuida. A continuación se muestra una imagen aclaratoria que hace uso de ellos.



Una vez que tenemos clara cuál es la funcionalidad de esta clase, vamos a ver su código fuente.

Código 23.1: (LibroDTO.java)

```

package com.arquitecturajava.aplicacion.serviciosexternos;

public class LibroDTO {
    private String isbn;
    private String titulo;
    private String categoria;
    public LibroDTO() {
    }
    public LibroDTO(String isbn, String titulo, String categoria) {
        super();
        this.isbn = isbn;
        this.titulo = titulo;
        this.categoria = categoria;
    }
    public String getCategoría() {
        return categoria;
    }
}

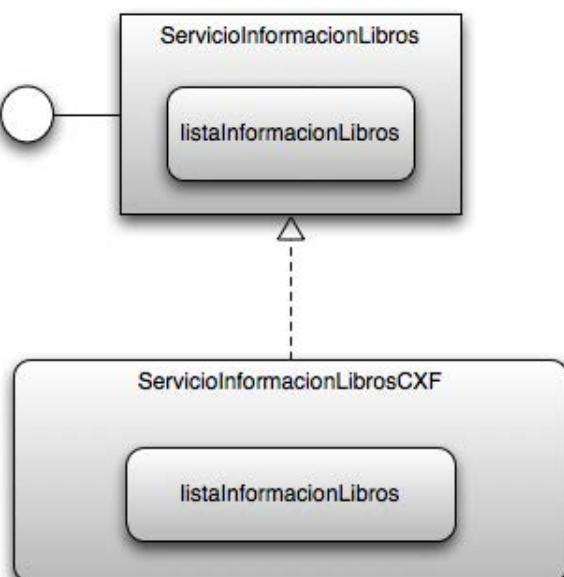
```

```
public String getIsbn() {
    return isbn;
}

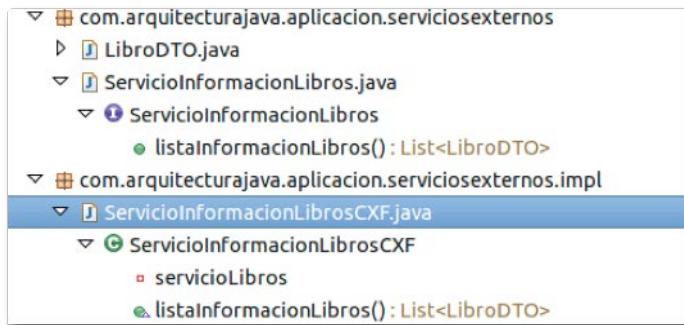
public String getTitulo() {
    return titulo;
}
public void setCategoria(String categoria) {
    this.categoria = categoria;
}
public void setIsbn(String isbn) {
    this.isbn = isbn;
}

public void setTitulo(String titulo) {
    this.titulo = titulo;
}
}
```

Construida la clase que vamos a utilizar para transferir la información entre las dos aplicaciones (LibroDTO), es momento de construir una clase de servicio que tenga un método que nos devuelva una lista de LibroDTO con toda la información referente a los libros. Al tratarse de una clase de servicio la denominaremos ServicioInformacionLibros y al método que nos devuelve la lista de LibroDTO le denominaremos listaInformacionLibros. Para construirla nos apoyaremos en el uso de interfaces (ver imagen)



Como podemos ver, dispondremos de un interface ServicioInformacionLibros y de una clase que lo implementa. A continuación se muestra una imagen con los paquetes y clases que debemos añadir al proyecto



Una vez tenemos claro qué interfaces y clases vamos a construir, es momento de introducir un par de anotaciones adicionales que pertenecen al estándar JAX-WS y son las encargadas de simplificar la publicación de servicios web en este tipo de entorno.

- **@WebService** : Anotación que marca una clase como servicio web
- **@ WebMethod** : Anotación que marca un método como público a nivel del servicio web.

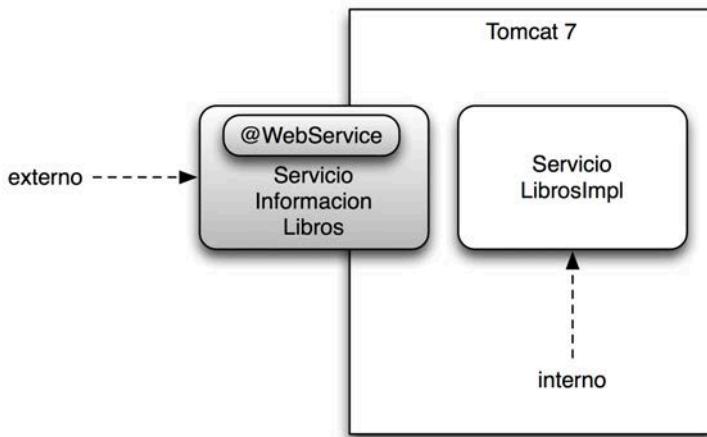
Vamos a usar a continuación estas anotaciones para crear un Servicio Web que cumpla con los requisitos que nosotros necesitamos. A continuación se muestra el código fuente de éste.

Arquitectura Java

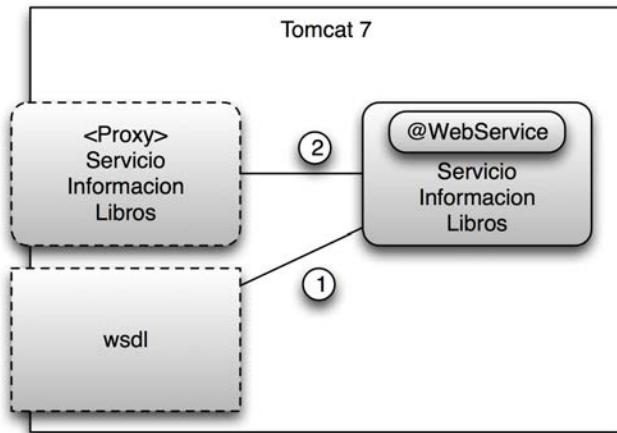
Código 23.2: (ServicioInformacionLibrosCXF.java)

```
//omitimos imports
@Service
@WebService(endpointInterface="com.arquitecturajava.aplicacion.serviciosexternos.ServicioInformacionLibros")
public class ServicioInformacionLibrosCXF implements ServicioInformacionLibros{
    @Autowired
    private ServicioLibros servicioLibros;
    @WebMethod
    public List<LibroDTO> listaInformacionLibros() {
        List<Libro> listaLibros=servicioLibros.buscarTodosLosLibros();
        List<LibroDTO> listaDestino= new ArrayList<LibroDTO>();
        for(Libro libro: listaLibros) {
            LibroDTO libroExterno= new LibroDTO(libro.getIsbn(),libro.getTitulo(),
                libro.getCategoría().getDescripción());
            listaDestino.add(libroExterno);
        }
        return listaDestino;
    }
}
```

Es evidente que nos hemos apoyado en las anotaciones `@Autowired` y `@Service` para acceder a la información de `ServicioLibros`. Una vez disponemos de esta información usaremos la anotación `@WebService` para publicar esta información hacia el exterior de la aplicación (ver imagen).



Al haber anotado la clase con `@WebService` JAX-WS se encargará de generar los proxies y el fichero WSDL necesario para acceder al servicio. A continuación se muestra una imagen aclaratoria.



Hemos terminado de construir nuestro servicio web .Sin embargo para que se generen los proxies y el fichero wsdl deberemos dar una serie de pasos en cuanto a Tomcat y Spring se refiere.

4. Publicación del servicio web.

Tomcat 7 de por sí no implementa soporte para el uso de servicios web con JAX-WS, ya que no cumple con todos los estándares de JEE : necesitaremos utilizar algún framework o librería adicional para poder usarlo.Para ello vamos utilizar Apache CXF como framework de servicios web, ya que se integra de una forma muy natural con Spring. La siguiente url nos permite obtener este framework.

<http://cxf.apache.org>

Una vez obtenido el framework pasamos a instalar sus librerías dentro de nuestra aplicación.

5. Instalación de Apache CXF

Apache CXF incorpora un grupo amplio de librerías, sin embargo las que necesitamos utilizar para nuestro ejemplo son bastante menas ya que parte nos las proporciona el propio framework Spring. En concreto tendremos que añadir las siguientes a nuestro proyecto:

- Cxf-2.4.0
- Neethi-3.0.0
- Wsdl4j-1.6.2
- Xmlschema-core-2.0

6. Configuración del framework

En este apartado nos vamos a encargar de configurar Apache CXF sobre Spring. Para ello es necesario realizar el siguiente conjunto de modificaciones:

- Modificar el fichero web.xml y añadir el servlet de apache CXF que facilita la integracion con nuestra aplicacion
- Modificar el fichero contextoAplicacion.xml y añadir las etiquetas necesarias de apache CXF

Vamos a modificar el fichero web.xml y configurar el servlet de Apache CXF que se encargara de gestionar los distintos servicios web. A continuacion se muestra el bloque de código que debemos añadir al web.xml.

Código 23.3: (web.xml)

```
<servlet-class>  
org.apache.cxf.transport.servlet.CXFServlet  
</servlet-class>  
<load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
<servlet-name>CXFServlet</servlet-name>  
<url-pattern>/servicios/*</url-pattern>  
</servlet-mapping>
```

Una vez hecho esto estamos en disposición de configurar la parte relacionada con Spring Framework y dar de alta nuestro servicio. Así pues vamos a ver qué elementos del fichero de configuración de Spring debemos modificar. En primer lugar es necesario añadir un nuevo namespace a los que ya tenemos para poder hacer uso de las etiquetas jaxws (ver código).

Código 23.4: (contextoAplicacion.xml)

```
xmlns:jaxws="http://cxf.apache.org/jaxws"  
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
```

Hecho esto, añadiremos varios ficheros xml de Apache CXF al fichero de Spring

Código 23.5: (contextoAplicacion.xml)

```
<import resource="classpath:META-INF/cxf/cxf.xml" />
<importresource="classpath:META-INF/cxf/cxf-extensionsoap.xml" />
<importresource="classpath:META-INF/cxf/cxf-servlet.xml" />
```

Tenemos ya importados los ficheros que Apache CXF necesita para trabajar. Únicamente necesitamos configurar el servicio web a nivel de Spring, lo cuál no presentará problemas (ver código).

Código 23.6: (contextoAplicacion.xml)

```
<jaxws:endpoint
    id="servicioInformacionLibros"
    implementor="com.arquitecturajava.aplicacion.serviciosexternos.
        impl.ServicioInformacionLibrosCXF"
    address="servicioInformacionLibros" />
</beans>
```

Como podemos ver la configuración es sencilla: en primer lugar asignamos un id al servicio que vamos a dar de alta

Código 23.7: (contextoAplicacion.xml)

```
id="servicioInformacionLibros"
```

Una vez que tenemos definido el identificador del servicio, definimos qué clase se encarga de implementar este servicio.

Código 23.8: (contextoAplicacion.xml)

```
implementor="com.arquitecturajava.aplicacion.serviciosexternos.
    impl.ServicioInformacionLibrosCXF"
```

Por ultimo, asignamos la dirección (url) desde la cuál accederemos al servicio web de nuestra aplicación.

Arquitectura Java

Código 23.9: (contextoAplicacion.xml)

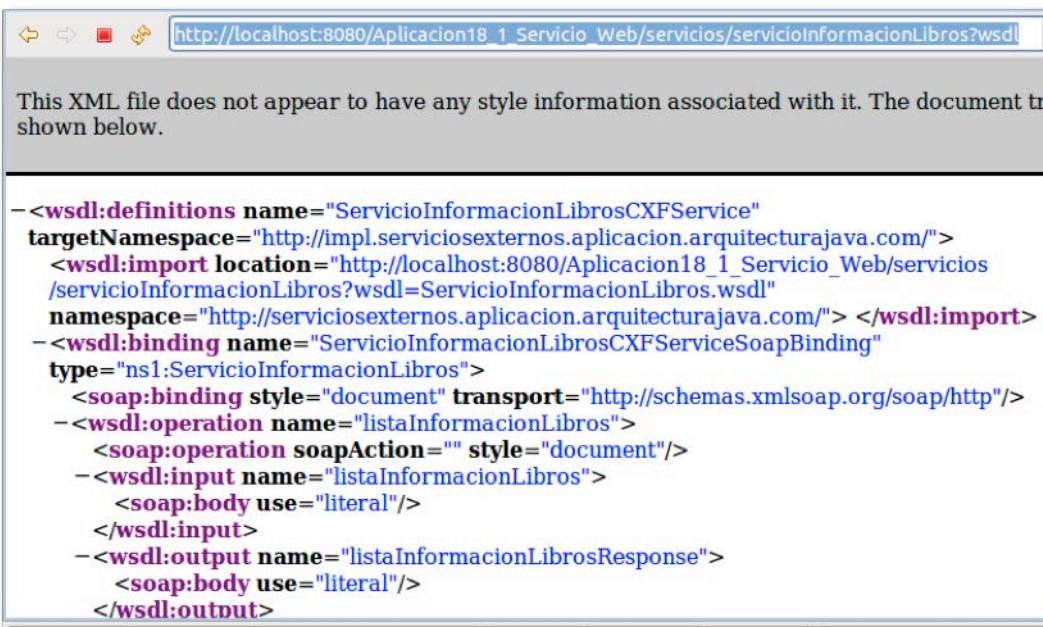
```
address="servicioInformacionLibros"
```

Con esta configuración hemos terminado de modificar el fichero de configuración de Spring. Podremos acceder al servicio a través de la siguiente URL:

Código 23.10: (contextoAplicacion.xml)

```
http://localhost:8080/Aplicacion18_1_Servicio_Web/servicios/servicioInformacionLibros?  
wsdl
```

Esta URL nos devuelve el fichero WSDL que necesitamos para construir un cliente y acceder al servicio (ver imagen).



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<wsdl:definitions name="ServicioInformacionLibrosCXFService"
    targetNamespace="http://impl.serviciosexternos.aplicacion.arquitecturajava.com/">
    <wsdl:import location="http://localhost:8080/Aplicacion18_1_Servicio_Web/servicios/servicioInformacionLibros?wsdl=ServicioInformacionLibros.wsdl"
        namespace="http://serviciosexternos.aplicacion.arquitecturajava.com/"> </wsdl:import>
    <wsdl:binding name="ServicioInformacionLibrosCXFServiceSoapBinding"
        type="ns1:ServicioInformacionLibros">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="listaInformacionLibros">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input name="listaInformacionLibros">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="listaInformacionLibrosResponse">
                <soap:body use="literal"/>
            </wsdl:output>
```

Resumen

En este capítulo hemos introducido brevemente el framework Apache CXF y como este framework se integra con Spring y permite la construcción de Servicios Web de una forma relativamente sencilla .Además hemos visto cómo definir un DTO que sirva para transferir información entre dos sistemas.Los DTO no son siempre necesarios a la hora de transferir la información, dependerá de nuestra situación. En otros momentos puede ser más práctico simplemente usar objetos de negocio.

24. Administración y pools

En el capítulo anterior hemos diseñado un servicio web usando Apache CXF como framework .No vamos a añadir más funcionalidad a la aplicación : podemos considerarla terminada.

Este capítulo intenta añadir algo más basado en mis experiencias trabajando con la plataforma JEE.Muchos desarrolladores construyen aplicaciones en entornos JEE sin pensar realmente en los entornos de producción donde éstas van a ser ejecutadas. Por así decirlo, trabajan de una forma totalmente independiente y finalmente entregan un war ,ear etc olvidándose del resto. Sin embargo muchas veces los administradores de sistemas tienen algo que decir y es importante disponer en nuestra organización de expertos en JEE a nivel de desarrollo, pero también es igual de importante tener expertos en JEE a nivel de sistemas ya que éstos tienen un peso clave en los entornos de producción .El objetivo de este capítulo no es ni más ni menos que revisar nuestra aplicación y ver como la podríamos configurar de mejor forma para un administrador de sistemas. Para ello nos centraremos en cómo gestionar los pools de conexiones y las diferencias de visión entre el equipo de desarrollo y el de sistemas.

Objetivos

- Configuración de la aplicación para administradores.

Tareas

1. El framework Spring y pools.
2. Pool de conexiones y Tomcat.
3. Configuración de Spring vía JNDI.

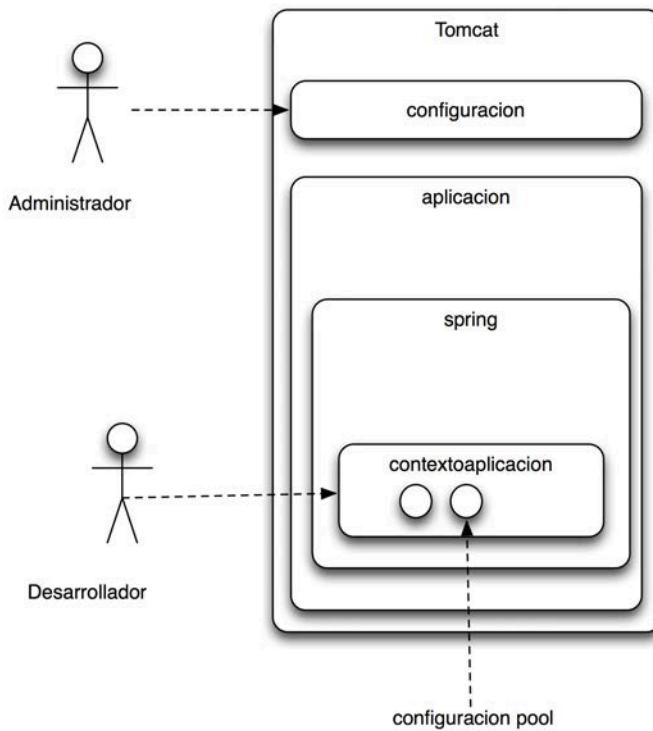
1. El framework Spring y Pools

Hemos gestionado los pools de conexiones o Datasource a través de las capacidades de inyección de dependencia que tiene Spring. Concretamente si recordamos hemos definido a nivel de Spring un pool de conexiones de la siguiente forma:

Código 24.1: (contextoAplicacion.xml)

```
<bean id="fuentedatos"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver" />
      <property name="url"
value="jdbc:mysql://localhost/arquitecturaJavaORM" />
      <property name="username" value="root" />
      <property name="password" value="java" />
</bean>
```

Nos puede parecer en un primer momento correcto como desarrolladores .Sin embargo esta configuración se sale de la configuración de nuestro servidor de aplicaciones : es algo completamente independiente y gestionado por un Framework concreto. Por lo tanto puede darse el caso de que el administrador de sistemas no esté al tanto ni sepa localizar estos recursos (ver imagen).



Como la imagen muestra únicamente el desarrollador tiene claro cómo se ha configurado el pool de conexiones usando el framework Spring. En cambio el administrador desconoce completamente que este pool de conexiones existe. Una vez hemos identificado el problema , debemos reconfigurar nuestra aplicación para que encaje de forma más natural con el servidor de aplicaciones que estemos usando y con las tareas de administración. En este caso nuestro servidor es fácil de configurar ya que se trata de Tomcat. Tomcat como servlet container soporta la configuración de Datasources o pools de conexiones a nivel de administración. Vamos a verlo en la siguiente tarea.

2. Pool de conexiones y Tomcat

Para configurar un pool de conexiones sobre Tomcat debemos crear un nuevo fichero de configuración que se denomina `context.xml` y que se encontrará ubicado en la carpeta `META-INF` de nuestra aplicación web (ver imagen).



El fichero en cuestión almacena la configuración necesaria para crear un pool de conexiones para nuestra aplicación a nivel de servidor. Veamos el contenido del fichero:

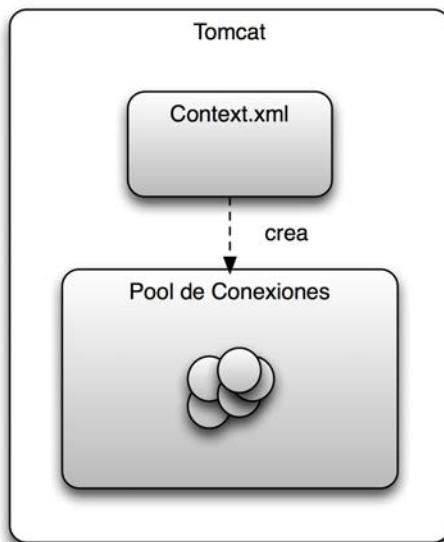
Código 24.2: (context.xml)

```
<Contextpath="/AplicacionFinal" docBase="AplicacionFinal"
reloadable="true" crossContext="true">
<Resource name="jdbc/MiDataSource" auth="Container"
          type="javax.sql.DataSource"
maxActive="100" maxIdle="30" maxWait="10000"
username="root" password="java" driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/arquitecturaJavaORM"/>
</Context>
```

El fichero define una etiqueta Context en la cuál se declara en nombre de la aplicación que va a disponer del pool de conexiones. Una vez tenemos definida esta parte, la etiqueta Resource se encarga de configurar el pool de conexiones en cuestión. A continuación se explican las distintas propiedades asignadas.

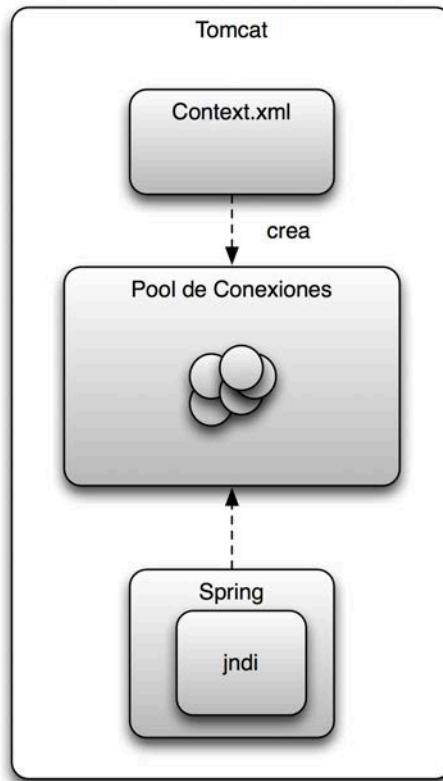
- **name** : Ruta de acceso al pool de conexiones
- **maxActive**: Número de conexiones activas en el pool
- **maxWait** : Tiempo máximo de espera por una conexión definido en milisegundos
- **user**: Usuario de acceso para la conexión
- **password** : Password de acceso para la conexión
- **driverClassName**: Clase a instanciar a nivel de driver JDBC
- **url** : Base de Datos a la que conectarnos

Una vez realizadas estas operaciones nuestro servidor ya dispone de forma pública de un pool de conexiones para nuestra aplicación (ver imagen).



3. Configuración de Spring vía JNDI.

Realizada esta operación, Tomcat ha dado de alta un nuevo recurso JNDI a nivel del servidor de aplicaciones , tendremos que modificar la estructura de nuestro fichero de configuración de spring (contextoAplicacion.xml) para que, en vez de crear un pool de conexiones propio, delegue en el pool de conexiones definido a nivel de servidor en el árbol jndi (ver imagen).

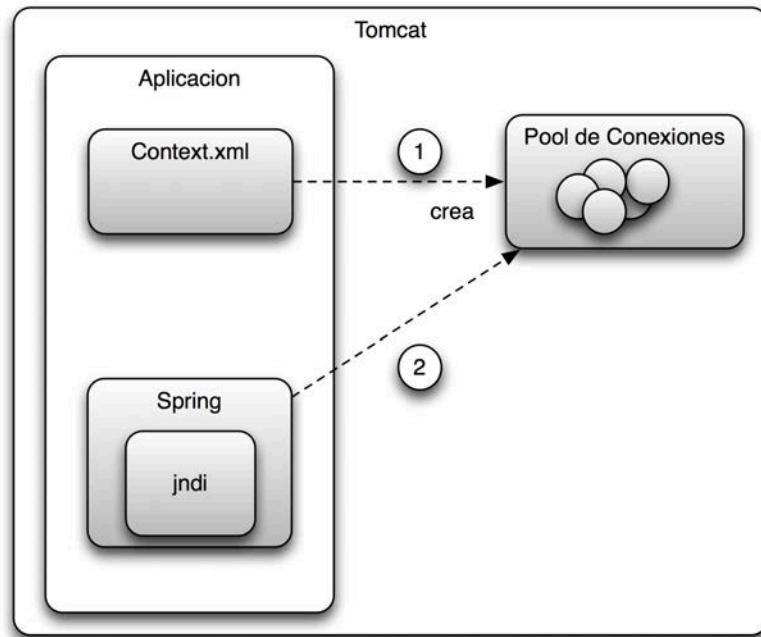


Para configurar el framework Spring para que haga uso de JNDI, debemos eliminar la etiqueta que hace referencia a un datasource que hemos creado con Spring y cambiarla por un datasource que delega en JNDI (ver imagen).

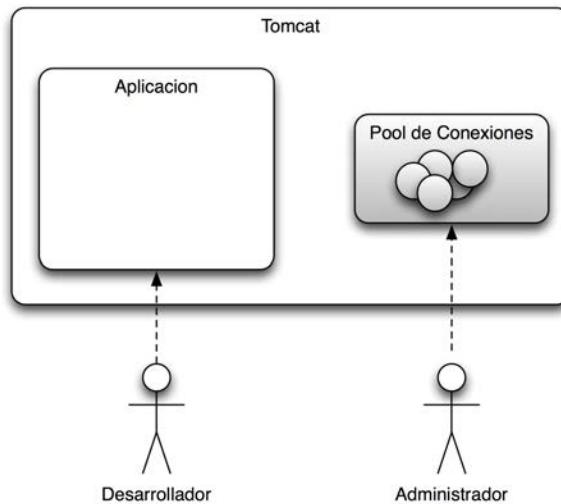
Código 24.4: (contextoAplicacion.xml)

```
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/MiDataSource"/>
```

De esta manera el framework spring delegará en los recursos JNDI del servidor de aplicaciones (ver imagen).



Así pues habremos configurado la gestión de pools de conexiones de una forma más amigable de cara a los administradores de sistemas y habremos asignado las responsabilidades de una manera más correcta (ver imagen).



Para que la etiqueta anterior nos funcione correctamente deberemos dar de alta un nuevo namespace a nivel del fichero de configuración de spring. A continuación se muestra el código:

Código 24.25: (contexto.xml)

```
xmlns:jee="http://www.springframework.org/schema/jee"  
http://www.springframework.org/schema/jee/spring-jee-2.0.xsd"
```

Una vez realizadas estas operaciones la aplicación funcionará correctamente utilizando el pool de conexiones de tomcat.

Resumen

En este caso el único cambio producido en la aplicación es la creación de un pool de conexiones a través de los ficheros de configuración del servidor ,de tal forma que la aplicación sea más fácil de gestionar por parte de los administradores y las responsabilidades estén mejor repartidas, algo que en los entornos reales es fundamental.

25. Conclusiones

En los capítulos de este libro hemos ido paso a paso desarrollando una pequeña aplicación JEE .Para ello hemos hecho uso de principios de ingeniería, de frameworks y de patrones de diseño diversos .Todos ellos juntos nos han permitido desarrollar una aplicación flexible. Ahora quizas una pregunta dificil de responder es la siguiente.

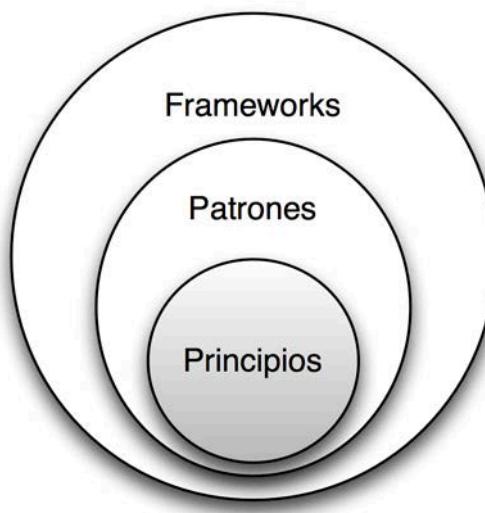
¿Que es lo mas importante?

1. Patrones de Diseño
2. Frameworks
3. Principios de Ingenieria

Para la mayor parte de los desarrolladores lo más importante es sin ninguna duda el conjunto de frameworks que ha de utilizar .Mi experiencia me indica que, aunque evidentemente esto es clave a la hora de desarrollar una arquitectura, es todavía más importante conocer y comprender los distintos patrones de diseño .Todos los frameworks están construidos apoyándose en los distintos patrones de diseño y un conocimiento sólido de éstos nos permitirá entender de una forma más natural el funcionamiento de los frameworks . Simplemente por recordar algunos de los ejemplos que hemos proporcionado en los distintos capítulos:

- JSF como framework se basa en un patron MVC2
- Spring y su integración con Hibernate se apoya en un patrón Template y en el patrón DAO

Si seguimos profundizando en este análisis, pronto nos daremos cuenta de que todos los patrones de diseño utilizados en los distintos capítulos aparecen a partir del uso de uno de los principios de ingeniería de software. Por ejemplo, el patron MVC aparece una vez que hemos dividido las responsabilidades usando el principio SRP .Por otro lado el patron Template aparece al hacer uso del principio DRY en una jerarquia de clases concreta. Por lo tanto lo más importante para los arquitectos es conocer estos principios de ingeniería de software ya que nos facilitará sobremanera el entender por qué un código se ha de construir de una manera u otra .He aquí un diagrama de la relación entre los tres conceptos:



Hemos visto muchos principios durante los distintos capítulos del libro, a continuación los enumeramos:

- DRY (Dont Repeat Yourself)
- SRP (Simple responsibility Principle)

Arquitectura Java

- IOC (Inversion of Control)
- DIP (Dependency Inversion Principle)
- COC (Convention over Configuration)
- OCP (Open Closed Principle)
- LSP (Liskov Substitution Principle)
- ISP (Interface Segregation Principle)

Un subconjunto de este grupo de principios se encuentra unificado en un acrónimo al cuál cada día se hace más referencia : “**SOLID**” o diseño “SOLID”, que hace referencia a un desarrollo que cumple con los siguientes principios.

- SRP
- OCP
- LSP
- ISP
- DIP

Apoyarnos en estos principios nos mantendrá en la senda correcta a la hora de abordar los distintos desarrollos : de ahí el deriva el título de este libro.

1. JEE un ecosistema

Hemos desarrollado durante los distintos capítulos una aplicación JEE .Es probable que mucha gente pueda pensar que son Spring, JPA y JSF los frameworks o estándares a utilizar a la hora de abordar una solución JEE. Sin embargo hay que recordar que JEE no es a día de hoy tanto una plataforma como un ecosistema donde convergen soluciones muy distintas ; las usadas en el presente libro no tienen por qué ser las idóneas para todo el mundo o en todas las situaciones. Seguidamente vamos a comentar algunas que pueden substituir o ser complementarias a las que hemos visto:

Enterprise Java Beans (3.0, 3.1): En el libro hemos usado Spring como framework principal, sin embargo los estándares se orientan hacia una solución basada en EJB ya que éstos han madurado mucho y son una opción tan válida como Spring a la hora de abordar muchos proyectos.

Spring MVC: Hemos usado JSF en nuestra aplicación pero a veces los estándares se pueden quedar cortos o no ser lo suficientemente flexibles para lo que necesitamos . Spring MVC puede ser a día de hoy una alternativa muy adecuada a la capa de presentación y sobre todo muy distinta a lo que JSF representa.

JBoss Seam: Seams es un framework ubicado a nivel de abstracción por encima de los EJBs y complementa de forma sólida la definición de capa de presentación definida por JSF.

Groovy y Grails: Groovy es un lenguaje dinámico basado en las ideas de Ruby y Grails es la versión de Rails para este lenguaje. Si estamos interesados en desarrollos agiles y lenguajes compactos que aceleren el desarrollo, ésta puede ser también una buena alternativa a otras soluciones.

Android: No hemos cubierto la parte de movilidad en este libro pero a día de hoy es obligatorio hacer una referencia a Android a la hora de desarrollar versiones móviles de nuestras aplicaciones.

HTML 5 Frameworks de movilidad: Por último serán cada dia más importantes los frameworks orientados a movilidad de HTML 5 que permiten el desarrollo de una única aplicación que funcione en varias plataformas, ya que recordemos Android solo representa una parte de este pastel. Ejemplos de estos nuevos frameworks son PhoneGap y JQuery Mobile.

Resumen

Hemos utilizado frameworks ,patrones y principios en este Libro. Los frameworks, aunque nos parezcan importantes, tienen un ciclo de vida unos vienen y otros se van . En cambio los principios y patrones nos acompañarán y nos serán útiles durante la carrera profesional que tengamos. Apostemos de una forma mas sólida por conocerlos. Para terminar, si algo he aprendido estos años de las plataformas J2EE y JEE, es a no ser dogmático y a estar abierto a ideas y soluciones nuevas que van llegando. Puede ser que el próximo Spring Framework o el próximo Android estén a la vuelta de la esquina : es necesario tener la mente abierta para aceptar estas evoluciones.

Bibliografía

DOM Scripting: Web Designwith JavaScript and the Document Object Model by Jeremy Keith and Jeffrey Sambells (Paperback - Dec 27, 2010)

Head First Servlets and JSP: Passing the Sun Certified Web Component Developer Exam by Bryan Basham, Kathy Sierra and Bert Bates(Paperback - Apr 1, 2008)

Java Persistence with Hibernate by Christian Bauer and Gavin King (Paperback - Nov 24, 2006)

Spring in Action by Craig Walls(Paperback - Jun 29, 2011)

Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition) by Deepak Alur, Dan Malks and John Crupi(Hardcover - May 10, 2003)

Apache CXF Web ServiceDevelopment

Murach's Java Servlets and JSP, 2nd Edition by Andrea Steelman and Joel Murach(Paperback - Jan 21, 2008)

Dependency Injection by Dhanji R. Prasanna(Paperback - Sep 4, 2009)

DesignPatterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides(Hardcover - Nov 10, 1994)

Core Java Server Faces (3rd Edition)by David Geary and Cay S. Horstmann(Paperback - Jun 6, 2010)

Aspectj in Action: Enterprise AOP with Spring Applications by RamnivasLaddad and Rod Johnson(Paperback - Oct 5, 2009)

The Pragmatic Programmer: From Journey to Masterby Andrew Hunt and David Thomas(Paperback - Oct 30, 1999)

Patterns of Enterprise ApplicationArchitecture by Martin Fowler(Hardcover - Nov 15, 2002)

El objetivo de este libro es permitir al lector obtener una visión global sobre la plataforma JEE. Adquiriendo los conocimientos necesarios para comenzar a trabajar con los distintos frameworks que soporta (Hibernate, Spring, JSF, etc.).

A diferencia de otros libros que se centran en el manejo de un framework en concreto **Arquitectura Java Sólida** hace hincapié en los principios de ingeniería y patrones de diseño que permiten integrar unos con otros a la hora de construir una solución enterprise.

Para más información visita:

www.arquitecturajava.com



Cecilio Álvarez Caules es Sun Certified Enterprise Architech (J2EE/JEE), Sun Certified Business Component Developer, Sun Certified Web Component Developer y Sun Certified Java Programmer.

Es además Microsoft Certified Solution Developer, Microsoft Certified Enterprise Developer y Microsoft Certified Trainer. Trabaja como consultor, arquitecto y trainer desde hace más de 15 años para distintas empresas del sector.