Spring framework

29/12/2004

1 Introducción.

Con este documento se pretende hacer una breve introducción al "Framework *Spring*". No se pretende hacer un documento que profundice en todos los aspectos de "*Spring*", sólo se desarrollarán aquellos detalles suficientes para comprender la forma de utilizar *Spring* y poder posteriormente profundizar en detalles más concretos usando la documentación existente en su sitio web.
Toda la documentación de *Spring* la podemos encontrar en:
http://www.*Spring*framework.org

1.1 ¿Qué es *Spring*?

Spring es un framework de aplicaciones Java/J2EE desarrollado usando licencia de OpenSource.
Se basa en una configuración a base de *javabeans* bastante simple. Es potente en cuanto a la gestión del ciclo de vida de los componentes y fácilmente ampliable. Es interesante el uso de programación orientada a aspectos (IoC). Tiene plantillas que permiten un más fácil uso de *Hibernate*, iBatis, JDBC..., se integra "de fábrica" con Quartz, Velocity, Freemarker, Struts, Webwork2 y tienen un plugin para eclipse.
Ofrece un ligero contenedor de bean para los objetos de la capa de negocio, *DAOs* y repositorio de *Datasources JDBC* y sesiones *Hibernate*. Mediante un *xml* definimos el contexto de la aplicación siendo una potente herramienta para manejar objetos Songleton o "factorias" que necesitan su propia configuración.
El objetivo de *Spring* es no ser intrusito, aquellas aplicaciones configuradas para usar beans mediante *Spring* no necesitan depender de interfaces o clases de *Spring*, pero obtienen su configuración a través de las propiedades de sus beans. Este concepto puede ser aplicado a cualquier entorno, desde una aplicación J2EE a un *applet*.
Como ejemplo podemos pensar en conexiones a base de datos o de persistencia de datos, como *Hibernate*, la gestión de transacciones genérica de *Spring* para DAOs es muy interesante.

La meta a conseguir es separar los accesos a datos y los aspectos relacionados con las transacciones, para permitir objetos de la capa de negocio reutilizables que no dependan de ninguna estrategia de acceso a datos o transacciones.
Spring ofrece una manera simple de implementar DAOs basados en *Hibernate* sin necesidad de manejar instancias de sesion de *Hibernate* o participar en transacciones. No necesita bloques “try-catch”, innecesario para el chequeo de transacciones. Podríamos conseguir un método de acceso simple a *Hibernate* con una sola línea.

1.2 ¿Que proporciona?

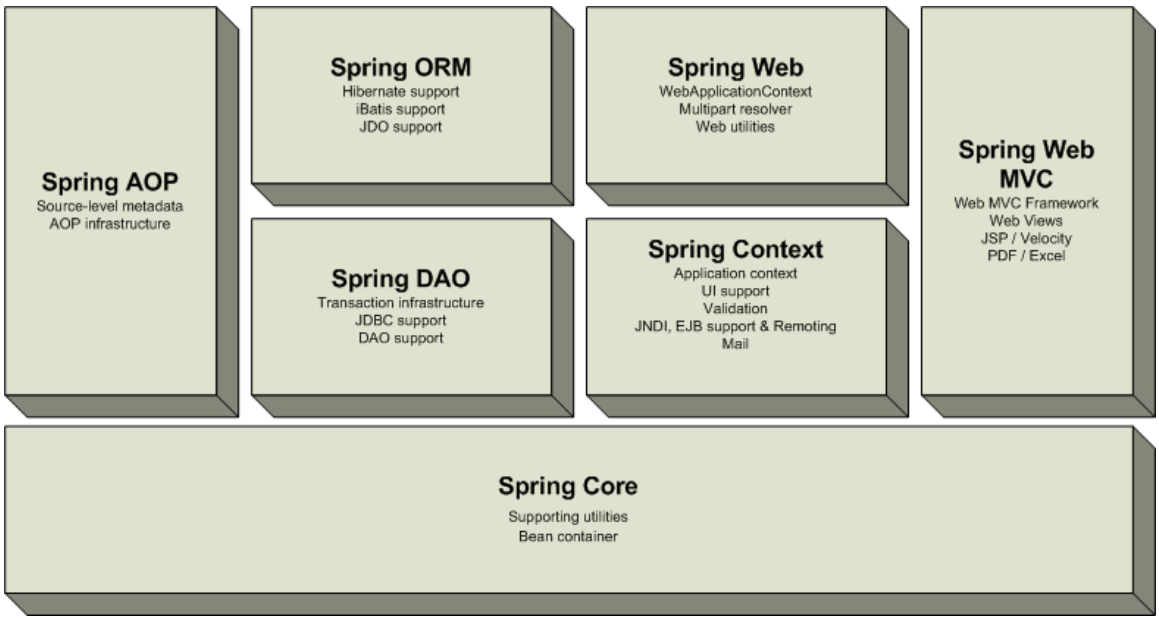
Spring proporciona:

- Una potente gestión de configuración basada en JavaBeans, aplicando los principios de Inversión de Control (*IoC*). Esto hace que la configuración de aplicaciones sea rápida y sencilla. Ya no es necesario tener *singletons* ni ficheros de configuración, una aproximación consistente y elegante. Estas definiciones de *beans* se realizan en lo que se llama el contexto de aplicación.
- Una capa genérica de abstracción para la gestión de transacciones, permitiendo gestores de transacción añadibles (*pluggables*), y haciendo sencilla la demarcación de transacciones sin tratarlas a bajo nivel. Se incluyen estrategias genéricas para JTA y un único JDBC DataSource. En contraste con el JTA simple o EJB CMT, el soporte de transacciones de *Spring* no está atado a entornos J2EE.
- Una capa de abstracción JDBC que ofrece una significativa jerarquía de excepciones (evitando la necesidad de obtener de SQLException los códigos que cada gestor de base de datos asigna a los errores), simplifica el manejo de errores, y reduce considerablemente la cantidad de código necesario.
- Integración con *Hibernate*, JDO e iBatis SQL Maps en términos de soporte a implementaciones DAO y estrategias con transacciones. Especial soporte a *Hibernate* añadiendo convenientes características de *IoC*, y solucionando muchos de los comunes problemas de integración de *Hibernate*. Todo ello cumpliendo con las transacciones genéricas de *Spring* y la jerarquía de excepciones DAO.

- Funcionalidad AOP, totalmente integrada en la gestión de configuración de *Spring*. Se puede aplicar AOP a cualquier objeto gestionado por *Spring*, añadiendo aspectos como gestión de transacciones declarativa. Con *Spring* se puede tener gestión de transacciones declarativa sin EJB, incluso sin JTA, si se utiliza una única base de datos en un contenedor Web sin soporte JTA.
- Un framework MVC (*Model-View-Controller*), construido sobre el núcleo de *Spring*. Este framework es altamente configurable vía interfaces y permite el uso de múltiples tecnologías para la capa vista como pueden ser JSP, Velocity, Tiles, iText o POI. De cualquier manera una capa modelo realizada con *Spring* puede ser fácilmente utilizada con una capa web basada en cualquier otro framework MVC, como Struts, WebWork o Tapestry.

Toda esta funcionalidad puede usarse en cualquier servidor J2EE, y la mayoría de ella ni siquiera requiere su uso. El objetivo central de *Spring* es permitir que objetos de negocio y de acceso a datos sean reutilizables, no atados a servicios J2EE específicos. Estos objetos pueden ser reutilizados tanto en entornos J2EE (Web o EJB), aplicaciones “standalone”, entornos de pruebas, etc.… sin ningún problema.
La arquitectura en capas de *Spring* ofrece mucha de flexibilidad. Toda la funcionalidad está construida sobre los niveles inferiores. Por ejemplo se puede utilizar la gestión de configuración basada en JavaBeans sin utilizar el framework MVC o el soporte AOP.

Ilustración 1: Arquitectura en capas



1.3 ¿Qué es Ioc?

Spring se basa en IoC. IoC es lo que nosotros conocemos como
El Principio de Inversión de Dependencia, Inversion of Control" (IoC) o patrón Hollywood ("No nos llames, nosotros le llamaremos") consiste en:

- Un Contenedor que maneja objetos por ti.
- El contenedor generalmente controla la creación de estos objetos. Por decirlo de alguna manera, el contenedor hace los “new” de las clases java para que no los realices tu.
- El contenedor resuelve dependencias entre los objetos que contiene.

Estos puntos son suficientes y necesarios para poder hablar de una definición básica de IoC. *Spring* proporciona un contenedor que maneja todo lo que se hace con los objetos del IoC. Debido a la naturaleza del IoC, el contenedor más o menos ha definido el ciclo de vida de los objetos. Y, finalmente, el contenedor resuelve las dependencias entre los servicios que él controla.

2 Herramientas necesarias.

Para poder realizar los siguientes ejemplos necesitaremos varias librerias. Para facilitar la ejecución recomiendo tener instalado un ide de desarrollo java como eclipse o netBeans para poder navegar por el código con soltura. En el caso concreto de este documento recomiendo usar eclipse, puesto que es el el IDE que usaré para compilar y ejecutar

Para el conjunto de los ejemplos necesitaremos las librerias:

- *Spring* – en http://www.springframework.org existe un fichero *"Springframework-whith-dependences.zip"* que contiene todas las clases necesarias para ejecutar todas las herramientas de spring.
- Log4j – En el fichero anterior encontramos el jar necesario.
- Jakarta Common-logging – Lo mismo ocurre con esta librería.
- *Hibernate* – Podemos encontrarlo en su pagina web.
- Struts – Podremos localizarlo en la pagina de jakarta-struts.
- JUnit – Podemos encontrar las clases en el fichero de spring-with-dependences. En caso contrario las librerias las podemos encontrar en su sitio web. Si se usa eclipse como ide lo incluye.

3 Primer ejemplo de uso

Hasta ahora solo hemos visto la teoría de que es el framework *Spring*. Ahora vamos a realizar un ejemplo sencillo, muy básico para simular el uso de la capa de configuración de beans, el núcleo básico de Spring, para poder posteriormente ir añadiendo funcionalidades a la aplicación.

En este ejemplo no necesitaremos base de datos, simularemos los accesos a base de datos con una clase que devuelva unos datos constantes. Después intentaremos sustituir esta clase de datos por un acceso a *Hibernate*, para, posteriormente, incluir transacciones.

Para el desarrollo de este ejemplo usaremos JUNIT para el proceso de test de las clases, se podría usar una clase main(), pero se ha considerado más adecuado introducir JUNIT por las posibilidades de test que ofrece.

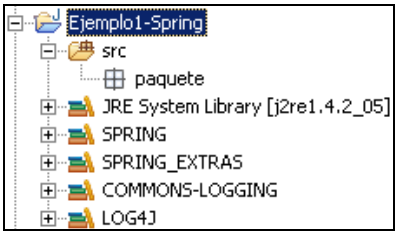
3.1 Librerias necesarias.

En este ejemplo necesitaremos:

- JUNIT
- *SPRING* (con todas las librerías que necesita)
- Common-logging
- Log4j

3.2 La estructura de directorios

La estructura de nuestro primer ejemplo será bastante simple:



Siendo "src" la carpeta donde irán los fuentes.

Las librerías se cargarán configurando eclipse o el IDE que se use.

3.3 Configurando LOG4J

Para nuestro uso personal vamos a configurar log4j para que nos vaya dejando trazas. El fichero será el siguiente.

```
log4j.rootCategory=INFO, Console
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=(%-35c{2} %-4L) %m%n
log4j.logger.paquete=ALL
```

Con este fichero, que colocaremos en la raiz de nuestra carpeta fuente (src), nos mostrara todas nuestras trazas en la consola.

3.4 Programando nuestra clase.

Spring se basa mucho en la programación mediante interfaces, de forma que nosotros crearemos los interfaces y la implementación que los crea. Así que crearemos un interfaz y una clase de modelo de datos. Estas son:

```
/* Clase que representa al usuario */
public class Usuario {
    private Integer id;

    private String nombre;
/* faltan los get y st correspondientes. */

}

/* Interface de acceso a los datos */
public interface UsuarioDao {

    public void saveUsuario (Usuario usuario);
    public Usuario findUsuario (Integer id);
    public void deleteUsuario (Integer id);
    public List listAll ();

}
```

Con estas clases realizamos una primera implementación de acceso a datos. Esta clase almacena los datos en una clase interna de almacenamiento:

```
public class UsuarioDaoStatic implements UsuarioDao {
    private static final Log log = LogFactory.getLog(UsuarioDaoStatic.class);

    private static HashMap tabla;
    public UsuarioDaoStatic ()
    {
        log.debug("Constructor de la implementacion DAO");
        tabla = new HashMap ();
    }

    public void saveUsuario (Usuario usuario) {
        log.debug("Guardamos el usuario "+usuario);
       if (usuario != null)
           tabla.put(usuario.getId(),usuario);
    }

    public Usuario findUsuario (Integer id) {
        log.debug("Estamos buscando usuario "+id);
        return (Usuario) tabla.get(id);
    }

    public void deleteUsuario (Integer id) {
        log.debug ("Borramos el usuario "+ id);
        tabla.remove(id);
    }

}
```

Esta seria una forma normal de cualquier aplicación que accede a una capa de acceso a datos. Ahora configuraremos *Spring* para que cada vez que se solicite acceso al interfaz *UsuarioDao* se haga mediante la implementación que nosotros deseamos.

3.5 Configuración de *Spring*.

Para este primer ejemplo, bastante básico, debemos de configurar *Spring* para que al solicitar el bean UsuarioDao, en este fichero es donde le especificamos la implementación concreta.

El fichero lo llamaremos applicationContext.xml y tendrá como contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
beans.dtd">
<beans>
        <bean id="usuarioDao" class="paquete.dao.impl1.UsuarioDaoStatic" />
</beans>
```

3.6 Clases de test

Ahora vamos a ver como se combinan en un uso normal. Para ello crearemos una clase *TestUsuarioDao*. El código es el siguiente:

```
public class TestUsuarioDao extends TestCase {
    private ClassPathXmlApplicationContext ctx;
    private UsuarioDao dao;
    private Usuario usuario;
    private static final Log log = LogFactory.getLog(TestUsuarioDao.class);

    protected void setUp() throws Exception {
        log.debug("SETUP del test");
        String[] paths = {"applicationContext.xml"};
        ctx = new ClassPathXmlApplicationContext(paths);
        dao = (UsuarioDao) ctx.getBean("usuarioDao");
        log.debug("hemos obtenido el objeto que implementa usuarioDao");
    }
    protected void tearDown() throws Exception {
        usuario = null;
        dao = null;
        }

    public void testAddFindBorrar ()
    throws Exception
    {
        usuario = dao.findUsuario(new Integer(1));
        log.debug("-----------> "+usuario);

        // Solo para verificar que hay conexión y no salta excepción
        usuario = new Usuario ();
        usuario.setId(new Integer (1));
        usuario.setNombre("Nombre usuario");
        dao.saveUsuario(usuario);
        assertTrue(usuario != null);
        Usuario usuario2 = dao.findUsuario(new Integer (1));
        log.debug("Recuperado usuario"+usuario2);
        assertTrue(usuario2 != null);
        log.debug ("Comparamos : "+usuario2 + " con : "+usuario);
        assertTrue (usuario2.equals(usuario));
        // recuperamos el mismo usuario
        dao.deleteUsuario(new Integer(1));
        usuario2 = dao.findUsuario(new Integer(1));
        assertNull("El usuario no debe de existir",usuario2);

    }
    public static void main (String[] args)
    {
        junit.textui.TestRunner.run(TestUsuarioDao.class);
```

```
    }
}
```

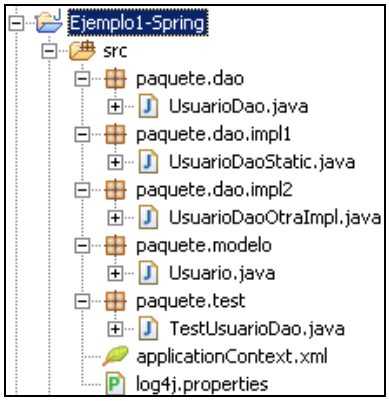
El test lo podemos realizar usando el interfaz gráfico que proporciona eclipse o directamente desde la linea de comando. En cualquier caso, en la salida de la consola obtenemos:

```
(test.TestUsuarioDao          48) SETUP del test
 (impl1.UsuarioDaoStatic      46) Constructor de la implementacion DAO
(test.TestUsuarioDao          52) hemos obtenido el objeto que implementa usuarioDao
(impl1.UsuarioDaoStatic       51) Guardamos el usuario paquete.modelo.Usuario@15212bc
(impl1.UsuarioDaoStatic       57) Estamos buscando usuario 1
(impl1.UsuarioDaoStatic       62) Borramos el usuario 1
(impl1.UsuarioDaoStatic       57) Estamos buscando usuario 1
```

Podemos comprobar como *Spring*, usando el fichero de configuración que hemos generado, nos carga la implementación que nosotros le hemos pedido.

3.7 Programando implementación alternativa.

Para probar el cambio de una implementación sin tener que modificar ni una sola línea de código haremos lo siguiente. Creamos una nueva clase que llamaremos *UsuarioDaoOtraImpl* que para ahorrar tiempo tendrá el mismo contenido que *UsuarioDaoStatic* , solo que lo situaremos en otro paquete, de forma que la estructura total de nuestro proyecto quede.



Si ahora modificamos el fichero applicationContext.xml cambiando la linea:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
beans.dtd">
<beans>
        <bean id="usuarioDao" class="paquete.dao.impl2.UsuarioDaoOtraImpl" />
</beans>
```

Y volvemos a lanzar el test, comprobamos en las trazas obtenidas que hemos cambiado de implementación cambiando un fichero de configuración.
Este aspecto es muy interesante para la aplicación final porque así se centralizan los controles de implementaciones en un fichero y simplificamos el código.

> NOTA: Existe un método *.refresh()* que nos permite recargar el fichero de configuración haciendo una llamada a este metodo. Es decir, que se podria cambiar de implementacion (por ejemplo de acceso con *Hibernate* a otro tipo de acceso) "en caliente".

4 Segundo ejemplo.

El ejemplo anterior es sólo una pequeña muestra de cómo se puede usar *Spring*, las posibilidades sólo se dejan intuir con este ejemplo. Con el siguiente vamos a intentar que nuestra aplicación de ejemplo anterior, usando *Spring* se conecte a base de datos mediante *Hibernate*.
Para poder Realizar este proceso vamos a procurar realizar la menor cantidad de modificaciones en el código anterior, para así apreciar el proceso de integración de *Spring*, que se anuncia como no intrusivo.

4.1 Creacion de base de datos.

Hay que considerar que la creación de una base de datos usando MySql, HSQLDB, Oracle o cualquier otro método se escapa de la finalidad de este documento. Solo comentar que para mi ejemplo concreto usé MySQL.
El script de creación de la tabla es el siguiente.

```
CREATE TABLE `atril`.`USUARIO` (
  `id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  `nombre` VARCHAR(45) NOT NULL,
  PRIMARY KEY(`id`)
)
TYPE = InnoDB;
```

4.2 Configurando *Hibernate*

Para realizar la configuración de *Hibernate*, necesitaríamos crear dos documentos de configuración, uno para la configuración y otro para el mapeo de los datos.

Sin embargo *Spring* se nos anuncia como un método de centralizar la configuración, por lo que no crearemos un fichero de configuración de conexión a *Hibernate*.

Sí crearemos el fichero de mapeo de clase:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
    <class name="paquete.modelo.Usuario" table="USUARIO">
    <id name="id" column="id" type="integer" unsaved-value="0">
        <generator class="assigned"  />
    </id>
    <property name="nombre" column="nombre" type="string" not-null="true"/>
    </class>
</hibernate-mapping>
```

El resto de configuraciones de *Hibernate* la haremos usando *Spring*.

4.3 Programando clases de negocio.

En la documentación existente de *Spring* se indica que *Spring* proporciona integración con *Hibernate*, JDO y iBATIS para el mantenimiento de recursos, soporte para implementación de clases DAO y estrategias de transacción. *Spring* esta especialmente integrado con *Hibernate* proporcionando una serie de características muy prácticas.

Así que seguiremos el ejemplo existente en dicha documentación para integrar *Hibernate*.

Asi que creamos la siguiente clase:

```
package paquete.dao.hibernate;

import java.util.List;

import org.springframework.orm.hibernate.HibernateTemplate;
import org.springframework.orm.hibernate.support.HibernateDaoSupport;

import paquete.dao.UsuarioDao;
import paquete.modelo.Usuario;
// Extiende de una clase que proporciona los métodos necesarios para acceder a Hibernate
public class UsuarioDaoHibernate extends HibernateDaoSupport implements
        UsuarioDao {

    public void saveUsuario (Usuario usuario) {
        this.logger.debug("Intentamos guardar el usuario "+usuario);
        HibernateTemplate temp = getHibernateTemplate();
        if (usuario!= null)
        {
            List listado = temp.find("FROM "+Usuario.class.getName()+" as usuario where usuario.id ="+usuario.getId());
            if (listado.isEmpty())
            {
                this.logger.debug("No contieneo, hacemos un save");
                temp.save(usuario);
            } else {
```

```
            this.logger.debug("Contiene, hacemos un update");
            temp.update(usuario);
          }
        }
    }
    public Usuario findUsuario (Integer id) {
        this.logger.debug("Buscamos el usuario "+id);
        return (Usuario) getHibernateTemplate()
            .get (Usuario.class,id);
    }

    public void deleteUsuario (Integer id) {
        this.logger.debug("Borramos el usuario "+id);
       Usuario usu = (Usuario) getHibernateTemplate().load(Usuario.class,id);
       getHibernateTemplate().delete(usu);

    }
}
```

Esta será la implementación de acceso a *Hibernate*, que sustituye a las implementaciones hechas anteriormente. Como se puede comprobar extiende de *HibernateDaoSupport*, una clase que *Spring* proporciona para facilitar la integración con *Hibernate*.

4.4 Modificando configuración.

El fichero de configuración debemos de modificarlo para que use la implementación de la clase que hemos escrito anteriormente.

La configuración quedaría como:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="usuarioDao"  class="paquete.dao.hibernate.UsuarioDaoHibernate">
    <property name="sessionFactory">
      <ref local="sessionFactory" />
    </property>
  </bean>
<!--<bean id="usuarioDao" -->
<!-- class="paquete.dao.impl1.UsuarioDaoStatic" /> -->
<!-- class="paquete.dao.impl2.UsuarioDaoOtraImpl" /> -->
<!-- Aqui configuramos hibernate -->
<!-- Conección a base de datos -->
  <bean id="myDataSource"  class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"><value>org.gjt.mm.mysql.Driver</value></property>

    <property name="url"><value>jdbc:mysql://localhost/atril</value></property>
    <property name="username"><value>root</value></property>
    <property name="password"><value>root</value></property>
  </bean>

<!-- Hibernate SessionFactory -->
  <bean id="sessionFactory"  class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource"><ref local="myDataSource" /></property>
<!-- Must references all OR mapping files. -->
    <property name="mappingResources"><list>
       <value>paquete/modelo/Usuario.hbm.xml</value>
     </list></property>
```

```
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
        <prop key="hibernate.connection.pool_size">1</prop>
        <prop key="hibernate.show_sql">false</prop>
      </props>
    </property>
  </bean>
</beans>
```

Como se puede apreciar, básicamente se ha añadido al bean "usuarioDao" un parámetro más, "sessionFactory", que no tenian las implementaciones anteriores. Este nuevo parámetro toma su valor de otro bean que a su vez necesita de otro, "myDataSource". Usando estos dos beans extras la implementación con *Hibernate* quedará configurada.

4.5 Ejecución.

Si volvemos a ejecutar la clase de test de los ejemplos anteriores, el funcionamiento debe de ser el mismo. Podemos comprobarlo consultando la base de datos y revisando las trazas obtenidas.

5 *Spring* AOP

5.1 ¿Qué es AOP?

AOP son las siglas en ingles de Programación orientada al aspecto (*Aspect Oriented Programming*).
La definición más simple de AOP es "una manera de eliminar codigo duplicado". Java es un lenguaje orientado a objetos y permite crear aplicaciones usando una determinada jerarquía de objetos, sin embargo esto no permite una manera simple de eliminar código repetido en aquellos objetos que no pertenecen a la jerarquía. AOP permite controlar tareas.
En el siguiente ejemplo modificaremos nuestra aplicación para permitir que se ejecuten tareas de escritura de trazas adicionales, que nos servirá para comprender el mecanismo de AOP, para poder ampliar luego nuevas funcionalidades, como transacciones.
En AOP usaremos conceptos como *interceptor*, que inspeccionará el codigo que se va a ejecutar, permitiendo por lo tanto realizar ciertas acciones como : escritura de

trazas cuando el método es llamado, modificar los objetos devueltos o envio de notificaciones.

5.2 Ejemplo de AOP.

Modificaremos nuestra aplicación anterior para introducir un control de aspecto, en concreto añadiremos una llamada a un sistema de trazas que nos dejará una traza de que un método ha sido llamado.

Para es

Para ello primero crearemos una clase que escribirá la traza. Esta extiende el interfaz MethodInterceptor que pertenece a la api de AOPAlliance (http://aopalliance.sourceforge.net/) que utiliza internamente *Spring*. La clase seria:

```
public class Intereceptor implements MethodInterceptor {
    private static final Log log = LogFactory.getLog(Intereceptor.class);
    public Object invoke (MethodInvocation metodo) throws Throwable {
        log.info(" ---> Metodo solicitado "+metodo.getMethod().getName());
        Object obj = metodo.proceed();
        log.info (" ---> El método ha devuelto : "+obj);
        return obj;
    }

}
```

Spring proporciona una clase ya implementada para realizar esta operación, entre otros interceptores ya implementados. Se recomienda un vistazo a la API de *Spring*

5.3 Modificando Configuración.

Ahora debemos de modificar la configuración de *Spring* para que realice las operaciones de la nueva clase de AOP.

Para ello debemos de realizar los siguientes cambios en el fichero de configuracio:

1. Nombrar el bean que representa nuestro interceptor.
2. Cambiar AutorDao a un proxy-bean que nos cargue la implementación de AutorDao y permitir la intercepción.
3. Indicar al proxy de AutorDao que interceptor debe de “vigilarlo”.

Estas modificaciones se pueden ver en el siguiente fichero, se han puesto comentarios para aclarar la función que realiza cada bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
```

```
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<!-- DEfinimos la clase de proxy que carga la implementacion -->
<bean id="usuarioDao" class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="proxyInterfaces"><value>paquete.dao.UsuarioDao</value></property>
      <property name="target"><ref local="usuarioImpl"/></property>
  <!-- Definimos los interceptores del bean -->
      <property name="interceptorNames">
         <list><value>interceptor</value></list>
      </property>
</bean>

<!-- Implementacion actual de UsuarioDao -->

<bean id="usuarioImpl" class="paquete.dao.hibernate.UsuarioDaoHibernate">
    <property name="sessionFactory"><ref local="sessionFactory" /></property>
</bean>


<!-- Aqui configuramos hibernate -->
<!-- Conección a base de datos usando un datasource de apacheCommons -->
  <bean id="myDataSource"  class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"><value>org.gjt.mm.mysql.Driver</value></property>
    <property name="url"><value>jdbc:mysql://localhost/atril</value></property>
    <property name="username"><value>root</value></property>
    <property name="password"><value>root</value></property>
  </bean>

<!-- Hibernate SessionFactory -->
  <bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource"><ref local="myDataSource" /></property>
<!-- Must references all OR mapping files. -->
    <property name="mappingResources">
        <list>
         <value>paquete/modelo/Usuario.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
        <prop key="hibernate.connection.pool_size">1</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
        <prop key="hibernate.show_sql">false</prop>
      </props>
    </property>
  </bean>
<!-- AOP , definimos nuestro interceptor -->
  <bean id="interceptor" class="paquete.aop.Interceptor" />
</beans>
```

5.4 Ejecución.

Volvemos a ejecutar nuestra clase de Test, si comprobamos nuestra salida de trazas, apreciamos que se están realizando las llamadas a nuestro interceptor, mostrando las líneas de trazas añadidas para mostrar esto.

6 Tercer ejemplo.

6.1 Transacciones.

Siguiendo con los ejemplos de AOP vamos a añadir a nuestra aplicación control de transacciones, de forma que podamos controlar, en caso de excepción, un "rollback" de la transacción realizada.

Tradicionalmente en las transacciones es necesario un método tx.begin(), tx.commit() y tx.rollback() (o nombres similares), la idea es forzar las transacciones con AOP sin necesidad de insertar estas llamadas en el código, controlando así cuando realizar transacciones y cuando no.

6.2 Programando clases sin trasnsacciones

Para poder apreciar el efecto de las transacciones vamos a modificar nuestra aplicación para que realice un proceso de inserción masiva y así verificar el efecto del proceso de transacciones.

Añadimos a nuestro interfaz un método nuevo:

```
public interface UsuarioDao {

    public void saveUsuario (Usuario usuario);
    public Usuario findUsuario (Integer id);
    public void deleteUsuario (Integer id);
    public List listAll ();

    public void saveListaUsuarios (Usuario[] usuario);

}
```

Este método pretende realizar el proceso de inserción masiva de usuarios en la Base de datos. Al modificar el interface debemos modificar la clase de Hibernate que la implementa. Este método será:

```
    public void saveListaUsuarios (Usuario[] usuario) {

        this.logger.debug("Guardamos todos los usuarios");
        HibernateTemplate temp = getHibernateTemplate();
        for (int i = 0; i < usuario.length; i++) {

            this.logger.debug("Guardando "+usuario[i]+ " i : "+i);
            temp.saveOrUpdate(usuario[i]);
        }

    }
```

Como se puede apreciar el proceso guarda cada uno de los usuarios a añadir. Hibernate lo tenemos configurado de forma que debemos de asignarle nosotros a

mano el ID del usuario, por lo que un campo null en este valor provocará una excepción.

Para apreciar este efecto vamos a modificar nuestra clase de TEST añadiendo el siguiente método de test:

```
    /**
     * Realiza el test de inserción de datos malos que provoquen un efecto
     * de rollback en las transacciones.
     * @throws Exception excepción generada.
     */
    public void testTransaccion () throws Exception
    {
        log.debug("Iniciamos guardar todos los usuarios:");
        Usuario[] usuarios = this.ListadoUsuariosMalos ();
        log.debug (usuarios);
        try {
        dao.saveListaUsuarios(usuarios);

        } catch (Exception e)
        {
            log.error (e);
            List listado = dao.listAll();
            log.debug ("La lista debe de estar vacia");
            assertTrue(listado.isEmpty());
        }

    }

    /**
     * @return Un listado de usuarios con algunos datos malos
     *
     */
    private Usuario[] ListadoUsuariosMalos () {

        Usuario usu1 = new Usuario ();
        usu1.setId(new Integer(100));
        usu1.setNombre("Nombre uno");
        Usuario usu2 = new Usuario ();
        usu2.setId(new Integer (101));
        usu2.setNombre("Nombre 2");
    // Es es un usuario no valido
        Usuario usu3 = new Usuario ();
        usu3.setId(null);
        usu3.setNombre("Usuario no valido");

        Usuario usu4 = new Usuario ();
        Usu4.setId(new Integer (103));
        Usu4.setNombre("Nombre 4");

        Usuario[] usuarios = { usu1,usu2,usu3};
        return usuarios;
    }
```

Si ahora lanzamos el test observaremos que termina con error (fail) debido a una excepción de hibernate:

```
org.springframework.orm.hibernate.HibernateSystemException: ids for this class must be manually assigned before calling save(): paquete.modelo.Usuario;
```

Es decir, al ejecutar la inserción del tercer usuario, al no tener este un valor de id asignado, nos provoca un error que provoca la parada del proceso de inserción. Este es un caso típico que debemos de iniciar una transacción, debido a que, si consultamos la base de datos obtenemos que:

ID	Nombre
100	Nombre uno
101	Nombre 2

Es decir, que nos ha insertado los dos primeros datos. En algunos casos esto no es deseable, es por lo que deberiamos de iniciar un proceso de transacción y hacer un rollback () en caso de excepción.

6.3 Modificando Configuración

Para configurar nuestra aplicación de ejemplo de forma que use transacciones, debemos de configurar nuestro fichero de *Spring* para que use transacciones. El proceso de las transacciones esta relacionado con el AOP. Asi que básicamente la modificación consiste en sustituir nuestra clase que hacia de proxy de usario bean por otro Proxy que proporciona herramientas de transacciones, ademas de configurar los administradores de transacciones necesarios, para ello el cambio realizado es:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

<!-- Aqui configuramos hibernate -->
<!-- Conexión a base de datos usando un datasource de apacheCommons -->
  <bean id="myDataSource"
  class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"><value>org.gjt.mm.mysql.Driver</value></property>
    <property name="url"><value>jdbc:mysql://localhost/atril</value></property>
    <property name="username"><value>root</value></property>
    <property name="password"><value>root</value></property>
  </bean>

<!-- Hibernate SessionFactory -->
  <bean id="sessionFactory"
  class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource"><ref local="myDataSource" /></property>

<!-- Must references all OR mapping files. -->
    <property name="mappingResources">
      <list>
        <value>paquete/modelo/Usuario.hbm.xml</value>
      </list>
    </property>

    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
        <prop key="hibernate.connection.pool_size">1</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
        <prop key="hibernate.show_sql">false</prop>
      </props>
    </property>
  </bean>
```

```
<!-- Configuracion del interfaz y la implementacion -->
       <bean id="usuarioDaoTarget" class="paquete.dao.hibernate.UsuarioDaoHibernate">
              <property name="sessionFactory"><ref local="sessionFactory"/></property>
       </bean>

    <bean id="transactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
       <property name="sessionFactory"><ref bean="sessionFactory"/></property>
    </bean>

    <bean id="usuarioDao"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
              <property name="transactionManager"><ref local="transactionManager"/></property>
              <property name="target"><ref local="usuarioDaoTarget"/></property>
              <property name="transactionAttributes">
                     <!-- define the transaction specs here -->
                     <props>
                            <prop key="save*">PROPAGATION_REQUIRED</prop>
                            <prop key="delete*">PROPAGATION_REQUIRED</prop>

                     </props>
              </property>
              <property name="preInterceptors"><ref local="interceptor" /></property>

       </bean>



<!-- AOP , definimos nuestro interceptor -->
  <bean id="interceptor" class="paquete.aop.Interceptor" />

</beans>
```

6.4 Ejecución.

Si lanzamos en este caso la ejecución del test y verificamos el resultado final de la base de datos, esta debe de estar vacia de forma que la transacción se ha realizado correctamente.

6.5 Conclusiones

PENDIENTE

7 *Spring* MVC

7.1 ¿Qué es MVC?

PENDIENTE

7.2 Configurando Tomcat

PENDIENTE

7.3 Estructura directorios

PENDIENTE

7.4 Configurando Aplicación web.

PENDIENTE

7.5 Programando clases.

PENDIENTE

7.6 Modificando configuración

PENDIENTE

7.7 Ejecutando.

PENDIENTE

7.8 Conclusiones.

PENDIENTE

8 Conclusiones

8.1 Ventajas

8.2 Inconvenientes