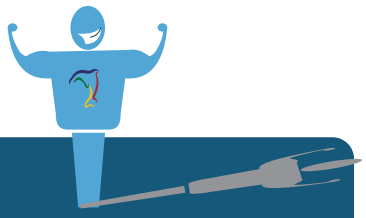
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

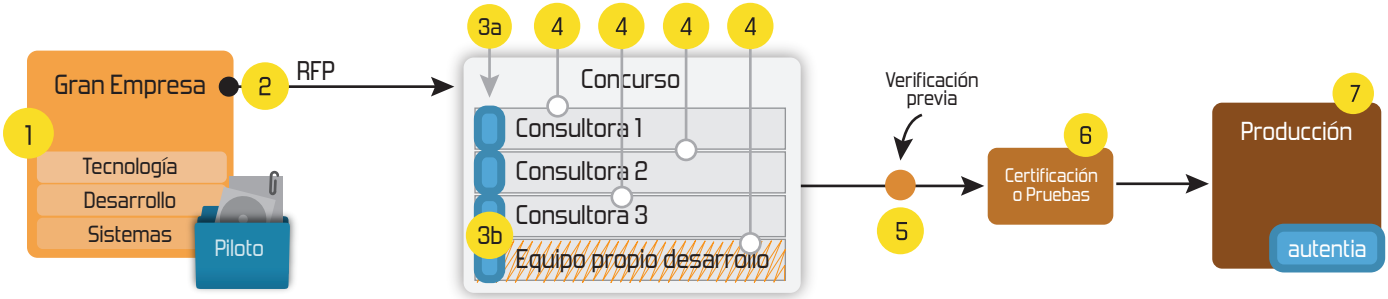
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

DESARROLLADO POR:
Alejandro Pérez García

Alejandro es socio fundador de Autentia y nuestro experto en J2EE, Linux y optimización de aplicaciones empresariales.

Ingeniero en Informática y Certified ScrumMaster

Si te gusta lo que ves, puedes contratarle para darte ayuda con soporte experto, impartir **cursos presenciales** en tu empresa o para que **realicemos tus proyectos como factoría** (Madrid). Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Fecha de publicación del tutorial: 2011-01-05

161

Share | Regístrate para votar

Spring @Configurable y los modelos de dominio anémicos

Creación: 27-12-2010

Índice de contenidos

1. Introducción

Recientemente, en el AOS 2010 en Barcelona, participé en la charla de "Los frameworks son Evil!!!". En esta charla había dos bandos "enfrentados", donde uno defendía los frameworks (Roberto Canales y yo) y el otro bando decía que los frameworks son Evil (Xavi Gost y Enrique Comba). La charla la verdad es que estuvo bastante bien porque la conclusión que se sacó al final es que las cosas hay que saber usarlas o sino ciertamente se convierten en Evil; pero esto pasa no sólo con los frameworks, sino que pasa igual con cualquier librería, herramienta de desarrollo, lenguaje, etc. De hecho al final de la charla Xavi y yo cambiamos cada uno de bando, dándole así la razón al otro (la potencia sin control no sirve de nada).

A raíz de esa charla me he animado a escribir este pequeño tutorial donde vamos a mostrar como luchar contra los modelos anémicos usando Spring e Hibernate.

En primer lugar me gustaría definir un poco que se entiende por "Modelo de Dominio anémico".

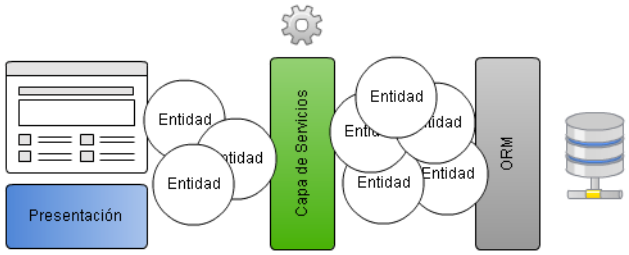
El **Modelo de Dominio** es aquél que define el negocio que estamos resolviendo. Al final podríamos decir que el Modelo de Dominio viene definido por las clases que implementan los conceptos del negocio que estamos resolviendo mediante un programa informático. Por esto también se le denomina en muchas ocasiones "Dominio del Problema", haciendo así referencia al problema que estamos resolviendo (modelando mediante clases).

Un punto a tener en cuenta del Modelo de Dominio es que lo creamos para representar el vocabulario y los conceptos del dominio del problema. Es decir, debe estar escrito en el lenguaje del cliente, representando así lo conceptos que este maneja (un Modelo de Dominio bien hecho debería ser legible para el cliente, aunque este no sea técnico, ya que está escrito en su lenguaje).

El **Modelo de Dominio anémico** es aquel que está especialmente "flaco" (de ahí lo de anémico;) Esto ocurre cuando nuestras clases del dominio son meros contenedores de información y sólo tienen un puñado de atributos con getters y setters (podéis encontrar varios artículos sobre por qué los getter y setters son Evil, por ejemplo: http://www.javaworld.com/javaworld/jw-09-2003/jw-0905-toolbox.html). Pero en este caso no podemos hablar realmente de clases, más bien se trata de simples estructuras de datos.

Por esto decimos que los modelos son anémicos, que están especialmente "flacos"; ya que no tienen ninguna lógica de negocio y se limitan a contener información (nos saltamos el patrón experto, ya que la lógica que manipula los datos no está en la misma clase que los contiene, y además con esto estamos creando clases donde no se cumple el principio de alta cohesión y bajo acomplamiento).

Spring y los Modelos de Dominio anémicos. Con Spring podríamos decir que es relativamente fácil caer en la tentación de construir Modelos de Dominio anémicos. Esto es debido a un tipo de diseño que se ha vuelto bastante típico cuando se usa un ORM (Hibernate, JPA, ...) + Spring + presentación (tipo Struts, JSF, JSP, ...). Ponemos un pequeño esquema:



Es muy habitual que tengamos una serie de Entidades (objetos persistentes que obtenemos gracias a algún ORM como Hibernate o JPA), que al final no son más que un reflejo de las tablas de la base de datos. Estas entidades las manipulamos con una capa de servicios que gestionamos con Spring, y las acabamos pintando con alguna tecnología de presentación (como puede ser Struts o JSF o JSP).

En este diseño no existen objetos de dominio, y la orientación a objetos escasea, ya que lo que se hace es manipular unas estructuras de datos (las entidades), con un conjunto de procedimientos (los servicios). Además las tecnologías de presentación ayudan a la degradación de este modelo ya que nos fuerzan a que tengamos getters y setters para acceder a todo lo que se quiere pintar.

Se tiende a este tipo de modelos porque el ciclo de vida de las entidades no lo gestionamos nosotros, sino el ORM, por lo que en principio parece difícil inyectar dependencias para convertirlos en clases del modelo de dominio. De esta manera se acaba con un conjunto de entidades cuyo ciclo de vida gestiona el ORM y un conjunto de servicios cuyo ciclo de vida gestiona el contenedor de inyección de dependencias (Spring).

Se suele caer en este diseño porque es cómodo (y nosotros vagos), pero los inconvenientes son muchos. El más destacado podría ser la dificultad del mantenimiento, ya que hay mucho acoplamiento entre las piezas del sistema. Otro gran problema es que no estamos haciendo realmente orientación a objetos, por lo que nos perdemos toda la potencia que ofrecen este tipo de lenguajes.

Algunas formas de evitar este tipo de diseños:

- No hay porque arrastrar las Entidades por todas las capas, es decir, una Entidad no es más que un objeto persistente, es decir que se guarda en algún soporte, como una base de datos, para que si apagamos la máquina y la volvemos a encender, este objeto perdure. Pero los objetos del dominio no tienen porque ser estas entidades, es decir no tienen porque ser el mismo objeto, ya que una cosa es como modelamos nuestro negocio y otra diferente es como guardamos los datos.
- Otra forma de evitar este tipo de diseño es usar correctamente las herramientas que tenemos a nuestra disposición. En este tutorial vamos a ver como podemos inyectar dependencias en objetos cuyo ciclo de vida no es gestionado por Spring. Esto nos ayuda a romper con este diseño "procedural", ya que vamos a poder poner en los objetos de dominio toda la lógica que sea necesaria, y dará igual si estos objetos de dominio los creamos con 'new' o los crea el ORM.

El código del ejemplo lo podéis descargar aquí.

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 17' (2.93 GHz Intel Core 2 Duo, 4GB DDR3 SDRAM, 128GB Solid State Drive).
- NVIDIA GeForce 9400M + 9600M GT with 512MB
- Sistema Operativo: Mac OS X Snow Leopard 10.6.5
- JDK 1.6.0_22
- Maven 3
- Spring 3.0.5.RELEASE
- Hibernate 3.5.1-Final
- AspectJ 1.6.10

3. Configuración

En el pom.xml hemos puesto las dependencias necesarias para usar los jar de Spring, de Hibernate y de AspectJ.

- Utilizaremos Spring como contenedor de inyección de dependencias,
- usaremos Hibernate como ORM para guardar nuestros objetos en una base de datos,
- y AspectJ para hacer la inyección en los objetos de dominio que vamos a crear con new o que va a crear Hibernate. Es decir la inyección la vamos a hacer mediante AOP (Programación Orientada a Aspectos), y en concreto mediante la librería AspectJ que modifica el bytecode de nuestras clases para conseguir la inyección. Esta es la única forma de hacerlo, ya que como Spring no gestiona el ciclo de vida de estos objetos, no puede intervenir para hacer él la inyección.

Aquí vamos a ver la configuración de Maven para lanzar AspectJ. Vamos a ver dos métodos, y tendremos que elegir uno de ellos (yo por ahora he preferido usar el weaving en tiempo de compilación):

3.1. Weaving en compilación

El weaving es el proceso mediante el cual se "retoca" el bytecode de la clase para hacer la inyección de las dependencias.

En este caso lo vamos a hacer en tiempo de compilación, de forma que las modificaciones estarán dentro de nuestro .class. Para ello modificamos el **pom.xml** para indicarle que cuando compile nuestras clases o nuestras clases de test, inmediatamente después use el compilador de AspectJ para modificar el bytecode.

```
...
<plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>aspectj-maven-plugin</artifactId>
        <configuration>
                <source>${compileSource}</source>
                <target>${compileSource}</target>
                <aspectLibraries>
                        <aspectLibrary>
                                <groupId>org.springframework</groupId>
                                <artifactId>spring-aspects</artifactId>
                        </aspectLibrary>
                </aspectLibraries>
        </configuration>
        <executions>
                <execution>
                        <goals>
                                <goal>compile</goal>
                                <goal>test-compile</goal>
                        </goals>
                </execution>
        </executions>
</plugin>
...
```

Ahora en el **applicationContext.xml** de Spring tendremos que poner:

```
...
<context:annotation-config />

<context:spring-configured />
<aop:aspectj-autoproxy />

<tx:annotation-driven />
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />

<context:component-scan base-package="com.autentia.tutorial" />
...
```

- En la línea 02 indicamos que se van a usar anotaciones.
- En la lína 04 y 05 son la que realmente le dicen a Spring que vamos a usar al anotación **@Configurable** para inyectar dependencias en objetos no gestionados por Spring, y que vamos a usar AspectJ.
- Las líneas 08 y 09 son para indicar que vamos a usar transacciones con anotaciones y que queremos transformar las excepciones de persistencia de Hibernate en excepciones genéricas de Spring.
- Con la línea 10 le estamos indicando el paquete a partir del cual queremos escanear las clases para ver si tienen anotaciones. El escaneo es recursivo, de forma que también ser revisan todos los subpaquetes.

3.2. Weaving en tiempo de carga de la clase

En este caso el bytecode se modifica en runtime al cargar la clase en memoria. El bytecode del .class no se modifica, por lo que hay que hacerlo cada vez que se arranca la aplicación.

En el **pom.xml** ya no declaramos `aspectj-maven-plugin`, pero lo que si vamos a hacer es modificar el plugin que lanza los test (el Surefire) para que también se haga el weaving al ejecutar los test.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <argLine>-javaagent:${user.home}/.m2/repository/org/springframework/spring-
instrument/${spring.version}/spring-instrument-${spring.version}.jar</argLine>
    </configuration>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
```

```
            <artifactId>spring-instrument</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Se puede ver como estamos indicando un **javaagent**, esto siempre es necesario porque lo que vamos a hacer "modificar" el classloader de la JVM para que al cargar las clases se haga el weaving. Si utilizamos un servidor de aplicaciones, cada uno suele tener su propia configuración (revisar la documentación de Spring para ver como es), en el caso del Jetty no hay configuración propia, así que usaremos este mismo javaagent modificando el script de arranque del Jetty.

Ahora en el **applicationContext.xml** tendremos:

```
<context:annotation-config />

<context:spring-configured />
<context:load-time-weaver />
<aop:aspectj-autoproxy />

<tx:annotation-driven />
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />

<context:component-scan base-package="com.autentia.tutorial" />
```

Vemos que todo es igual salvo que hemos añadido la línea 04 para indicar a Spring que queremos hacer el weaving en tiempo de carga de las clases.

El único inconveniente de hacer el weaving en tiempo de carga es que si por algún motivo la clase se carga antes de que esté levantado el contexto de Spring, ya no se podrá hacer la inyección en esta clase ya que AspectJ no podrá hacer el weaving.

4. Nuestros objetos de dominio

He preparado un simple objeto de dominio que además lo vamos a guardar en la base de datos.

```
@Entity
@Configurable(preConstruction = true)
class DomainObject {

    @Id
    @GeneratedValue
    private Long id;

    @Inject
    @Transient
    private ServiceToInject injectedDependendy;

    private final String name;

    DomainObject(String name) {
        System.out.println("Creando una instancia de " + getClass().getSimpleName() + " (" + injectedDependendy +
")");
        this.name = name;
    }

    Long getId() {
        return id;
    }

    Object getInjectedDependency() {
        return injectedDependendy;
    }

    String getName() {
        return name;
    }
}
```

Lo importante es destacar en la linea 02 como indicamos que el objeto es **@Configrable**, es decir, que Spring no gestiona su ciclo de vida, y que le vamos a hacer el weaving para inyectarle las dependencias. Con el atributo `preConstruction = true`, le estamos indicando que queremos hacer la inyección antes de que se llame al constructor, de esta forma podemos usar esos atributos en el propio constructor. Esto no suele ser necesario, pero aquí os lo he querido enseñar para que veáis como en el mensaje que sacamos por consola en el constructor el valor ya está inicializado.

En la lína 09 estamos usando la anotación **@Inject** como suele ser habitual para hacer una inyección cualquiera. Pero en este caso la clase no esta marcada como un `@Component`, y el ciclo de vida no está gestionado por Spring.

En la lína 10 usamos **@Transient** para indicar a Hibernate que no tiene que intentar persistir este atributo. No tendría sentido siquiera intentarlo puesto que este atributo es precisamente el que vamos a inyectar.

Ahora vamos a ver la clase que estamos inyectando.

```
@Service
class ServiceToInject {

    ServiceToInject() {
        System.out.println("Creando una instancia de " + getClass().getSimpleName());
    }
}
```

Podéis ver que esta clase no tiene nada de especial. Es un típico bean de Spring (Spring gestiona su ciclo de vida).

5. El test

También he creado un pequeño test para comprobar que todo funciona correctamente.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration({ "classpath:applicationContext.xml", "classpath:applicationContext-infrastructure-
test.xml" })
public class DomainObjectTest {

    @Inject
    private DomainObjectRepository domainObjectRepository;

    private Long persistedDomainObjectId;

    @Inject
    private ServiceToInject serviceToInject;

    @Test
    @Transactional
    public void a_dependency_is_injected_to_my_domain_objects_when_find_from_hibernate() {
        insertDataInTheDatabase();

        List<DomainObject> domainObjects = domainObjectRepository.findByExact("Autentia");
        assertEquals(3, domainObjects.size());
        for (DomainObject domainObject : domainObjects) {
            assertInjectedDependencyIn(domainObject);
        }
    }

    @Test
    @Transactional
    public void a_dependency_is_injected_to_my_domain_objects_when_get_from_hibernate() {
        insertDataInTheDatabase();

        assertInjectedDependencyIn(domainObjectRepository.getBy(persistedDomainObjectId));
```

```
    }

    @Test
    @Transactional
    public void a_dependency_is_injected_to_my_domain_objects_when_load_from_hibernate() {
        insertDataInTheDatabase();

        assertInjectedDependencyIn(domainObjectRepository.loadBy(persistedDomainObjectId));
    }

    @Test
    public void a_dependency_is_injected_to_my_domain_objects_when_new_is_done() {
        DomainObject domainObject = new DomainObject("Autentia");
        assertInjectedDependencyIn(domainObject);
    }

    private void assertInjectedDependencyIn(DomainObject domainObject) {
        assertNotNull(domainObject);

        final Object injectedDependency = domainObject.getInjectedDependency();

        assertNotNull(injectedDependency);
        assertSame(serviceToInject, injectedDependency);
    }

    private void insertDataInTheDatabase() {
        for (int i = 0;  i < 3; i++) {
            final DomainObject domainObject = new DomainObject("Autentia");
            domainObjectRepository.save(domainObject);
            persistedDomainObjectId = domainObject.getId();
        }
    }
}
```

Se puede ver como tenemos cuatro métodos de test, donde en cada uno de ellos conseguimos el objeto de dominio de una forma diferente (o con un new, o distintas formas de recuperar el objeto de la base de datos con Hibernate) y se comprueba que la dependencia se ha inyectado correctamente.

6. Conclusiones

Hemos visto como con este método podemos inyectar dependencias en objetos no gestionados por Spring. Mi recomendación sería usar esto lo menos posible, es decir:

- Si ciclo de vida lo gestionamos nosotros, deberíamos inyectarle nosotros las dependencia, preferentemente a través del constructor o sino mediante setters.
- Si el ciclo de vida lo gestiona Spring, deberíamos inyectarle las dependencias con los mecanismos normales de Spring (preferente a través del constructor).
- Si el ciclo de vida no lo gestionamos ni nosotros ni Spring, por ejemplo porque son objetos que recuperamos de la base de datos con Hibernate, entonces y sólo entonces es cuando tiene sentido usar este mecanismo.

Además recordar que los frameworks no son realmente Evil, no son ni buenos ni malos, sólo son herramientas que si las usamos mal si se pueden convertir en Evil, pero bien usadas nos pueden simplificar mucho el trabajo.

Esto mismo nos pasa con todo, por ejemplo si no usamos ningún framework ni librería y lo hacemos todo a mano, podemos cometer los mismos errores y hacer igualmente modelos anémicos.

Por eso lo realmente importante es saber de TDD, de refactorización, de patrones, de anitpatrones, leer libros como el Clean Code, practicar, practicar, practicar, ...

7. Sobre el autor

Alejandro Pérez García, Ingeniero en Informática (especialidad de Ingeniería del Software) y Certified ScrumMaster

Socio fundador de Autentia (Formación, Consultoría, Desarrollo de sistemas transaccionales)

mailto:alejandropg@autentia.com

Autentia Real Business Solutions S.L. - "Soporte a Desarrollo"

http://www.autentia.com

COMENTARIOS