

# Spring y el principio de Hollywood

Mayo de 2005

@author: Jorge Rodríguez

En mi artículo anterior “To EJB or not to EJB?”<sup>1</sup>, se menciona al framework “*Spring*”<sup>2</sup> como solución substituta a un contenedor EJB en aplicaciones J2EE. A pesar que dicho framework abarca muchos aspectos que pueden ser aplicados en el desarrollo de una aplicación empresarial, el presente artículo tiene como objetivo introducir el principio en el que se basa Spring, que constituye su núcleo y lo hace útil en diversos escenarios de desarrollo en java. Éste es el primero de una serie donde iremos mostrando las diferentes características del framework y sus aplicaciones.

## Un poco sobre Spring.

Spring nació del código que acompaña el best seller “Expert One-on-One J2EE Design and Development (Wrox, 2002)”, escrito por Rod Johnson, quien desde los inicios de la plataforma J2EE, es un crítico activo de los EJBs, y que basa sus artículos y libros en la experiencia práctica del desarrollo de aplicaciones empresariales. A pesar de que Spring ha evolucionado mucho desde la publicación del libro, este continua siendo un buen lugar para entender la motivación que hay detrás del framework.

Spring es un framework java, ligero, cuyo concepto principal es implementar el patrón *Inversión del Control*, también conocido como *Inyección de Dependencias*. Spring, como veremos nos permite disfrutar de servicios de middleware que de otra forma estaríamos obligados a usar mediante contenedores EJB.

Spring nos permite diseñar y escribir código en capas, abstrayendonos de conceptos cruzados como la búsqueda de componentes, el manejo de transacciones, la seguridad, el uso directo de sentencias SQL, el manejo de excepciones, y otros.

## El principio de Hollywood o Inversión del Control

El principio de Hollywood: “No me llames, yo te llamo” es como se le conoce en algunas literaturas al patrón de diseño “*Inversión del Control*”<sup>3</sup>, patrón que describe la forma en que un objeto resuelve sus dependencias con otros objetos. La idea se basa en que dado un objeto, “algo”<sup>4</sup> resuelva sus dependencias, inyectandose ya sea a través de métodos setXXX o de uno de los constructores del objeto; para que esto sea posible, ese “algo” debe manejar el ciclo de vida del objeto. Este patrón puede parecer confuso, incluso parecer que va en contra de paradigmas de la programación orientada a objetos como la “encapsulación”<sup>5</sup>, sin embargo, constituye una poderosa filosofía de trabajo.

Para hacer la descripción mas concreta, veamos el principio a través de un pequeño ejemplo ireal, pero suficiente para visualizar lo que ocurre en escenarios reales.

---

1 Ediciones Diciembre - Febrero 2004, Marzo – Abril 2005, Revista Informática.

2 <http://www.springframework.org>

3 Martin Fowler (<http://martinfowler.com/articles/injection.html>)

4 Concretamente un contenedor de inversión de control como Spring o PicoContainer.

5 Tema de discusión para otro artículo.

En la figura 1, se muestra una parte de un sistema bancario, donde la clase Banco, usa la interface ManagerCuentas que le provee de servicios cuya implementación depende del tipo de cuenta, en este caso contamos con dos implementaciones: ManagerCuentasVista y ManagerCuentaCredito, aquí para simplificar y ahorrar espacio hemos dibujado solo ManagerCuentasCredito.

De esta forma, la clase Banco necesita en algún momento resolver cual es la implementación del servicio ManagerCuentas, para lo cual, nuestro sistema ejemplo implementa el patrón de diseño *Service Locator*, dejando que la clase ServiceLocator se encargue de buscar y crear el objeto implementación del servicio(vista o crédito).

Hasta aquí todo bien, pero ahora imaginemos que nuestro desarrollo está terminado y nos piden extender el sistema para además ahora usar un manager para cuentas de tipo ahorro. Aquí nos encontramos un problema, porque a pesar de que la clase Banco no tiene relación alguna con la nueva clase ManagerCuentasAhorro, no obstante necesita decirle a la clase ServiceLocator que servicio usar, o sea que, en nuestro sistema solo se ha movido la responsabilidad de la búsqueda de los servicios ManagerCuentas a otra clase de nuestro modelo, en este caso a ServicesLocator. Por tanto para extender este sistema sistema necesitamos modificar el código, limitando así las posibilidades de extensión de nuestra aplicación.

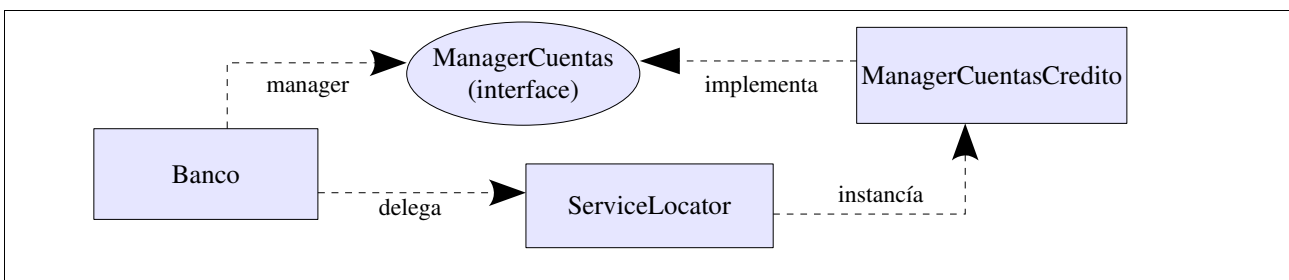


Figura 1. Clase Banco delega la búsqueda del manager de cuentas al ServiceLocator

Como resolver este dilema?, pues bien, si miramos ahora la figura 2, vemos el mismo sistema, pero ahora usando un contenedor de *Inversión del Control*, en este caso Spring. Ahora la clase Banco no necesita saber que manager de cuentas va a usar, es responsabilidad del instalador de la aplicación, a través de un xml, configurar la búsqueda o resolución de componentes de servicio en el sistema. De esta forma pueden aparecer nuevos managers de cuentas y nuestro sistema quedar funcional sin modificación alguna, incluso podemos cambiar la localización de los managers de cuentas sin sufrir cambio alguno el sistema. En nuestro ejemplo, bastaría con colocar a ManagerCuentasAhorro en el archivo de configuración y quedaríamos listo para cumplir el nuevo requerimiento.

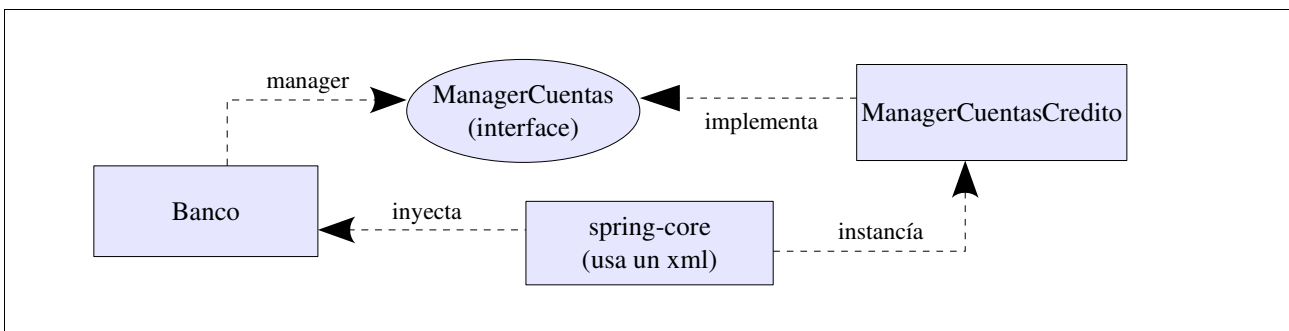


Figura 2. Alguien le dice al contenedor en un xml que manager de cuentas usar.

## Beneficios del Principio

El principio antes expuesto tiene varios beneficios entre los que podemos destacar:

- Quita la responsabilidad de buscar o crear objetos dependientes, dejando estos configurables. De esta forma búsquedas de componentes complejas como es el uso de JNDI pueden ser delegadas al contenedor.
- Reduce a cero las dependencias entre implementaciones, favoreciendo el uso de diseño de modelos de objetos basados en interfaces.
- Permite a nuestra(s) aplicación(es) ser re-configurada(s) sin tocar el código.
- Estimula la escritura de componentes testables<sup>6</sup>, pues la *Inversión del Control* facilita el uso de pruebas unitarias.

## Spring como actor

Pero como Spring nos maneja el código?. Pues bien, el núcleo de este framework lo constituye su principal concepto o interface BeanFactory<sup>7</sup>, que no es otra cosa que una fábrica de objetos JavaBeans que puede ser configurada a través de un archivo xml y creada en una única sentencia o línea de código<sup>8</sup>. Es a través de este concepto que Spring implementa la *Inversión del Control*.

Volviendo al esquema de la figura 2, es posible decirle a Spring que al crear una instancia de la clase Banco, no se olvide que para este caso necesitamos que el manager de cuentas sea un objeto de tipo ManagerCuentasAhorro. Esto es posible a través de un archivo de configuración<sup>9</sup> que contenga algo como la figura 3.

```
<bean id="banco" class="cl.banco.sistema.logic.Banco">
  <property name="manager"><ref local="manager"/></property>
</bean>

<bean id="manager" class="cl.banco.sistema.manager.ManagerCuentasAhorro"/>
```

Figura 3. Descriptor de Spring, donde declaramos los objetos(beans) y sus dependencias.

De esta forma, Spring es capaz de:

- 1.- Instanciar nuestra clase Banco con nombre banco.
- 2.- Crear una instancia de ManagerCuentasAhorro con nombre manager
- 3.- Y por último inyectar manager a nuestra clase banco<sup>10</sup>

Usando esta funcionalidad de Spring seremos capaces ahora de separar los conceptos de “búsqueda” y “uso” de los servicios, potenciando la extensibilidad y el buen diseño de nuestras aplicaciones.

Un beneficio no menos grande gracias a la separación de conceptos de la que hablabamos en el párrafo anterior es relacionado al desarrollo. Spring nos permite separar casi en su totalidad los roles de los programadores. Volviendo a nuestro sistema de ejemplo, podemos tener un equipo trabajando en la clase Banco, mientras otro lo hace en las clases ManagerCuentasVista, ManagerCuentasCredito y ManagerCuentasAhorro, sin embargo como ambos equipos los separa un contrato bien definido(interface ManagerCuentas), el equipo que desarrolla la cuenta Banco no tiene porque esperar que esten listas las clases managers de cuentas para probar su programación, pues

6 del término Test(Prueba).

7 Fábrica de beans.

8 En una aplicación web, puede incluso configurarse en el web.xml

9 Pueden ser varios, incluso varias versiones del mismo para diferentes escenarios (Desarrollo, Testing, Producción)

10 Puede ser a través de un método setManager o a través del constructor de la clase (para esto habría que configurar)

ellos pueden implementar una clase prototipo, digamos `ManagerCuentasProt` configurandola en el archivo xml de Spring. Luego de terminado el desarrollo del sistema basta con cambiar unas lineas del xml para cambiar a una implementación real del `ManagerCuentas`.

## Otras funcionalidades

Pero Spring ofrece además:

- Una capa de abstracción para el manejo de transacciones, permitiendo enchufar varias soluciones.
- Integración con varios frameworks open sources como: Hibernate, JDO, iBatis SQL Map, JMS, JNDI, RMI, JAX-RPC, Axis, Struts, WebWork, Tapestry, etc.
- Un framework MVC capaz de integrar multiples tecnologías de despliege como JSP, Velocity, XSLT.
- Una implementación AOP. Aunque es posible el uso de AOP de forma abstracta sin ni siquiera entender sus conceptos mediante Spring.

y muchas otras funcionalidades<sup>11</sup>, eso si, todas basadas en el simple principio que hemos visto aquí.

## Conclusiones

Spring es un framework open source que resuelve muchos de los problemas que nos encontramos a diario en J2EE. Provee una forma consistente de manejar nuestros objetos y nos motiva a usar buenas prácticas como el uso de interfaces en los modelos.

La base de la arquitectura es el concepto de *Inversion del Control*, o también conocido como el *Principio de Hollywood*, que nos permite usar a Spring como configurador de nuestros objetos, delegando al mismo la resolución de las dependencias del grafo de objetos, y permitiendonos configurar nuestras aplicaciones mediante el uso de archivos xml simples, que incluso pueden cambiar en dependencia del ambiente o la etapa del desarrollo en que nos encontremos.

Spring posee además otras funcionalidades como son un framework MVC, una implementación AOP, y una capa de manejo de transacciones, pero todas basadas en su principio hollywodiense.

Todo esto y mucho más hacen de Spring un candidato fuerte a la hora de escoger una tecnología para desarrollar nuestras aplicaciones empresariales. Puede saberse más en su sitio de internet "<http://www.springframework.org>", donde se incluyen los javadocs, y algunos tutoriales.

### Acerca del autor:

Jorge Rodriguez es Sun Certified Programmer for the Java 2 Platform 1.4 y Sun Certified Web Component Developer for the Java 2 Platform Enterprise Edition, además cuenta con una certificación en WebLogic Workshop 8.1. Trabaja como Arquitecto J2EE para Interplanet S.A y Xperience Consulting Services y puede ser contactado en [jrodriguez@interplanet.cl](mailto:jrodriguez@interplanet.cl).

---

<sup>11</sup> To EJB or not To EJB?