



# **Curso Spring - RichFaces - Introducción**

*Abril 2011 – JoeDayz  
Susan Inga*

02/02/2011

El taller Core Spring 3.0 de JoeDayz es  
licenciado bajo Creative Commons  
Attribution-Noncommercial-No Derivative  
Works 3.0 United States License

# Contenido

- ✓ Introducción
- ✓ ¿Qué es Spring?
- ✓ Módulos de Spring
- ✓ El taller
- ✓ Introducción a Conceptos
- ✓ ¿IoC o DI?
- ✓ Demo 1: Hello World
- ✓ Demo 2: Caballeros
- ✓ Spring AOP
- ✓ Demo 3: Caballero + Trovador



# ***Spring Framework***

*Simplificando JEE*

# Introducción

- ✓ Gran parte de los desarrolladores Java desarrollan para Web.
- ✓ Retos:
  - Gestión de estados
  - Flujo de trabajo
  - Validaciones
- ✓ Spring es la solución de Spring para el desarrollo Web.
- ✓ Resultado: aplicaciones flexibles, altamente cohesivas y débilmente acopladas.

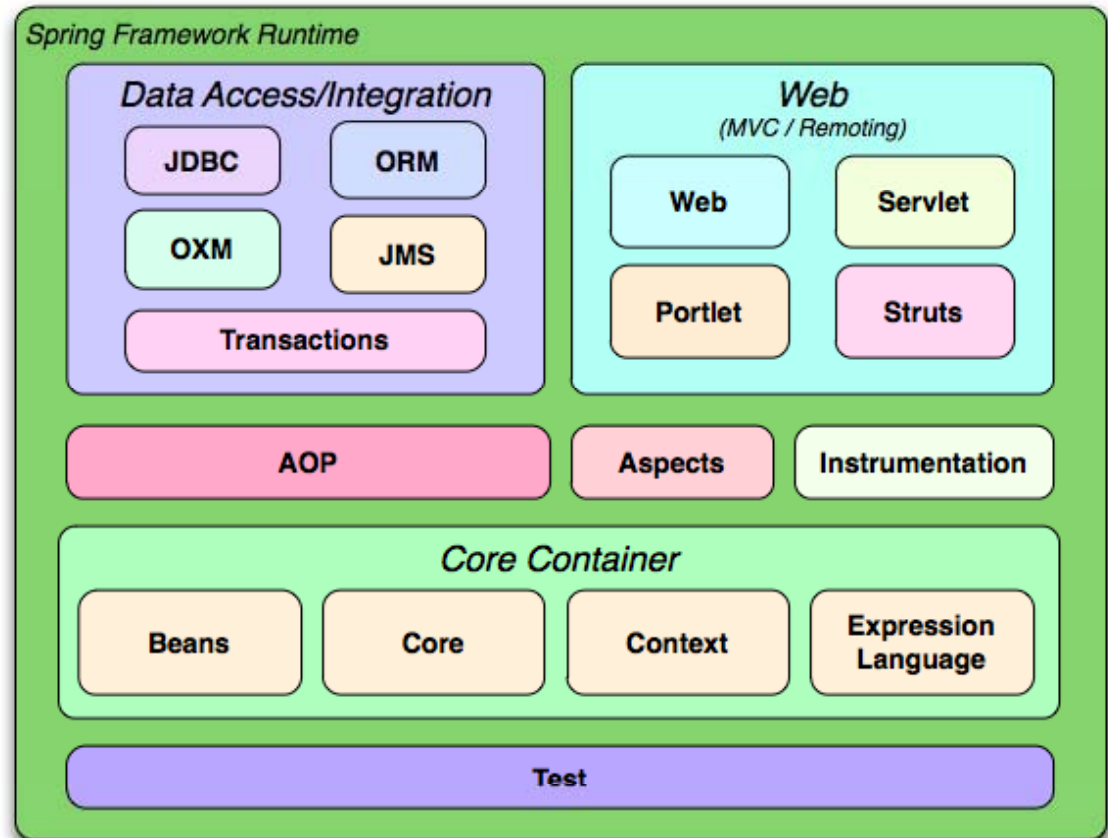


# ¿Qué es Spring?

- ✓ Spring es un framework open source creado por Rod Johnson y descrito en su libro [Expert One-on-One: J2EE Design and Development](#). Fue creado para solucionar la complejidad en el desarrollo de aplicaciones empresariales.
- ✓ Spring hace posible el uso de simples JavaBeans para conseguir cosas que antes era sólo posible con los EJBs.
  - Para evitar caer en ambigüedad. Usaremos "bean" para referirnos a JavaBeans simples y EJB para referirnos a Enterprise JavaBeans. También usaremos el termino "POJO" (Plain-old java object) de vez en cuando.

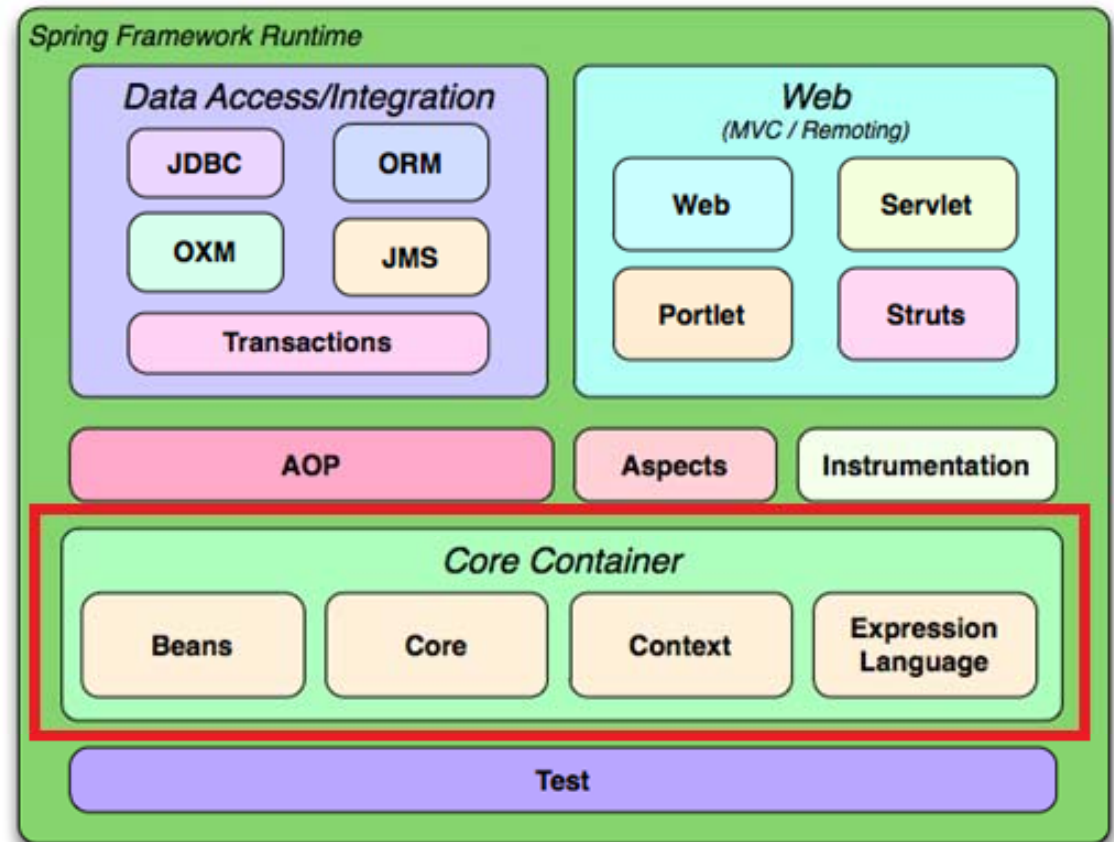
# ¿De qué se compone Spring?

- ✓ Spring está compuesto por diferentes y bien definidos módulos, contruidos sobre la base del contenedor core.
- ✓ Ser modular, es precisamente la característica que hace posible el uso de todo o una parte de Spring Framework, según las necesidades de una aplicación en particular.



# Módulos de Spring: Core

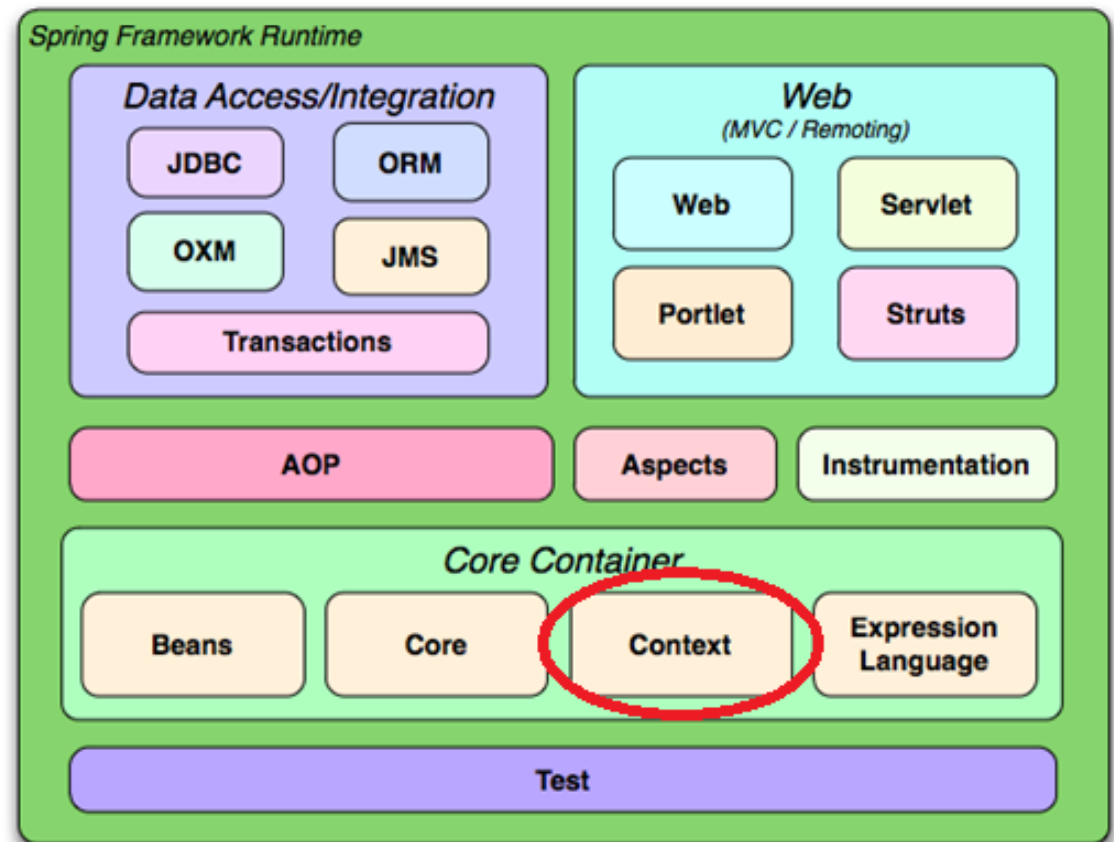
- ✓ El contenedor de Spring provee la funcionalidad más importante del Spring Framework: BeanFactory, la base para la DI en Spring.





# Módulos de Spring: Context

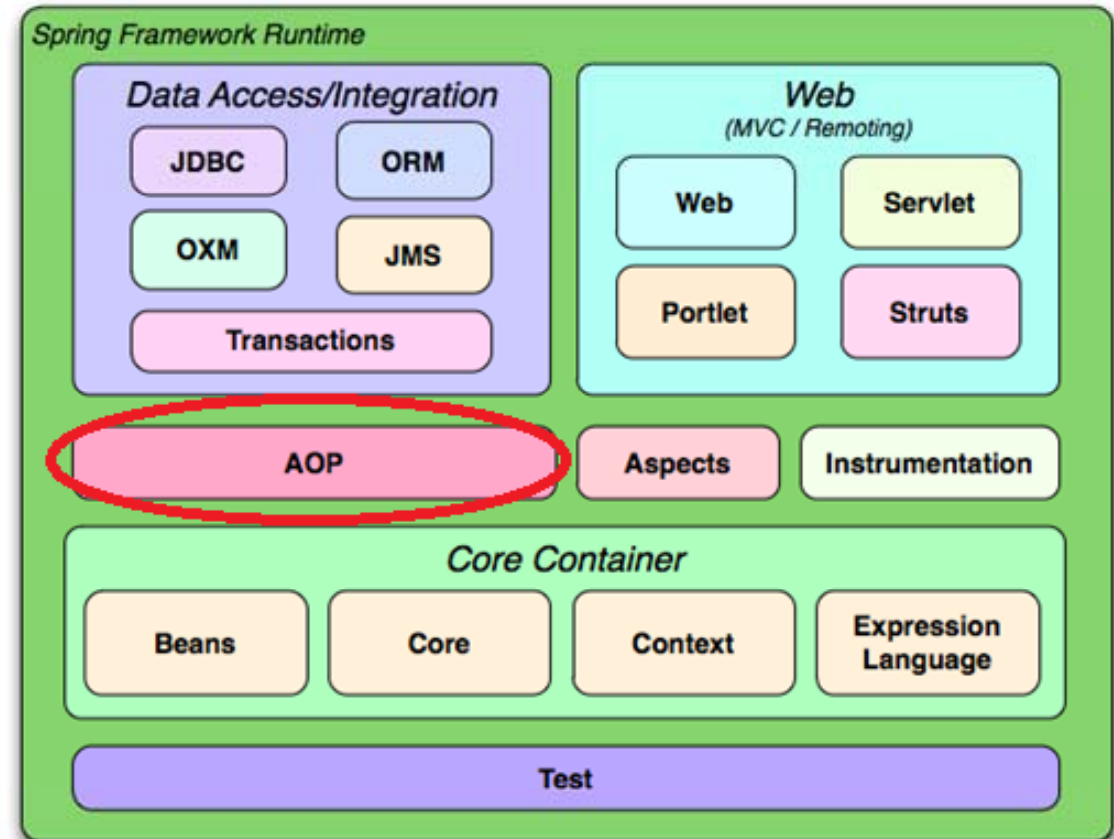
- ✓ Este módulo extiende del BeanFactory, añadiendo soporte para mensajes (I18N), eventos de aplicación, validaciones.
- ✓ Provee muchos de los servicios empresariales como email, acceso vía JNDI, integración EJB, remoting y scheduling.
- ✓ Incluye del soporte para integrarse con otros frameworks de vista como Velocity y FreeMarker.





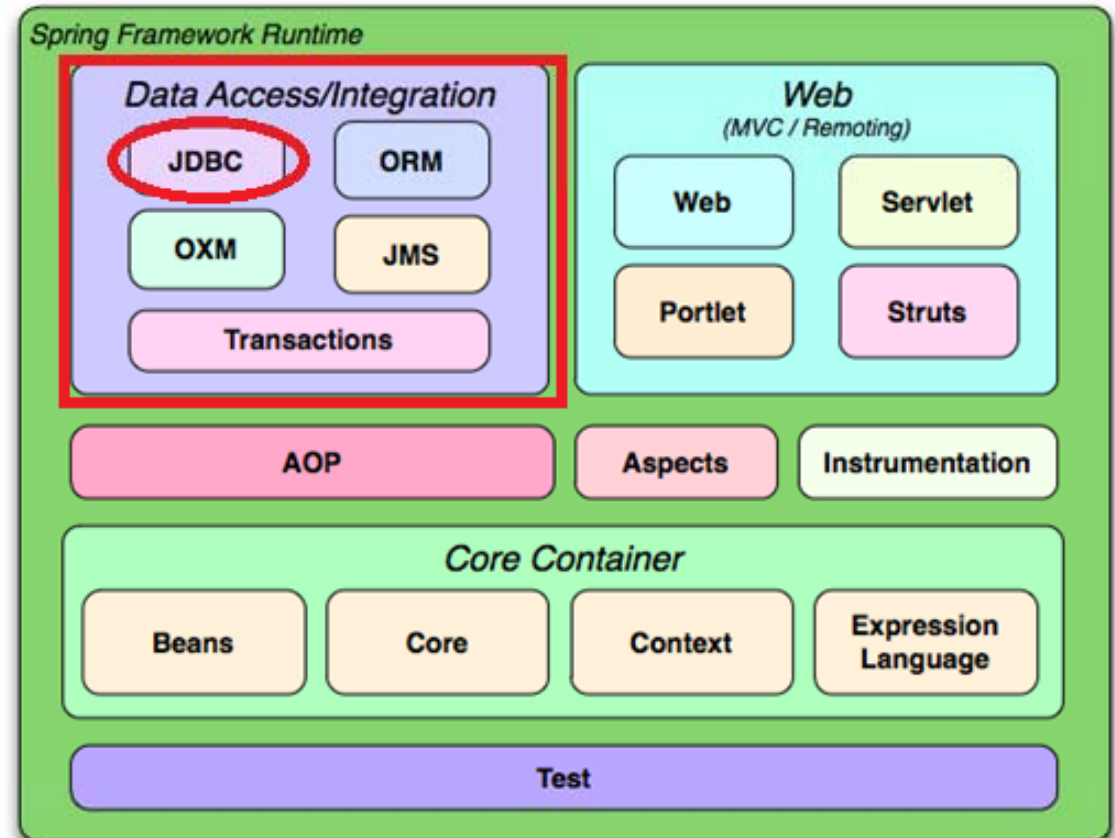
# Módulos de Spring: AOP

- ✓ Spring provee un soporte muy enriquecido para la programación orientada a aspectos con su módulo AOP.
- ✓ Nos permite desarrollar nuestros propios aspectos, y, así cómo, DI, AOP nos permite desacoplar los objetos de nuestra aplicación.
- ✓ Temas como manejo de transacciones, seguridad, etc, son desacoplados de los objetos donde se debería aplicar estos.



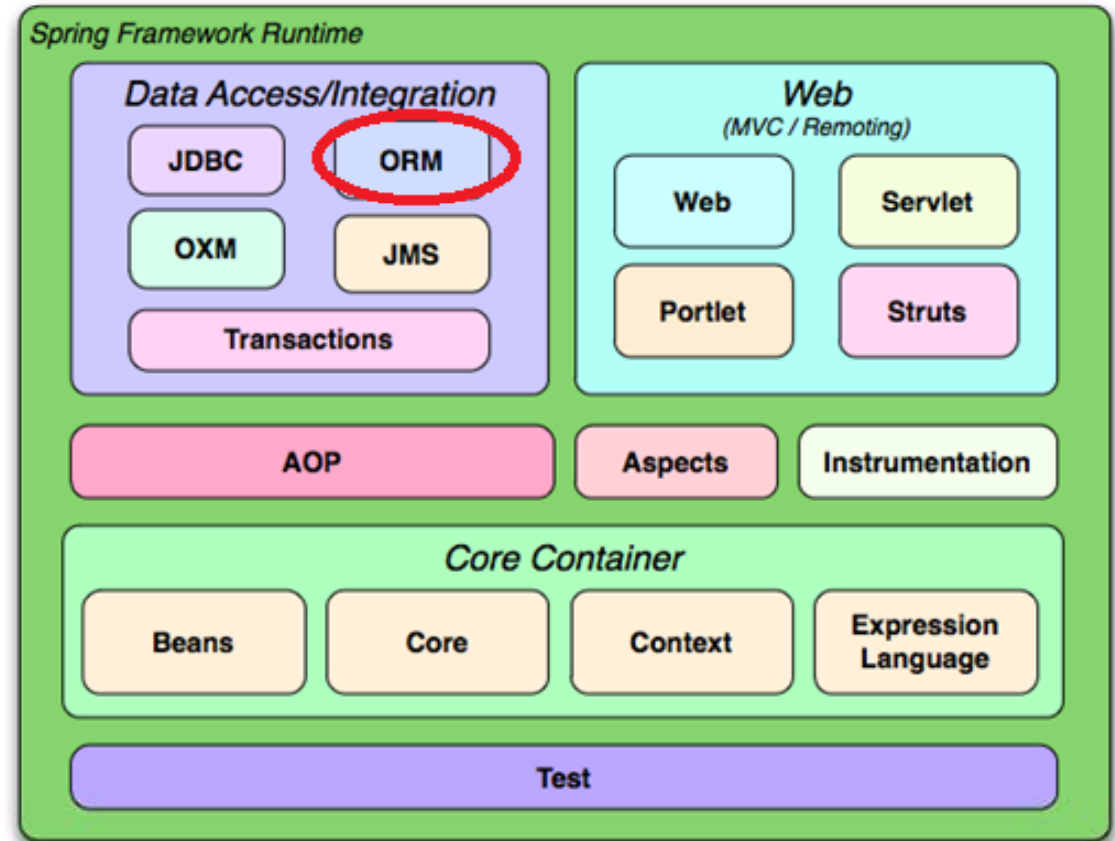
# Módulos de Spring: DAO

- ✓ Trabajar con JDBC, crear un statement, procesar un result set, cerrar conexiones a base de datos, siempre ha sido un tema crítico en el manejo de la persistencia.
- ✓ Spring JDBC y Data Access Objects (DAO) abstraen esa complejidad de manera que nuestro código sea limpio y simple.
- ✓ Previene los problemas de fallas en la liberación de los recursos de base de datos.
- ✓ También proporciona un soporte significativo para manejo de excepciones y mensajes de error dados por diferentes servidores de base de datos. No tendremos que lidiar con mensajes SQL propietarios.
- ✓ Adicionalmente, este módulo provee el servicio de manejo de transacciones para objetos en una aplicación Spring.



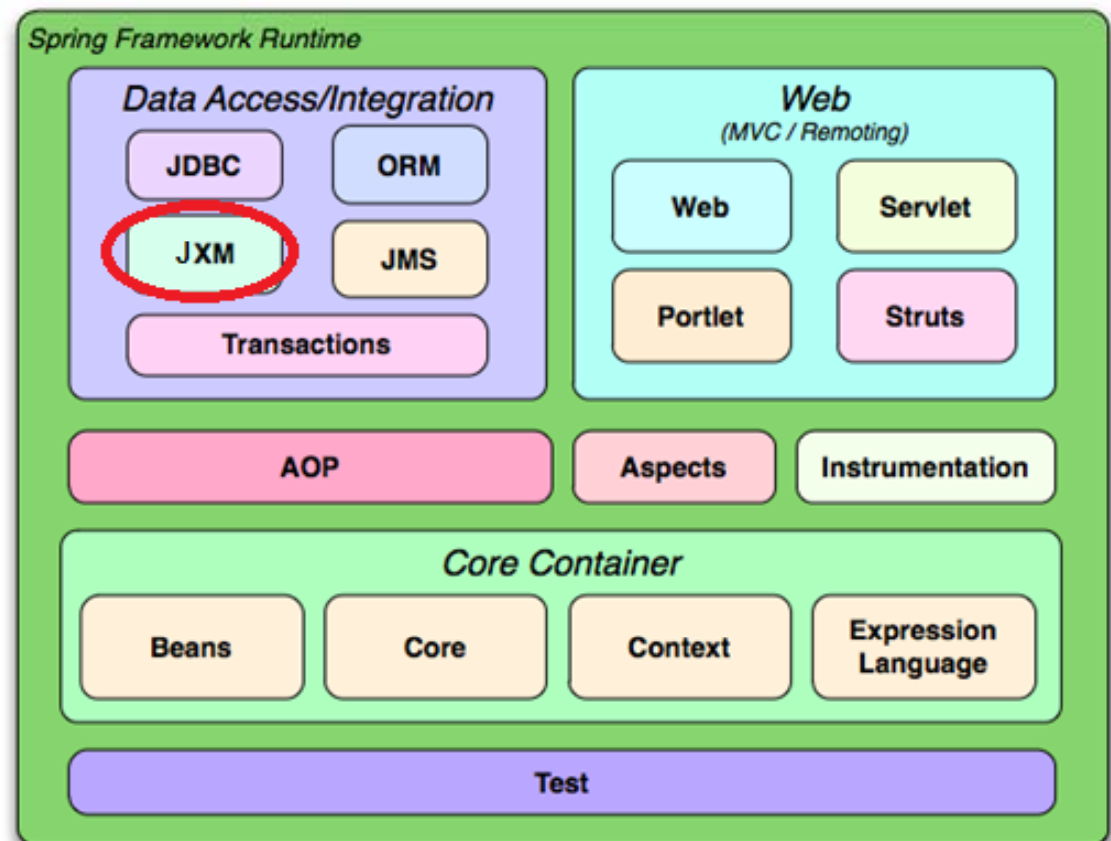
# Módulos de Spring: ORM

- ✓ Este módulo permite construir el DAO para diferentes soluciones ORM.
- ✓ Entre los diferentes frameworks ORM que soporta tenemos: Hibernate, Java Persistence API, Java Data Objects, e iBATIS SQL Maps.
- ✓ Spring Transaction Management soporta cada uno de estos frameworks ORM, así como JDBC.



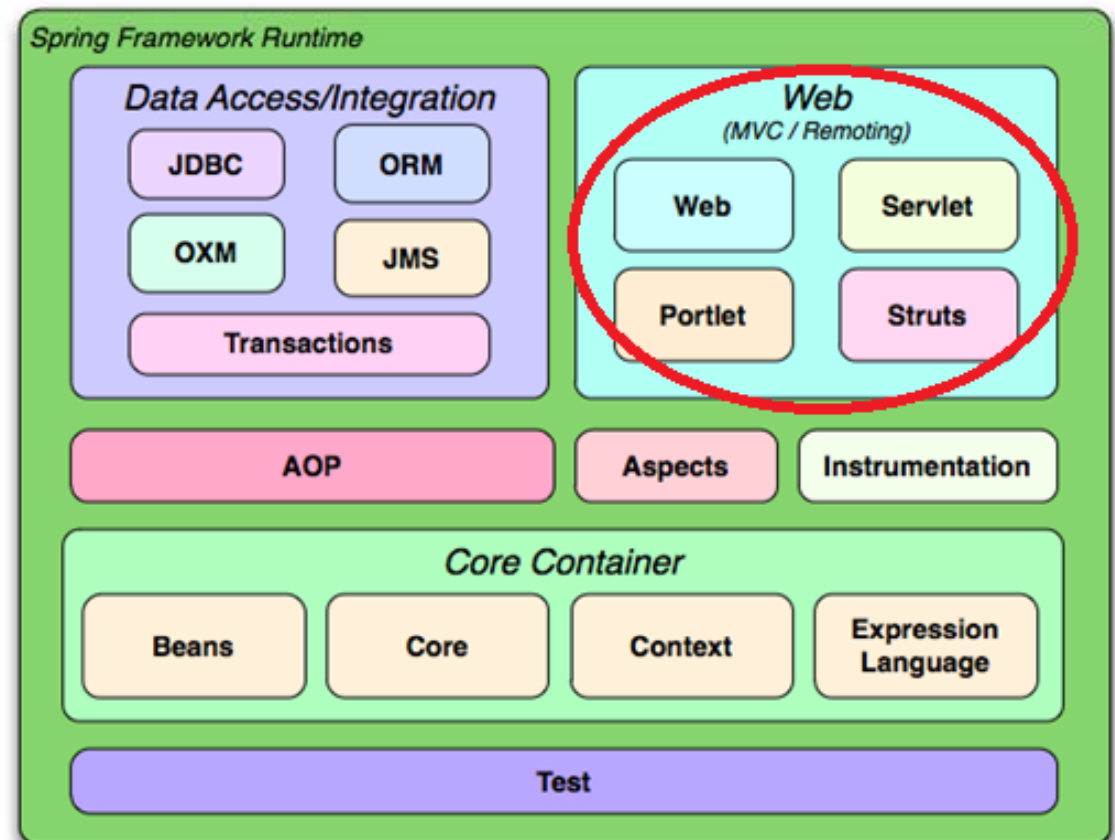
# Módulos de Spring: JMX

- ✓ Spring JMX permite exponer los beans de nuestra aplicación como JMX MBeans.
- ✓ Esto hace posible el monitoreo y la reconfiguración de aplicaciones en tiempo de ejecución.



# Módulos de Spring: Web

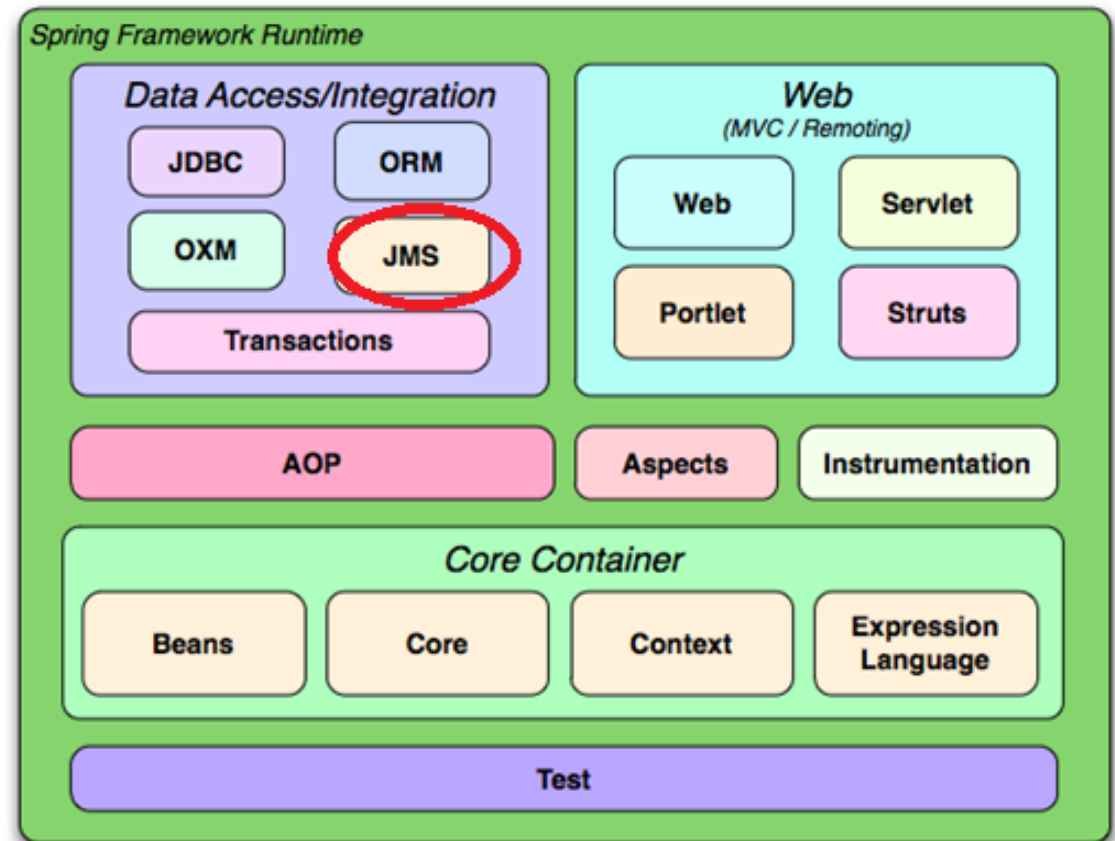
- ✓ Spring da soporte a diferentes frameworks MVC como Struts, JSF, WebWork y Tapestry.
- ✓ Viene con su propio framework MVC que promueve técnicas de desacoplamiento en la capa web de una aplicación.





# Módulos de Spring: JMS

- ✓ Remoting es viable si se dispone de la red y ambos puntos de comunicación están siempre disponibles.
- ✓ La comunicación orientada a mensajes por otro lado, nos da más garantía y sostenibilidad para la entrega de mensajes, aún si la red y los puntos de intercambio (endpoints) no están disponibles.
- ✓ Spring JMS nos permite enviar mensajes a colas JMS. Al mismo tiempo, este módulo nos ayuda a crear POJOs orientados al mensaje que son capaces de consumir mensajes asíncronos.



# ***Spring - RichFaces: El taller***

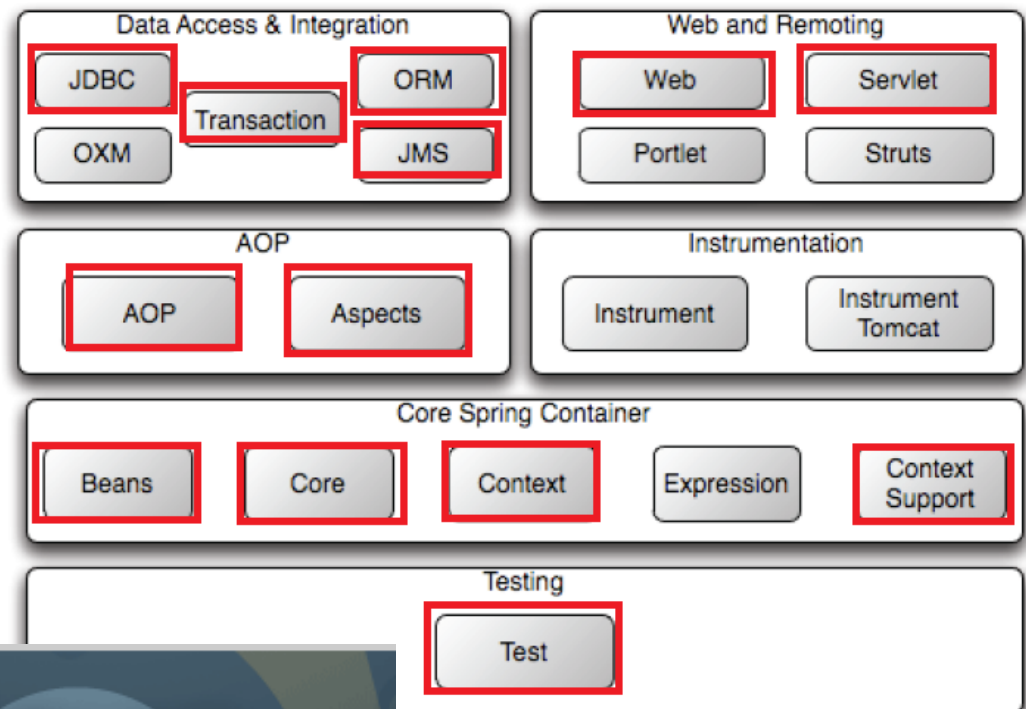
***Alcance del Curso Spring - RichFaces***



# Módulos de Spring y RichFaces:

## ¿Qué veremos nosotros?

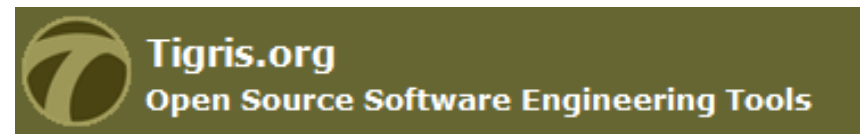
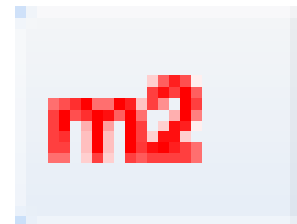
- ✓ Core
- ✓ Context y Context support
- ✓ Testing
- ✓ Aspect Oriented Programming (AOP)
- ✓ ORM con Hibernate
- ✓ JSF
- ✓ RichFaces
- ✓ Spring Security



# Tools



- ✓ En el taller se usará como IDE de desarrollo Eclipse, específicamente STS de [springsource.com](http://springsource.com)
- ✓ También, antes de comenzar con el ejercicio, debemos instalar los siguientes plugins:
  - Plugin de subversion:  
[http://subclipse.tigris.org/update\\_1.6.x](http://subclipse.tigris.org/update_1.6.x)
  - Plugin de maven:  
<http://m2eclipse.sonatype.org/update/>



# *Tools: Maven*



- ✓ Es una herramienta de gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002.
- ✓ Parecido a Ant, pero mucho más simple.
- ✓ Project Object Model (POM) = archivo de descripción, construcción y organización de dependencias

# ***maven***

# ***Clase 1: Introducción a Conceptos***

*IoC y AOP*

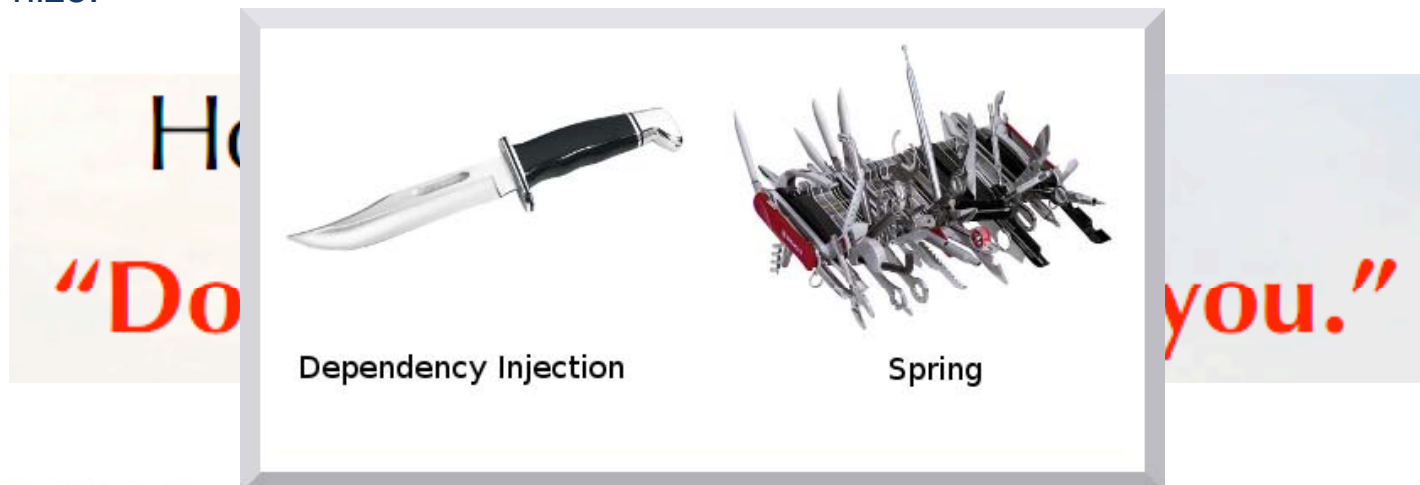
# ¿Qué vas a aprender?

- ✓ Inyección de Dependencias
- ✓ Desacoplamiento con Interfaces
- ✓ AOP



# Spring: ¿IoC o DI?

- ✓ Al inicio la Inyección de Dependencias era llamada Inversión de Control (principio Hollywood).
- ✓ En el 2004 Martin Fowler en un artículo escrito por él cuestionó qué aspecto era el que se estaba invirtiendo. Finalmente, concluye que era la adquisición de dependencias lo que se invertía y finalmente acuña esa frase DI, lo cual describe mejor lo que realmente ocurre.
- ✓ Cuando se aplica DI, se le dan las dependencias a los objetos en el momento de su creación mediante alguna entidad externa que coordina cada objeto del sistema. En otras palabras, las dependencias se inyectan en los objetos.
- ✓ Una ventaja que existe es el desacoplamiento de las clases: si solo se conoce las dependencias **vía interfaz**, al cambiar la implementación de la dependencia el objeto dependiente no se entera que se hizo.





# Tipos de DI I





# Tipos de DI II

- ✓ Constructor: los valores son inyectados a través del constructor hacia el objeto en construcción.

```
public SaludoServiceImpl() {  
    this.saludo = "Hola a todos =)";  
}
```

```
<bean id="saludoService" class="com.joedayz.tema1.saludo.SaludoServiceImpl">  
</bean>
```

```
public SaludoServiceImpl(String saludo){  
    this.saludo = saludo;  
}
```

```
<bean id="saludoService" class="com.joedayz.tema1.saludo.SaludoServiceImpl">  
    <constructor-arg value="Buen día a todos!!!!!!"/>  
</bean>
```

# *Tipos de DI III*

- ✓ Setter: los objetos son creados, luego los valores son asignados a través de cada setter.

```
public void setSaludo(String saludo) {  
    this.saludo = saludo;  
}
```

```
<bean id="saludoService" class="com.joedayz.temal.saludo.SaludoServiceImpl">  
    <property name="saludo" value="Buenos Dias Man!" />  
</bean>
```

# Tipos de DI IV



- ✓ Atributo: el objeto es creado, los valores de sus dependencias son reflectados asignándolos a sus respectivos atributos.

```
public class InicioController {  
  
    @Autowired  
    private ImagenManager imagenManager;
```

# Demo 1: Hello World



# Demo 1: Hello World



- ✓ La primera clase de nuestro ejemplo necesita una clase de servicio con el propósito de imprimir un mensaje familiar. A continuación mostramos la interface SaludoService la cual define el contrato para la clase de servicio.

```
SaludoService.java ✖  
  
package com.joedayz.tema1.saludo;  
  
public interface SaludoService {  
  
    public void saludar();  
}
```

# Demo 1: Hello World



- ✓ SaludoServiceImpl implementa la interface SaludoService. De esta manera separamos la implementación de su contrato.

```
SaludoServiceImpl.java SaludoApp.java
1 package com.joedayz.temal.saludo;
2
3 public class SaludoServiceImpl implements SaludoService {
4
5     private String saludo;
6
7     public SaludoServiceImpl() {}
8
9
10    public SaludoServiceImpl(String saludo){
11        this.saludo = saludo;
12    }
13
14    public void saludar() {
15        System.out.println(saludo);
16    }
17
18    public void setSaludo(String saludo) {
19        this.saludo = saludo;
20    }
21
22
23 }
```

# Demo 1: Hello World



- ✓ La clase SaludoServiceImpl tiene una sola propiedad: saludo.
- ✓ Esta propiedad es un String que almacena el texto del mensaje que será impreso cuando llamemos al método saludar().
- ✓ Notaremos que el saludo puede ser configurado de dos formas: vía constructor o vía el setter de la propiedad. Lo que no es aparente es quién hará la llamada al constructor o al setter para establecer la propiedad.

```
SaludoServiceImpl.java SaludoApp.java
1 package com.joedayz.tem1.saludo;
2
3 public class SaludoServiceImpl implements SaludoService {
4
5     private String saludo;
6
7     public SaludoServiceImpl() {}
8
9
10    public SaludoServiceImpl(String saludo){
11        this.saludo = saludo;
12    }
13
14    public void saludar() {
15        System.out.println(saludo);
16    }
17
18    public void setSaludo(String saludo) {
19        this.saludo = saludo;
20    }
21
22
23 }
```



# Demo 1: Hello World



- ✓ Es en este punto, donde le permitiremos al contenedor de Spring establecer la propiedad saludo.
- ✓ El archivo de configuración de Spring (saludo.xml) que le dirá al contenedor como configurar el servicio saludo.

```
saludo.xml
<?xml version="1.0" encoding="UTF-8"?>
<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-2.5.xsd">

    <bean id="saludoService"
          class="com.joedayz.tem1.saludo.SaludoServiceImpl">
        <property name="saludo" value="Buenos Dias Man!" />
    </bean>
</beans>
```

# Demo 1: Hello World

- ✓ El archivo XML declara una instancia de un SaludoServiceImpl en el contenedor de Spring y configura su propiedad saludo con un valor de "Buenos Días Man!".
- ✓ El atributo id en <bean> sirve para notificar al contenedor de Spring acerca de una clase y como esta debe ser configurada. Aquí es usada para nombrar al bean como saludoService y el atributo class para el nombre completo del bean.
- ✓ Dentro de <bean>, el elemento <property> es usado para configurar una propiedad. En nuestro caso la propiedad saludo. Como se observa el elemento <property> le dice al contenedor de Spring que llamará al setSaludo(), pasando a este "Buenos Días Man!" cuando instancia el bean.
- ✓ Todo lo descrito anteriormente se resume en lo siguiente:

```
SaludoServiceImpl saludoService = new SaludoServiceImpl();
saludoService.setSaludo("Buenos Dias Man!");
```
- ✓ Alternativamente, nosotros podemos establecer la propiedad usando el constructor.

# Demo 1: Hello World



- ✓ La ultima pieza del ejemplo, es la clase que carga el contenedor de Spring y usa a este para recuperar el servicio saludo.

```
SaludoApp.java

package com.joedayz.temal.saludo;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class SaludoApp {

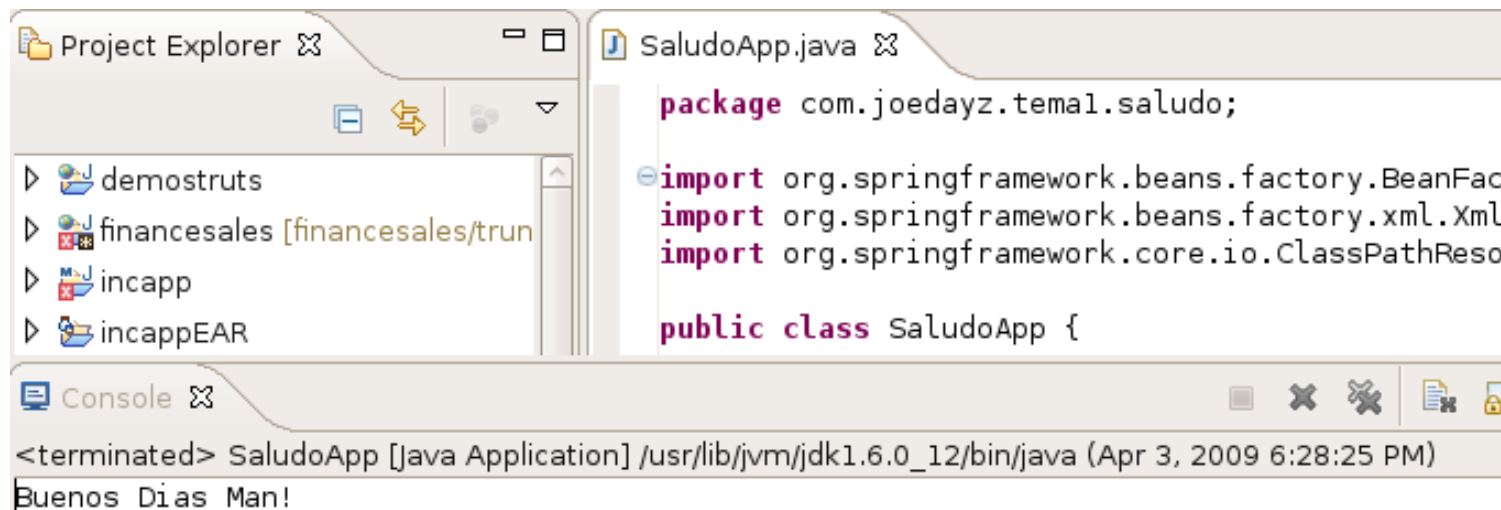
    /**
     * @param args
     */
    public static void main(String[] args) {
        BeanFactory factory =
            new XmlBeanFactory(new ClassPathResource("saludo.xml"));

        SaludoService saludoService =
            (SaludoService) factory.getBean("saludoService");

        saludoService.saludar();
    }
}
```

# Demo 1: Hello World

- ✓ La clase BeanFactory es usada aquí como el contenedor de Spring.
- ✓ Después de cargar el saludo.xml dentro del contenedor, el método main() llama al método getBean() en el BeanFactory para recuperar una referencia al servicio saludo. Con esta referencia en mano, finalmente se llama al método saludar(). Cuando ejecutamos la aplicación Saludo esta imprime:



```
Project Explorer
└─ demostruts
└─ financesales [financesales/trun
└─ incapp
└─ incappEAR

SaludoApp.java
package com.joedayz.temal.saludo;

import org.springframework.beans.factory.BeanFac
import org.springframework.beans.factory.xml.Xml
import org.springframework.core.io.ClassPathReso

public class SaludoApp {

Console
<terminated> SaludoApp [Java Application] /usr/lib/jvm/jdk1.6.0_12/bin/java (Apr 3, 2009 6:28:25 PM)
Buenos Dias Man!
```

# Demo 2: Caballeros



- ✓ Explicaremos DI con un ejemplo novelesco e inspirado en los caballeros de la mesa redonda.
- ✓ Tendremos pues una clase Java que represente a un Caballero el cual se enrola o participa de una aventura para conseguir, por ejemplo, el Santo Grial.





# Demo 2: Caballeros...Round 1

- ✓ Implementamos la clase CaballeroDeLaMesaRedonda, la cual tendrá un nombre y se podrá embarcar a una aventura. En este caso le daremos la aventura de buscar el "Santo Grial".

```
CaballeroDeLaMesaRedonda.java ✖  
  
package com.joedayz.temal.caballero;  
  
public class CaballeroDeLaMesaRedonda {  
  
    private String nombre;  
    private SantoGrialAventura aventura;  
  
    public CaballeroDeLaMesaRedonda(String nombre) {  
        this.nombre = nombre;  
        aventura = new SantoGrialAventura();  
    }  
  
    public SantoGrial embarcarEnAventura() throws GrialNotFoundException {  
        return aventura.embarcar();  
    }  
}
```

# Demo 2: Caballeros...Round 1

- ✓ La implementación de SantoGrialAventura sería así:

```
SantoGrialAventura.java ✕  
  
package com.joedayz.temal.caballero;  
  
public class SantoGrialAventura {  
    public SantoGrialAventura() {}  
  
    public SantoGrial embarcar() throws GrialNotFoundException {  
        // do whatever it means to embark on a quest  
        return new SantoGrial();  
    }  
  
}
```

- ✓ Y listo, hemos terminado, entregamos nuestro código, pero el encargado del proyecto nos pide nuestro Test Case y nosotros.... uff, no tenemos ningún test case.



# Demo 2: Caballeros...Round 2

- ✓ El unit testing es una parte importante del desarrollo. No sólo nos asegura que cada unidad individual funcione como esperábamos, sino que sirva para documentar cada unidad de la mejor forma posible. A continuación un test case para nuestra clase Caballero.

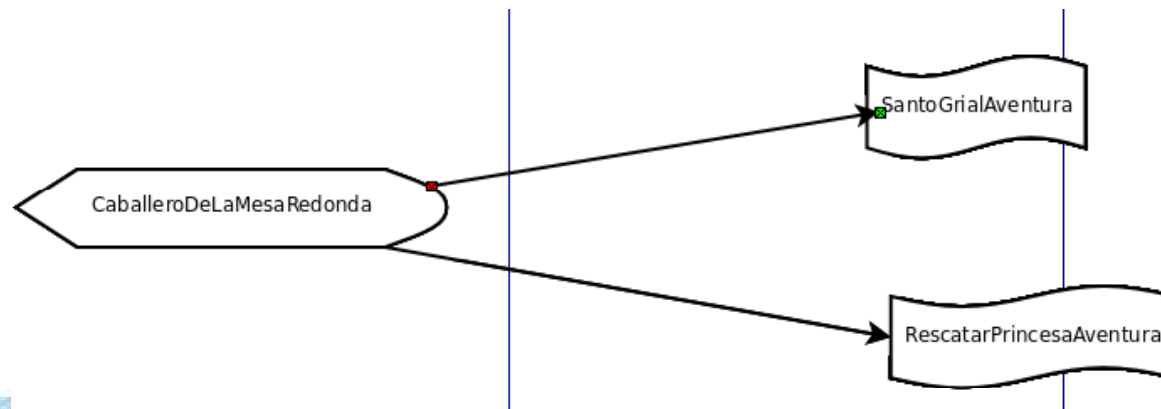
```
CaballeroDeLaMesaRedondaTest.java ✖  
  
package com.joedayz.temal.caballero;  
  
import junit.framework.TestCase;  
  
public class CaballeroDeLaMesaRedondaTest extends TestCase {  
  
    public void testEmbarcarEnAventura() throws GrialNotFoundException {  
        CaballeroDeLaMesaRedonda caballero =  
            new CaballeroDeLaMesaRedonda("Amadeo");  
  
        SantoGrial grial = caballero.embarcarEnAventura();  
        assertNotNull(grial);  
        assertTrue(grial.isSanto());  
    }  
}
```

- ✓ Después de escribir este test case, nos damos cuenta que indirectamente también testeamos el SantoGrialAventura. Pero, que pasaría si el embarcar de SantoGrialAventura retorna null o si arroja GrialNotFoundException.

# Demo 2: Caballeros... ¿Quién llama a quien?



- ✓ El principal problema del test es cómo el CaballeroDeLaMesaRedonda obtiene su SantoGrialAventura. Ya sea creando una nueva instancia SantoGrial u obteniéndola vía JNDI, el punto clave es que cada Caballero es responsable en obtener su propia aventura.
- ✓ Adicionalmente, no hay forma de testear la clase Caballero aisladamente, puesto que, cada vez que testeamos el CaballeroDeLaMesaRedonda, también testeas SantoGrialAventura.
- ✓ Lo más preocupante es que no hay forma de decirle al SantoGrialAventura que retorne null o GrialNotFoundException para otros tests. Para solucionar esto, tal vez, sea necesario crear una implementación mock (simulada) de SantoGrialAventura que nos permita decidir como esta se comportará. Pero, aún si tuviéramos una implementación mock, CaballeroDeLaMesaRedonda, aún recuperaría su propio SantoGrialAventura, lo cual nos lleva a que tendríamos que cambiar CaballeroDeLaMesaRedonda para recuperar la mock aventura para propósitos de testing.



# Demo 2: Caballeros...Round 3

- ✓ El problema anterior que se describió se llama acoplamiento. En este punto CaballeroDeLaMesaRedonda esta acoplado estáticamente a SantoGrialAventura.
- ✓ Una técnica común usada para reducir acoplamiento es ocultar los detalles de la implementación usando interfaces, de manera que la implementación actual pueda ser cambiada sin impactar al cliente de la clase. Por ejemplo, supongamos que deseamos crear una interface Aventura:

```
Aventura.java ✖  
  
package com.joedayz.temal.caballero;  
  
public interface Aventura {  
    public abstract Object embarcar() throws AventuraFailedException;  
}
```

# Demo 2: Caballeros...Round 3

- ✓ Entonces, tendríamos que cambiar SantoGrialAventura para implementar esta interface. También hay que notar que embarcar() retorna un Object y throws un AventuraFailedException.

```
SantoGrialAventura.java ✖  
  
package com.joedayz.temal.caballero;  
  
public class SantoGrialAventura implements Aventura{  
    public SantoGrialAventura() {}  
  
    public Object embarcar() throws GrialNotFoundException {  
        // do whatever it means to embark on a quest  
        return new SantoGrial();  
    }  
}
```

# Demo 2: Caballeros...Round 3

- ✓ También, el siguiente método debería cambiar a CaballeroDeLaMesaRedonda para ser compatible con estos tipos de Aventura:

```
CaballeroDeLaMesaRedonda.java
package com.joedayz.temal.caballero;

public class CaballeroDeLaMesaRedonda {

    private String nombre;
    private Aventura aventura;

    public CaballeroDeLaMesaRedonda(String nombre) {
        this.nombre = nombre;
    }

    public Object embarcarEnAventura() throws AventuraFailedException {
        return aventura.embarcar();
    }

    public String getNombre() {
        return nombre;
    }

    public void setAventura(Aventura aventura) {
        this.aventura = aventura;
    }
}
```

# Demo 2: Caballeros...Round 3

- ✓ En este punto, si vemos CaballeroDeLaMesaRedonda el problema está en cómo obtiene su aventura, porque, muy mal haríamos en volver a instanciar SantoGrialAventura en el constructor. Nos preguntamos, entonces, ¿El Caballero debería obtener su aventura? o ¿Debería asignársele una aventura? ¿Qué hay si mañana más tarde se le quiere asignar al Caballero que rescate una princesa? (Fiona!)
- ✓ Es aquí donde DI entra a nuestro rescate, debemos poder relacionarlo a sus dependencias indirectamente y no directamente.

```
public void setAventura(Aventura aventura) {  
    this.aventura = aventura;  
}
```

- ✓ El Caballero, por lo tanto, debe relacionarse con sus dependencias, entonces, es nuestra responsabilidad crear las asociaciones respectivas entre componentes. A este acto se le llama wiring. Y una de las formas más comunes de hacer esto es vía XML.

```
<bean id="aventura"  
    class="com.joedayz.temal.caballero.SantoGrialAventura"/>  
  
<bean id="caballero"  
    class="com.joedayz.temal.caballero.CaballeroDeLaMesaRedonda">  
    <constructor-arg value="Amadeo" />  
    <property name="aventura" ref="aventura" />  
</bean>
```

# Demo 2: Caballeros...Round 4

- ✓ Ahora creamos CaballeroApp para cargar la definición de beans usando XmlBeanFactory.

```
CaballeroApp.java
package com.joedayz.temal.caballero;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class CaballeroApp {

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        ApplicationContext ctx =
            new ClassPathXmlApplicationContext(
                "com/joedayz/temal/caballero/caballero.xml");

        Caballero caballero =
            (Caballero) ctx.getBean("caballero");

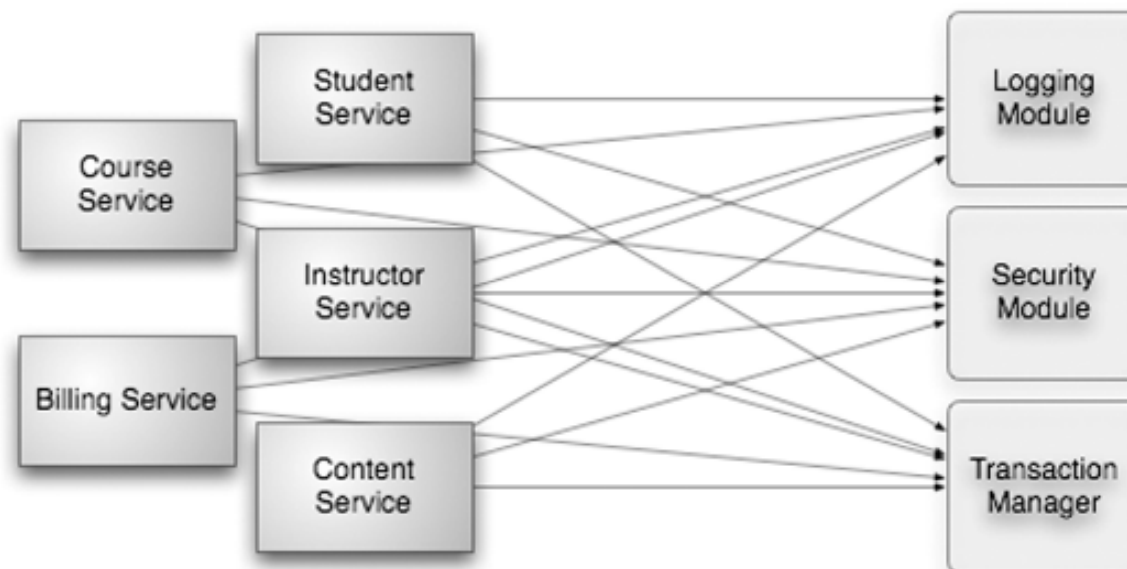
        caballero.embarcarEnAventura();
    }
}
```

- ✓ Esto tan simple que hemos visto de caballeros, princesas, aventuras, nos ha permitido ver como funciona la inyección de dependencias.



# Spring: AOP I

- ✓ Como hemos visto DI nos permite usar componentes de software pobremente acoplados, AOP viene para permitirnos capturar la funcionalidad que es usada a lo largo de toda nuestra aplicación en componentes reutilizables.
- ✓ Aspect-Oriented programming se define como una técnica de programación que promueve la separación de funcionalidades (concerns) dentro de un sistema de software.
- ✓ Los sistemas están compuestos por diferentes componentes, cada uno responsable por una pieza específica de funcionalidad. Sin embargo, muy frecuentemente, estos componentes traen responsabilidades colaterales a su funcionalidad principal.
- ✓ Servicios del sistema como logging, administración de transacciones, y seguridad se encuentran frecuentemente localizados en componentes cuya verdadera responsabilidad es otra cosa. Estos servicios del sistema son comúnmente denominados como cross-cutting concerns porque ellos suelen impactar en múltiples componentes de un sistema.



# Spring: AOP II

- ✓ Como vemos cada componente esta involucrado con estos servicios del sistema y tiene que manejar dicho servicio muy a pesar de la responsabilidad que tiene.
- ✓ AOP nos soluciona esto, viendo estos servicios como capas o layers de funcionalidad. Estas capas se pueden aplicar declarativamente sin afectar la aplicación de manera que ni este enterado de que se tiene estos servicios.

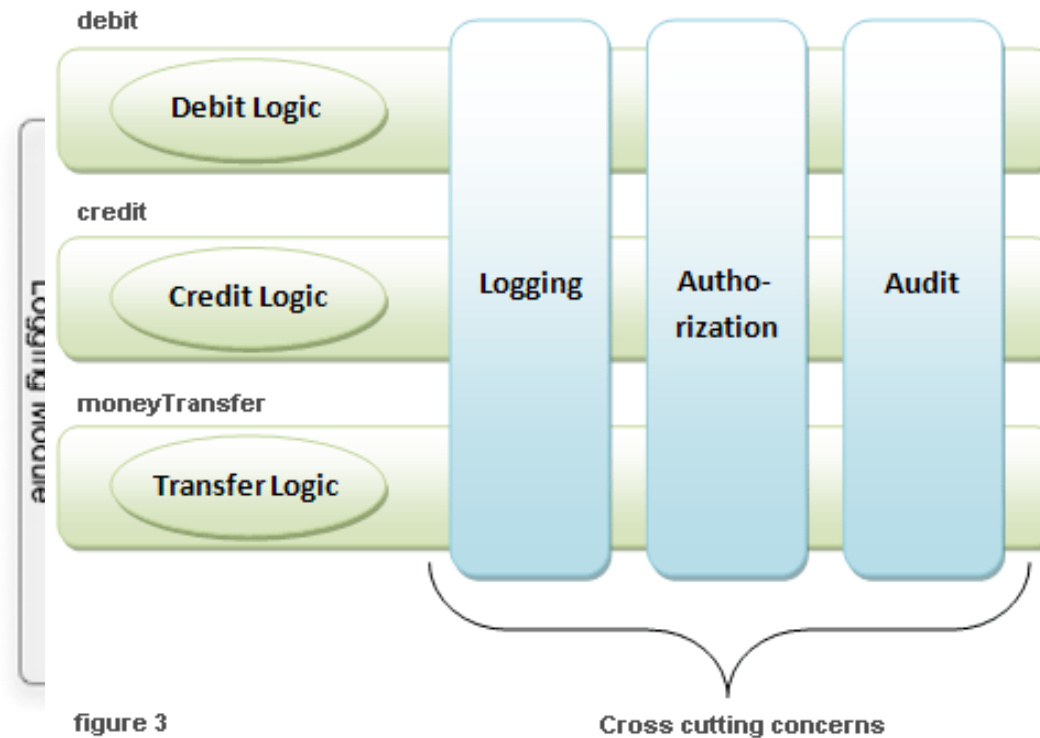


figure 3

Cross cutting concerns

# Demo 3: Caballeros + Trovador



+



=

# AOP

# Demo 3: Caballeros + Trovador

- ✓ Qué hay si nos piden que tengamos un Trovador, como se muestra aquí:

```
Trovador.java
package com.joedayz.temal.caballero;

import org.apache.log4j.Logger;

public class Trovador {

    private static final Logger SONG =
        Logger.getLogger(Trovador.class);

    public Trovador() {}

    public void cantarAntes(Caballero caballero) {
        SONG.info("Fa la la; Sir " + caballero.getNombre() +
            " es tan valiente!");
    }

    public void cantarDespues(Caballero caballero) {
        SONG.info("Tee-hee-he; Sir " + caballero.getNombre() +
            " termino su aventura!");
    }
}
```

# Demo 3: Caballeros + Trovador

- ✓ Para mantenernos pensando en inyección de dependencias, vamos a modificar CaballeroDeLaMesaRedonda dándole una instancia de Trovador y luego hacemos las modificaciones para que el Trovador cante antes de tomar la aventura y al finalizar la aventura.

```
CaballeroDeLaMesaRedonda.java ✕  
  
public class CaballeroDeLaMesaRedonda implements Caballero{  
  
    private String nombre;  
    private Aventura aventura;  
    private Trovador trovador;  
  
    public CaballeroDeLaMesaRedonda(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public Object embarcarEnAventura() throws AventuraFailedException {  
        trovador.cantarAntes(this);  
        Object obj = aventura.embarcar();  
        trovador.cantarDespues(this);  
        return obj;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setAventura(Aventura aventura) {  
        this.aventura = aventura;  
    }  
  
    public void setTrovador(Trovador trovador) {  
        this.trovador = trovador;  
    }  
}
```

# Demo 3: Caballeros + Trovador

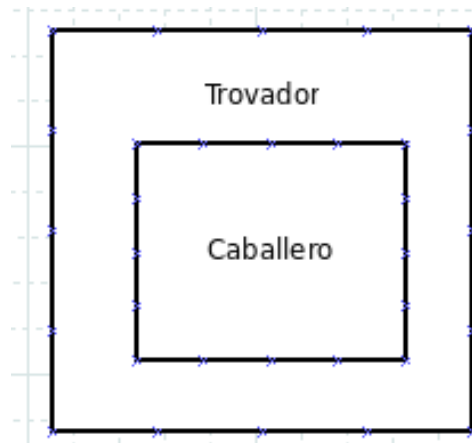
- ✓ Cómo vemos tenemos el problema de que cada Caballero debe detenerse y decirle al Trovador que cante antes de que el Caballero tome su aventura y luego de terminar la aventura, el Caballero debe de recordar decirle al Trovador que continúe cantando.
- ✓ Si hubiésemos querido solucionar el problema como se muestra en el siguiente código, Java no lo permitiría, es un error de sintaxis:

```
public SantoGrial embarcarEnAventura() throws AventuraFailedException {  
    trovador.cantarAntes(this);  
    return aventura.embarcar();  
    trovador.cantarDespues(this);  
}
```



# Demo 3: Caballeros + Trovador

- ✓ Idealmente, un trovador debería tener más iniciativa y automáticamente cantar canciones sin explícitamente decirle que lo haga.
- ✓ En resumen, los servicios del trovador son ortogonales a los del Caballero. Por lo tanto, tiene más sentido tratar de convertir al trovador en un aspecto que permita añadir los servicios del trovador a un Caballero.



- ✓ Luego del preámbulo, es hora de configurar la clase Trovador como un aspecto usando Spring AOP.



# Demo 3: AOP...Round 1



- ✓ En Spring sería de la siguiente forma, primero veamos los namespaces que hay que declarar:

```
caballero.xml ⌕  
  
<?xml version="1.0" encoding="UTF-8"?>  
  
<beans xmlns="http://www.springframework.org/schema/beans"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xmlns:aop="http://www.springframework.org/schema/aop"  
  xsi:schemaLocation="http://www.springframework.org/schema/beans  
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  
    http://www.springframework.org/schema/aop  
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
```

# Demo 3: AOP...Round 2



- ✓ Luego, añadimos el bean y la configuración para que sea un aspecto en el sistema.

```
caballero.xml
<property name="aventura" ref="aventura" />
</bean>

<bean id="trovador"
      class="com.joedayz.temal.caballero.Trovador"/>

<aop:config>
  <aop:aspect ref="trovador">
    <aop:pointcut
      id="aventuraPointcut"
      expression="execution(* *.embarcarEnAventura(..) and target(bean))" />

    <aop:before
      method="cantarAntes"
      pointcut-ref="aventuraPointcut"
      arg-names="bean" />

    <aop:after-returning
      method="cantarDespues"
      pointcut-ref="aventuraPointcut"
      arg-names="bean" />
  </aop:aspect>
</aop:config>
</beans>
```

- ✓ De esta forma declaramos un aspecto y lo relacionamos al bean Trovador. Un aspecto está compuesto de pointcuts (lugares del código donde la funcionalidad del aspecto se aplicará) y advice (como aplicar la funcionalidad). El `<aop:pointcut>` es el elemento que define un pointcut que es disparado por la ejecución de un método `embarcarEnAventura()`.

# Demo 3: AOP...Round 3



- ✓ Finalmente, nosotros tenemos dos AOP advice. El <aop:before> declara que el método cantarAntes() del Trovador deberá ser llamado antes del pointcut, mientras que <aop:after> que declara cantarDespues() del método Trovador deberá ser llamado después del pointcut. El pointcut en ambos casos es una referencia a aventuraPointcut, el cual es la ejecución de embarcarEnAventura().
- ✓ Gracias a AOP, nuestra clase CaballeroDeLaMesaRedonda no sabe nada acerca de la existencia de trovador (como ven ya quite la referencia a Trovador).

```
CaballeroDeLaMesaRedonda.java
package com.joedayz.temal.caballero;

public class CaballeroDeLaMesaRedonda implements Caballero{

    private String nombre;
    private Aventura aventura;

    public CaballeroDeLaMesaRedonda(String nombre) {
        this.nombre = nombre;
    }

    public Object embarcarEnAventura() throws AventuraFailedException {
        return aventura.embarcar();
    }

    public String getNombre() {
        return nombre;
    }

    public void setAventura(Aventura aventura) {
        this.aventura = aventura;
    }
}
```

*Demo 3: AOP forever!*



***Enjoy!***

# *Finalmente... Bienvenidos!*



# *Gracias!*