

# La Charpente Printanière

---

Olivier Liechti  
Open Source Frameworks  
Master Of Science in Engineering (MSE)



MASTER OF SCIENCE  
IN ENGINEERING

# Planning

Date	Java EE Frameworks	Gamification Project
20.02.13	Overview + Business Tier ( <b>EJB</b> )	Group formation + domain model analysis
27.02.13	Web Services ( <b>JAX-RS</b> / <b>JAX-WS</b> )	<b>10' presentation</b> of the domain model + review
06.03.13	Design of the gamification REST API / project setup on Github	
13.03.13	<b>15' presentation</b> of the REST API + 15' discussion / documentation on Github	
20.03.13	Intro to javascript / node.js frameworks	node.js + express.js tutorial
27.03.13	<b>Spring Framework</b>	Implementation of the REST API
03.04.13	<i>Eastern Break</i>	<i>Eastern Break</i>
10.04.13	Persistence Tier ( <b>JPA</b> )	NoSQL tutorial + relevance to the project
17.04.13	Project implementation	
24.04.13	Message Oriented Middleware ( <b>JMS</b> )	Project implementation
01.05.13	<b>20' presentation</b> of your NoSQL back-end and its role in your project	
08.05.13	Presentation Tier (MVC / Frameworks)	javascript framework tutorial
15.05.13	Project implementation	
22.05.13	Project implementation	
29.05.13	<b>Final Presentations &amp; Demos (30')</b>	

# Spring Framework

---

- **Spring**
  - From a book to a company
  - Core framework + ecosystem
- **Core concepts**
  - Inversion of Control (IoC) & Dependency Injection (DI)
  - Aspect Oriented Programming (AOP)
- **The Web Tier**
  - Spring MVC
  - Spring WebFlow

# The Spring Framework

---

- **When was it developed?**

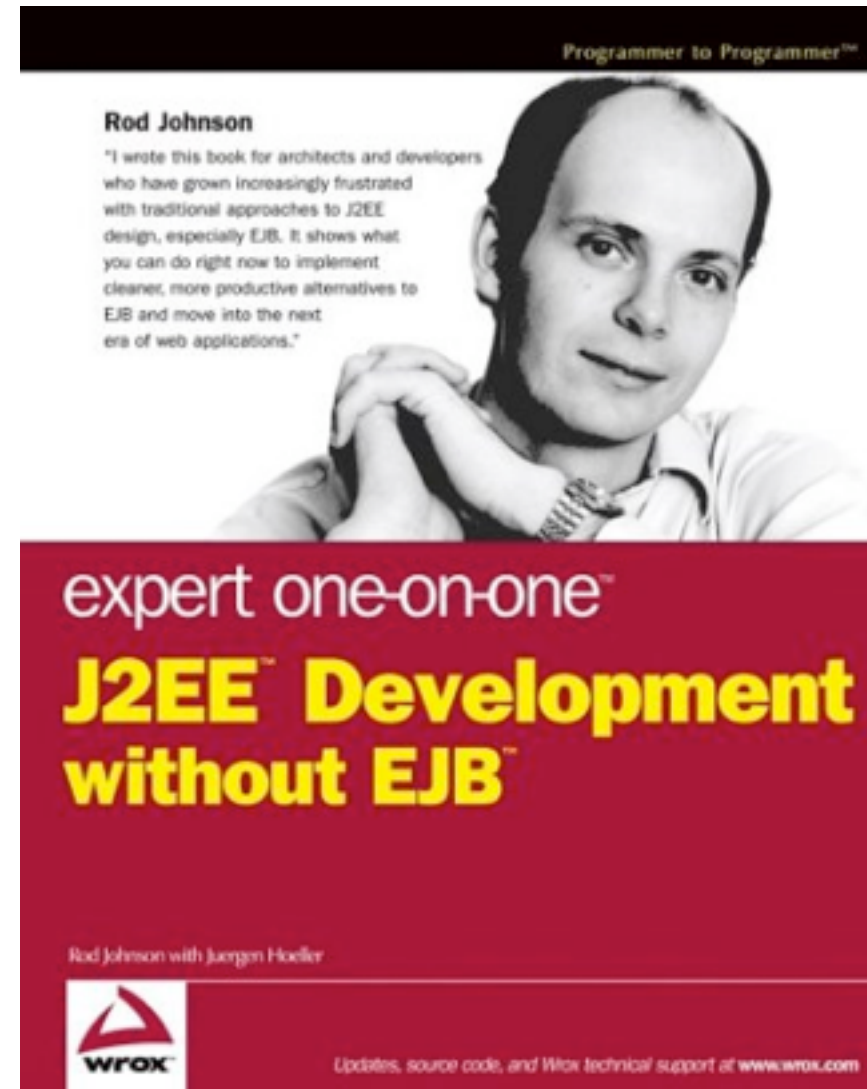
- The Spring Framework was released in 2003.
- It was developed by Rod Johnson and presented in the book “Expert One-on-One J2EE Design and Development”.
- The framework has quickly become very popular and has expanded a lot since its inception (also through “acquisitions” of open source projects)

- **Why was it developed?**

- The Spring Framework was developed at the time of J2EE and EJB 2.
- At the time, using Enterprise Java Beans was rather “painful”.
- The Spring Framework proposed a lightweight approach, which was appropriate in many situations (for which J2EE was overkill).

# Rod Johnson

---



# “What Do We Really Want from EJB”

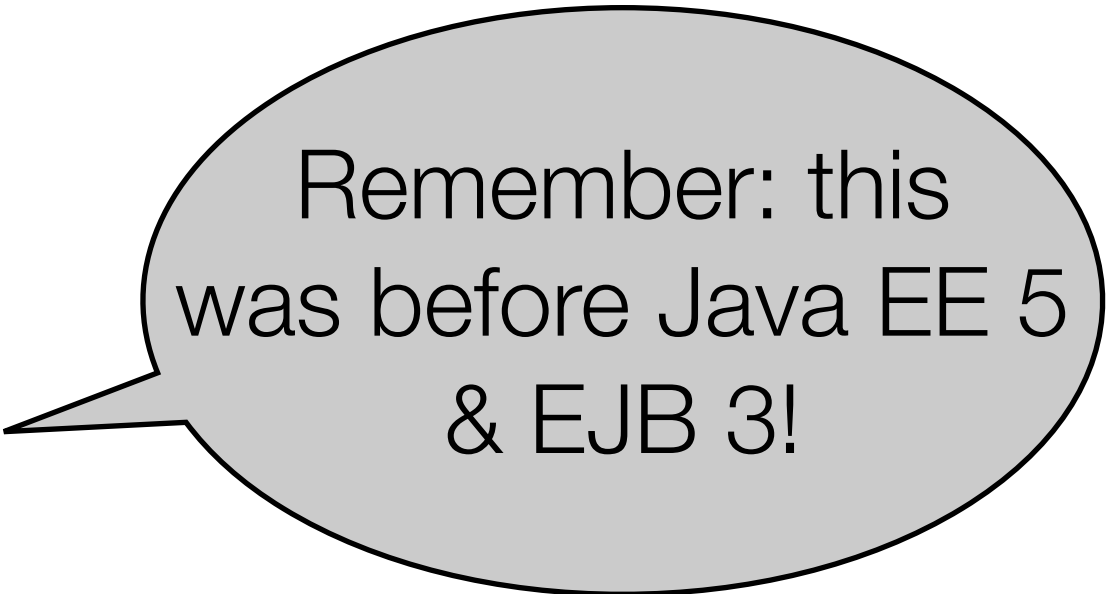
---

- **Declarative Transaction Management**
- Remoting (RMI)
- Clustering
- Thread Management
- EJB Instance Pooling
- Resource Pooling
- Security
- Business Object Management

# “What Don’t we Want from EJB?”

---

- **The Monolithic, Distinct Container Problem**
- Inelegance and the Proliferation of Classes
- Deployment Descriptor Hell
- Class Loader Hell
- **Testing**
- EJB Overdose
- **Complex Programming Model**
- Simple Things Can Be Hard
- Is the Goal of Enabling Developers to Ignore the Complexity of Enterprise Applications Event Desirable?
- Loss of Productivity
- **Portability Problems**



Remember: this  
was before Java EE 5  
& EJB 3!

# Spring: Core Framework vs. Ecosystem

---

- **The core Spring framework**

- provides a solution for building elegant object-oriented systems;
- supports inversion of control and aspect-oriented programming as key underlying mechanisms.

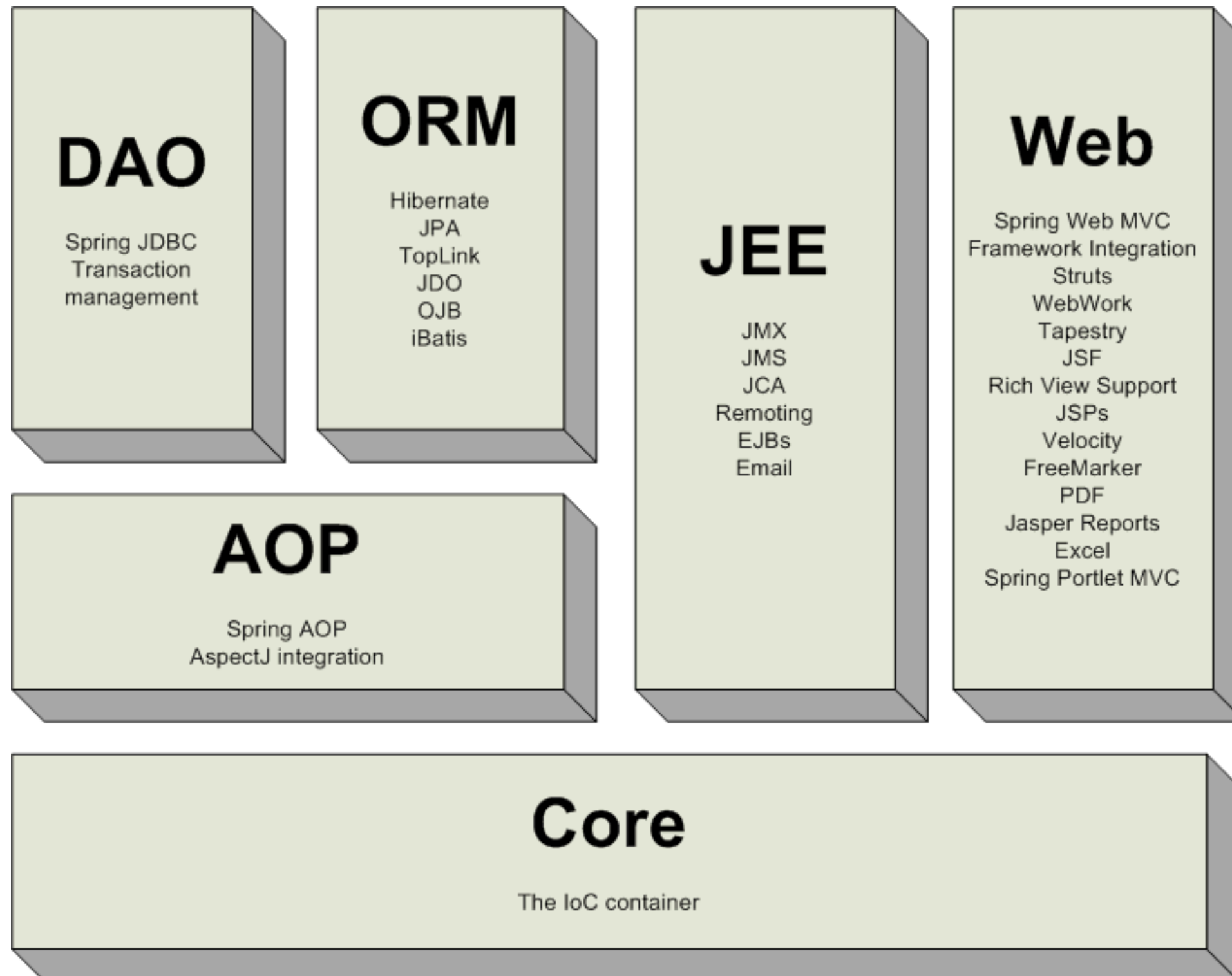
- **The Spring ecosystem**

- is a set of modules and frameworks built on top of the core framework;
- provides solution in many different domains: data access, web tier, messaging, security, etc.

So... then I can use only the core framework (which is lightweight)

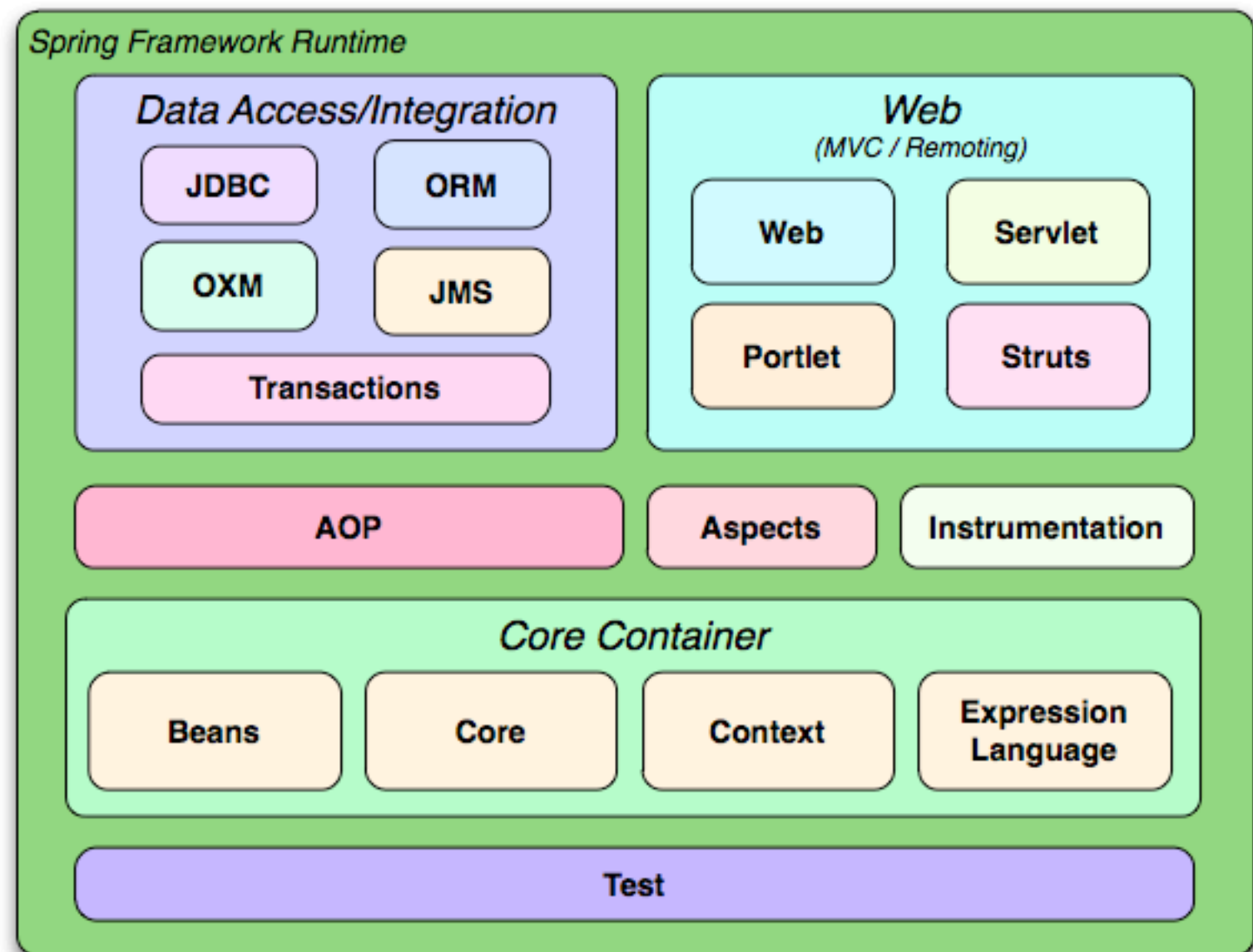


# Spring Framework



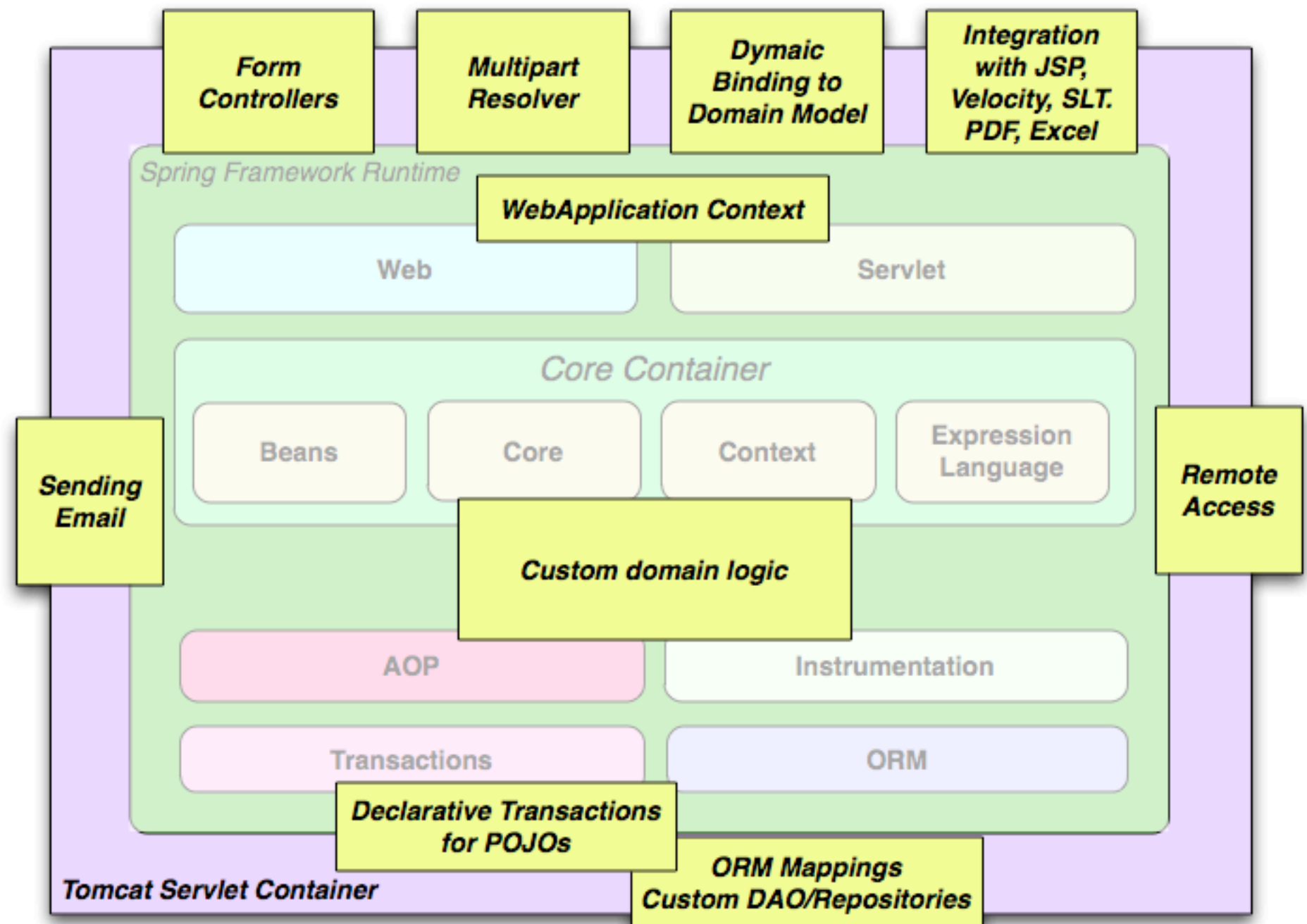
# Spring Framework 3.0

- Spring enables you to build applications from POJOs and to apply enterprise services non-invasively.
- This capability applies to the Java SE programming model and to full and partial Java EE.
- Spring 3.0 released at the end of 2009



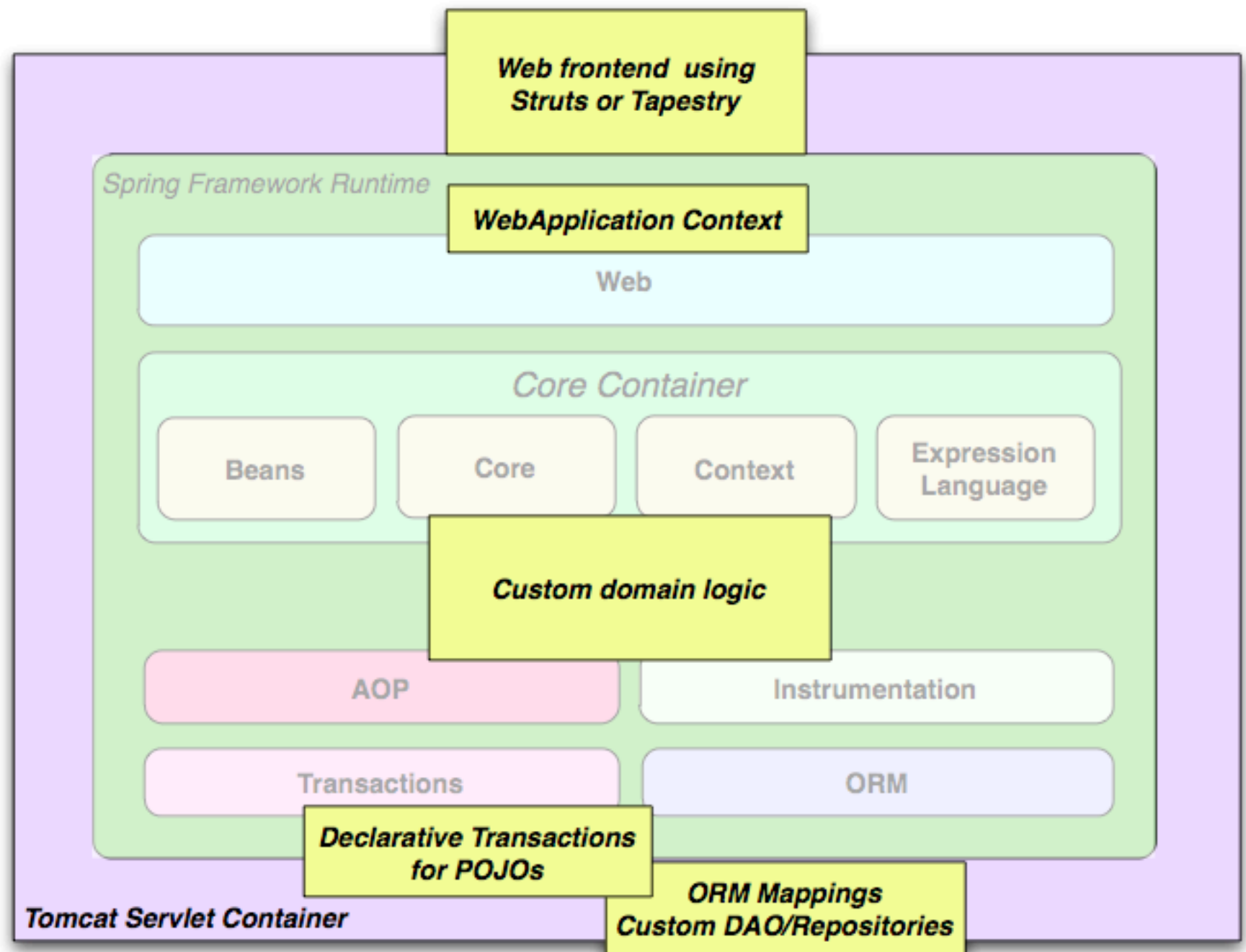
# Spring Framework 3.0

- Using the full framework, also in the web tier.



# Spring Framework 3.0

- Integrating with third-party web frameworks.



# Spring Projects

---

Spring Security

Spring Web Flow

Spring Dynamic  
Modules

Spring Integration



Spring .NET

Spring Batch

# Inversion of Control (IoC)

---



The Hollywood Principle

# Example

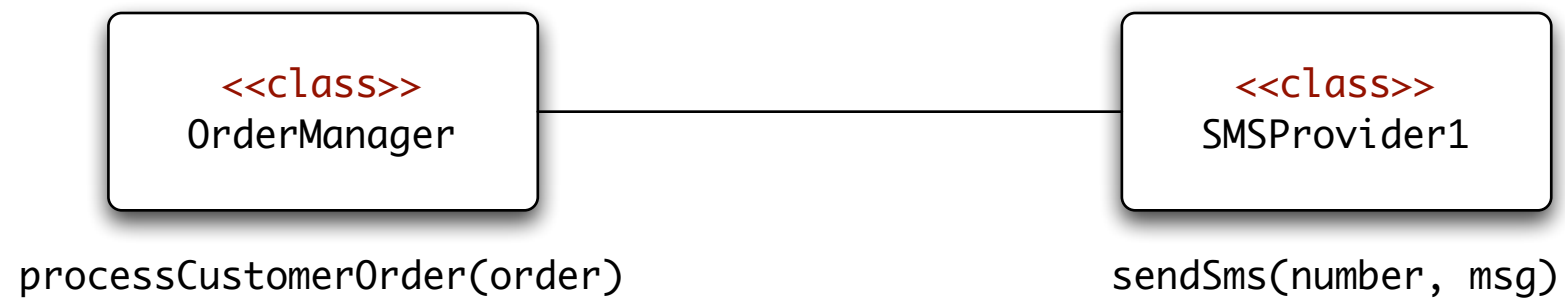
---

- You develop an **online web store** and design an **order processing** workflow.
- During the order processing, you want to **send a SMS** to the customer.
- You can use “SMS Providers” also called “**SMS Brokers**”, who make it easy to send SMS through an HTTP interface.
- Question: how do you **design** the underlying system?



# Step 1: share responsibilities

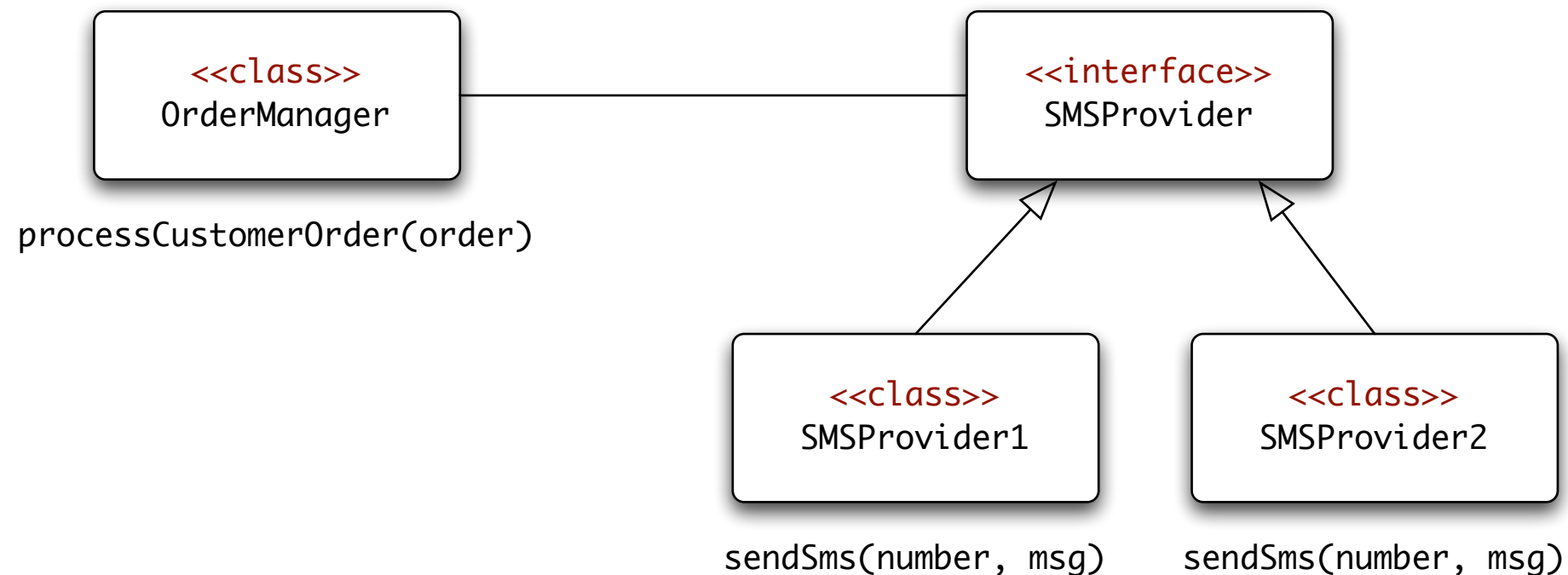
---



Ok... but what if want to change the SMS provider?



## Step 2: program against interfaces



Ok... but how does the OrderManager get a reference to the proper SMS provider?

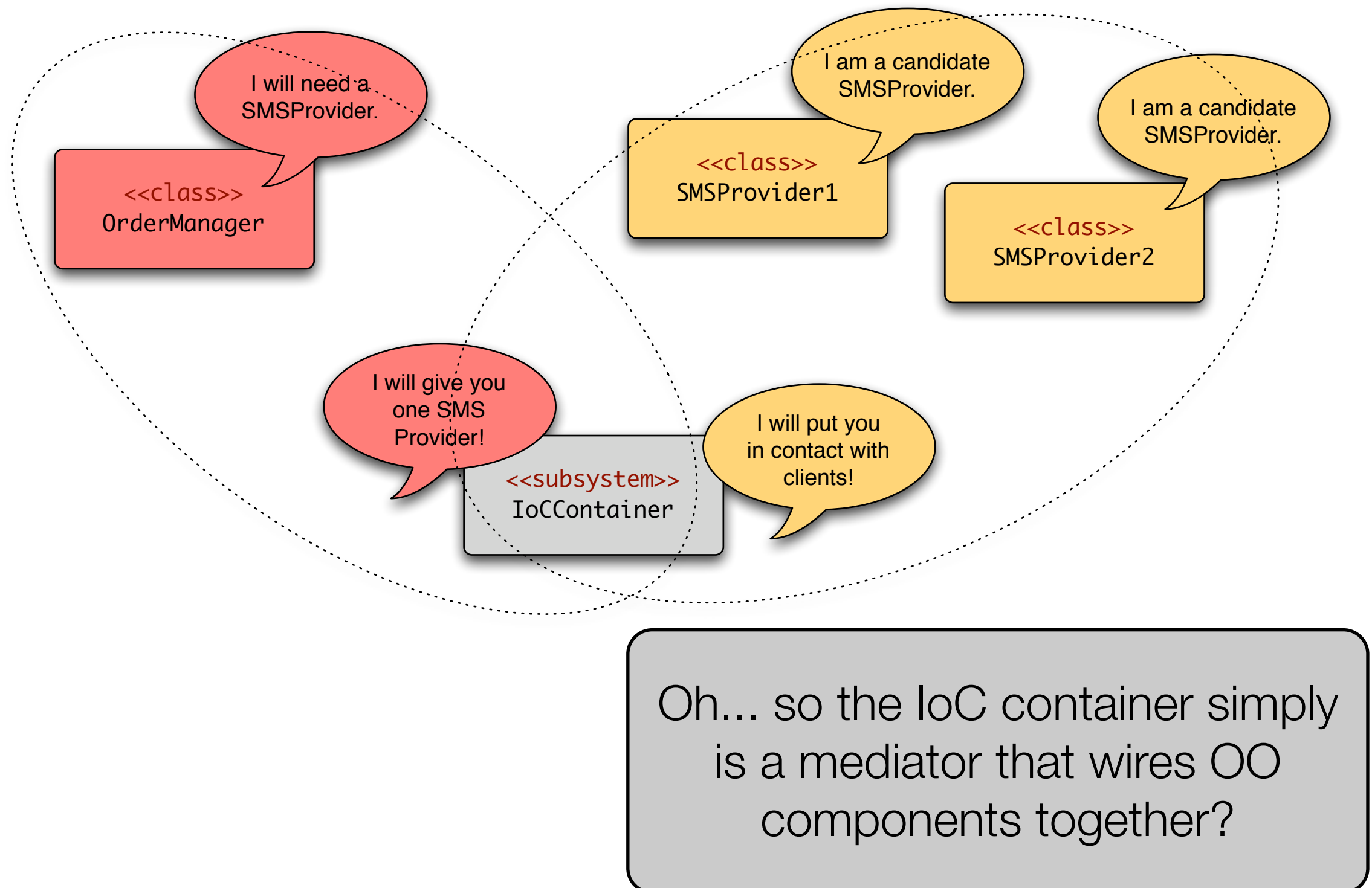
## Step 2: program against interfaces

---

```
public class OrderManager {  
    public void processOrder(Order o) {  
        ...  
        SMSProvider p1 = new SMSProvider1();  
        p1.sendSms(o.getCustPhoneNumber(), o.getTotal(), message);  
        ...  
    }  
}
```

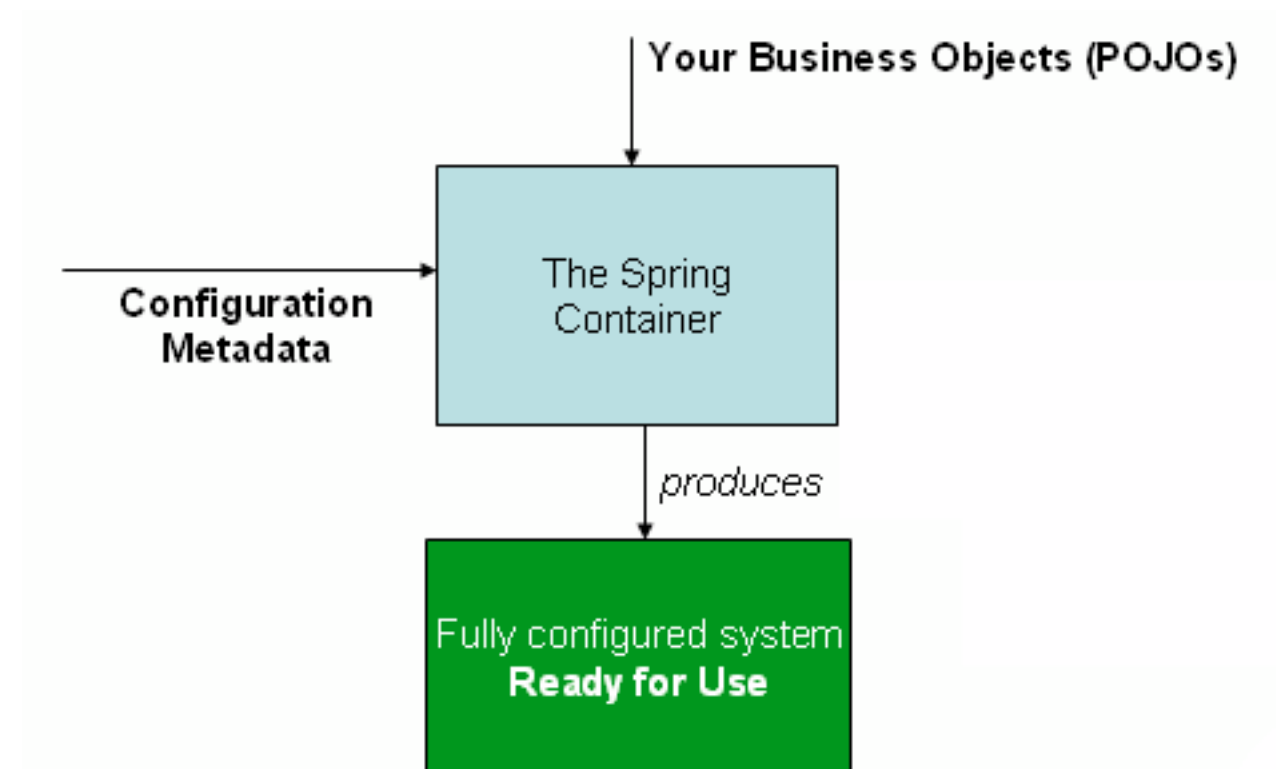
Not ideal... if we need to change the SMSProvider implementation, we need to go back to code.

# Step 3: use Inversion of Control (IoC)



# IoC in the Spring Framework

- The IoC Container manages “**Spring Beans**”, which are standard classes (POJOs).
- The IoC knows about all “**components**” or “**services**” defined in the system.
- The BeanFactory interface provides methods for interacting with the IoC Container.
- `ApplicationContext` extends `BeanFactory` and adds lots of “magic” behind the scenes (e.g. lifecycle management). **Use this!**



<http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html>

# BeanFactory vs ApplicationContext

---

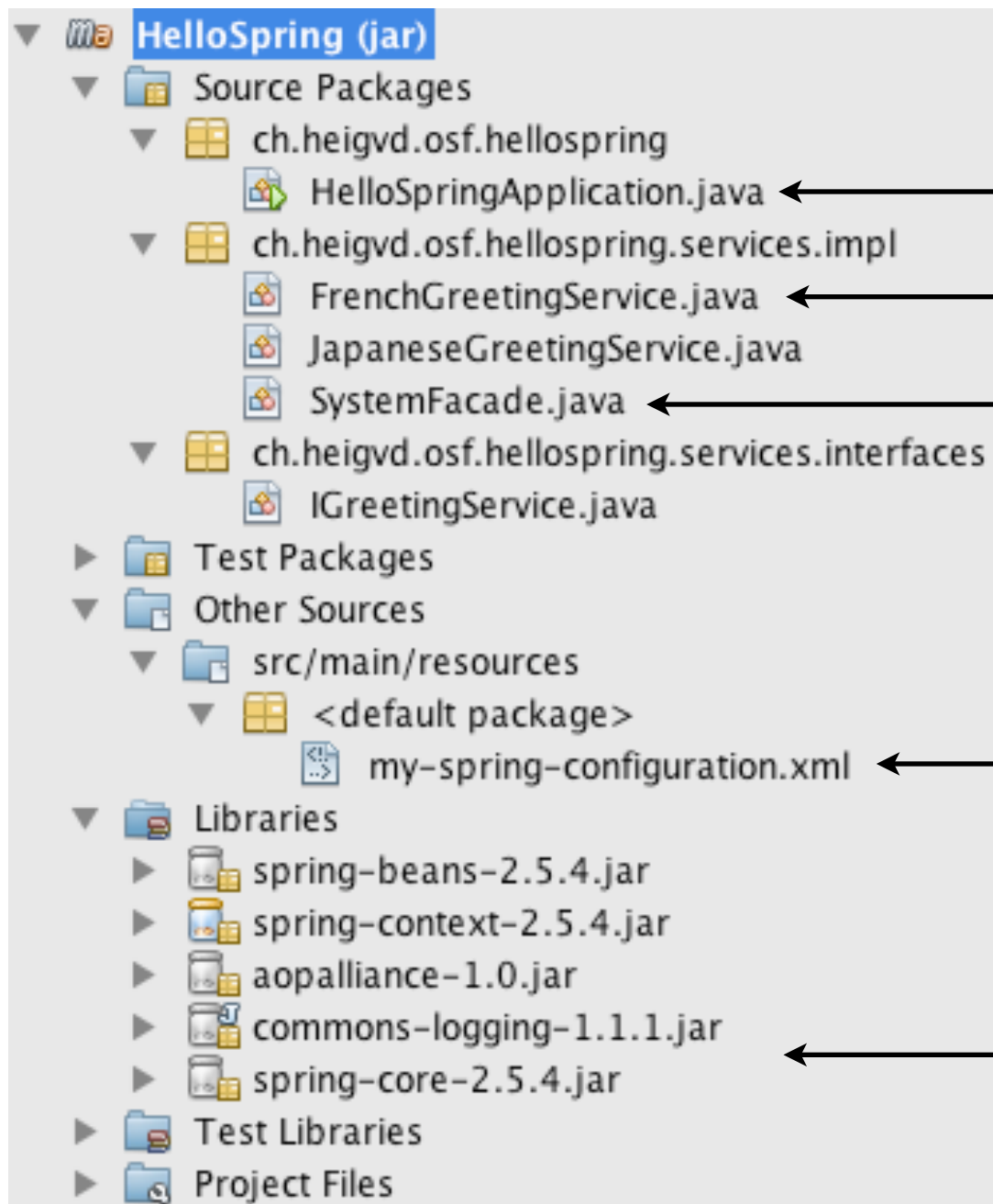
## BeanFactory or ApplicationContext?

Users are sometimes unsure whether a BeanFactory or an ApplicationContext is best suited for use in a particular situation. A BeanFactory pretty much just instantiates and configures beans. An ApplicationContext also does that, and it provides the supporting **infrastructure** to enable lots of enterprise-specific features such as **transactions** and **AOP**.

**In short, favor the use of an ApplicationContext.**

From the doc: <http://static.springsource.org/spring/docs/2.5.6/reference/beans.html>

# Sample project (Java SE)



the entry point that creates the IoC container

2 spring beans implementing the same interface

1 spring bean that needs a IGreetingService implementation

the configuration file that declares and wires the spring beans

the core spring libraries

# Sample project: the entry point

```
package ch.heigvd.osf.hellospring;

import ch.heigvd.osf.hellospring.services.impl.SystemFacade;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class HelloSpringApplication {

    private Log log = LogFactory.getLog(HelloSpringApplication.class);

    public void startApplication() {
        log.info("Starting HelloSpring application... Welcome!");
        ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{"my-spring-configuration.xml"});

        log.info("Getting a reference to the 'mySystem' bean (facade)");
        SystemFacade facade = (SystemFacade)context.getBean("mySystem");
        log.info("Invoking doStuff method on the facade; processing will be delegated to wired IGreetingService bean");
        facade.doStuff();

        log.info("Done.");
    }

    public static void main(String[] args) {
        new HelloSpringApplication().startApplication();
    }
}
```

This is where we launch the IoC container

The Spring Beans are accessible through the IoC container

# Sample project: the configuration file

```
<?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="aFrenchService" class="ch.heigvd.osf.hellospring.services.impl.FrenchGreetingService">
</bean>
```

```
<bean id="aJapaneseService" class="ch.heigvd.osf.hellospring.services.impl.JapaneseGreetingService">
</bean>
```

```
<bean id="mySystem" class="ch.heigvd.osf.hellospring.services.impl.SystemFacade">
  <property name="greetingService" ref="aFrenchService"/>
</bean>
```

```
</beans>
```

we use this name when we do the  
lookup in the Java code

```
SystemFacade facade = (SystemFacade)
    context.getBean("mySystem");
```

this is a “dependency injection”: we  
provide a IGreetingService  
implementation to the mySystem bean



# Sample project: bean implementation

```
package ch.heigvd.osf.hellospring.services.impl;

import ch.heigvd.osf.hellospring.services.interfaces.IGreetingService;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

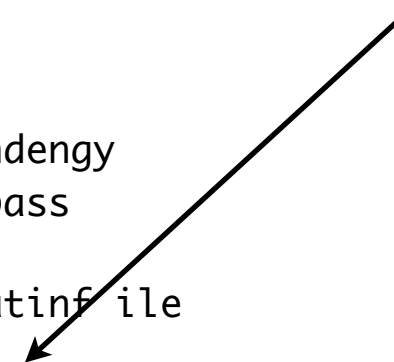
public class SystemFacade {

    private Log log = LogFactory.getLog(IGreetingService.class);
    private IGreetingService greetingService;

    /**
     * Setter for the greetingService property. This is what makes dependency
     * injection possible: the IoC container will call this method and pass
     * an instance of the bean defined in the XML configuration file
     * @param greetingService a spring bean defined in the XML configuration file
     */
    public void setGreetingService(IGreetingService greetingService) {
        this.greetingService = greetingService;
    }

    /**
     * This method will be called from the application. It does not do much, except
     * for delegating processing to the wired beans (thank you dependency injection)
     */
    public void doStuff() {
        log.info("System facade invoked... delegating work to the wired greeting service");
        log.info("Wired greeting service says: " + greetingService.greet());
        log.info("System facade done with processing.");
    }
}
```

This method enables “setter” based  
dependency injection



# Spring IoC Configuration

- Different ways to declare Spring Beans, both in XML and through annotations.
- These different ways have been added over time, so they are not available in all versions of the Spring Framework.
- Essentially, you need to declare your components, how they are instantiated and how they depend on each other.
- For some frameworks, you have lots of components. XML schemas make configuration less verbose.

`<bean></bean>`

XML with  
DTD

`<mvc:view-controller>`

XML with  
schemas

bean wiring

Annotation  
based

bean declaration

Java based

# XML Schema Based Configuration

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
    <list>
      <value>pechorin@hero.org</value>
      <value>raskolnikov@slums.org</value>
      <value>stavrogin@gov.org</value>
      <value>porfiry@gov.org</value>
    </list>
  </property>
</bean>
```



before

```
<!-- creates a java.util.List instance with the supplied values -->
<util:list id="emails">
  <value>pechorin@hero.org</value>
  <value>raskolnikov@slums.org</value>
  <value>stavrogin@gov.org</value>
  <value>porfiry@gov.org</value>
</util:list>
```



after

# XML Schema Based Configuration

---

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/MyDataSource"/>
</bean>

<bean id="userDao" class="com.foo.JdbcUserDao">
  <!-- Spring will do the cast automatically (as usual) -->
  <property name="dataSource" ref="dataSource"/>
</bean>
```

before

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/MyDataSource"/>

<bean id="userDao" class="com.foo.JdbcUserDao">
  <!-- Spring will do the cast automatically (as usual) -->
  <property name="dataSource" ref="dataSource"/>
</bean>
```

after

# XML Schema Based Configuration

---

```
<bean id="simple"  
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">  
  <property name="jndiName" value="ejb/RentalServiceBean"/>  
  <property name="businessInterface" value="com.foo.service.RentalService"/>  
</bean>
```

before

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"  
               business-interface="com.foo.service.RentalService"/>
```

after

# Want to integrate your framework?

---

- Appendix D in the Spring Framework documentation tells you
  - how to specify your own XML schema
  - how to write handlers to process XML configuration files

## D. Extensible XML authoring

### D.1. Introduction

### D.2. Authoring the schema

### D.3. Coding a NamespaceHandler

### D.4. Coding a BeanDefinitionParser

### D.5. Registering the handler and the schema

#### D.5.1. 'META-INF/spring.handlers'

#### D.5.2. 'META-INF/spring.schemas'

### D.6. Using a custom extension in your Spring XML configuration

### D.7. Meatier examples

#### D.7.1. Nesting custom tags within custom tags

#### D.7.2. Custom attributes on 'normal' elements

### D.8. Further Resources

# Spring Framework - AOP

---

- **Aspect Oriented Programming**
  - Separation of concerns, cross-cutting concerns
  - Terminology
  - AOP frameworks
- **Aspect Oriented Programming & Java EE**
  - Declarative enterprise services
  - EJBs and interceptors
- **AOP with Spring**
  - Spring AOP vs. Spring with AspectJ
  - XML vs. Annotations

# Aspect Oriented Programming (AOP)

---

- In all applications, there are “things” that need to be done over and over and that are **orthogonal** to **business logic**.
- Examples:
  - Logging and auditing
  - Security checks (authorization)
  - Transaction management
- In traditional object-oriented design, the common approach is to implement the pure business logic and these orthogonal functions **at the same place** (in class methods).

***Separation of concerns: business logic vs. other “aspects”***



# AOP Frameworks

---

- AspectJ created at Xerox PARC in 2001 (Gregor Kiczales)
- Several other frameworks and projects have been developed (e.g. AspectWerkz), for different languages.
- The Spring Framework makes it possible to use AOP concepts and relies itself on AOP for some of its features.
- As always with Spring, there are lots of different ways to use AOP (pure Spring vs. AspectJ integration, XML vs. annotations, etc.)

**aspectj** *crosscutting objects for better modularity*

<http://eclipse.org/aspectj/>

# AOP & Java EE

---



Remember the  
notion of “container”



Remember  
how transaction and  
security services are  
provided?



Remember EJB3  
interceptors?

# Aspect Oriented Programming (AOP)

- Where is my business logic? It's hard to find... What do I have to bother with all these infrastructure concerns?
- How can I get a global view for security management in my application?
- What if I need to change the way I do the auditing? I will have to go in every single method...
- *What a nightmare!!*



```
<<class>>  
ProductManager
```

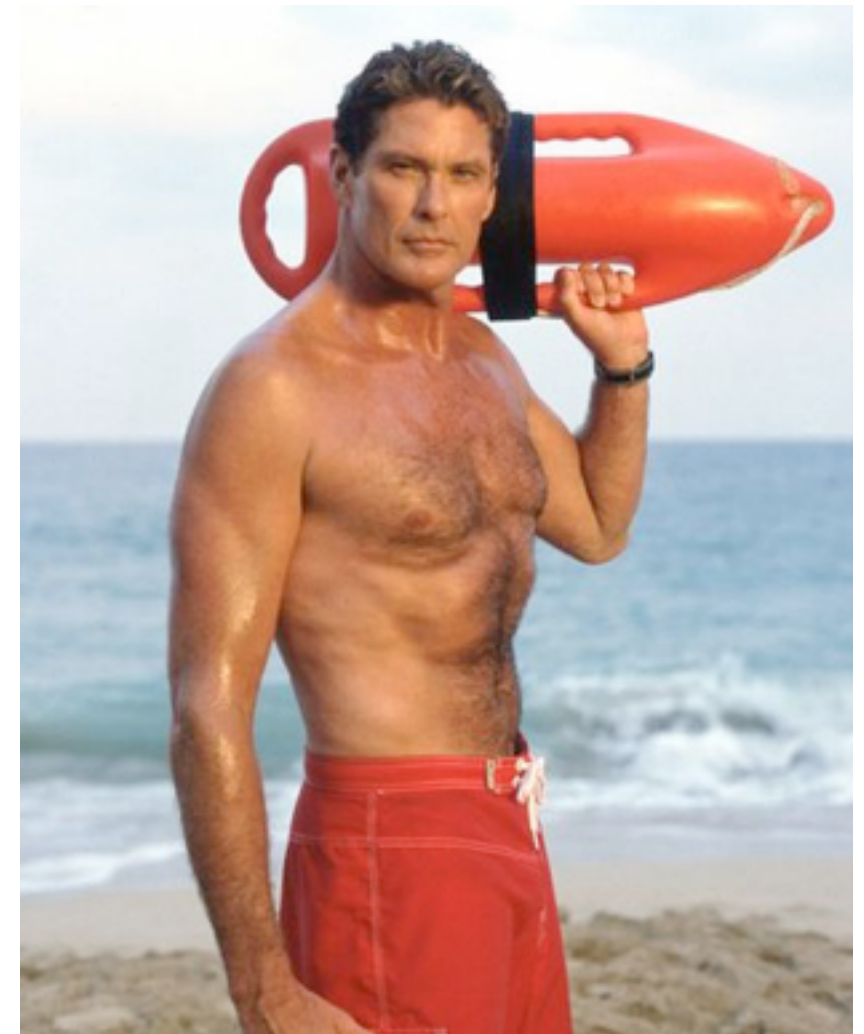
```
public void addProduct(Product p) {  
    // check if the user is authenticated and authorized  
    ...  
    // start transaction  
    ...  
    // finally, some business logic  
    ...  
    // commit transaction  
    ...  
    // leave a trace in the audit trail  
    ...  
}
```

```
public void removeProduct(Product p) {  
    // check if the user is authenticated and authorized  
    ...  
    // start transaction  
    ...  
    // finally, some business logic  
    ...  
    // commit transaction  
    ...  
    // leave a trace in the audit trail  
    ...  
}
```

# AOP to the rescue

---

- **AOP supports the separation of concerns.** In other words, it gives a way to split the implementation of the business logic from the implementation of system-level functions.
- **Terminology**
  - An **aspect** or **cross-cutting concern** refers to **something** that needs to be done throughout the application code. Security, logging and transaction management are examples of cross-cutting concerns.
  - An **advice** is the **orthogonal logic** that is executed when a certain join point is executed (advice can be executed **before**, **after** or **around** the join point).
  - A **pointcut** is an **expression** used to define a set of join points. With a pointcut, one can specify which join points (i.e. which methods)
  - A **join point** defines **when** the orthogonal logic could be executed. For instance, the execution of a `processOrder()` **method** is a join point.



# AOP to the rescue

- **AOP supports the separation of concerns.** In other words, it gives a way to split the implementation of the business logic from the implementation of system-level functions.

- **Terminology**

- An **aspect** or **cross-cutting concern** refers to **something** that needs to be done throughout the application code. Security, logging and ... are examples of cross-cutting concerns.

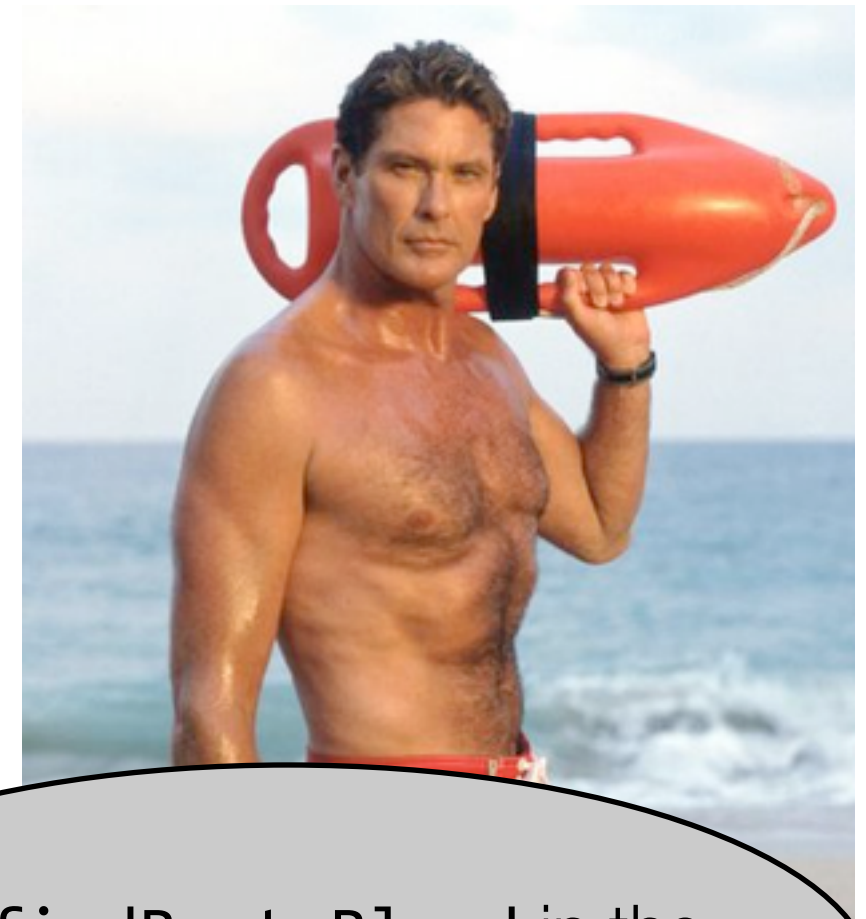
- An **advice** is the code that is executed at certain join points. It can be executed **before, after** or **around** the join point.

All methods that start with  
“find” in the `ch.heigvd.osf`  
package

- A **pointcut** is an **expression** used to define a set of join points. With a pointcut, one can specify which join points (i.e. which methods)

- A **join point** defines **when** the orthogonal logic could be executed. For instance, the execution of a process or a **method** is a join point.

the `findBustyBlond` in the  
`ch.heigvd.osf.BayWatch` class





# Join Point

```
<<class>>
```

```
ch.heigvd.osf.service.OrderManager
```

```
public void processOrder(Order o) {}
public void processOrder(Order o, Params p) {}
public void processOrder(Order o, Helper h) {}
public void cancelOrder(Order o) {}
public void cancelOrder(Order o, Params p) {}
public Stats getStatistics() {}
public List listLargeOrders() {}
```

These are all join points...

```
<<class>>
```

```
ch.heigvd.osf.service.ProductManager
```

```
public void createProduct(Product p) {}
public void updateProduct(Product p) {}
public void deleteProduct(Product p) {}
public void listAllProducts() {}
public List listExpensiveProducts() {}
```

...surprise, surprise,  
these too!

# Pointcut

---

Pointcuts can be declared with an annotation (or with XML...)

```
@PointCut(expression)
private void aNameForThisSetOfMethods {}
```

The **expression** is based on the AspectJ pointcut language. Here are some examples:

the execution of any public method:

```
execution(public * *(..))
```

the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

the execution of any method defined by the AccountService interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

the execution of any method on a Spring bean named 'tradeService':

```
bean(tradeService)
```

the execution of any method on a Spring bean with a name matching the wildcard expression

```
bean(*Service)
```

`execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)`

<http://static.springsource.org/spring/docs/2.5.6/reference/aop.html#aop-pointcuts>

# How Can it Work?

---

- There are different ways to implement AOP.
- Remember that we want to “**combine**” two pieces of orthogonal code - located in two different artifacts (a “business” class and an “advice class”).
- One possibility is to use a **special compilation process**. This is called “**weaving**”, since the aspect code is weaved into the main business logic. As an alternative, it is possible to do the weaving as an **after-compilation** process. “Weaving” is what the AspectJ framework and toolset is doing.
- Another approach is to use proxies that are dynamically generated. This is what Spring is doing - so there is no special compilation process

**Note:** Spring Roo, the RAD tool that we mentioned last week uses AspectJ to separate developer code from Roo-generated code. Spring Roo actually uses a special feature of AspectJ called Load-Time Weaving, so it does not require a special compiling process.



# AOP in the Spring Framework

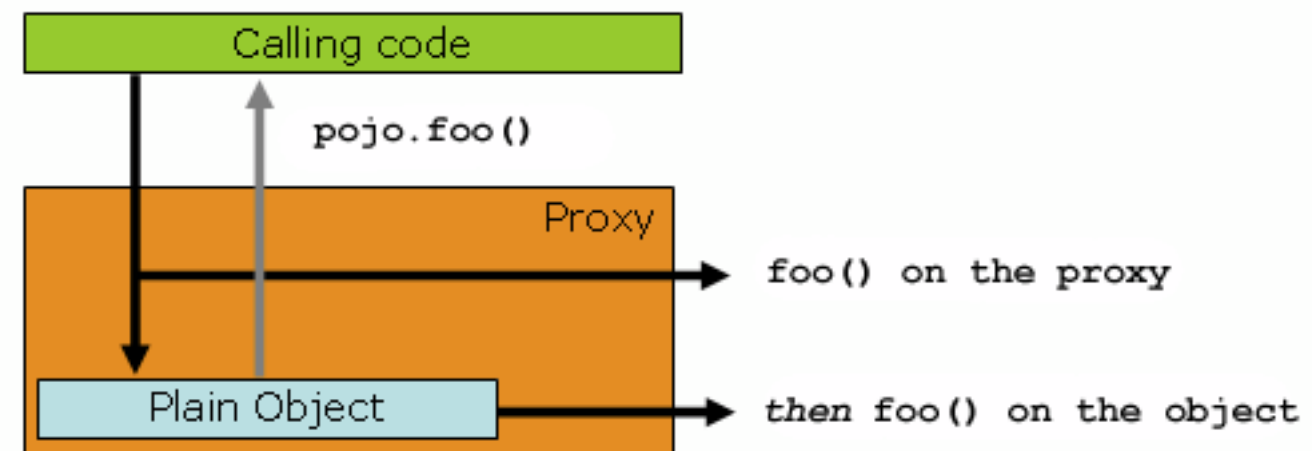
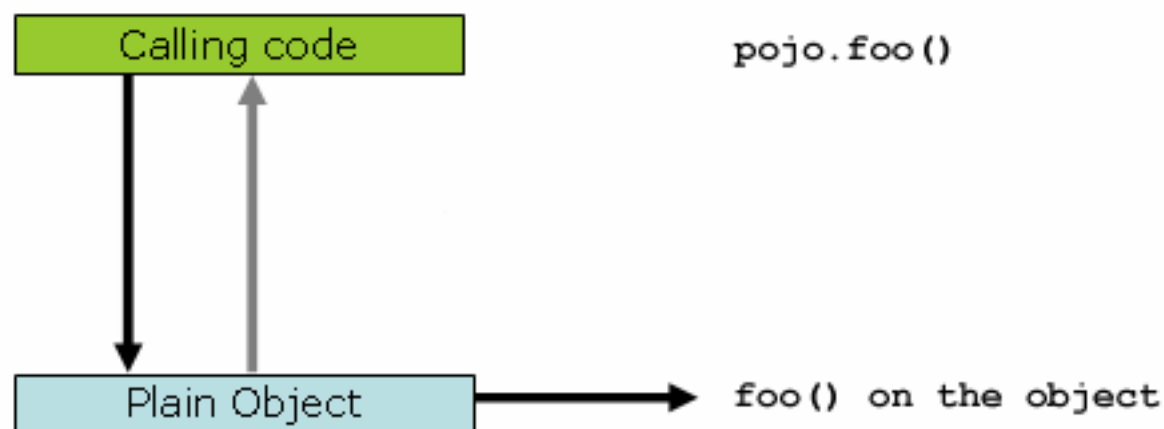
---

- AOP is used in the Spring Framework to:
  - provide **declarative** enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative **transaction management**
  - allow users to implement **custom aspects**, complementing their use of OOP with AOP

*“If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP, and can skip most of this chapter.”*

# Spring AOP Capabilities and Goals (1)

- Spring AOP is implemented in **pure Java**
- There is **no need for a special compilation process**
- Spring AOP does not need to control the class loader hierarchy, and is thus **suitable for use in a Java EE web container or application server**



# Spring AOP Capabilities and Goals (2)

---

- Spring AOP currently supports only **method execution join points** (advising the execution of methods on Spring beans).
- Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs.
- If you need to advise field access and update join points, **consider a language such as AspectJ**.

# Spring AOP Capabilities and Goals (3)

---

- Spring AOP's approach to AOP differs from that of most other **AOP frameworks**.
- The aim is **not** to provide the most complete AOP implementation (although Spring AOP is quite capable);
- It is rather to provide a **close integration** between AOP implementation and Spring IoC to help solve common problems in enterprise applications.
- Spring AOP will never strive to compete with **AspectJ** to provide a comprehensive AOP solution.

# @AspectJ support

---

- @AspectJ refers to a **style of declaring aspects** as regular Java classes annotated with Java 5 annotations.
- The @AspectJ style was introduced by the **AspectJ project** as part of the AspectJ 5 release.
- Spring 2.0 interprets **the same annotations as AspectJ 5**, using a library supplied by AspectJ for pointcut parsing and matching.
- The AOP runtime is still pure Spring AOP though, and there is **no dependency on the AspectJ compiler or weaver**.

# Spring AOP or full AspectJ?

---

- **Use the simplest thing that can work.**
- **Spring AOP is simpler than using full AspectJ** as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes.
- If you only need to **advise the execution of operations on Spring beans**, then Spring AOP is the right choice.
- If you need to **advise objects not managed by the Spring container** (such as domain objects typically), then you will need to use **AspectJ**.
- You will also need to use **AspectJ** if you wish to advise join points other than simple method executions (for example, **field get or set join points**, and so on).
- When using **AspectJ**, you have the choice of the **AspectJ** language syntax (also known as the "code style") or the **@AspectJ** annotation style.

# @AspectJ or XML for Spring AOP

---

- If you have chosen to use Spring AOP, then you have a **choice** of **@AspectJ** or **XML style**.
- Clearly if you are not running on Java 5+, then the XML style is the appropriate choice; for Java 5 projects there are **various tradeoffs** to consider.
  - The XML style will be most familiar to existing Spring users.
  - It can be used with any JDK level.
  - The XML style is slightly more limited in what it can express than the @AspectJ style
  - The @AspectJ style has the advantage of keeping the aspect as a modular unit.

Example



# Project Setup



```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>3.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>3.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>com.springsource.org.aspectj.runtime</artifactId>
  <version>1.6.6.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.6.6</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>1.6.6</version>
</dependency>
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>2.2</version>
</dependency>
```

# Pointcut

---

Pointcuts can be declared with an annotation (or with XML...)

```
@PointCut(expression)
private void aNameForThisSetOfMethods {}
```

The **expression** is based on the AspectJ pointcut language. Here are some examples:

the execution of any public method:

```
execution(public * *(..))
```

the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

the execution of any method defined by the AccountService interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

the execution of any method on a Spring bean named 'tradeService':

```
bean(tradeService)
```

the execution of any method on a Spring bean with a name matching the wildcard expression

```
bean(*Service)
```

`execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)`

<http://static.springsource.org/spring/docs/2.5.6/reference/aop.html#aop-pointcuts>

# Defining a Pointcut “Inline”

```
@Aspect
public class MyFirstAspect {

    @Before("execution(public * ch.heigvd.osf..*.*(..))")
    public void myMethod(JoinPoint jp) {
        System.out.println("My advice has been applied...");
        System.out.println("target: " + jp.getTarget());
        System.out.println("this: " + jp.getTarget());
        System.out.println("signature: " + jp.getSignature());
    }
}
```

```
<aop:aspectj-autoproxy/>
```

```
<bean id="myFirstAspect" class="ch.heigvd.osf.hellospringaop.aspects.MyFirstAspect">
</bean>
```

## Notes:

- myMethod will be executed before **any public method** in **any class** in the `ch.heigvd.osf` package (or in a **sub-package**) is called.
- myMethod has access to runtime information

# Using an @Aspect to Define Pointcuts

```
package ch.heigvd.osf.system;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemPointCuts {

    @Pointcut("execution(* create*(..))")
    public void createMethods() {}

    @Pointcut("execution(* update*(..))")
    public void updateMethods() {}

    @Pointcut("execution(* delete*(..))")
    public void deleteMethods() {}

    @Pointcut("createMethods() && updateMethods() && deleteMethods()")
    public void allCRUDMethods() {}

}
```

# Using an @Aspect to Implement Advices

```
package ch.heigvd.osf.system.logging;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class MyLoggingAspect {
    @Before("ch.heigvd.osf.system.SystemPointCuts.allCRUDMethods()")
    public void doLogOperation() {
        log.info("About to call a CRUD method....");
    }
}
```

Here, we work with:

- one pointcut, which is defined in the SystemPointCuts aspect (see previous slide)
- this pointcut defines a set of several join points: all the methods with a name starting with either create, update or delete
- one advice, which states that before every execution of the join points matching the pointcut, we will execute the doLogOperation

# References

---

- <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html>
- <http://www.javalobby.org/java/forums/t44746.html>
- [http://en.wikipedia.org/wiki/Aspect-oriented\\_programming](http://en.wikipedia.org/wiki/Aspect-oriented_programming)

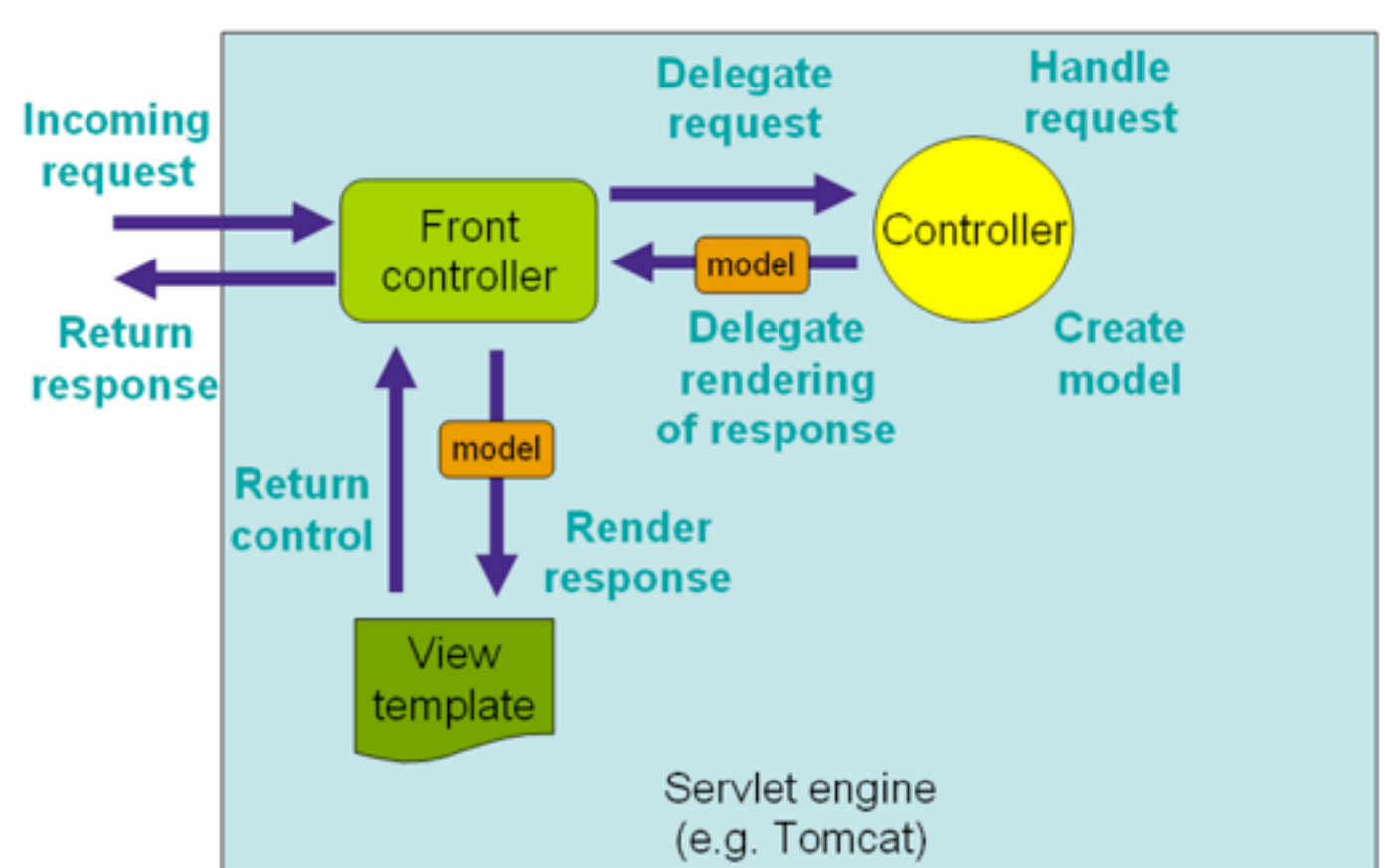
# Spring and the Web Tier

---

- **Two different Spring components**
  - Spring MVC
  - Spring Web Flow
- **Integration with third party frameworks and technologies**
  - Struts
  - Tapestry
  - JSF
  - ...
- **Support for Portlets (JSR 168, JSR 286)**

# Spring MVC

- Remember the MVC pattern in the first OSF lecture?
- Spring provides a FrontController - it's called the DispatcherServlet.
- It is what enables the flexible configuration of your system.
- Spring MVC supports different view technologies. You can use JSPs, but also Velocity or XSLT views.



The Dispatcher Servlet

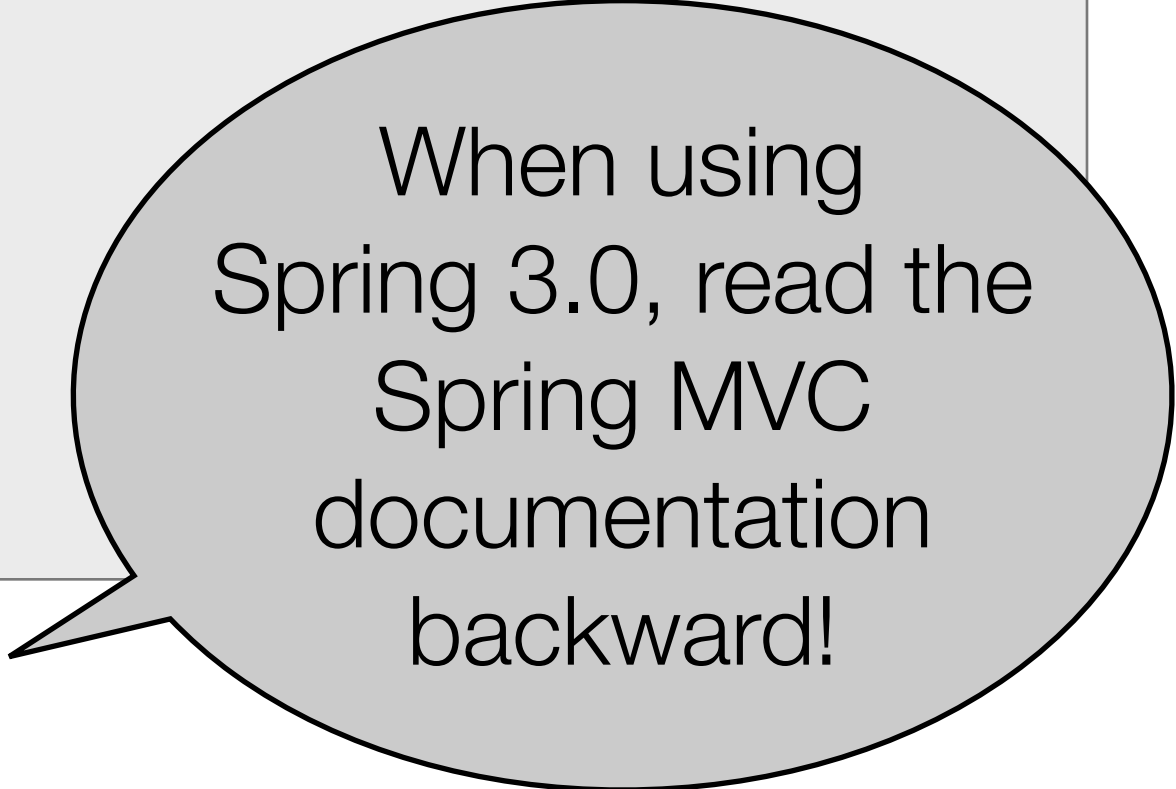


# Implementing Controllers

- Again, the way you will do that depend on the version of the Spring Framework you are using (XML vs annotations).
- Today, you can do every thing with annotations, in a style that reminds JAX-RS (but it is not the same!)

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public ModelAndView helloWorld() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("helloWorld");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}
```



When using Spring 3.0, read the Spring MVC documentation backward!

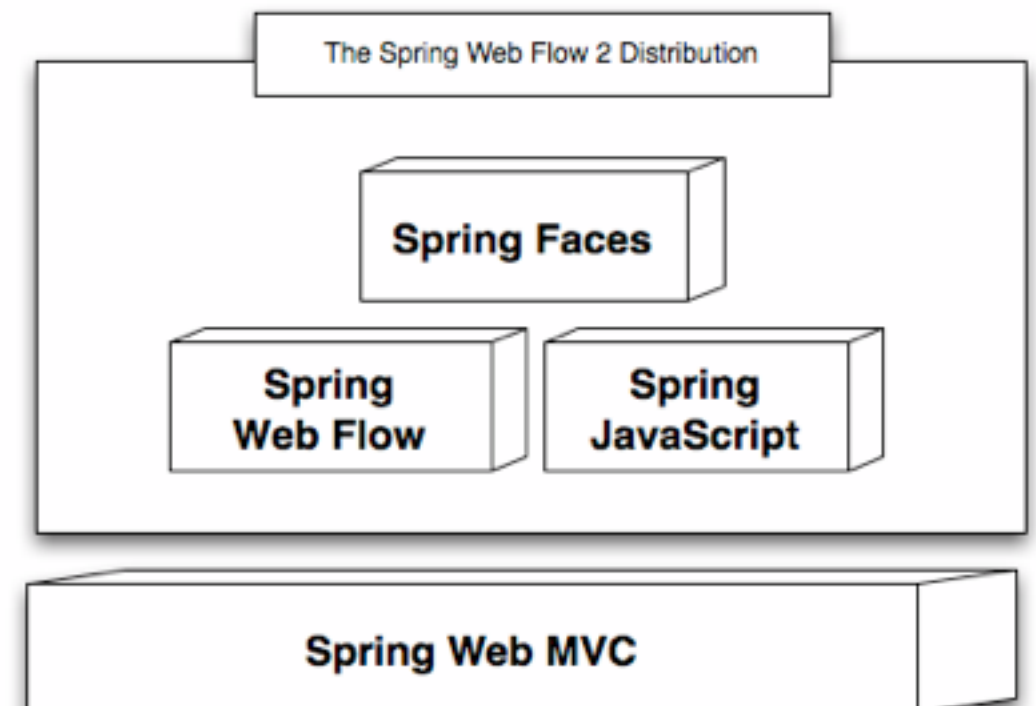
# Spring MVC vs Spring Web Flow

- **Spring MVC**

- “Traditional” MVC framework, which decouples the views and the business logic.
- FrontController on steroids
- Evolved from XML-based configuration to annotation-based configuration (different but similar to JAX-RS...)

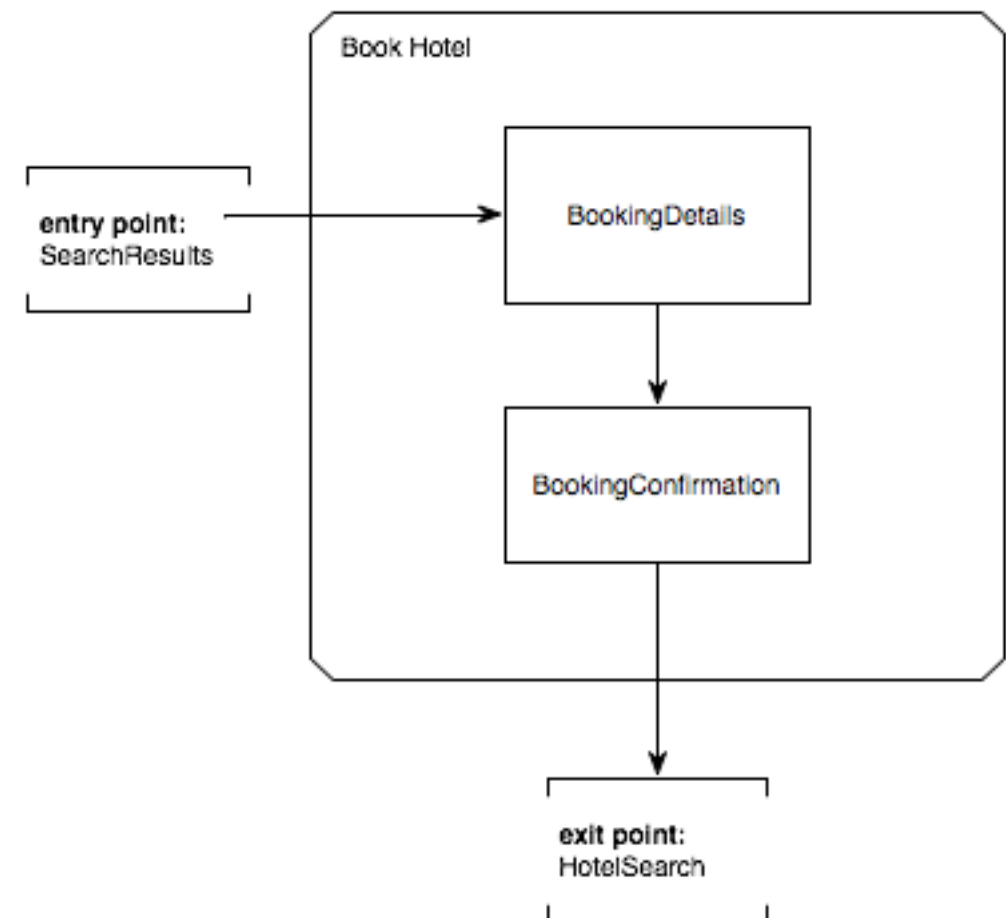
- **Spring Web Flow**

- Used for “conversations”
- View transitions expressed in a configuration file
- Conversation scope to deal with multiple tabs.

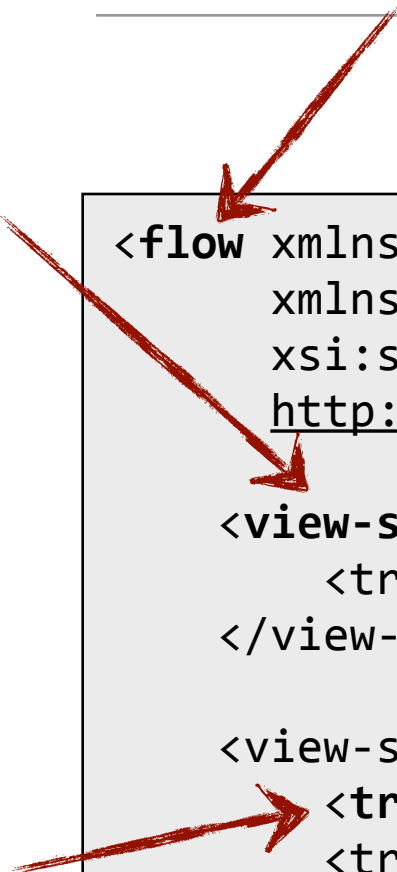


# Spring Web Flow

- **What is a flow?**
  - A set of states
  - A set of transitions between states
  - Views that need to be displayed when entering a state
  - Actions that need to be performed when entering or leaving a state.



# Spring Web Flow



```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
      http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

  <view-state id="enterBookingDetails">
    <transition on="submit" to="reviewBooking" />
  </view-state>

  <view-state id="reviewBooking">
    <transition on="confirm" to="bookingConfirmed" />
    <transition on="revise" to="enterBookingDetails" />
    <transition on="cancel" to="bookingCancelled" />
  </view-state>

  <end-state id="bookingConfirmed" />

  <end-state id="bookingCancelled" />

</flow>
```

When HTTP requests processed by the framework trigger the “events” referenced in the flow definition. With Spring MVC, everything is handled behind the scenes.

# Spring Web Flow

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
      http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <input name="hotelId" />

    <on-start>
        <evaluate expression="bookingService.createBooking(hotelId, currentUser.name)"
            result="flowScope.booking" />
    </on-start>

    <view-state id="enterBookingDetails">
        <transition on="submit" to="reviewBooking" />
    </view-state>

    <view-state id="reviewBooking">
        <transition on="confirm" to="bookingConfirmed" />
        <transition on="revise" to="enterBookingDetails" />
        <transition on="cancel" to="bookingCancelled" />
    </view-state>

    <end-state id="bookingConfirmed" />

    <end-state id="bookingCancelled" />

</flow>
```

# Spring Web Flow - Setup

---

- Spring Web Flow is configured via the Spring IoC
- Lots of flexibility, but you can start with default behavior
- One thing you need to do is to indicate where your flow definitions are:
  - by explicitly referencing a definition file (e.g. `booking.xml`)
  - by specify a pattern that will be used to find definition files (e.g. `*-flow.xml`)

```
<webflow:flow-registry id="flowRegistry">  
  <webflow:flow-location path="/WEB-INF/flows/booking/booking.xml" />  
</webflow:flow-registry>
```

```
<webflow:flow-location-pattern value="/WEB-INF/flows/**/*-flow.xml" />
```