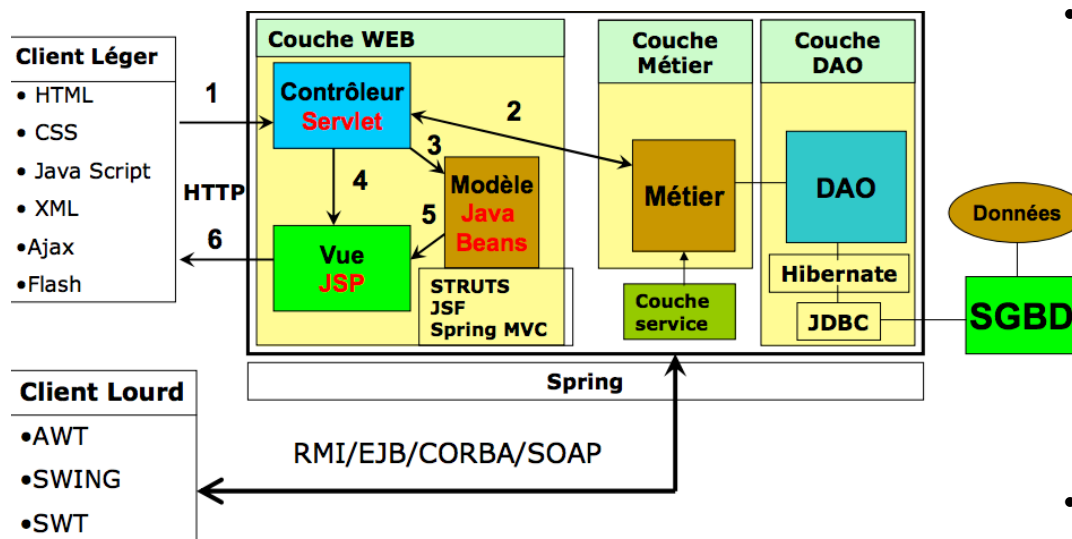


Chapitre 6 : Spring framework et les design patterns

Rappel de l'architecture logicielle des Apps JEE



- La couche [**dao**] s'occupe de l'accès aux données, le plus souvent des données persistantes au sein d'un SGBD.
- La couche [**métier**] implémente les traitements " métier " de l'application. Cette couche est indépendante de toute forme d'interface avec l'utilisateur.
 - C'est généralement la couche la plus stable de l'architecture.
 - Elle ne change pas si on change l'interface utilisateur ou la façon d'accéder aux données nécessaires au fonctionnement de l'application.
- La couche [**Web**] qui représente les interfaces (graphiques souvent) qui permet à l'utilisateur de piloter l'application et d'en recevoir des résultats ainsi que la partie contrôle qui orchestre les accès à l'application.

Problématiques de développement JEE

- ❑ JEE n'encourage pas une bonne séparation des préoccupations, c'est-à-dire l'isolation des différentes problématiques qu'une application doit gérer.
- ❑ C'est une plate-forme complexe à maîtriser, qui pose des problèmes de productivité.
- ❑ Une grande part des développements sont consacrés aux problématiques techniques.
- ❑ JEE impose une plate-forme d'exécution lourde, qui pose des problèmes d'interopérabilité entre différentes implémentations.
- ❑ Les développements utilisant JEE s'avèrent souvent difficiles à tester et faire évoluer.

N.B.: Pour plus de détails sur ces problèmes (Livre : Spring pour la pratique)

La réponse Spring !

- ❑ Pour résoudre ces problèmes que nous venons d'évoquer, des solutions ont émergé (Struts ...).
 - ❑ Spring Framework est l'une des solutions,
 - ❑ En rupture avec les conteneurs JEE qui sont disqualifiés par de nombreux experts pour leur lourdeur, Spring est qualifié d'un conteneur dit **léger**,
 - ❑ Cette solution a été étendue de manière à supporter la POA (**programmation orientée aspect**), ou AOP (Aspect Oriented Programming), un nouveau paradigme de programmation permettant d'aller au-delà de l'approche objet en terme de modularisation des composants.



Programmation orientée aspect

- La **programmation orientée aspect**, ou **POA** est un [paradigme](#) de [programmation](#) qui permet de traiter séparément les [préoccupations transversales](#) (en anglais, [cross-cutting concerns](#)), qui relèvent souvent de la technique, des préoccupations métier, qui constituent le cœur d'une application^{[1](#),[trad 1](#),[trad 2](#)}. Un exemple classique d'utilisation est la [journalisation](#), mais certains principes architecturaux ou [modèles de conception](#) peuvent être implémentés à l'aide de ce paradigme de programmation, comme l'[inversion de contrôle](#)^{[trad 3](#)}.

Présentation de Spring

- ❑ SPRING est un conteneur dit « léger », c'est-à-dire une infrastructure similaire à un serveur d'application JEE.
- ❑ Il prend en charge la création d'objets et la mise en relation d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets.
- ❑ Le gros avantage par rapport aux serveurs d'application est qu'avec SPRING, vos classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'applications JEE et des EJBs).
- ❑ C'est en ce sens que SPRING est qualifié de conteneur « léger ».

Présentation de Spring

❑ Outre cette espèce de super fabrique d'objets, SPRING propose tout un ensemble d'abstractions permettant de gérer entre autres :

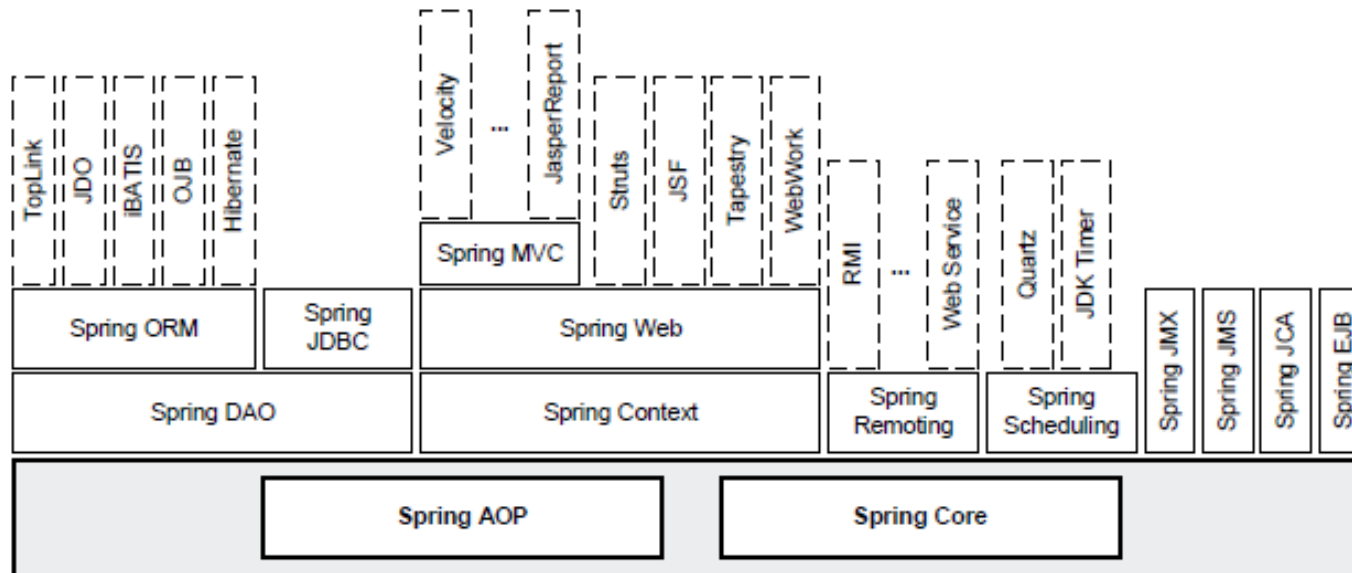
- Le mode transactionnel.
- L'appel d'EJBs.
- La création d'EJBs.
- La persistance d'objets
- La création d'une interface Web.
- L'appel et la création de WebServices.

❑ Pour réaliser tout ceci, SPRING s'appuie sur les principes du design patterns, l'IoC et sur la programmation par aspects (AOP).

Les modules de Spring

Comme nous pouvons le voir, Spring repose sur un socle technique constitué des modules :

- Spring Core, le conteneur léger ;
- Spring AOP, le framework de POA.



Les patrons de conception

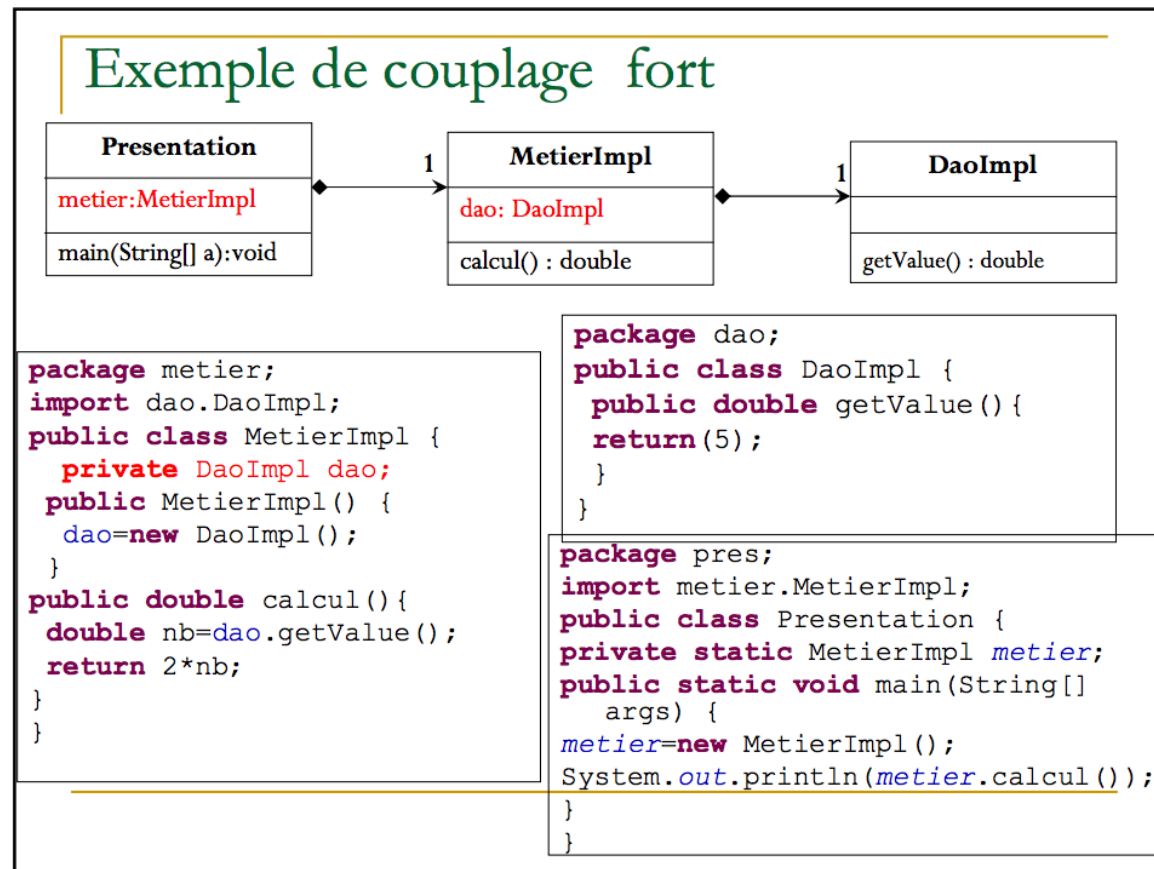
- ❑ En anglais, on parle de **Design Pattern**.
- ❑ Spring repose sur des concepts éprouvés, patrons de conception et paradigmes, dont les plus connus sont
 - IoC (Inversion of Control),
 - le patron « fabrique »
 - le singleton,
 - la programmation orientée Aspect,
 - ou encore le modèle de programmation dit "par template".
- ❑ Ces concepts n'étant pas propre à Spring, ils s'appliquent également à d'autres frameworks.

IoC

Problème de Couplage fort

- Quand une classe A est liée à une classe B, on dit que la classe A est fortement couplée à la classe B.
- La classe A ne peut fonctionner qu'en présence de la classe B.
- Si une nouvelle version de la classe B (soit B2), est créée, on est obligé de modifier dans la classe A.
- Modifier une classe implique:
 - Il faut disposer du code source.
 - Il faut recompiler, déployer et distribuer la nouvelle application aux clients. Ce qui engendre des problèmes de la maintenance de l'application

Couplage fort : exemple

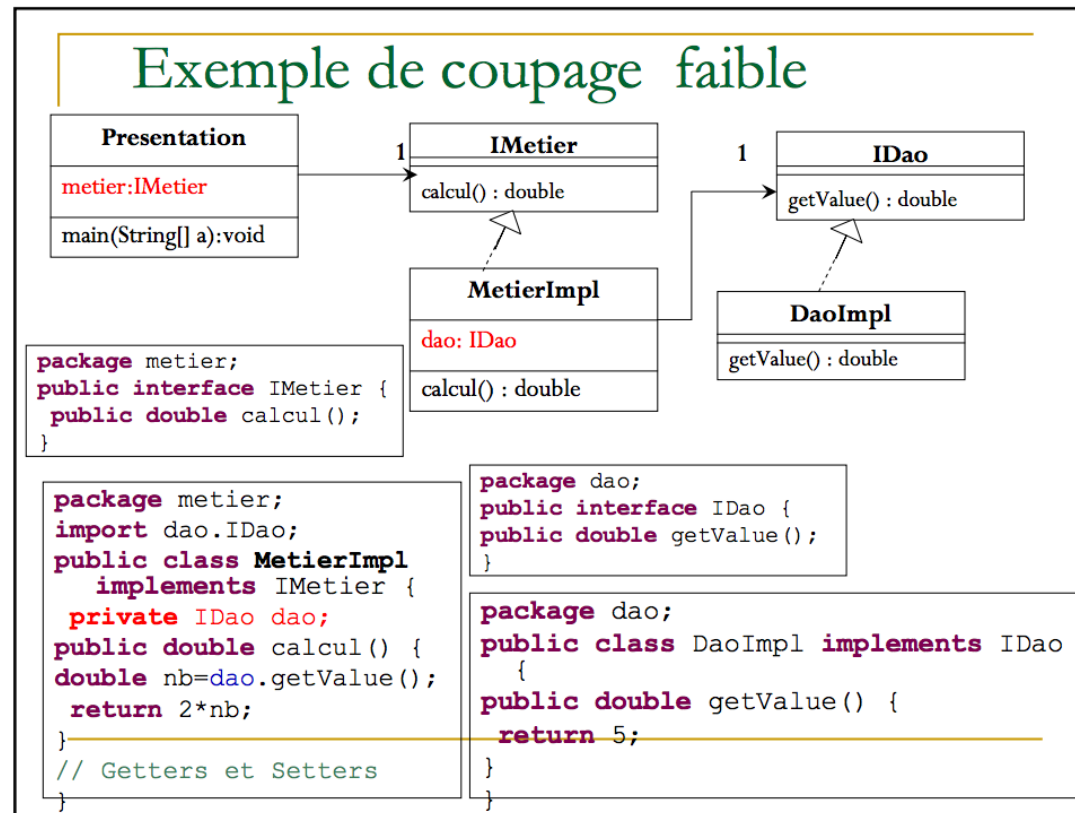


De ce fait nous avons violé le principe « une application doit être fermée à la modification et ouverte à l'extension »

Couplage faible

- Pour utiliser le couplage faible, nous devons utiliser les interfaces. Considérons une classe A qui implémente une interface IA, et une classe B qui implémente une interface IB.
- Si la classe A est liée à l'interface IB par une association, on dit que la classe A et la classe B sont liées par un couplage faible.
- Cela signifie que la classe A peut fonctionner avec n'importe quelle classe qui implémente l'interface IB.
- En effet la classe A ne connaît que l'interface IB. De ce fait n'importe quelle classe implémentant cette interface peut être associée à la classe A, sans qu'il soit nécessaire de modifier quoi que se soit dans la classe A.
- Avec le couplage faible, nous pourrions créer des application fermée à la modification et ouvertes à l'extension.

Couplage faible : exemple

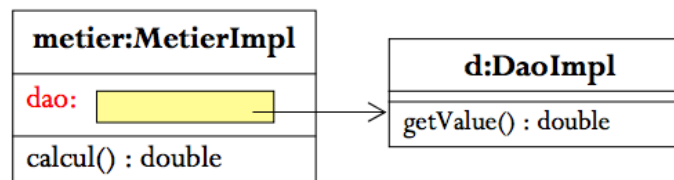


Injection des dépendances avec Spring

- L'injection des dépendance, ou l'inversion de contrôle est un concept qui intervient généralement au début de l'exécution de l'application.
- Spring IOC commence par lire un fichier XML qui déclare quelles sont les différentes classes à instancier et d'assurer les dépendances entre les différentes instances.
- Quand on a besoin d'intégrer une nouvelle implémentation à une application, il suffirait de la déclarer dans le fichier xml de beans spring.

Injection des dépendances dans une application java standard

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-
    2.0.dtd" >
<beans>
  <bean id="d" class="dao.DaomImpl2"></bean>
  <bean id="metier" class="metier.MetierImpl">
    <property name="dao" ref="d"></property>
  </bean>
</beans>
```



Injection des dépendances dans une application java standard

```
package pres;
import metier.IMetier;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
public class Presentation {
    public static void main(String[] args) {
        ApplicationContext appContext = new
FileSystemXmlApplicationContext("AppContext.xml");
        IMetier m=(IMetier) appContext.getBean("metier");
        System.out.println(m.calcul());
    }
}
```

Injection des dépendances dans une application web

- Dans une application web, SpringIOC est appelé au démarrage du serveur en déclarant le listener ContextLoaderListener dans le fichier **web.xml**

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring-beans.xml</param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

- Dans cette déclaration, ContextLoaderListener est appelé par Tomcat au moment du démarrage de l'application. Ce listener cherchera le fichier de beans spring « spring-beans.xml » stocké dans le dossier WEB-INF. ce qui permet de faire l'injection des dépendances entre MetierImpl et DaoImpl
-

Types d'injection de dépendances

- Il existe 4 types d'injections de dépendances :
 - Injection par constructeur
 - **Injection par interface**
 - Injection par mutateur
 - Injection par champs

BN : Java ne support pas tous ces types !

Factory

Le patron de conception « fabrique »

Factory

- ❑ C'est grâce à ce modèle que Spring peut produire des objets respectant un contrat mais indépendants de leur implémentation.
- ❑ En réalité, ce modèle est basé sur la notion d'interface et donc de contrat.
- ❑ L'idée est simplement d'avoir un point d'entrée unique qui permet de produire des instances d'objets.
- ❑ Tout comme une usine produit plusieurs types de voitures, cette usine a comme caractéristique principale de produire des voitures
- ❑ De la même façon une fabrique d'objets produira n'importe quel type d'objet pour peu qu'ils respectent le postulat de base.
- ❑ Ce postulat (le contrat) pouvant être très vague ou au contraire très précis.

Le patron de conception « fabrique »

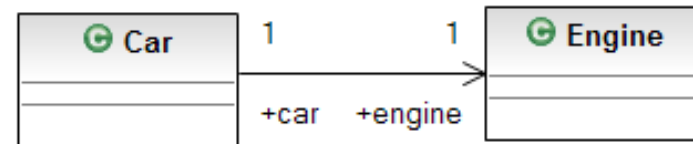
Factory

Avantages : Rapide à développer

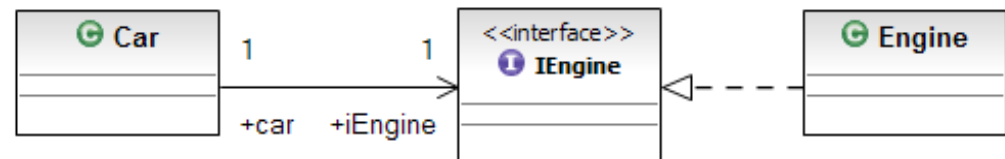
Inconvénients : Statique Disperse les dépendances dans le code

Solution

Les interfaces : Le java fournit un moyen de simplifier la gestion des dépendances



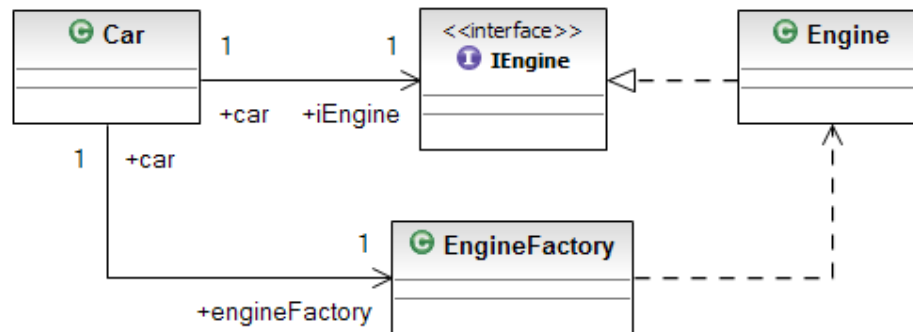
```
public class Car{
    public static void main(String[] args)
    Engine e = new Engine ();
    e.run ();
}
```



```
public class Car{
    public static void main(String[] args)
    IEngine e = new Engine ();
    e.run ();
}
```

Le patron de conception « fabrique »

Factory



```

public class factory {
    public IEngine getDependency() {
        return new Engine();
    }
}

public class A {
    public static void main(String[] args) {
        IEngine b = new factory().getDependency();
        b.someMethod();
    }
}
  
```

❑ Comme nous pouvons le constater, nous avons gardé le new ce qui fait qu'il reste une dépendance indirecte dans A.

❑ Avantages : Toujours rapide à développer, Possibilité de changer d'implémentation

❑ Inconvénients : Dépendance toujours là. Disperse les dépendances dans le code

❑ Factory : Ce pattern permet d'avoir une classe **factory** qui va gérer les dépendances. Cette factory possède des méthodes qui vont instancier la dépendance et la retourner. Chaque fois qu'une dépendance devra être résolue, la classe appelante utilisera la factory.

❑ Dès lors le pattern IoC (Inversion of control) sera réalisable

Singleton

Le singleton

- ❑ Il s'agit certainement du patron de conception le plus connu.
- ❑ En effet, il est (très) utilisé dans beaucoup de domaines.
- ❑ Ce modèle revient à s'assurer qu'il n'y aura toujours qu'une instance d'une classe donnée qui sera instanciée dans la machine virtuelle.
- ❑ Les objectifs sont simples :
 - ❑ Eviter le temps d'instanciation de la classe.
 - ❑ Eviter la consommation de mémoire inutile.
- ❑ Ce modèle impose cependant une contrainte d'importance, la classe qui fournit le service ne doit pas avoir de notion de session.
- ❑ L'objectif est de garantir l'unicité de l'instance, pour cela il faut interdire la création de toute instance en dehors du contrôle de la classe.
- ❑ Dans ce but, le constructeur est rendu privé et une méthode qui retourne une instance de la classe est mise à disposition.

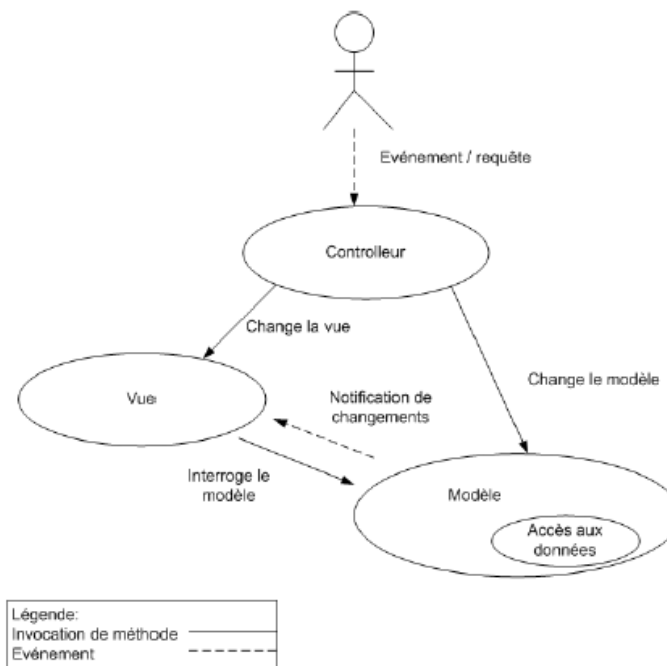
MVC selon Spring

Spring MVC

- ▶ Ce n'est peut-être pas le concept le plus important dans Spring, dans la mesure où il a été largement démocratisé par Struts.
- ▶ Cependant encore trop de programmeurs ont tendance à mélanger toutes les couches, à mettre des traitements métiers dans la jsp, ou encore des servlets dans lesquelles ils mélangent allégrement html, javascript, métier et bien d'autres choses.
- ▶ Étant donné que l'un des objectifs les plus importants de Spring est la séparation des couches, la partie MVC et le concept d'un point de vue général semblent indispensables.
- ▶ Comme cité précédemment l'objectif est de séparer les couches et les technologies.

MCV1

► Le modèle MVC version 1 :



Problème du MVC1

- ▶ Le problème de ce design pattern est la partie concernant les notifications de changement dans le modèle.
- ▶ En effet, si ce point ne pose pas de problème dans le cadre de swing par exemple, où les composants de la vue sont connectés et capables d'intelligence, il n'en est pas de même pour les applications web.
- ▶ Dans le cadre d'une application web, c'est le design pattern MVC2 qui est utilisé car il ne nécessite pas l'emploi du design pattern observer.
- ▶ Ce dernier permet la notification sur les composants : il observe le modèle et permet à la vue de réagir pour se mettre à jour.

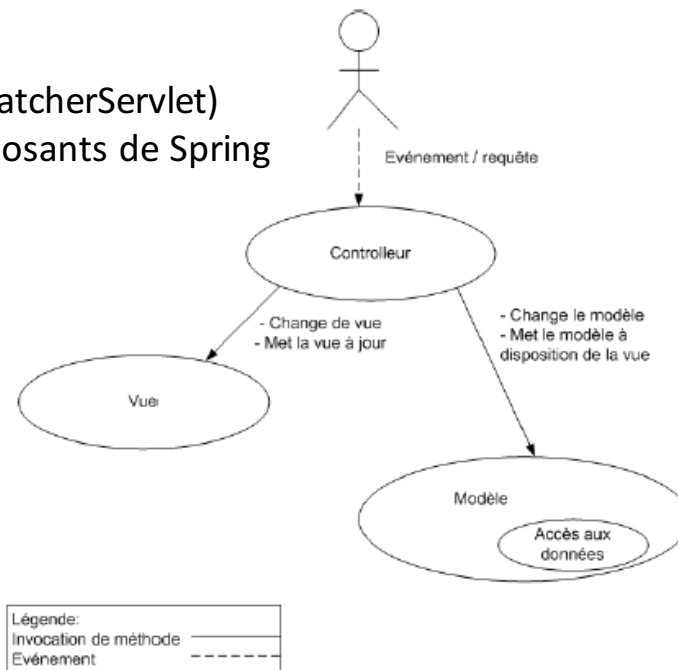
MVC2

► Le modèle MVC version 2 :

Il s'agit d'un Framework

Baser sur le pattern : Front contrôle (dispatcherServlet)

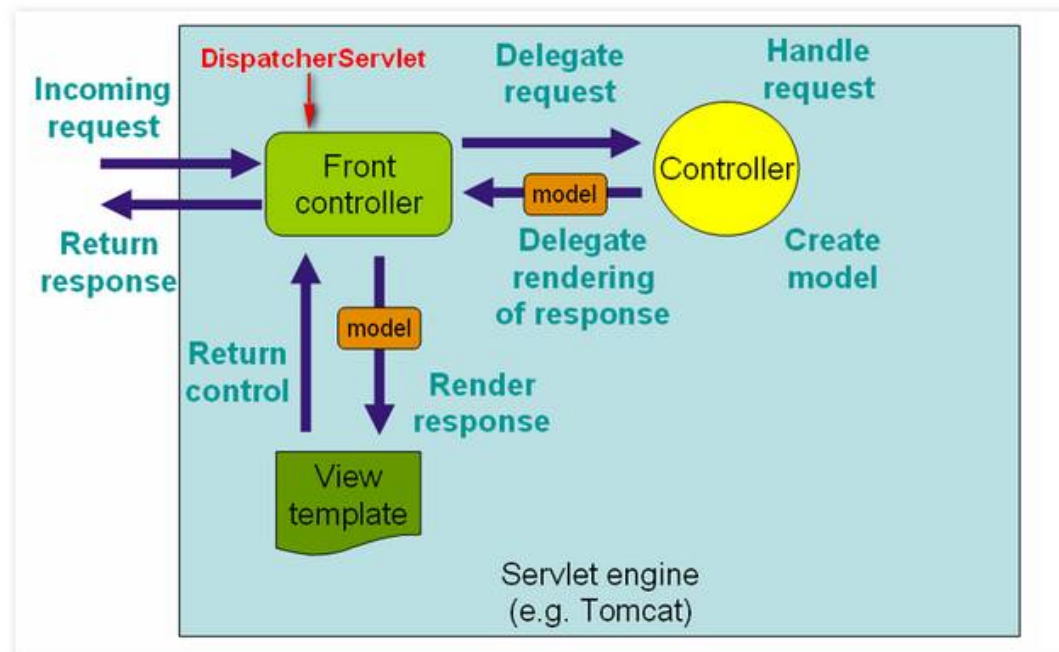
S'intègre facilement avec les autres composants de Spring



MVC2

- ▶ On remarque que c'est le contrôleur qui devient le module central.
- ▶ De fait, la vue peut à présent se contenter d'afficher sans aucune intelligence.
- ▶ Cependant, même si le design MVC permet de mieux séparer les couches, il ne faut pas oublier qu'il ne s'agit pas de la façon de procéder la plus intuitive.
- ▶ Elle induit donc d'investir du temps dans la réflexion sur la façon de séparer les différentes couches et technologies et surtout de bien réfléchir dans quelle couche s'effectue quel traitement.
- ▶ De plus les frameworks génèrent souvent plus de fichiers et naturellement plus de configuration.
- ▶ Ce surplus de complexité est cependant bien contrebalancé par la flexibilité supérieure, la plus grande fiabilité, une plus grande facilité pour tester et débbugger (puisque que l'on peut tester les bouts un à un).

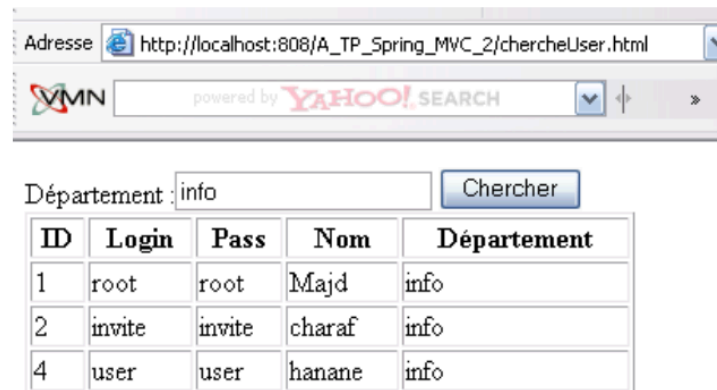
Spring MVC



Application

Application

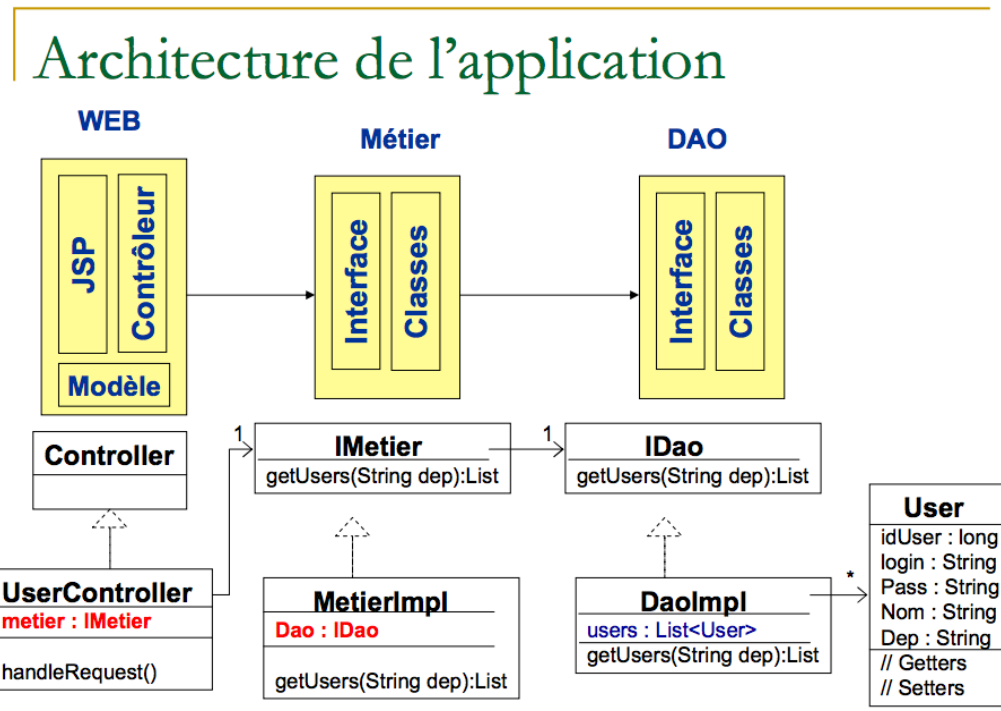
- Création d'une application web qui permet d'afficher les utilisateurs d'un département saisi.



The screenshot shows a web browser window with the address bar displaying `http://localhost:8080/A_TP_Spring_MVC_2/chercheUser.html`. Below the address bar is a search bar with the text "powered by YAHOO! SEARCH". The main content area of the browser displays a form with a label "Département :" followed by a text input field containing the value "info" and a "Chercher" button. Below the form is a table with the following data:

ID	Login	Pass	Nom	Département
1	root	root	Majd	info
2	invite	invite	charaf	info
4	user	user	hanane	info

Application



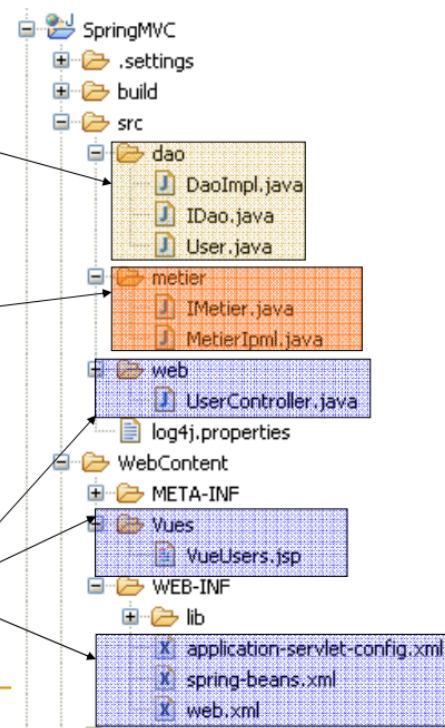
Application

Structure du projet

Couche DAO

Couche Métier

Spring MVC



Application

Couche DAO

■ L'entité User.java

```
package dao;

public class User {
    private Long idUser; private String login;
    private String pass; private String nom;
    private String departement;

    public User() {
    }
    public User(String login, String pass, String nom, String dep) {
        this.login = login;
        this.pass = pass;
        this.nom = nom;
        this.departement = dep;
    }
    // Getters et Setters
}
```

Application

Couche DAO

■ Interface de la couche DAO

```
package dao;  
import java.util.List;  
public interface IDao {  
    public void addUser(User u);  
    public List<User> getUsersByDep(String dep);  
}
```

Application

Couche DAO

■ Une implémentation DAO

```
package dao;
import java.util.*;
import org.apache.log4j.Logger;
public class DaoImpl implements IDao {
    private List<User> users=new ArrayList<User>();
    Logger log=Logger.getLogger(DaoImpl.class);
    public void addUser(User u) {
        u.setIdUser(new Long(users.size()+1));
        users.add(u);
    }
    public List<User> getUsersByDep(String dep) {
        List<User> urs=new ArrayList<User>();
        for (User u:users)
            if(u.getDepartement().equals(dep))
                urs.add(u);
        return urs;
    }
}
```

Application

Couche DAO

■ DaoImpl (Suite)

```
public void init(){
    this.addUser(new User("root", "123", "Alpha","math"));
    this.addUser(new User("user", "432", "Gamma","math"));
    this.addUser(new User("toto", "123", "wild","math"));
    this.addUser(new User("admin", "admin", "Mandour","info"));
    this.addUser(new User("user1", "user1", "Gamma","info"));
    log.info("Création de 5 Utilisateurs");
}
}
```

Application

Couche Métier

- Interface de la couche métier

```
package metier;  
import java.util.List;  
import dao.User;  
  
public interface IMetier {  
    public void addUser(User u);  
    public List<User> getUsersByDep(String dep);  
}
```

Application

Couche Métier

■ Implémentation de la couche métier

```
package metier;  
import java.util.List;  
import dao.IDao;  
import dao.User;  
public class MetierImpl implements IMetier {  
    private IDao dao;  
    public void setDao(IDao dao) {  
        this.dao = dao;  
    }  
    public void addUser(User u) {  
        dao.addUser(u);  
    }  
    public List<User> getUsersByDep(String dep) {  
        return dao.getUsersByDep(dep);  
    }  
}
```

Application

Le fichier web.xml

Pour L'injection des dépendances :

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring-beans.xml</param-value>
</context-param>
<listener>
  <listener-class>
org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

Application

Le fichier web.xml

Pour Spring MVC

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/application-servlet-config.xml</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

Application

Spring MVC

Structure de beans Spring pour l'injection des dépendances:

```
<?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:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd">
  <bean class="dao.Daolmpl" id="dao" init-method="init">
  </bean>
  <bean id="metier" class="metier.Metierlpml">
    <property name="dao" ref="dao"></property>
  </bean>
</beans>
```

Application

Spring MVC

■ Le Contrôleur:UserController.java

```
package web;
import .....
public class UserController implements Controller {
    private IMetier metier;
    public ModelAndView handleRequest(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        String dep=request.getParameter("departement");
        Map modele=new HashMap();
        modele.put("dep", dep);
        List<User> users=metier getUsersByDep(dep);
        modele.put("users", users);
        return new ModelAndView("vueUsers",modele);

    }
    // Getters et Setters
}
```

Application

Spring MVC

■ **application-servlet-config.xml:**

□ **Mapping des URL :**

```
<bean
  class="org.springframework.web.servlet.handler.SimpleUrlH
  andlerMapping">
  <property name="mappings">
    <props>
      <prop key="chercheUser.html">userController</prop>
    </props>
  </property>
</bean>
```

Application

Spring MVC

■ **application-servlet-config.xml** (suite)

- Déclaration du contrôleur :

```
<!-- LES CONTROLEURS -->
<bean id="userController" class="web.UserController">
  <property name="metier">
    <ref bean="metier"/>
  </property>
</bean>
```

- Déclaration du résolveur de vues:

```
<!-- le résolveur de vues -->
<bean class=
  "org.springframework.web.servlet.view.BeanNameViewResolver"/>
<!-- les vues -->
<bean id="vueUsers"
  class="org.springframework.web.servlet.view.JstlView">
  <property name="url">
    <value>/Vues/Users.jsp</value>
  </property>
</bean>
```

Application

■ La vue : Users.jsp

Spring MVC

```
<%@taglib uri="/WEB-INF/c.tld" prefix="c" %>
<html>
<body>
<form action="chercheUser.html" method="post">
Département :<input type="text" name="departement" value="${dep}">
<input type="submit" name="action" value="Chercher">
<table border="1" width="80%">
<tr>
<th>ID</th><th>Login</th><th>Pass</th>
<th>Nom</th><th>Département</th>
</tr>
<c:forEach items="${users}" var="u">
<tr>
<td><c:out value="${u.idUser}"/></td>
<td><c:out value="${u.login}"/></td>
<td><c:out value="${u.pass}"/></td>
<td><c:out value="${u.nom}"/></td>
<td><c:out value="${u.departement}"/></td>
</tr>
</c:forEach>
</table>
</form>
</body>
</html>
```


Application

Utilisation des annotations

```
package web
import ...
@Controller
public class UserController {
    @Autowired
    private IMetier metier;
    @RequestMapping(value="/chercheUsers")
    public String chercheUsers(@RequestParam String dep, Model model){
        model.addAttribute("dep",dep);
        model.addAttribute("users", metier.getUsersByDep(dep));
        return("VueUsers");
    }
    @RequestMapping(value="/chercher")
    public String chercher(){
        return("VueUsers");
    }
}
```

Application

Spring MVC

- Quand on utilise les annotations, le contenu du fichier application-servlet-config.xml est réduit au code xml suivant:

```
<!-- Spécifier les packages où Spring devrait chercher les
      contrôleurs au démarrage-->
```

```
<context:component-scan base-package="web"/>
```

```
<!-- Utiliser un résolveur de vues simple qui suppose que
      toutes les vues se terminent par .jsp et que qu'elles sont
      stockées dans le dossier Vues-->
```

```
<bean class=
      "org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/Vues/" />
  <property name="suffix" value=".jsp" />
</bean>
```

Application

■ La vue : Users.jsp

Spring MVC

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<form action="chercheUsers.html" method="get">
  Département:<input type="text" name="dep" value="${dep}">
  <input type="submit" value="OK">
</form>
<table width="80%" border="1">
<tr>
  <th>ID</th><th>Login</th><th>Pass</th><th>Nom</th><th>DEp</th>
</tr>
<c:forEach var="u" items="${users}">
  <tr>
    <td>${u.idUser}</td><td>${u.nom}</td><td>${u.pass}</td>
    <td>${u.nom}</td><td>${u.departement}</td>
  </tr>
</c:forEach>
</table>
</body>
</html>
```