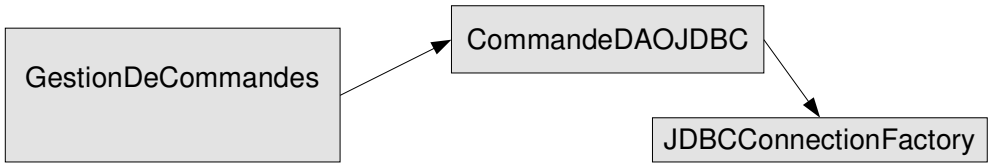
Le framework Spring
DUT info S4

Définitions

- Bibliothèque
  - peu d'incidence sur l'architecture des applications (facile à remplacer si code bien écrit)
- Framework
  - fournissent l'architecture d'une partie de l'application

Le problème des dépendances

- On souhaite avoir des composants les plus indépendants possibles. Mais :



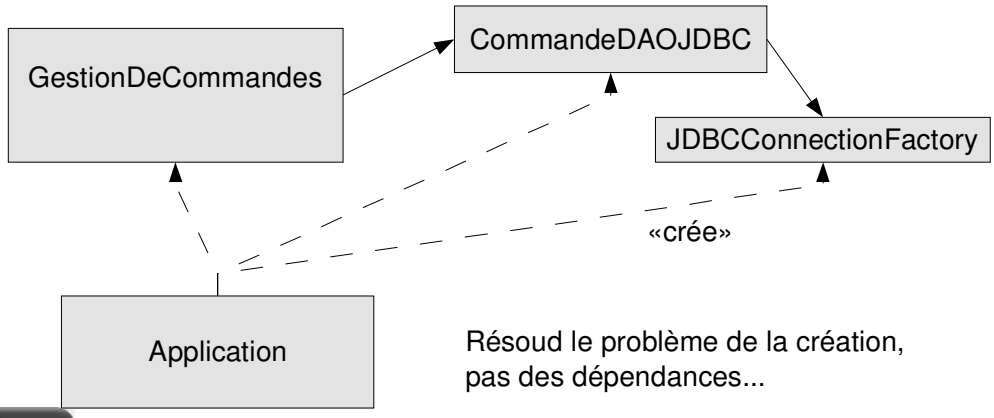
- La classe GestionDeCommande dépend de CommandeDAOJDBC et de JDBCConnectionFactory

Dépendances

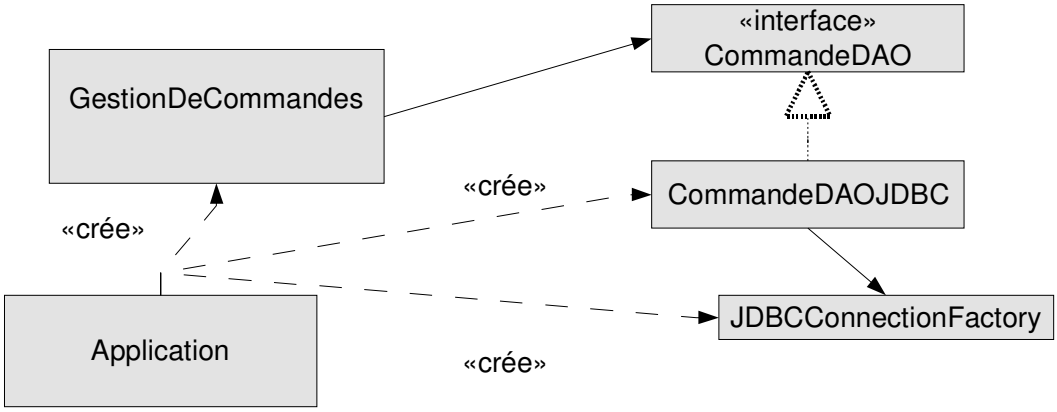
- C'est un problème, car
  - il est alors impossible de proposer plusieurs solutions de sauvegarde (XML, hibernate...) sans modifier GestionDeCommande
  - il est impossible de *tester* GestionDeCommande sans avoir écrit CommandeDAOJDBC
  - le test est compliqué : d'où proviennent les erreurs éventuelles ?
  - qui crée CommandeDAOJDBC ? qui crée GestionDeCommande ?

Solution 1

- Dans le programme de début d'année : l'application s'occupait de « cabler » les différents composants.



Solution 2



- Variante : GestionDeCommande admet ne dépend plus de CommandeDAOJDBC, mais seulement de son interface.

Inversion de contrôle

- Cette technique s'appelle Inversion de contrôle
- ou principe d'Hollywood : *ne nous appelez pas, nous vous appellerons*
- dans spring : injection de dépendance
- On remplace:

  dans GestionDeCommande

  dao= new CommandeDAOJDBC();

  par l'appel de la méthode setDAO(...) de GestionDeCommande.

Spring

- framework *léger :* impose peu de contraintes aux classes écrite par le programmeur
- basé sur l'inversion de contrôle
- framework modulaire pour traiter des aspects variés d'une application : couche web, client riche, persistence, transactions, sécurité...
- support de la programmation orientée aspect, en particulier pour les transactions.

Les beans

- Éléments de base des applications Spring. En gros, pour spring, des objets java dotés d'accesseurs.
- Spring permet de paramétrer les beans de manière déclarative (par ex. dans un fichier XML)

Création de l'infrastructure par XML

- Fichier de définition des beans

```
<beans>
    <bean id='commandeDAO' class='CommandeDAOJDBC'>
        <property name='connectionFactory' ref='cf'/>
    </bean>

    <bean id='cf' class='ConnectionFactory'>
        <property name='url' value='jdbc:hsqldb:hsql://localhost/db'/>
        .....
    </bean>

    <bean id='gestionDeCommande'>
        <property name='dao' ref='commandeDAO'/>
    </bean>
</beans>
```

Spring et les applications web

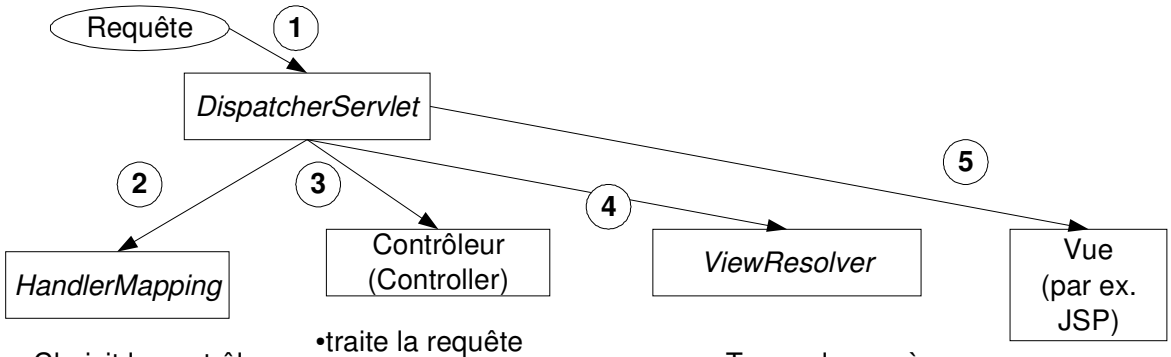
- Une application Spring web utilise typiquement la servlet

  org.springframework.web.servlet.DispatcherServlet

  (pattern dit « Front Controller »).
- Cette servlet, définie dans web.xml, a un nom, par exemple « monAppli ».
- le fichier de configuration des beans est alors

  monAppli-servlet.xml

Étapes du traitement d'une requête



Choisit le contrôleur

- traite la requête
- crée un objet modelAndView
- rempli de « modèle »
- choisit le *nom* de la vue

Trouve la vue à afficher à partir du *nom*.

classe fournie par Spring
classe écrite par le programmeur

Exemple de configuration (1)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <!-- Ensemble simple, mais complet et explicite, de beans pour une application web -->
    <!-- Un contrôleur -->
    <bean id="affiche4" class="simple.AfficheQuatreController"></bean>
    <!-- L'infrastructure -->
    <!-- Associe un contrôleur à une requête -->
    <bean id="handlerMapping"
            class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /**/quatre=affiche4
            </value>
        </property>
    </bean>
    <!-- Associe à un nom de vue la jsp correspondante -->
    <bean id="viewResolver"
            class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```


Exemple de configuration (2)

```
<!-- un contrôleur -->
<bean name="/afficheQuatre" class="simple.AfficheQuatreController">
</bean>

<!-- L'infrastructure -->

<!-- Associe un contrôleur à une requête par le nom du contrôleur -->
<bean id="handlerMapping"
      class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
</bean>

<!-- Associe à un nom de vue la jsp correspondante -->
<bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">
 <property name="prefix" value="/WEB-INF/pages/"/>
 <property name="suffix" value=".jsp"/>
</bean>
```

Exemple de contrôleur

```
public class AfficheQuatreController extends AbstractController {


    @Override
    protected ModelAndView
        handleRequestInternal(HttpServletRequest request,
                              HttpServletResponse response)
    throws Exception {
        ModelAndView nombre=
            new ModelAndView("afficherNombre", "nombre", new Integer(4));
        return nombre;
    }
}
```

nom de la vue

nom d'un bean request

valeur du précédent

« Convention over Configuration » en Spring 2.0

- Inspiré de frameworks comme *Ruby on Rails*
- Pour les cas simples, on remplace la *configuration* XML par l'application de *conventions*, en gros :
- pour l'URI « /appli/affichequatre »
  - contrôle: AfficheQuatreController
  - vue « normale » : afficheQuatre

Conventions (de base)

- pour les contrôleurs simples :
  - le nom de la classe se termine par « Controller »
  - url =

  base + « / » + nom du contrôleur en *minuscule* sans « Controller ».

  AfficheQuatreController → /appli/affichequatre

Conventions (suite)

- Éléments du modèle
  - Dans le contrôleur (p.ex. **AfficheQuatre**Controller) :

  ModelAndView m= new ModelAndView();

  → la vue s'appellera « **afficheQuatre** »

  Personne p= ....; m.addObject(**p**);

  → l'élément du modèle s'appelle automatiquement « **personne** ».

  Set<Personne> tab= ....; m.addObject(tab);

  → l'élément s'appelle **personneList**

SimpleFormController

- Gère un formulaire
- les données sont sauvées dans un bean request, appelé la commande.

```
<bean class="facture.controle.CreerArticleController">
        <property name="commandClass" value="facture.modele.Article" />
        <property name="commandName" value="article"/>
        <property name="formView" value="creerArticle"/>
        <property name="successView" value="menu"/>
        <!-- «  valideur  » optionnel -->
        <property name="validator" ref="articleValidator"/>
        <!-- façade de la couche métier -->
        <property name="facturationFacade" ref="facturationFacade"/>
</bean>
```


SimpleFormController

```
public class CreerArticleController extends SimpleFormController {

    private FacturationFacade facturationFacade;

    @Override
    protected void doSubmitAction(Object command) throws Exception {
        Article a= (Article) command;
        facturationFacade.saveArticle(a);
    }

    public FacturationFacade getFacturationFacade() {
        return facturationFacade;
    }

    public void setFacturationFacade(FacturationFacade facturationFacade) {
        this.facturationFacade = facturationFacade;
    }
}
```

Les tags « form »

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Create user</title>
</head>
<body>
<form:form commandName="article">
    <p>Nom :<form:input path="designation" />
    <p>Adresse :<form:input path="prix" />
     <p><input type="submit" value="sauver" />
</form:form>
</body>
</html>
```

lit et écrit dans le bean «article» (c.f. configuration)

manipule la propriété article.prix

Validation (optionnelle)

```
public class ArticleValidator implements Validator {

    /**
     * Retourne vrai si le validateur gère la classe passée en argument.
     */
    public boolean supports(Class clazz) {
        return (clazz.equals(Article.class));
    }
    // accumule des erreurs si l'objet est mal formé.
    public void validate(Object target, Errors errors) {
        Article a= (Article) target;
        ValidationUtils.rejectIfEmpty(errors, "designation", "empty_field","empty field!");
        if (a.getPrix() <= 0)
            errors.rejectValue("prix", "incorrect_price", "incorrect price");
    }
}
```