Architectures des applications avec IHM

Cours IHM

Frédéric Moal

2011/2012

Objectif

Comment architecturer une application objet utilisant une interface graphique…

Plan de cette présentation

- L'Architecture MVC
- Un exemple interne Swing : UI-delegate
- Un exemple classique : les listes
- Un exemple complet de MVC 1 pour une application Swing
- Exemple MVC 2 : une application Web

Séparation du GUI/classes métier

- Les classes métier doivent être indépendantes des interfaces graphiques qui les utilisent
- L'architecture est la suivante :
  - classes métier (dans un paquetage)
  - classes pour l'interface graphique (GUI) (dans un autre paquetage)
  - les écouteurs du GUI font appel aux méthodes des classes métier

Classe « principale » schématique

```
public class Application {
   . . .
   public static void main(String[] args) {
   // Initialisation (utilise classes métier)
   . . .
   // Appelle la classe GUI principale
   new GUI(. métier .);
   }
}
```

Les traitements de fin éventuels sur les classes métiers peuvent être placés dans les écouteurs au moment de quitter le GUI

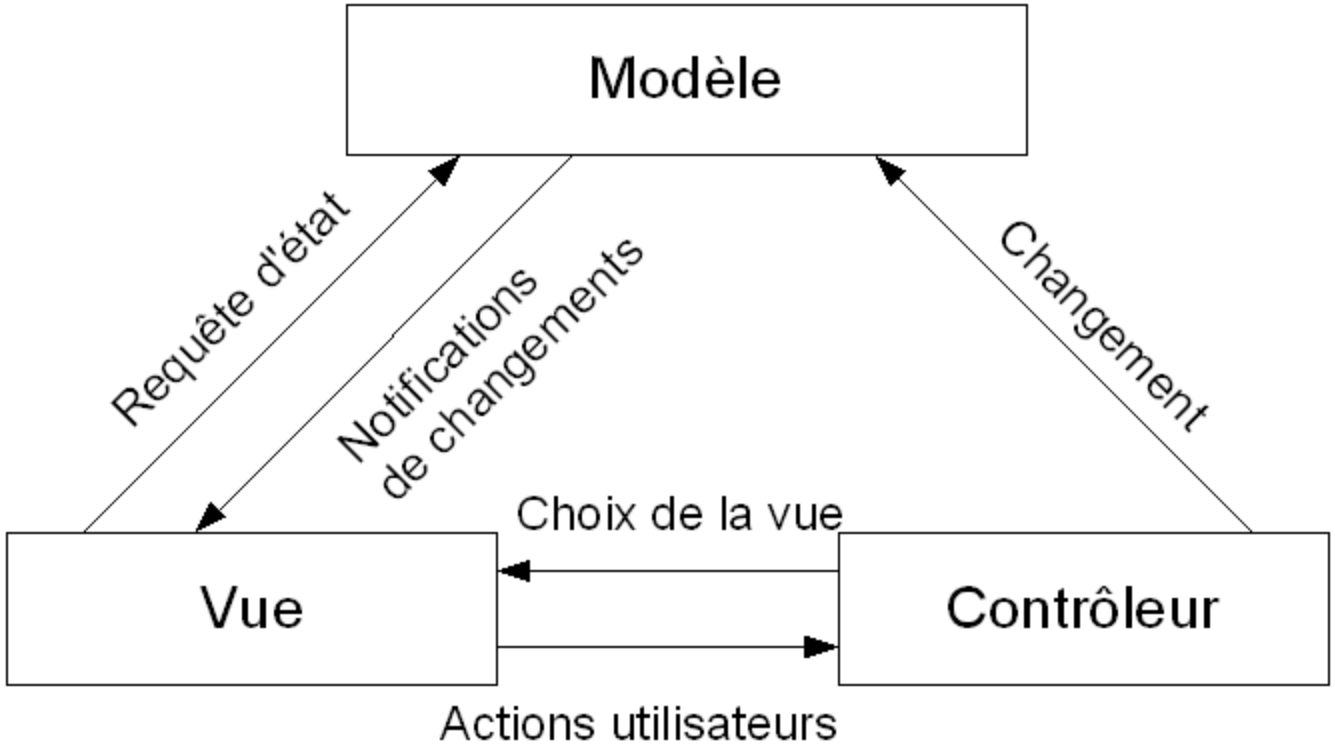
Plan de cette présentation

- L'Architecture MVC
- Un exemple Swing : UI-delegate
- Un exemple classique : les listes
- Un exemple complet de MVC 1 pour une application Swing
- Exemple MVC 2 : une application Web

Architecture MVC

- L'architecture MVC (Modèle-Vue-Contrôleur) est utilisée pour modéliser les composants graphiques qui contiennent des données (listes, tables, arbres,…) :
  - le modèle contient les données
  - les vues donnent une vue des données (ou d'une partie des données) du modèle à l'utilisateur
  - le contrôleur traite les événements reçus par le composant graphique

Variante Architecture MVC



Architecture MVC

- La synchronisation entre la vue et le modèle se passe avec le pattern Observer (modèle utilisé par Swing pour les écouteurs)

  Il permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour.

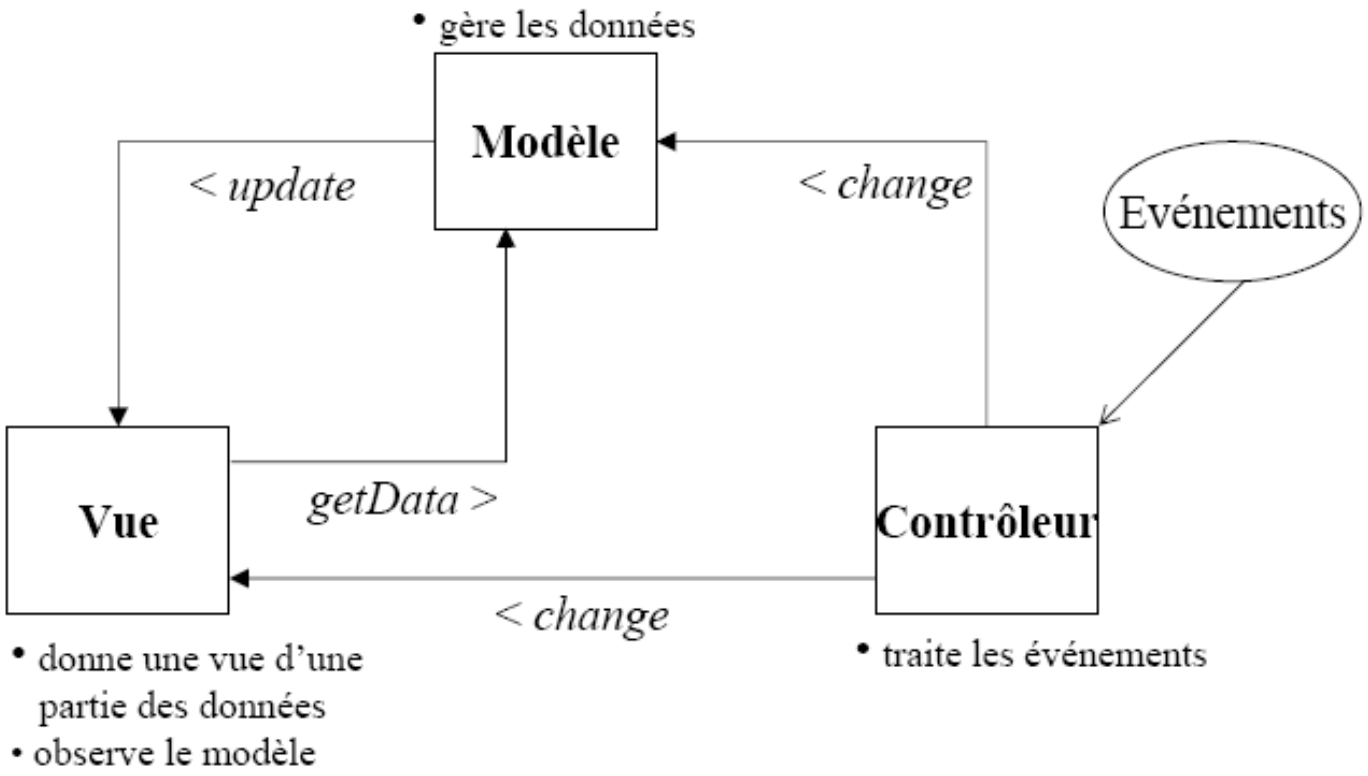
Architecture MVC

- Ce modèle de conception permet principalement 2 choses :
  - Le changement d'une couche sans altérer les autres. couches clairement séparées : facilement modifiable

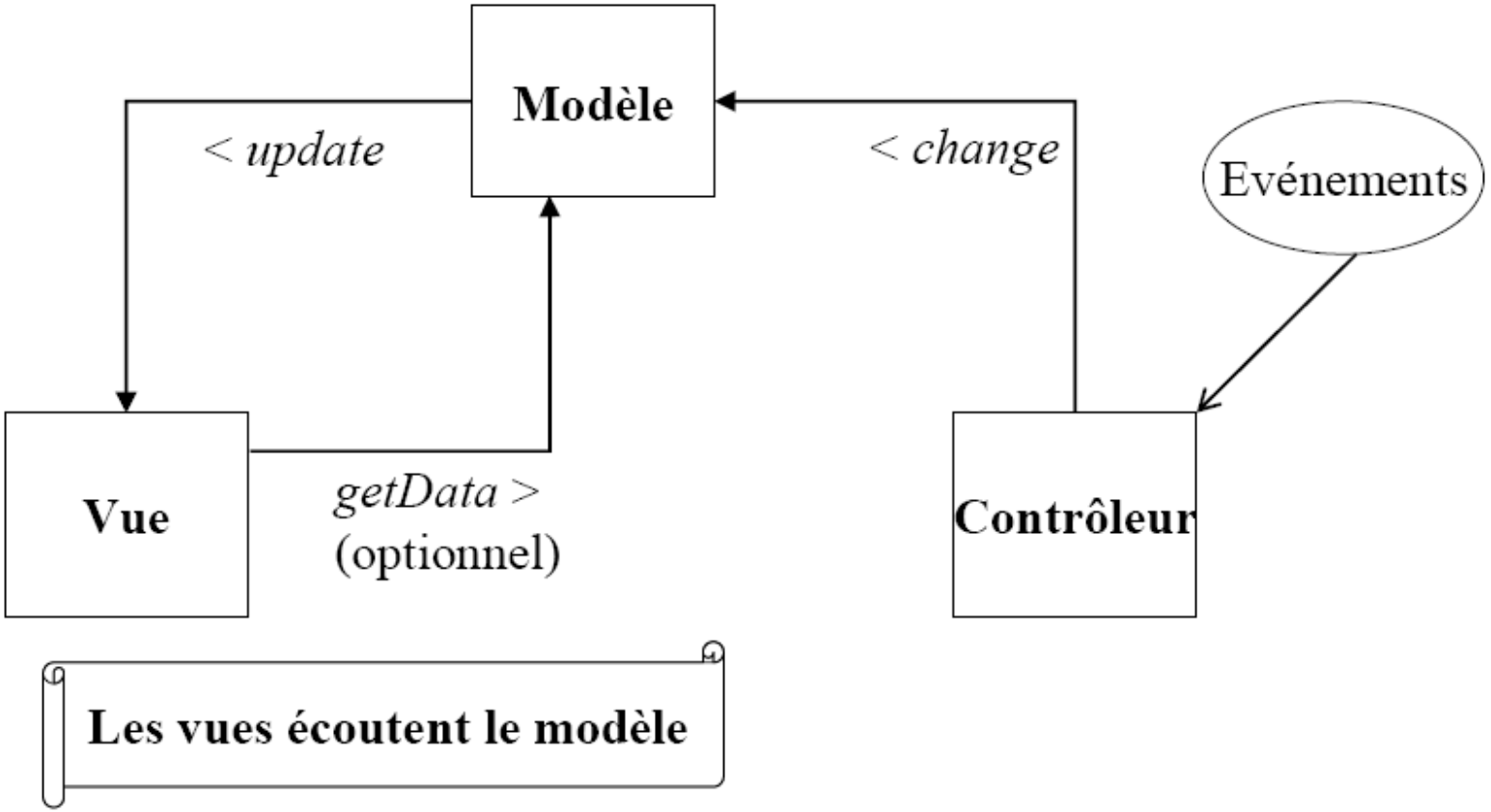
    eg : remplacer Swing par SWT sans porter atteinte aux autres couches, ou changer le modèle sans toucher à la vue et au contrôleur.

    => modifications plus simples.
  - La synchronisation des vues. Avec ce design pattern, toutes les vues qui montrent la même chose sont synchronisées.

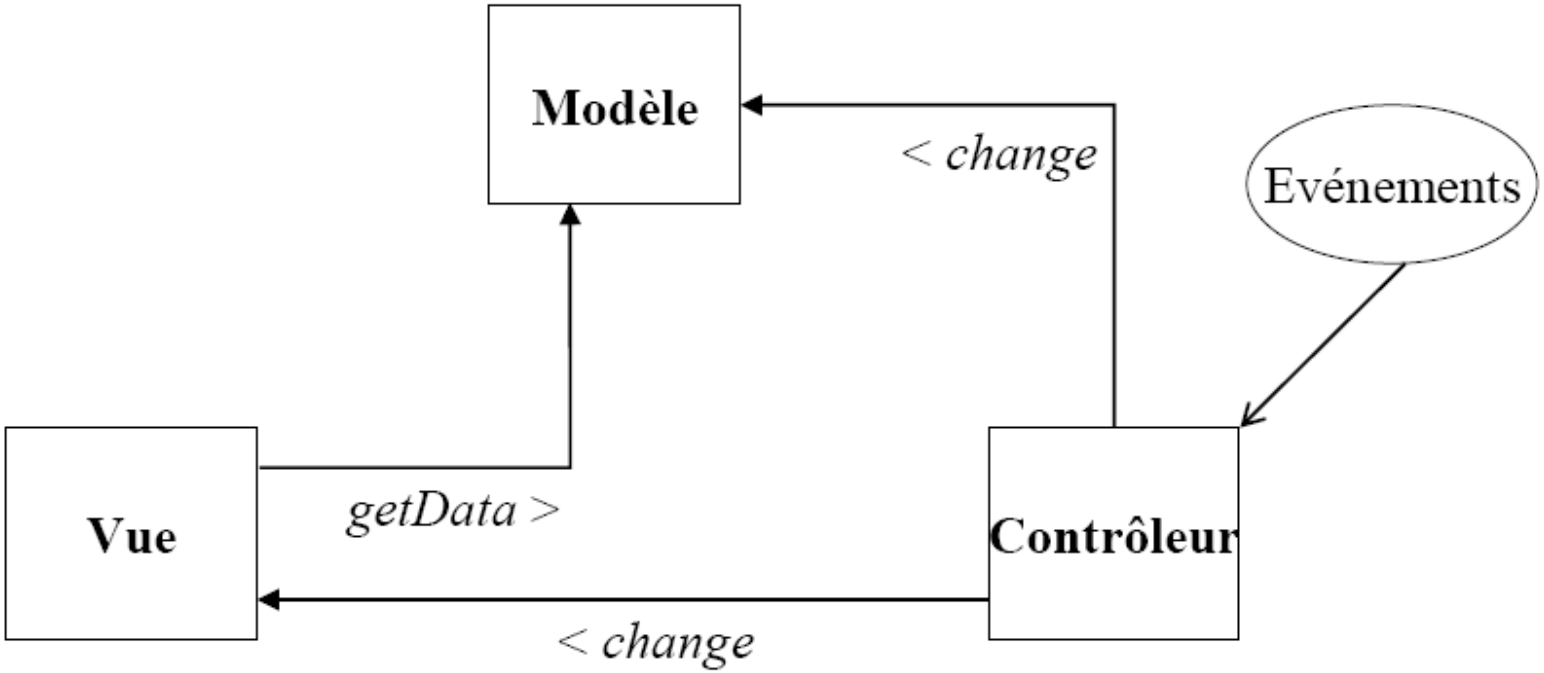
Architecture MVC



Architecture MVC



Variante Architecture MVC



Exemple de fonctionnement MVC

- Exemple de processus engendré par une action de l'utilisateur

1. Le contrôleur reçoit un événement
2. Il informe le modèle (*change*)
3. Celui-ci modifie ses données en conséquence
4. Le contrôleur informe la vue d'un changement (*change*)
5. La vue demande au modèle les nouvelles données (*getData*)
6. La vue modifie son aspect visuel en conséquence

Plan de cette présentation

- L'Architecture MVC
- Un exemple interne Swing : UI-delegate
- Un exemple classique : les listes
- Un exemple complet de MVC 1 pour une application Swing
- Exemple MVC 2 : une application Web

UI-delegate

- Pour représenter ses composants graphiques, Swing utilise une variante de MVC où contrôleur et vue sont réunis dans un objet *UIdelegate*
  - donne le *look and feel* du composant
  - Un *look and feel* est constitué de tous les objets *UI-delegate* associés

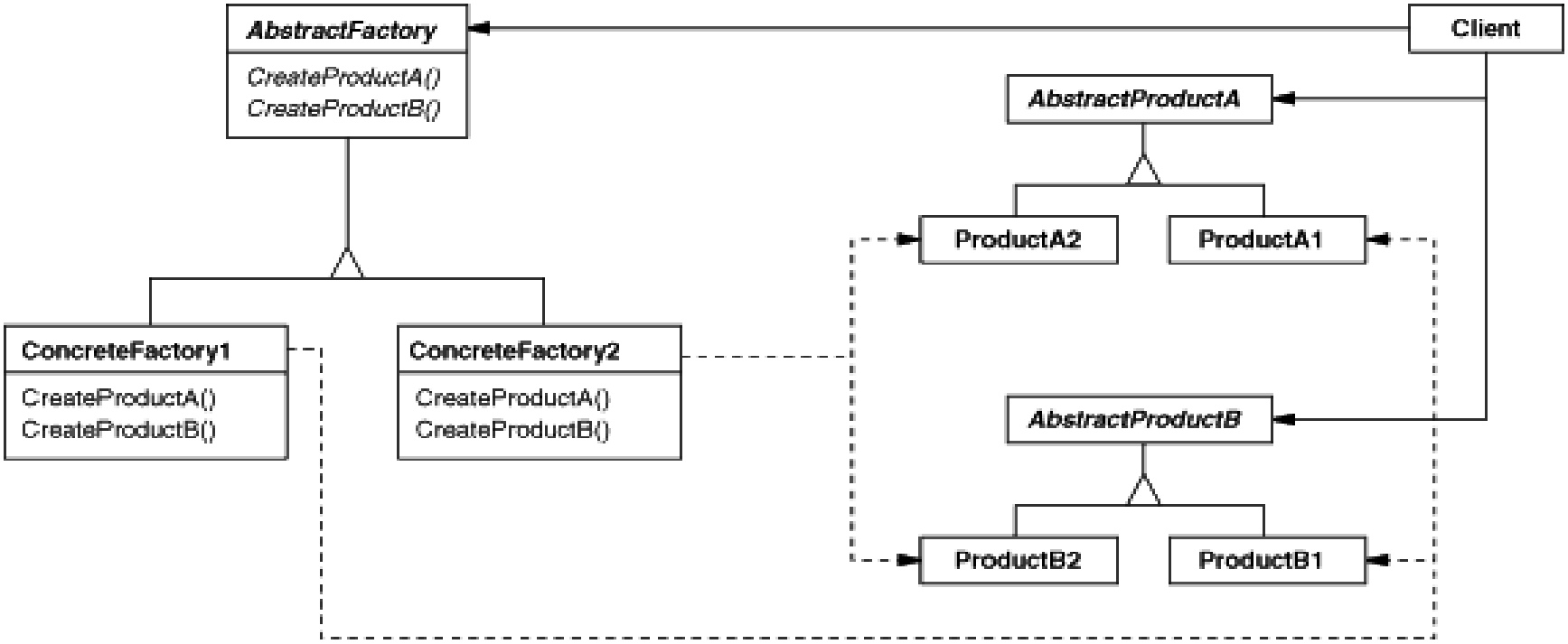
Pattern « fabrique abstraite »

- Le changement de *look and feel* utilise le modèle de conception (*design pattern*) fabrique abstraite
  - permet de choisir une famille de classes indépendamment du reste de l'application
  - Chaque *look and feel* correspond à une fabrique d'un certain type de *UI-delegate* (gérée par le *UI-manager*)
  - Pour changer de *look and feel*, il suffit de dire que l'on veut changer de fabrique

Abstract Factory

Objectif : obtenir des instances de classes implémentant des interfaces connues, mais en ignorant le type réel de la classe obtenue

Exemple : une application gérant des documents polymorphes générateur de composants graphiques supportant une multitude de *look-and-feels*



« Pluggable Look and feel »

- Pour changer de *look and feel* pour l'ensemble des composants d'une interface graphique :

```
UIManager.setLookAndFeel(
"javax.swing.plaf.metal.MetalLookAndFeel");

// Change pour les composants déjà affichés
SwingUtilities.updateComponentTreeUI(fenetre);
```

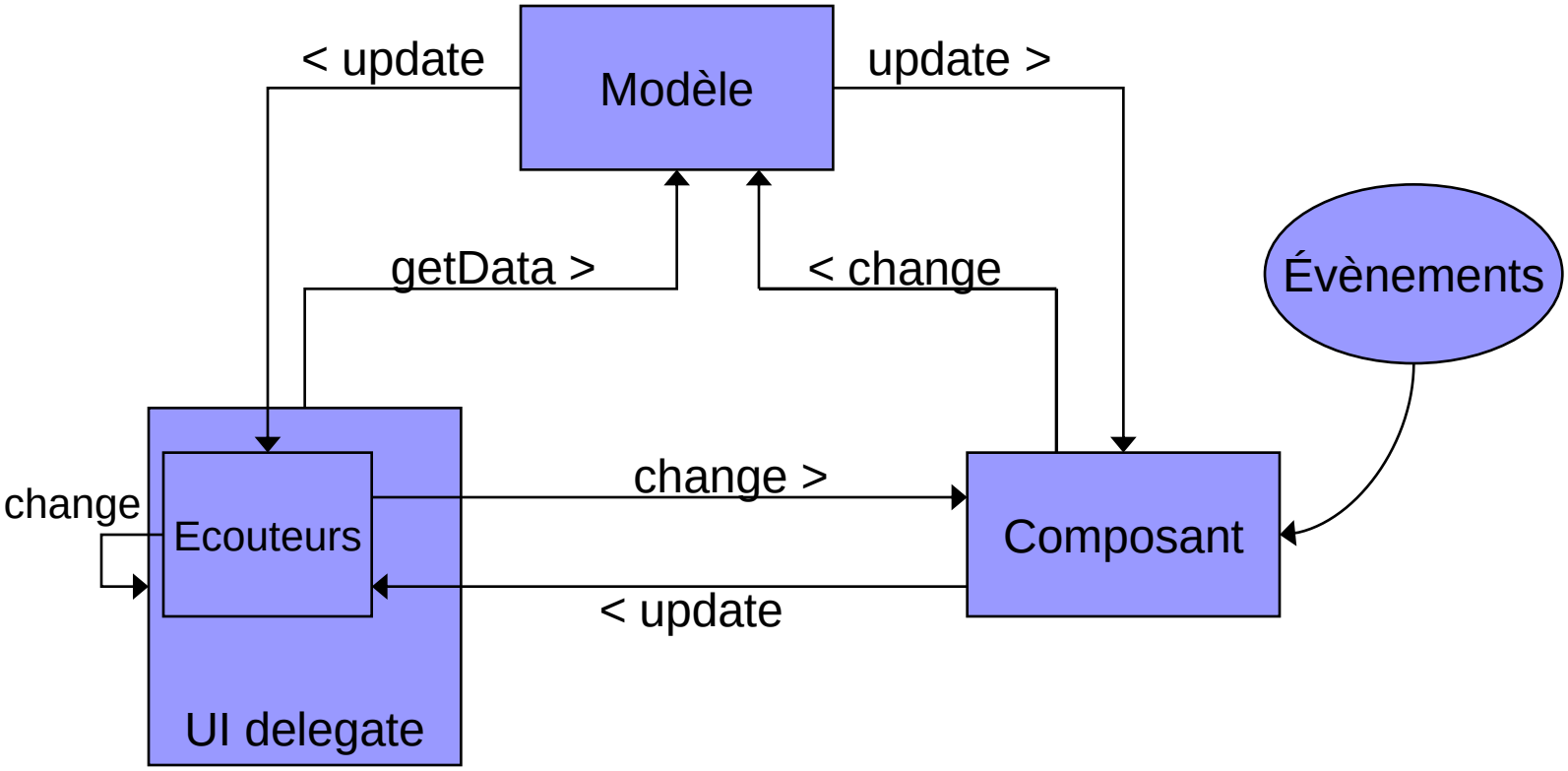
Architecture avec UI-delegate

- Le composant
  - est l'interlocuteur pour les objets extérieurs
  - contient le code pour les comportements de base du composant
  - reçoit les événements (générés par le système)
- Le modèle contient les données

Architecture avec UI-delegate

- Le UI-delegate
  - représente une façon de voir ces données et de traiter les événements
  - écoute le composant (pour traiter les événements en tant que contrôleur) ; il contient des classes internes auxquelles il délègue ce traitement
  - écoute le modèle (pour afficher une vue de ce modèle)

Architecture avec *UI delegate*



Plan de cette présentation

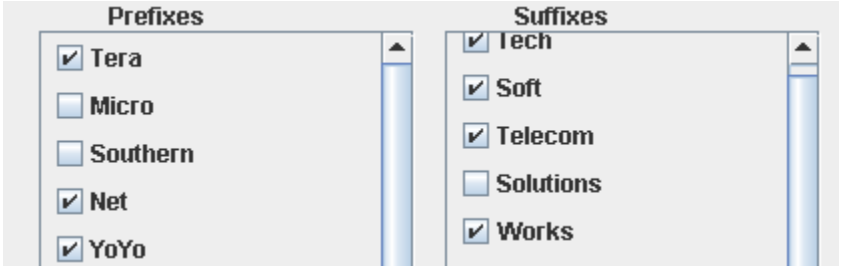
- L'Architecture MVC
- Un exemple interne Swing : UI-delegate
- Un exemple classique : les listes
- Un exemple complet de MVC 1 pour une application Swing
- Exemple MVC 2 : une application Web

Utilisation explicite du modèle

- Le modèle des composants n'est pas toujours utilisé explicitement dans le code
- Il est souvent caché par les implémentations utilisées dans le cas les plus simples (boutons par exemple)
- Dans le cas des listes, l'utilisation explicite du modèle n'est indispensable que lorsque la liste est modifiable

Listes

- Une liste permet de présenter une liste de choix à l'utilisateur



- Celui-ci peut cliquer sur un ou plusieurs choix

Barre de défilement pour les listes

- Le plus souvent une liste a des barres de défilement ; pour cela, il faut insérer la liste dans un **`ScrollPane`** :

```
JScrollPane sList =
    new JScrollPane(uneListe);
```

- Par défaut, 8 éléments de la liste sont visibles ; on peut modifier ce nombre :

```
uneListe.setVisibleRowCount(5);
```

Mode de sélection

- L'utilisateur peut sélectionner un ou plusieurs éléments de la liste suivant le mode de sélection de la liste :
  - **SINGLE_SELECTION**
  - **SINGLE_INTERVAL_SELECTION**
  - **MULTIPLE_INTERVAL_SELECTION** (mode par défaut)

```
uneListe.setSelectionMode(
   ListSelectionModel.SINGLE_SELECTION);
```

Listes non modifiables

- 2 constructeurs pour des listes non modifiables :
  - `public JList(Object[] listData)`
  - `public JList(Vector listData)`

Afficher les éléments des listes

- Une liste affiche ses éléments en utilisant **toString()** (elle sait aussi afficher des instances de **ImageIcon**)

- On peut aussi programmer des affichages particuliers avec un « *renderer* » (`setCellRenderer(ListCellRenderer)`)

Exemple de liste non modifiable

```
JList liste = new JList(new String[]
{"Un", "Deux", "Trois", "Quatre", …});

JScrollPane sp = new JScrollPane(liste);

liste.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// Pour cet exemple, la classe est l'écouteur
liste.addListSelectionListener(this);

// Ajoute le scrollPane dans le container,
// ce qui ajoutera la liste
c.add(sp);
```

Utilisation standard d'une liste

- Il est rare d'écrire un écouteur de liste
  - L'utilisation standard d'une liste est de demander à l'utilisateur de cliquer sur un bouton lorsqu'il a fini de faire ses choix dans la liste
  - On récupère alors la sélection de l'utilisateur par une des méthodes **getSelectedValue()** ou **getSelectedValues()** dans la méthode **actionPerformed()** de l'écouteur du bouton

Listes modifiables

- Une liste modifiable est associée à un modèle qui fournit les données affichées par la liste
- Ce modèle est une classe qui implémente l'interface **`ListModel`**
- On construit la liste avec le constructeur **`public JList(ListModel dataModel)`**

Modèle de données

```
public interface ListModel {
   int getSize();
   Object getElementAt(int i);
   void addListDataListener(ListDataListener l);
   void removeListDataListener(ListDataListener l);
}
```

- Pour faciliter l'écriture d'une classe qui implémente **ListModel**, le JDK fournit la classe abstraite **AbstractListModel** qui implémente les 2 méthodes de **ListModel** qui ajoutent et enlèvent les écouteurs

Listes modifiables simples

- Les plus simples ont un modèle de la classe **`DefaultListModel`** qui hérite de la classe **`AbstractListModel`**
- Avec **`DefaultListModel`** on gère les données avec des méthodes semblables aux méthodes **`add`** et **`remove`** des collections :
  - **`add(int, Object)`,**
  - **`addElement(Object)`,**
  - **`remove(int)`,**
  - **`removeElement(Object)`,**
  - **`clear()`**

Exemple de liste modifiable simple

```
pays = new DefaultListModel();

pays.addElement("France");
pays.addElement("Italie");
pays.addElement("Espagne");
pays.addElement("Maroc");

liste = new JList(pays);
```

Listes plus complexes

- On peut ne pas vouloir enregistrer tous les éléments de la liste en mémoire centrale
- Le modèle héritera alors directement de la classe **`AbstractListModel`**

En mémoire, sans enreg.

```
/** Les 1000 premiers entiers composent le
* modèle de données de la liste */
class Entiers1000 extends AbstractListModel {
  public int getSize() {
      return 1000;
  }
  public Object getElementAt(int n) {
      return new Integer(n + 1);
  }
}
. . .
JList liste = new JList(new Entiers1000());
```

En mémoire, sans enreg.

```
/** exemple réaliste */
class JListClient extends AbstractListModel {
  Modele modele = … ;
  public int getSize() {
      return modele.getNombreClient();
  }
  public Object getElementAt(int n) {
      Client c = modele.getClientByPos(n);
      return c.getNom();
  }
}
. . .
JList liste = new JList(new JListClient());
```

Architecture

JListClient n’est pas le MODELE de MVC !!!

Plan de cette présentation

- L'Architecture MVC
- Un exemple interne Swing : UI-delegate
- Un exemple classique : les listes
- Un exemple complet de MVC 1 pour une application Swing
- Exemple MVC 2 : une application Web

Application

- Un exemple simple :
  - une application permettant de modifier un volume
  - plusieurs vues pour représenter ce volume et après toute modification, les vues devront être synchronisées
  - interface développée avec Swing

Le modèle : version de base

VolumeModel.java :

```
public class VolumeModel {
    private int volume;
    public VolumeModel(){
      super();
      volume = 0;
    }
    public int getVolume() {
      return volume;
    }
    public void setVolume(int volume) {
      this.volume = volume;
    }
}
```

Le modèle : extension

il faut que notre modèle puisse notifier un changement de volume :

créer un nouveau listener (VolumeListener) et un nouvel événement (VolumeChangedEvent) :

VolumeListener.java

```
import java.util.EventListener;

public interface VolumeListener extends EventListener {
  public void volumeChanged(VolumeChangedEvent event);
}
```

Le modèle : extension

nouvel événement (VolumeChangedEvent) :

VolumeChangedEvent.java :

```
import java.util.EventObject;

public class VolumeChangedEvent extends EventObject{
	private int newVolume;
	public VolumeChangedEvent(Object source,
					int newVolume){
		super(source);
		this.newVolume = newVolume;
	}
	public int getNewVolume(){
		return newVolume;
	}
}
```

Le modèle : implémentation

- Il faut implémenter ce système d'écouteurs dans le modèle pour que d'autres entités puissent "écouter" les changements du modèle
- notre modèle avertit tous ses écouteurs à chaque changement de volume.
- Ensuite, en fonction de l'application, on peut tout à fait imaginer plusieurs listeners par modèles et d'autres événements dans les listeners, par exemple quand le volume dépasse certains seuils...

Le modèle : VolumeModel.java

```
import javax.swing.event.EventListenerList;

public class VolumeModel {
   private int volume;
   private EventListenerList listeners;
   public VolumeModel(){
       this(0);
   }
   public VolumeModel(int volume){
       super();
       this.volume = volume;
       listeners = new EventListenerList();
   }
   public int getVolume() {
       return volume;
   }
   ->
```

Le modèle : VolumeModel.java

```
    public void addVolumeListener(VolumeListener
    listener){
        listeners.add(VolumeListener.class, listener);
    }
    public void removeVolumeListener(VolumeListener
    l){
        listeners.remove(VolumeListener.class, l);
    }
    public void fireVolumeChanged(){
        VolumeListener[] listenerList =
        (VolumeListener[])listeners.getListeners(Volum
    eListener.class);
        for(VolumeListener listener : listenerList){
        listener.volumeChanged(new
    VolumeChangedEvent(this, getVolume()));
        }
    }
}
```

Le contrôleur

doit le moins possible être dépendant de Swing : on va créer une classe abstraite représentant une vue du volume

VolumeView.java

```
public abstract class VolumeView implements
   VolumeListener{
   private VolumeController controller = null;
   public VolumeView(VolumeController controller){
        super();
        this.controller = controller;
   }
   public final VolumeController getController(){
        return controller;
   }
   public abstract void display();
   public abstract void close();
}
```

Le Contrôleur

- Le contrôleur ne manipule que des objets de type View et non plus de type Swing
- Dans cet exemple, nous allons créer un seul contrôleur pour les 3 vues (plus simple car les 3 vues font toutes la même chose)
- Dans le cas de vue fondamentalement différentes, il est fortement conseillé d'utiliser plusieurs contrôleurs

Le Contrôleur : VolumeController.java

```
public class VolumeController {

   public VolumeView fieldView = null;
   public VolumeView spinnerView = null;
   public VolumeView listView = null;
   private VolumeModel model = null;

   public VolumeController (VolumeModel model){
       this.model = model;
       fieldView = new JFrameFieldVolume(this,
   model.getVolume());
       spinnerView = new JFrameSpinnerVolume(this,
   model.getVolume());
       listView = new JFrameListVolume(this,
   model.getVolume());
       addListenersToModel();
   }
```

Le Contrôleur : VolumeController.java

```
    public void displayViews(){
        fieldView.display();
        spinnerView.display();
        listView.display();
    }
    public void closeViews(){
        fieldView.close();
        spinnerView.close();
        listView.close();
    }
    public void notifyVolumeChanged(int volume){
        model.setVolume(volume);
    }
}
```

Les vues

- 3 vues différentes du même modèle :
  - Une vue permettant de modifier le volume avec un champ texte avec un bouton permettant de valider le nouveau volume : JFrameFieldVolume
  - Une vue permettant de modifier le volume à l'aide d'un spinner avec un bouton permettant de valider le nouveau volume : JFrameSpinnerVolume
  - Une vue listant les différents volumes et qui ajoutera chaque nouveau volume dans une liste déroulante : JFrameListVolume
- Toutes ces vues seront représentées par une JFrame

Les vues : JFrameFieldVolume.java

```
import java.awt.event.*; ...

public class JFrameFieldVolume extends VolumeView
   implements ActionListener{

   private JFrame                  frame = null;
   private JPanel                  contentPane = null;
   private JFormattedTextField     field = null;
   private JButton                 button = null;
   private NumberFormat            format = null;

   public JFrameFieldVolume(VolumeController controller) {
        this(controller, 0);
   }
   public JFrameFieldVolume(VolumeController controller,
   int volume){
        super(controller);
        buildFrame(volume);
   }
```

Les vues : JFrameFieldVolume.java

```
private void buildFrame(int volume) {
  frame = new JFrame();
  contentPane = new JPanel();
  format = NumberFormat.getNumberInstance();
  format.setParseIntegerOnly(true);
  format.setGroupingUsed(false);
  format.setMaximumFractionDigits(0);
  format.setMaximumIntegerDigits(3);
  field = new JFormattedTextField(format);
  field.setValue(volume);
  ((DefaultFormatter)field.getFormatter()).setAllowsInvali
  d(false);
  contentPane.add(field);
  button = new JButton("Mettre à jour");
  button.addActionListener(this);
  contentPane.add(button);
  frame.setContentPane(contentPane);
  frame.setTitle("JFrameSpinnerVolume");
  frame.pack();
}
```

Les vues : JFrameFieldVolume.java

```
    @Override
    public void close() {
        frame.dispose();
    }
    @Override
    public void display() {
        frame.setVisible(true);
    }
    public void volumeChanged(VolumeChangedEvent
    event) {
        field.setValue(event.getNewVolume());
    }
    public void actionPerformed(ActionEvent arg0) {

        getController().notifyVolumeChanged(Integer.pa
    rseInt(field.getValue().toString()));
    }
}
```

Les vues : JFrameListVolume.java

```
public class JFrameListVolume extends VolumeView {
   private JFrame frame = null;
   private JPanel contentPane = null;
   private JList listVolume = null;
   private JScrollPane scrollVolume = null;
   private DefaultListModel jListModel = null;
   public JFrameListVolume(VolumeController controller) {
        this(controller, 0);
   }
   public JFrameListVolume(VolumeController controller, int
   volume){
        super(controller);
        buildFrame(volume);
   }
   private void buildFrame(int volume) {
        frame = new JFrame();
        contentPane = new JPanel();
        jListModel = new DefaultListModel();
        jListModel.addElement(volume);
```

Les vues : JFrameListVolume.java

```
        listVolume = new JList(jListModel);
        scrollVolume = new JScrollPane(listVolume);
        contentPane.add(scrollVolume);
        frame.setContentPane(contentPane);
        frame.setTitle("JFrameListVolume");
        frame.pack();
    }
    @Override
    public void close() {
        frame.dispose();
    }
    @Override
    public void display() {
        frame.setVisible(true);
    }
    public void volumeChanged(VolumeChangedEvent event) {
        jListModel.addElement(event.getNewVolume());
    }
}
```

Les vues : JFrameSpinnerVolume.java

```
public class JFrameSpinnerVolume extends VolumeView implements
   ActionListener{
   private JFrame frame = null;
   private JPanel contentPane = null;
   private JSpinner spinner = null;
   private SpinnerNumberModel spinnerModel = null;
   private JButton button = null;
   public JFrameSpinnerVolume(VolumeController controller) {
        this(controller, 0);
   }
   public JFrameSpinnerVolume(VolumeController controller, int
   volume){
        super(controller);
        buildFrame(volume);
   }
   private void buildFrame(int volume) {
        frame = new JFrame();
        contentPane = new JPanel();
        spinnerModel = new SpinnerNumberModel(volume, 0, 100, 1);
        spinner = new JSpinner(spinnerModel);
        contentPane.add(spinner);
        button = new JButton("Mettre à jour");
        button.addActionListener(this);
        contentPane.add(button);
```

Les vues : JFrameSpinnerVolume.java

```
        frame.setContentPane(contentPane);
        frame.setTitle("JFrameSpinnerVolume");
        frame.pack();
    }
    @Override
    public void close() {
        frame.dispose();
    }
    @Override
    public void display() {
        frame.setVisible(true);
    }
    public void volumeChanged(VolumeChangedEvent event) {
        spinnerModel.setValue(event.getNewVolume());
    }
    public void actionPerformed(ActionEvent arg0) {
    getController().notifyVolumeChanged(spinnerModel.getNumber().in
    tValue());
    }
}
```

Classe lanceur

la classe "main" de l'application: elle crée un nouveau modèle, crée un nouveau contrôleur en lui passant le modèle et demande au contrôleur d'afficher les vues.

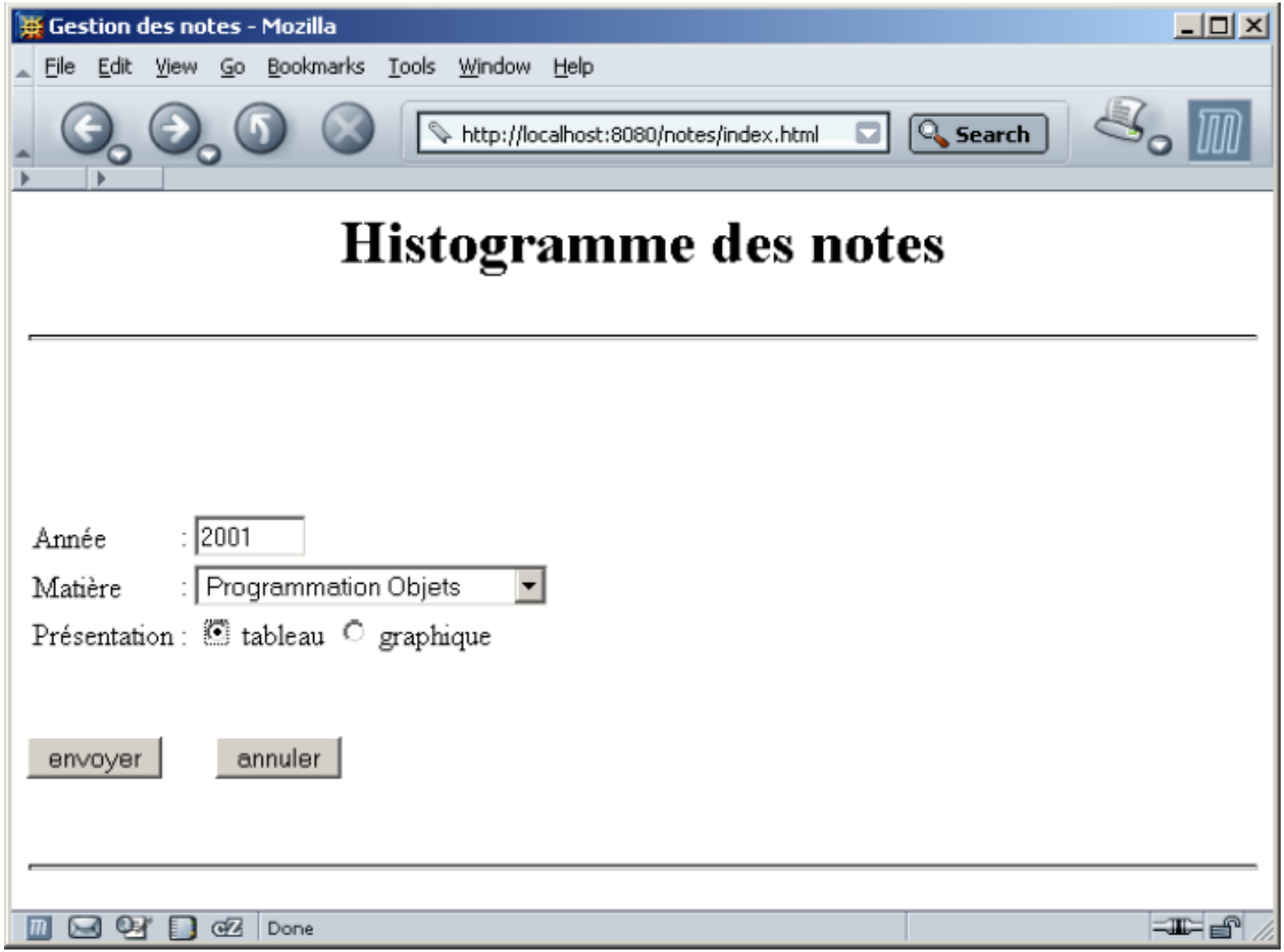
JVolume.java

```
public class JVolume {
public static void main(String[] args) {
  VolumeModel model = new VolumeModel(50);
  VolumeController controller = new
   VolumeController(model);
  controller.displayViews();
}
}
```

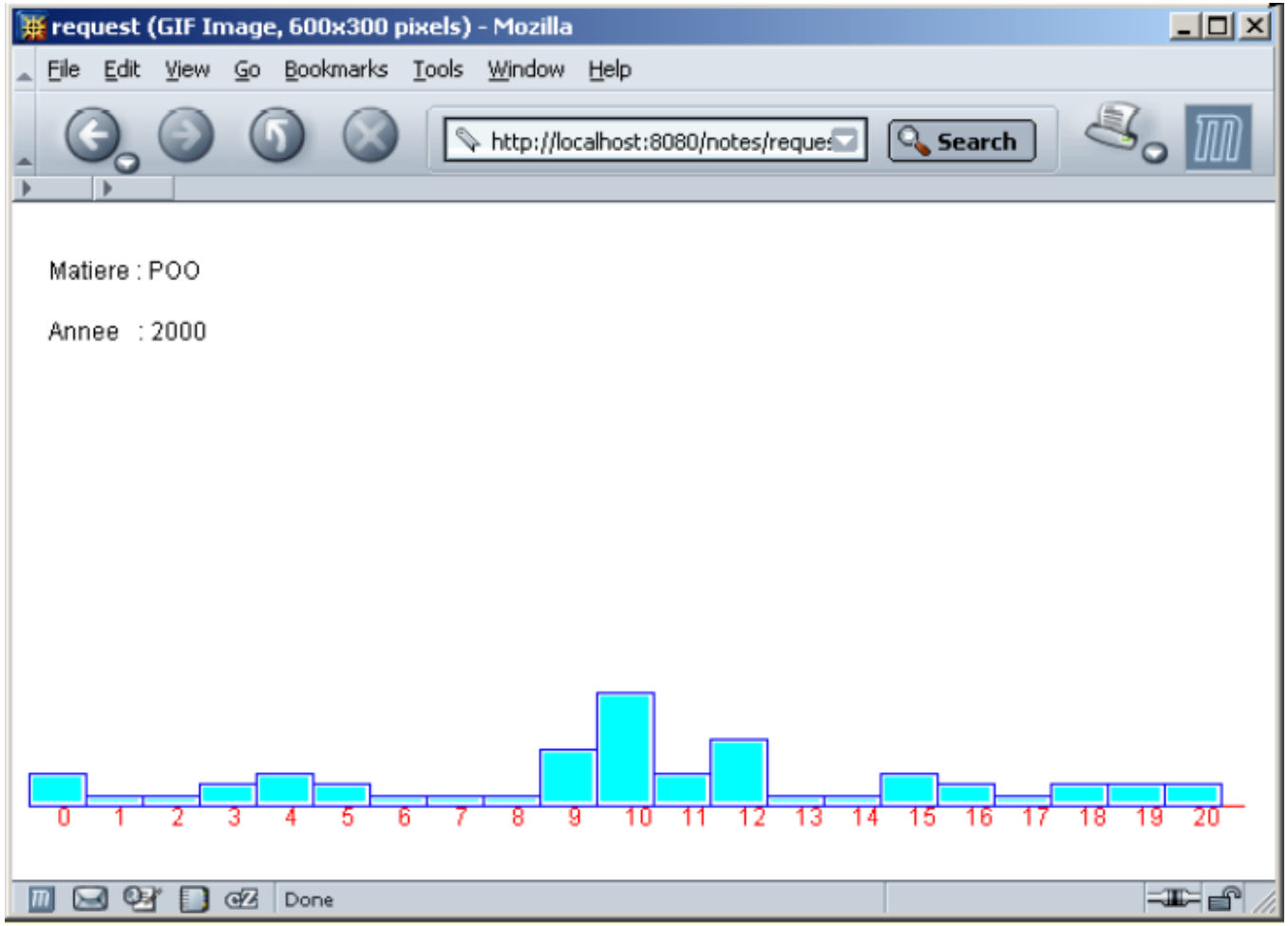
Plan de cette présentation

- L'Architecture MVC
- Un exemple interne Swing : UI-delegate
- Un exemple classique : les listes
- Un exemple complet de MVC 1 pour une application Swing
- Exemple MVC 2 : une application Web

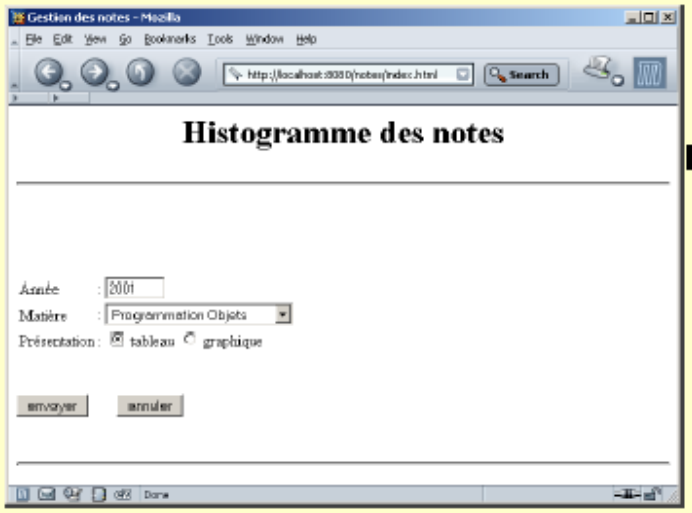
Exemple d'application Web



Deux vues des mêmes données



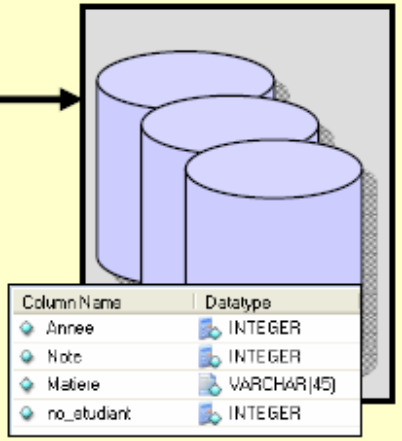
Architecture



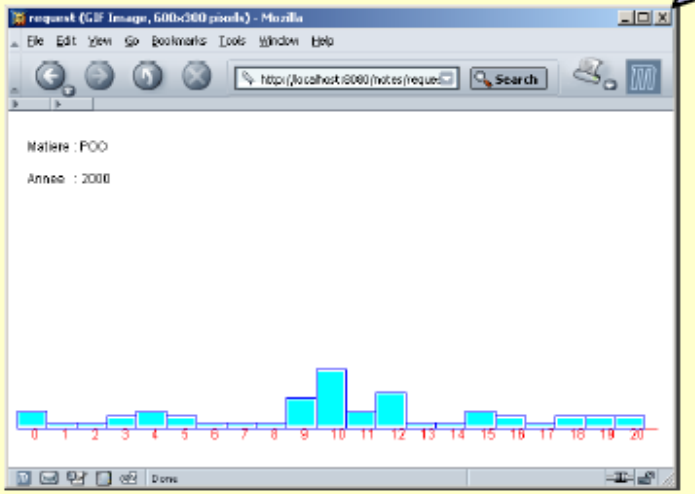
serveur Web

Quelle architecture pour le tier web ?

Serveur SGBD



Histogramme des notes image gif



Page HTML avec fréquences des notes dans un tableau

Modèle MVC

Modèle Model View Controler

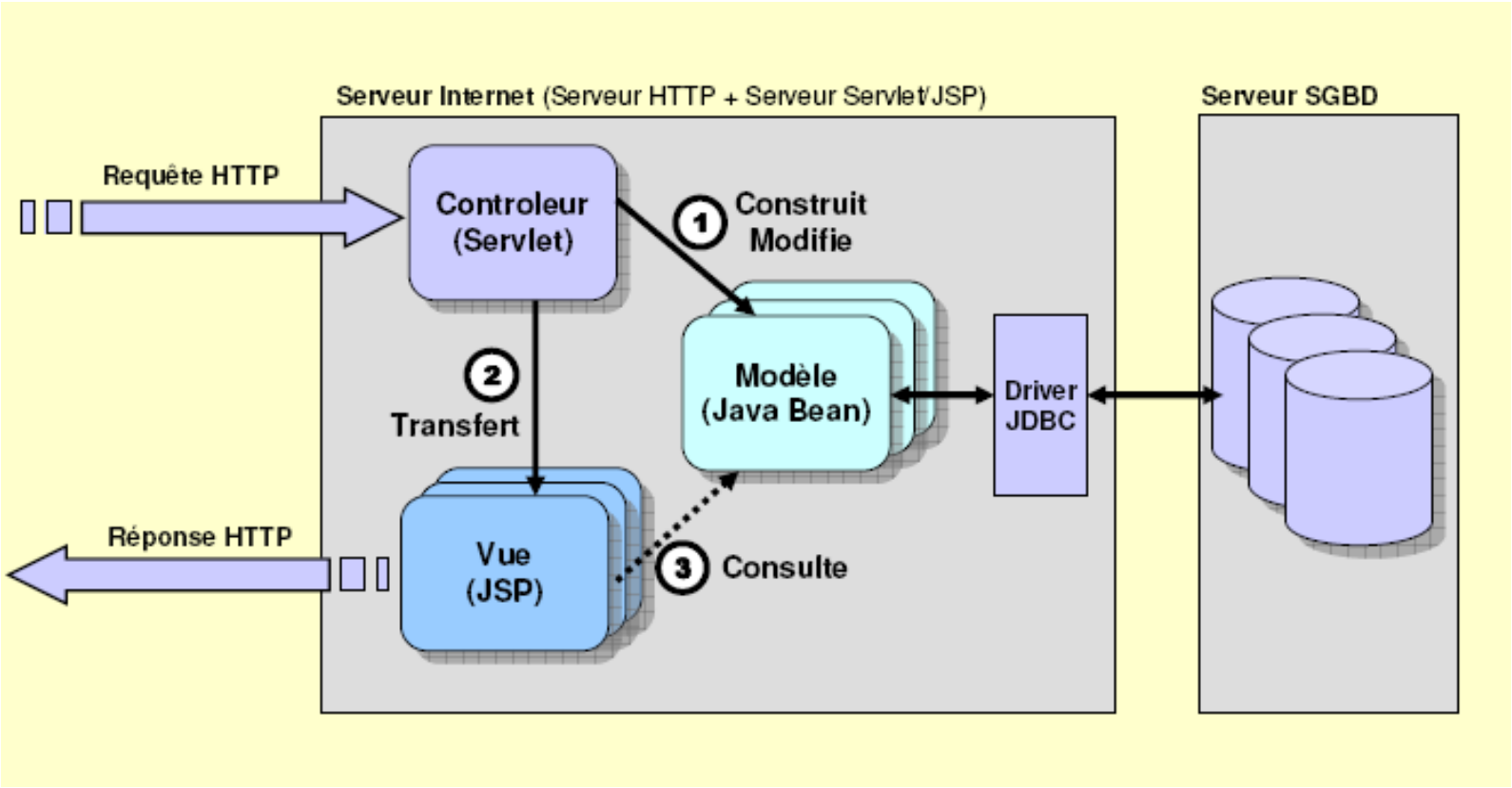
(Modèle 2 pour applications JSP)

- **Contrôle**
  - *Analyse des requêtes de l'utilisateur*
  - *Mise en place des traitements*
  - *Mise en place de la présentation*
- **Modèle**

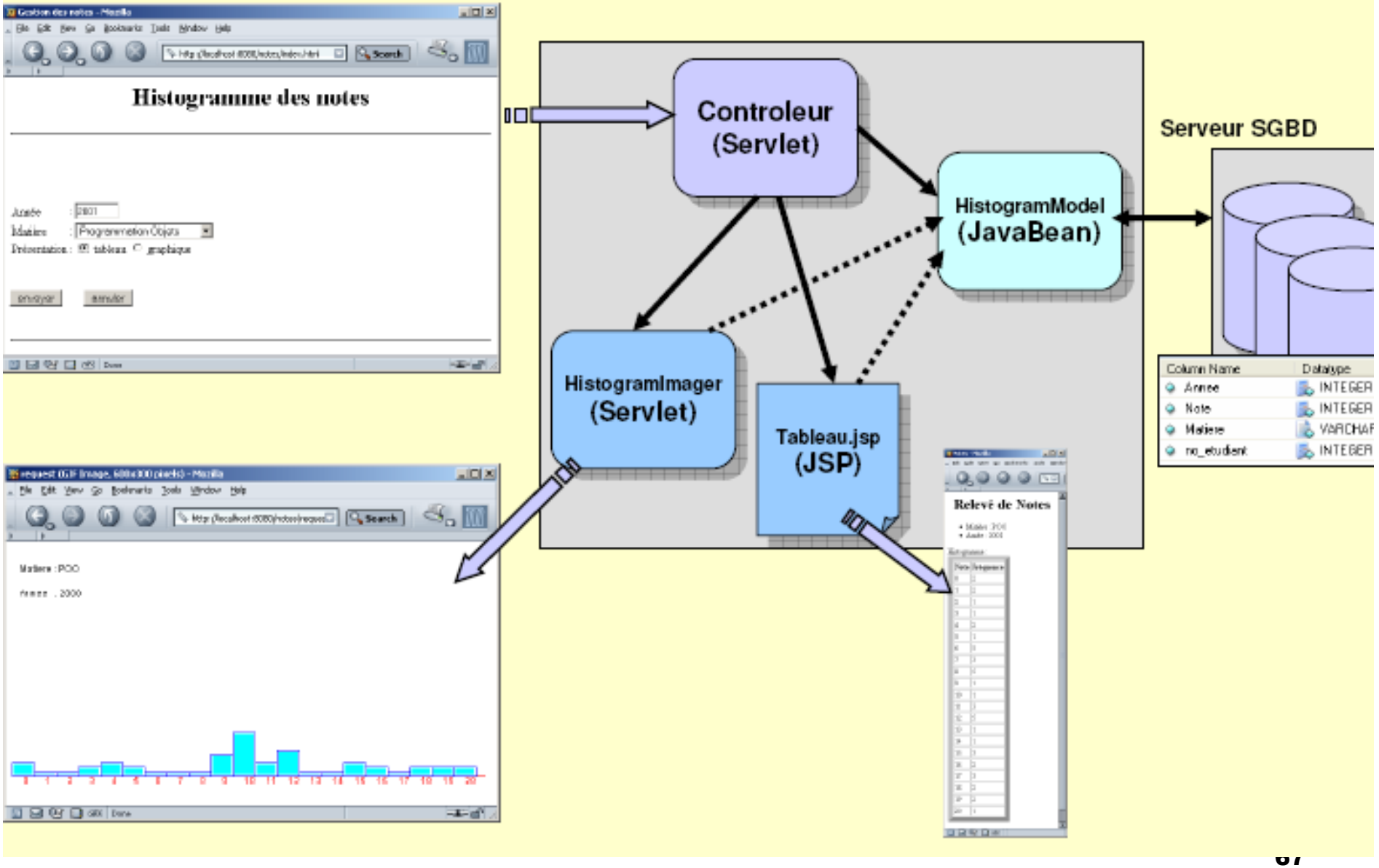
  Effectue les traitements – couche métier
- **Vue**

  Présentation des résultats

Architecture



Pour notre exemple :



Contrôleur

```
/** Réponse à une requête de type <CODE>get</CODE>
* @param request données associées à la requête HTTP
* @param response données associées à la réponse **/
public void doGet
        (HttpServletRequest request,HttpServletResponse response)
        throws IOException, ServletException {
   //-----------------------------------------------------------
   // Analyse de la requête HTTP
   //-----------------------------------------------------------
   ...1
   //-----------------------------------------------------------
   // Construction du modèle
   //-----------------------------------------------------------
   ...2
   //-----------------------------------------------------------
   // Génération de la présentation
   //-----------------------------------------------------------
   ...3
}
```

Contrôleur : analyse de la requète

1

```
int annee = 0 ;
String matiere = null ;
String presentation = null ;
try {
   annee = Integer.parseInt(request.getParameter("annee")) ;
   matiere = request.getParameter("matiere") ;
   presentation = request.getParameter("presentation") ;
} catch(NumberFormatException e) {
   // ne devrait jamais arriver : test nombre en javascript
   // côté client avec blocage de l'envoie du formulaire
   response.sendError(HttpServletResponse.SC_INTERNAL_SERVER
   _ERROR, "L'année demandée n'est pas un nombre") ;
   return ;
}
```

Controleur : construction du modèle

2

```
HistogramModel notes = null ;

// récupération de l'objet session ou création si il n'existe pas
HttpSession session = request.getSession(true) ;
notes = (HistogramModel) session.getAttribute("notes") ;

if (notes==null) { // on crée le beans et on l'enregistre dans la
   session courante
   notes = new HistogramModel() ;
   session.setAttribute("notes", notes) ;
}

//mise à jour du modèle
notes.setAnnee(annee) ;
notes.setMatiere(matiere) ;
try {
   notes.extraireNotes() ;
} catch (SQLException e) {
   response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
   e.getMessage()) ;
return;
}
```

Controleur : transfert vers la vue

3

```
if (presentation.equals("tableau")) {
   // redirige réponse vers la page JSP de type tableau
try {
   getServletContext().getRequestDispatcher("/tableau.jsp").forward(re
   quest, response);
} catch (Exception e) {
   response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
   "erreur de la présentation tableau\n" + e.getMessage()) ;
   return ;
}  else          // creation de la présentation histogramme
   try {
        getServletContext().getRequestDispatcher("/graphic").forward(r
   equest, response);
   } catch (Exception e) {
        response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR,
   "erreur de la présentation graphique\n" + e.getMessage()) ;
   return ;
}
```

Exemple de JSP


```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<%@ page import="java.util.*" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Liste des clients</title>
</head>
<body>
<TABLE>
<TR><TD> Note </TD> <TD> Fréquence </TD></TR>
<%  HistogramModel m = (HistogramModel)session.getAttribute("modele");
   for(int i=0; i<=20; i++) {
        out.println("<TR>");
        out.println("<TD>"+i+"</TD>");
        out.println("<TD>"+m.getFrequence(i)+"</TD>");
        out.println("</TR>");
   }
%>
</TABLE>
</body>
</html>
```

Servlet ou JSP : appels au modèle

- **Dans la servlet :**

```
HttpSession session = requete.getSession(true) ;
HistogramModel notes =
  (HistogramModel) session.getAttribute("notes") ;
…
```

- **Dans la page JSP :**

```
<%@ page import="model.HistogramModel" %>
<jsp:useBean id="notes" class="model.HistogramModel"
  scope="session"/>
<p>Matière : <%=notes.getMatiere() %></p>
<p>Année :   <jsp:getProperty name="notes"
                property="annee"/> </p>
```

Fichier de déploiement

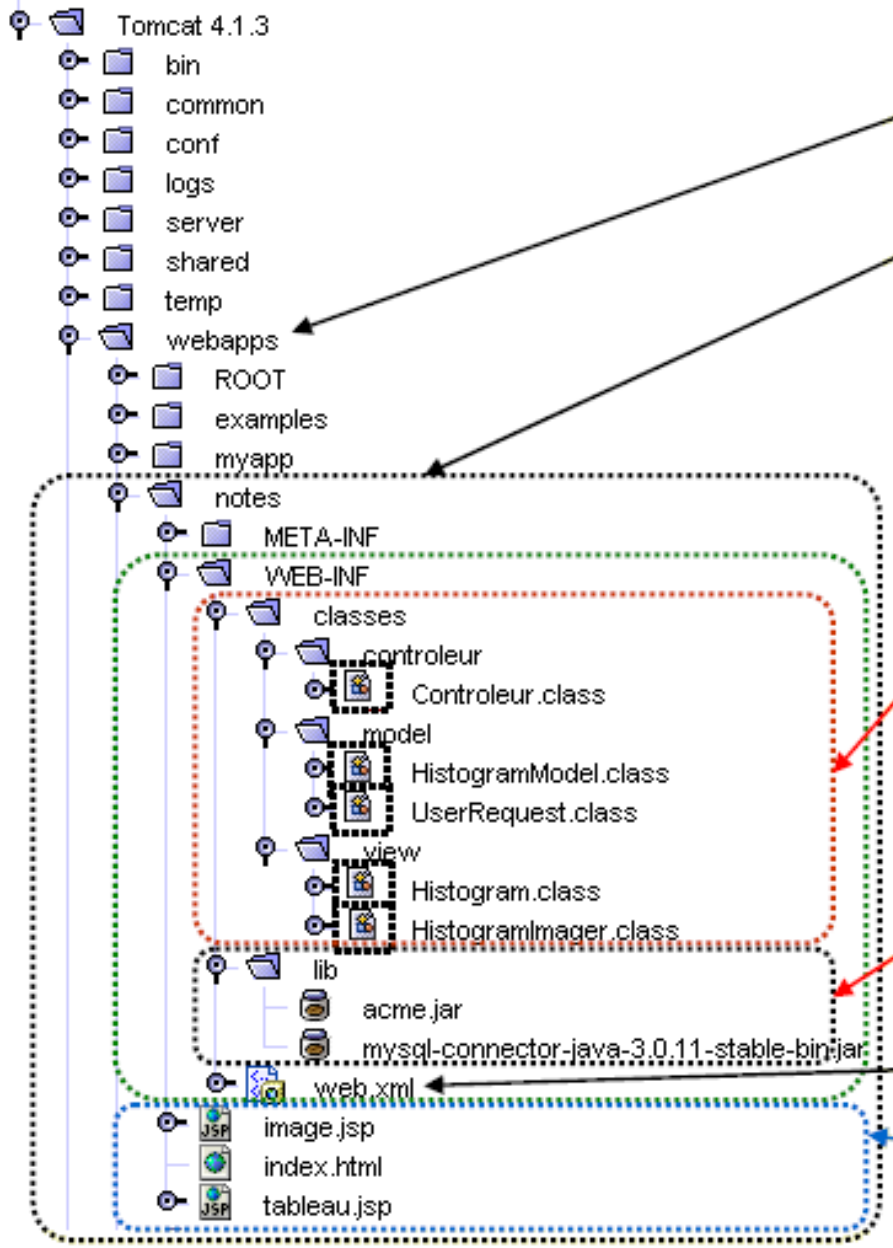
- Le fichier web.xml (WebContent/WEB-INF) contient la description de toutes les servlets et JSP à déployer
- Il définit les URLs d'accès à ces éléments
- Généré automatiquement (Eclipse…) ou défini « à la main »

Web.xml : exemple

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp1" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <display-name>ProjetServlets</display-name>
    <servlet>
        <description>
        </description>
        <display-name>MaServlet</display-name>
        <servlet-name>MaServlet</servlet-name>
        <servlet-class>controleur.MaServletImplV2</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>MaServlet</servlet-name>
        <url-pattern>/run</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Dans le navigateur :
http://localhost:8080/WebApp1/run

Organisation générale de l'appli.



Répertoire d'installation des applications web

A chaque application web correspond un répertoire dont le contenu respecte une structuration normalisée *à partir de la spécification 2.2 des servlets*

Classes java (et ressources associées) nécessaires à l'application

Si organisation en packages cela doit être traduit dans la hiérarchie des sous répertoires de classes

Les `.jar` des bibliothèques non standard de classes utilisées par l'application (Attention !! pas de .zip)

Descripteur de déploiement de l'application

Pages et tous les fichiers accessibles par client du serveur web
(`*.html`, `*.jsp`, `*.css`, images, etc)