

Architectures Logicielles pour les Applications Interactives

Marc Christie

marc.christie@univ-rennes1.fr

M2

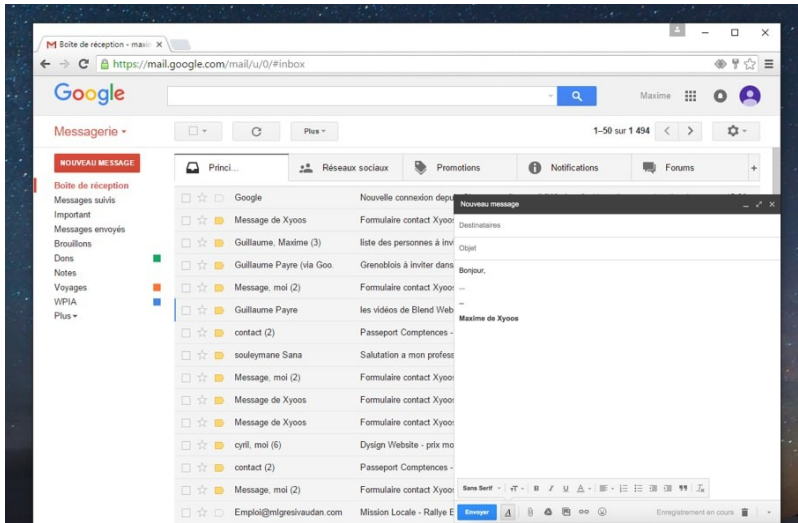
Contexte

=> Connaissez-vous une architecture logicielle pour les applications interactives?

=> En quoi cette architecture est une bonne architecture?

=> Quel est même l'intérêt de mettre en place une architecture logicielle pour les applications interactives?

Contexte



IHM

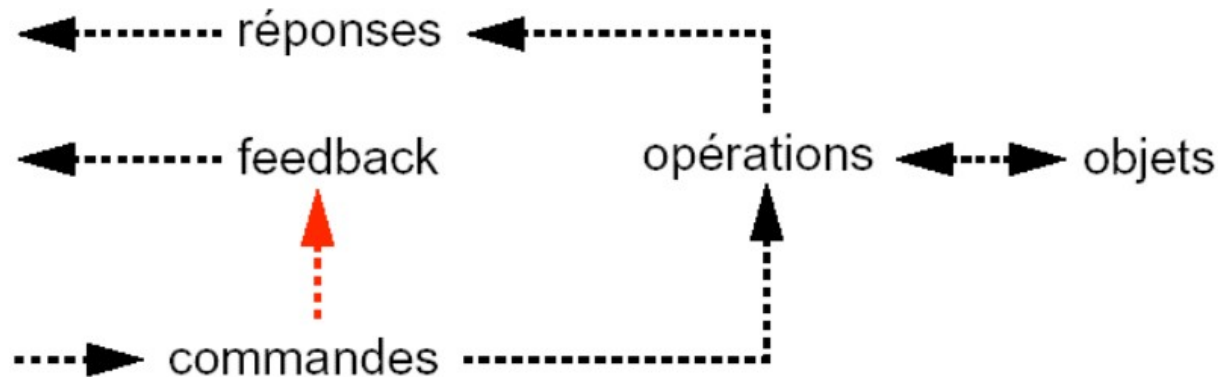


Données et
traitements

Noyau fonctionnel

=> vers une séparation des préoccupations (SoC = separation of concern)

Contexte



src: M. Beaudoin Lafon

- Complexité de la conception d'applications interactives
 - IHM=un processus de conception incrémental
 - implique de nombreuses modifications/extensions sur la partie fonctionnelle
 - Un noyau fonctionnel parfois partagé avec différentes IHM ou couches applicatives
 - les besoins et des contraintes peuvent être différents
 - Importance de la synchronisation entre l'état des objets et l'état de la vue sur les données

Implémenter une AL?

- « Oui, on fera ça à la fin » - *dit le développeur*
- « On code tout dans l'IHM, de toutes façons ça ne sera pas réutilisé » - *dit le visionnaire*
- « L'architecture? On code le noyau, on code l'interface, et on connecte...» - *dit le séparateur de préoccupations*

Quand ca ne fonctionne pas...

- **Contexte:** Re-développement d'une IHM pour les traders (société Calyon)
- **Objectif:** même noyau fonctionnel (en C++), passage à une IHM JAVA (au lieu de Qt/C++) via JNI (Java Native Interface)
- **Problème:** a nécessité la refonte **intégrale** de l'application, 2 ans de retard à cause du mécanisme évènementiel Qt utilisé **dans le modèle.**

Difficulté

- Comment lier efficacement un noyau fonctionnel et une interface utilisateur
 - En impactant au minimum le noyau?
 - En permettant une bonne réutilisabilité des développements (e.g. reprendre une partie de l'application)
 - En permettant une bonne modularité (remplacement de composants par d'autres)

Séparations

- Il faut assurer les séparations :
- Verticales
 - IHM \rightleftharpoons Noyau fonctionnel \rightleftharpoons Données
- Horizontales
 - Réutilisation d'une partie de l'application
 - eg. IHM tableau \rightleftharpoons noyau fonctionnel du tableau \rightleftharpoons données du tableau

Conception logicielle

- Nécessité de mettre en place des architectures de conception:
 - Modularité / flexibilité (SoC: separation of concerns)
 - Réutilisabilité
- Nécessité de communication interne à l'équipe
 - (rétro) Conception
 - Pérennité de l'application

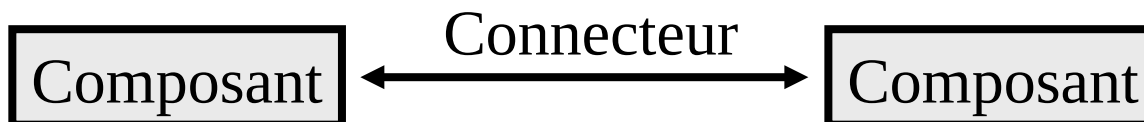
Architectures Logicielles ...

IEEE 1471 standard (2000)

« The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution »

Nécessite la définition:

- des composants
 - nature, contours et fonctionnalités des composants
- des relations (connecteurs)
 - nature des liens entre les composants
 - dépendances entre composants
- et d'un schéma directeur
 - pour guider le processus de conception



... pour l'IHM

Idée générale: séparation entre

- la sémantique :
 - données de l'application
 - fonctionnalités de l'application
- la présentation:
 - représentation graphique de l'application
 - interaction avec l'application

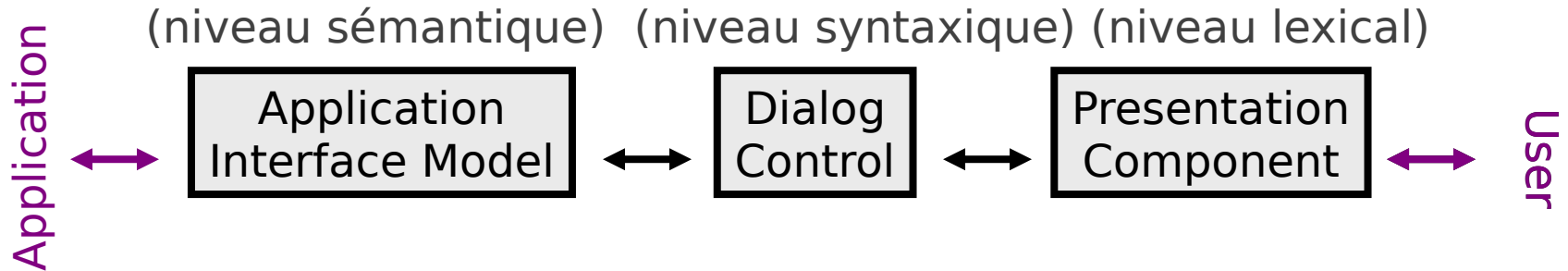
Partie 1 – Les clients lourds

Les différentes modèles d'architectures

- Les modèles basés « couches applicatives » à « grain macroscopique » travaillent à l'échelle des préoccupations
 - Modèle de Seeheim [83]
 - Modèle en Arche [92]
- Les modèles basés « agents » ou « à grain microscopique » travaillent à l'échelle des classes/fonctions
 - Modèle-Vue-Contrôleur (MVC)
 - Présentation-Abstraction-Contrôle (PAC, PAC Amodeus)
 - Modèle-Vue-Présentation (MVP)
 - Modèle-Vue-VueModèle (MVVM)

Modèle de Seeheim [83]

patron de conception abstrait



- **Application Interface Model:**
 - Offre une API pour accéder aux données/fonctionnalités de l'application (eg Design Pattern Facade)
- **Dialog Control**
 - Adapte le contenu des données à la présentation
 - Etablit et régule les communications entre l'interface graphique et l'interface du modèle
 - Garantit la cohérence d'accès aux fonctionnalités (e.g. en cas de mode Edition/Suppression...)
- **Presentation Component**
 - Offre l'ensemble des composants graphiques à l'utilisateur
 - Prend en charge les interactions utilisateur

Modèle en Arche

patron d'architecture logicielle [82]

régulation des tâches
séquencement de l'interaction
liés au modèle de tâches

modèle utilisateur versus
modèle système
réorganisation des données

Contrôleur de Dialogue

Possibly adapted Domain Objects

Logical Presentation Objects

Adaptateur

- Functional Core Adapter (Virtual Application Layer)

modifiable
portable
(versus efficacité)

Présentation

- Logical Presentation Component (Virtual Toolkit)

Domain Objects

Physical Interaction Objects

Domaine

- Application Functional Core

presentation
widgets
look and feel

Interaction

- Interaction Toolkit Component
- Physical Presentation Component

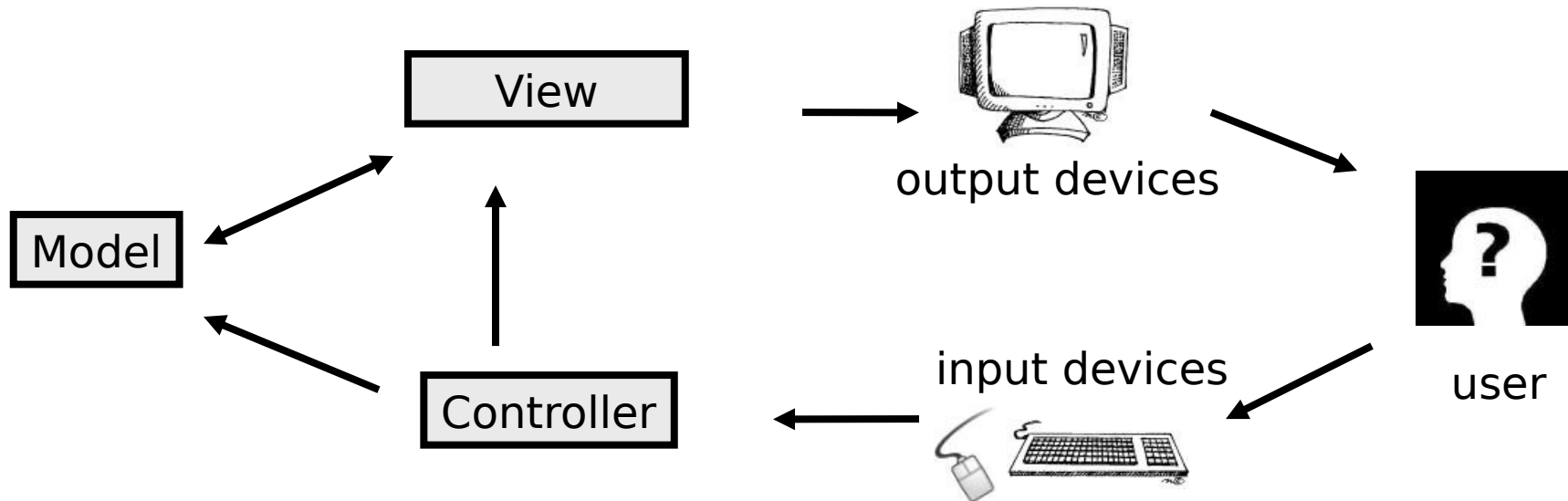
Component

- Domain-specific component
- **Domaine:** noyau fonctionnel
- **Adaptateur:** adaptation des objets au contrôleur (typage, différence de structures...)
- **Contrôleur:** établit le lien entre les objets de l'application et leur présentation logique
- **Présentation:** représentation logique des objets nécessaires à l'interaction
- **Interaction:** représentation interactive des objets (i.e. avec le toolkit cible)

Les différentes modèles d'architectures

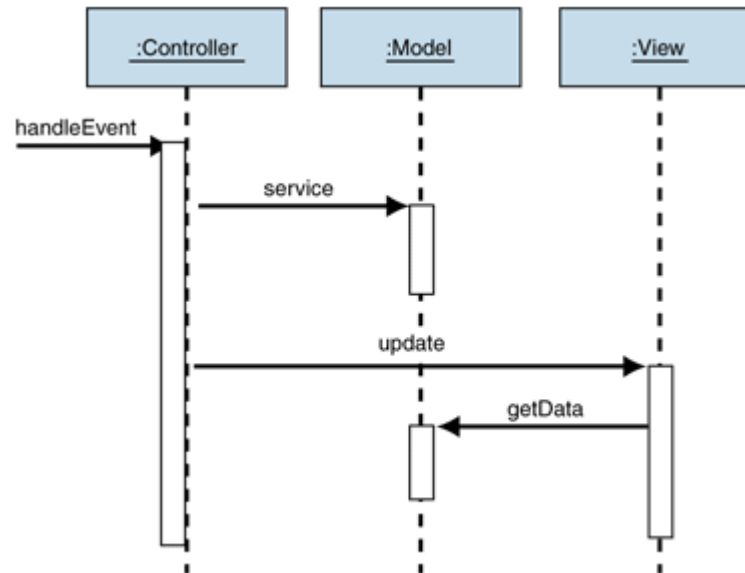
- Les modèles basés « couches applicatives » à « à grain macroscopique » travaillent à l'échelle des préoccupations
 - Modèle de Seeheim [83]
 - Modèle en Arche [92]
- Les modèles basés « agents » ou « à grain microscopique » travaillent à l'échelle des classes/fonctions
 - Modèle-Vue-Contrôleur (MVC)
 - Présentation-Abstraction-Contrôle (PAC, PAC Amodeus)
 - Modèle-Vue-Présentation (MVP)
 - Modèle-Vue-VueModèle (MVVM)

Model View Controller



- **Model:** données et fonctionnalités de l'application
 - Le modèle ne connaît pas le contrôleur, il peut connaître la vue (directement ou indirectement)
 - Le modèle est mis à jour uniquement par le contrôleur
- **View:** représentation graphique des éléments visuels
 - La vue peut connaître le modèle (pour aller chercher des données - *approche dirigée par la vue*)
 - La vue est mise à jour par le contrôleur (ou par le modèle - *approche dirigée par le modèle*)
- **Controller:** prise en compte de l'interaction utilisateur
 - Le contrôleur connaît la vue et le modèle
 - Le contrôleur est activé par l'utilisateur, modifie le modèle et modifie/notifie la vue

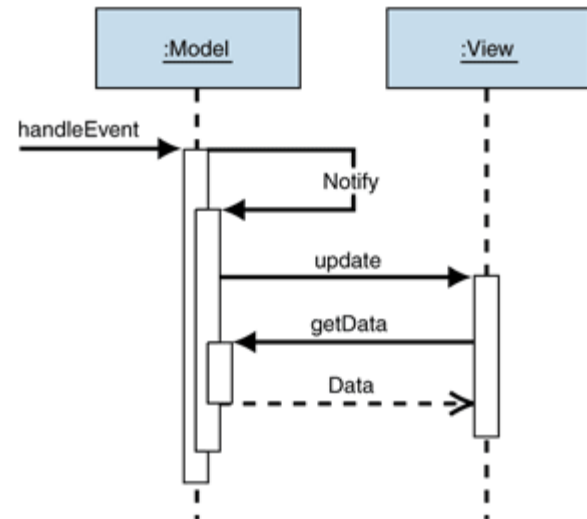
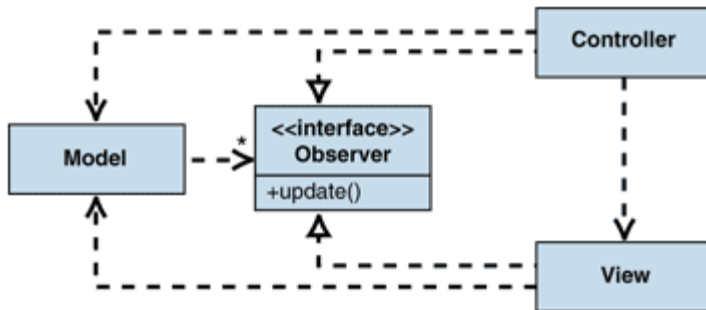
Model View Controlleur



- Approche « dirigée par la vue »: la vue connaît le modèle et récupère les données nécessaires à l’affichage

Model View Controller

- Approche « dirigée par la vue »: avec utilisation d'un pattern Observer:
 - Le Controller et la View implémentent l'interface « Observer »
 - Le Controller connaît la View et le Model
 - Le View connaît le Model (mais pas le Controller)
 - Le Model connaît seulement ses « Observers » (c'est l'observable)



Model View Controlleur

- Un exemple simple en Java: **le monitoring cardiaque:**
 - Modèle: classe qui représente le rythme cardiaque
 - Méthodes pour augmenter ou réduire le rythme des battements
 - Notifie la vue à chaque battement
 - Vue:
 - Affiche les pulsations par minutes
 - Est notifié par le modèle d'un nouveau battement
 - Contrôleur:
 - Offre les widgets pour augmenter ou réduire le rythme des battements

Modèle (1)

- L'objet modèle:
 - Dérive de la classe `Observable`
 - Implémente une méthode `updateObservers()` qui notifie tous les observateurs de l'objet:

```
private void updateObservers() {  
    setChanged();  
    notifyObservers(new Long(numberOfBeats));  
}
```

Modèle (2)

- En Java: class HeartModel

```
public void run() {
    while (running) {
        numberOfBeats++;
        updateObservers();
        try {
            Thread.sleep(SLEEP_DURATION * excitementLevel);
        } catch (InterruptedException e) {
            // do nothing if interrupted
        }
    }
}

public void stopHeart() {
    running = false;
}

public void adjustExcitementLevel(long delta) {
    excitementLevel += delta;
    if (excitementLevel < 1) {
        excitementLevel = 1;
    }
}
```

Modèle (2)

- Les méthodes `run()`, `stopHeart()` et `adjustExcitementLevel()` sont disponibles pour le contrôleur
- Les attributs de la classe `HeartModel`:
 - `excitementLevel`
 - `numberOfBeats`
 - `running`

Contrôleur

- Classe HeartController

Possède trois boutons:

- Stop: pour arrêter le cœur
- Excite: pour augmenter le rythme des battements
- Calm: pour diminuer le rythme des battements

```
stopButton.addActionListener(action);
exciteButton.addActionListener(action);
calmButton.addActionListener(action);
...
class MyAction implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        Object object = event.getSource();
        if (object == calmButton)
            calm_ActionPerformed(event);
        else if (object == exciteButton)
            excite_ActionPerformed(event);
        else if (object == stopButton)
            stop_ActionPerformed(event);
    }
}
...
void calm_ActionPerformed(ActionEvent event) {
    // myHeart est une reference vers le modèle (donné à la construction)
    myHeart.adjustExcitementLevel(INCREASE_AMOUNT);
}
```


Vue

- Classe HeartView:
 - Implémente Observer
 - S'enregistre auprès du modèle pour être informé
 - Implémente la méthode update() de Observer pour récupérer les données à mettre à jour

```
...
public HeartView(Observable model) {
    this();
    model.addObserver(this);
}

public void update(Observable model, Object data) {
    heartbeatText.setText("" + data);
}
...
```

MVC – Passage à l'échelle

Dans une application de taille conséquente:

- Modèle: problème de la notification
 - Que notifier à la vue? Quel niveau de granularité?
L'IHM doit-elle recharger l'ensemble de l'information?
 - Multiples notifications = multiples rafraichissements
(d'où perte de réactivité de l'interface)
 - Fréquence des notifications/complexité de la vue
- Vue:
 - Quelles données transmettre? veut-on toujours tout remettre à jour dans l'interface

MVC - Modularité

Avec MVC, le couplage entre les classes est **fort**

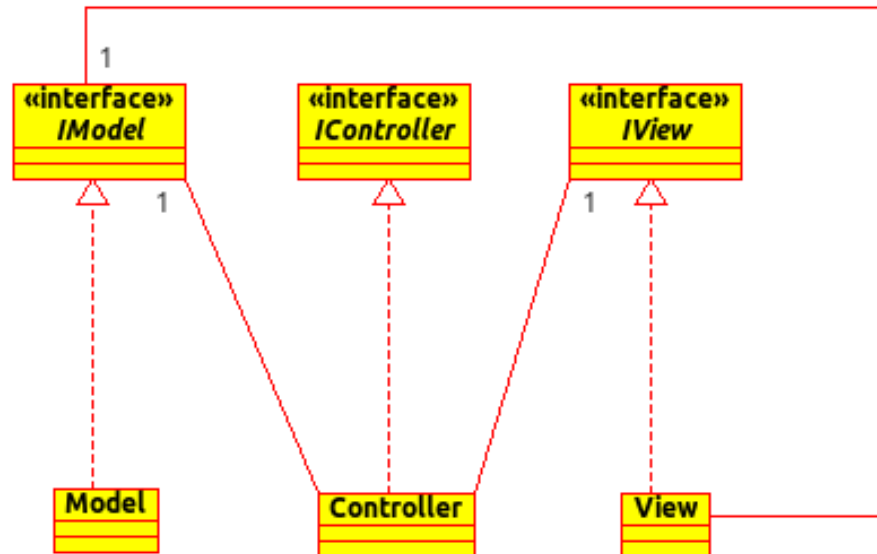
- Problème de la Vue qui connaît le Modèle
 - Et si le modèle doit être remplacé?
- Problème du Controller qui connaît la Vue
 - Et si la vue doit être remplacée?
- Problème de la dépendance aux représentations internes des vues
 - De nombreux widgets (eg. JTable en Swing) proposent une représentation de modèle interne
 - Il faudrait des adaptateurs entre les données (indépendantes) du modèle et les données de la vue

MVC et impact sur le modèle

- Le pattern MVC impacte le modèle:
 - Appel de `notifyObservers()` dans le modèle !!
- Il est *préférable* d'éviter la modification du modèle, ou parfois, le modèle n'est pas modifiable (classes existantes externes à la société)
- Utilisation d'une Interface sur le modèle:
 - On utilise le polymorphisme
 - La notification se fait dans l'interface du `Modele`

MVC et Interfaces

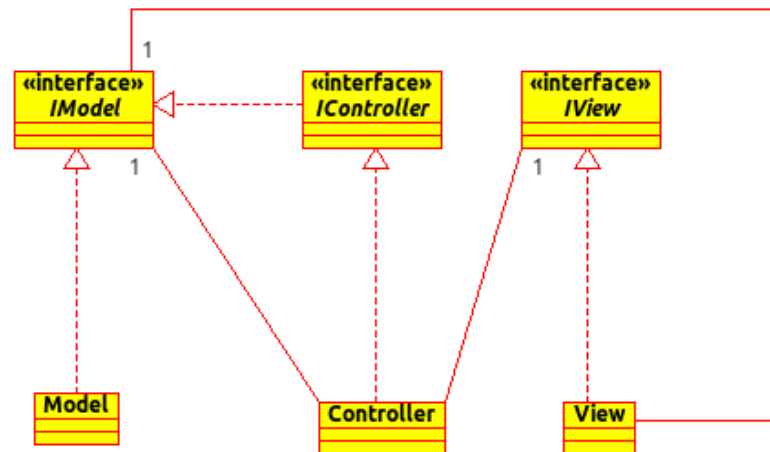
- Utilisation d'interfaces pour abstraire les composants Model/View/Controller



- Avantages:
 - Le controlleur ne connaît que des interfaces
 - La vue ne connaît pas le modèle
- Inconvénients:
 - Multiplie les classes / interfaces similaires entre IControlleur et IModel

MVC, interfaces et proxy

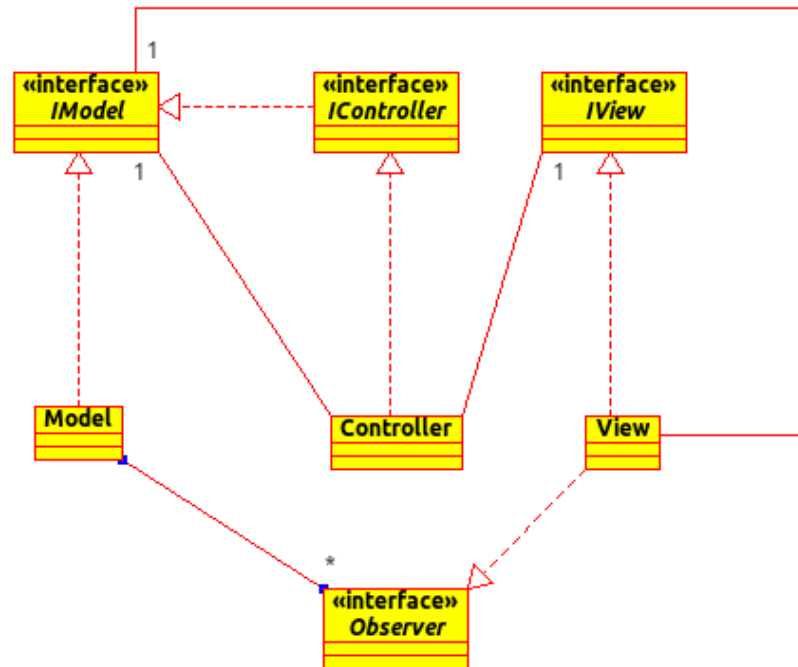
- Utilisation d'un design pattern proxy:
 - IController est un proxy du modèle (toutes les opérations sur le Model sont accessibles par le Controller)
 - Toutes les vérifications sont réalisées dans le Controller



- Problème: comment le modèle notifie la vue?

MVC, interfaces, proxy, observer

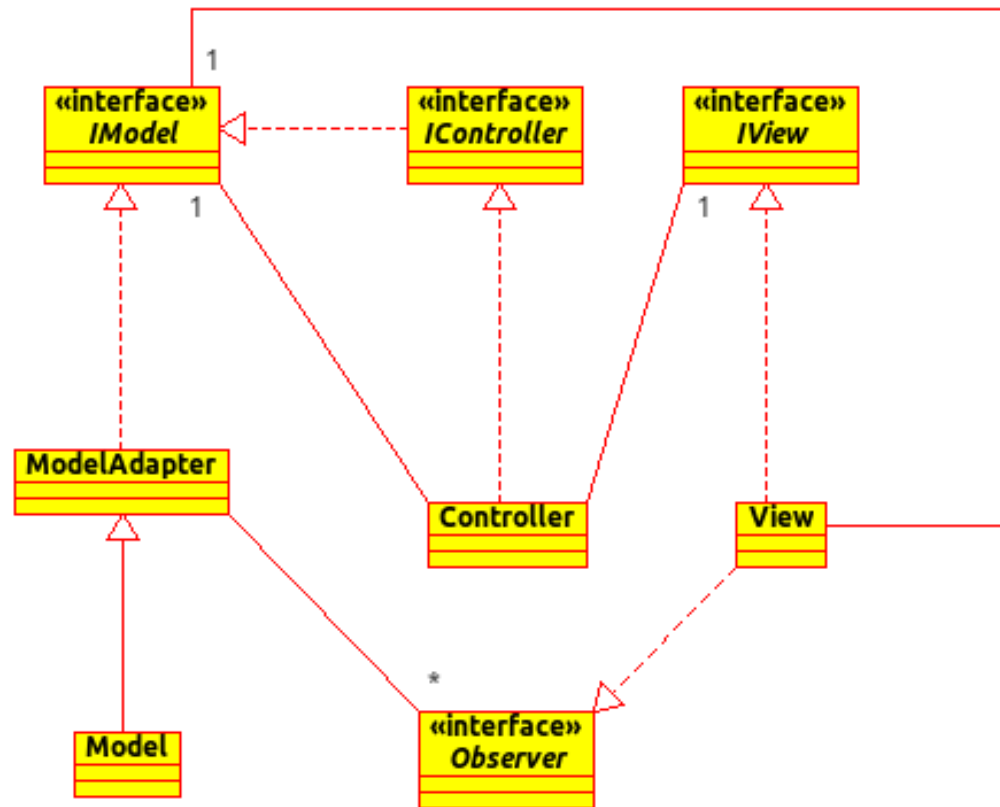
- Utilisation d'un observer pour notifier la vue



- Inconvénient: un mécanisme d'observable doit être ajouté au modèle (ce qui n'est pas toujours possible ou désiré)

MVC, interfaces, proxy, observer et délégation

- Utilisation d'une délégation pour éviter la notification dans le Model



Exemple pratique en Swing

- Un jeu de solitaire, composé de cartes qui peuvent être déplacées entre différents tas de cartes ou retournées sur un tas de cartes
- Le noyau fonctionnel **n'est pas modifiable**
- La carte (le modèle): Carte.java

Constructor Summary

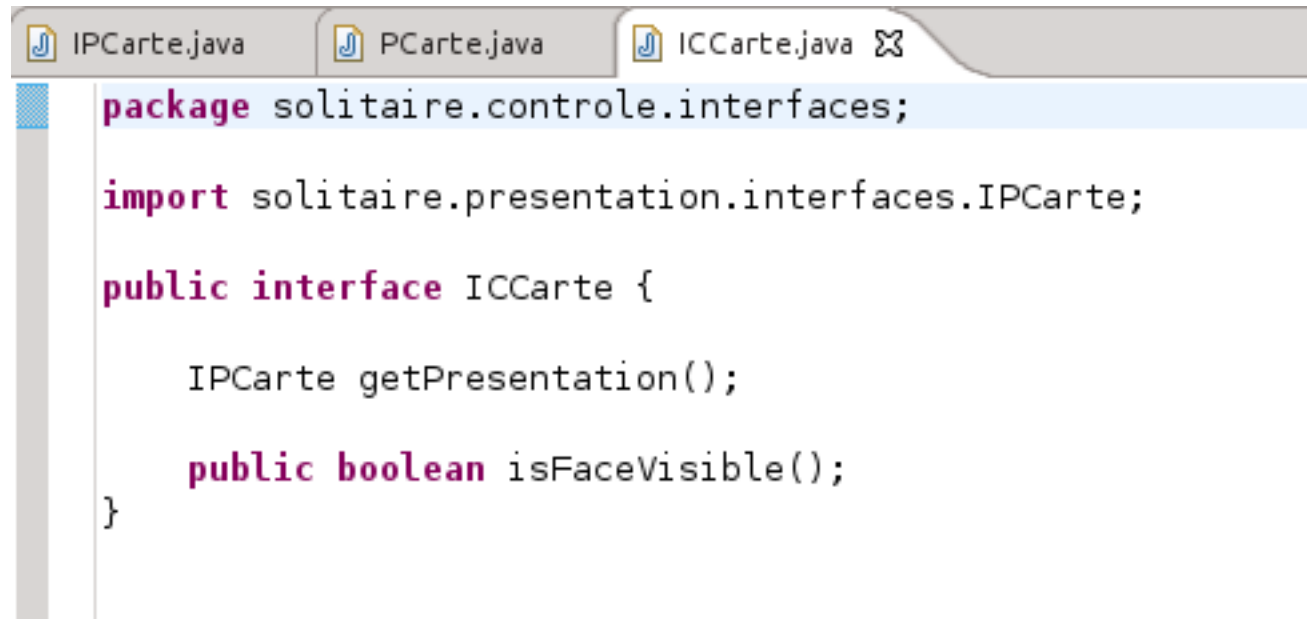
Carte (solitaire.application.Carte carte)
construction d'une carte par recopie d'une autre carte
Carte (int valeur, int couleur)
construction d'une carte à partir de 2 valeurs entières :

Method Summary

int	getCouleur ()	renvoie la couleur (entre 1 et 4) d'une carte
int	getValeur ()	renvoie la valeur (entre 1 et 13) d'une carte
boolean	isFaceVisible ()	retourne vrai si la carte est visible, faux si elle est retournée
static void	main (java.lang.String[] args)	permet de tester la classe avec un environnement minimal
void	setFaceVisible (boolean faceVisible)	permet rendre visible ou non une carte
java.lang.String	toString ()	méthode de conversion d'une carte en une chaîne de caractères, pour un affichage textuel

L'interface de controleur

- ICCarte.java



```
package solitaire.control.interfaces;

import solitaire.presentation.interfaces.IPCarte;

public interface ICCarte {

    IPCarte getPresentation();

    public boolean isFaceVisible();
}
```

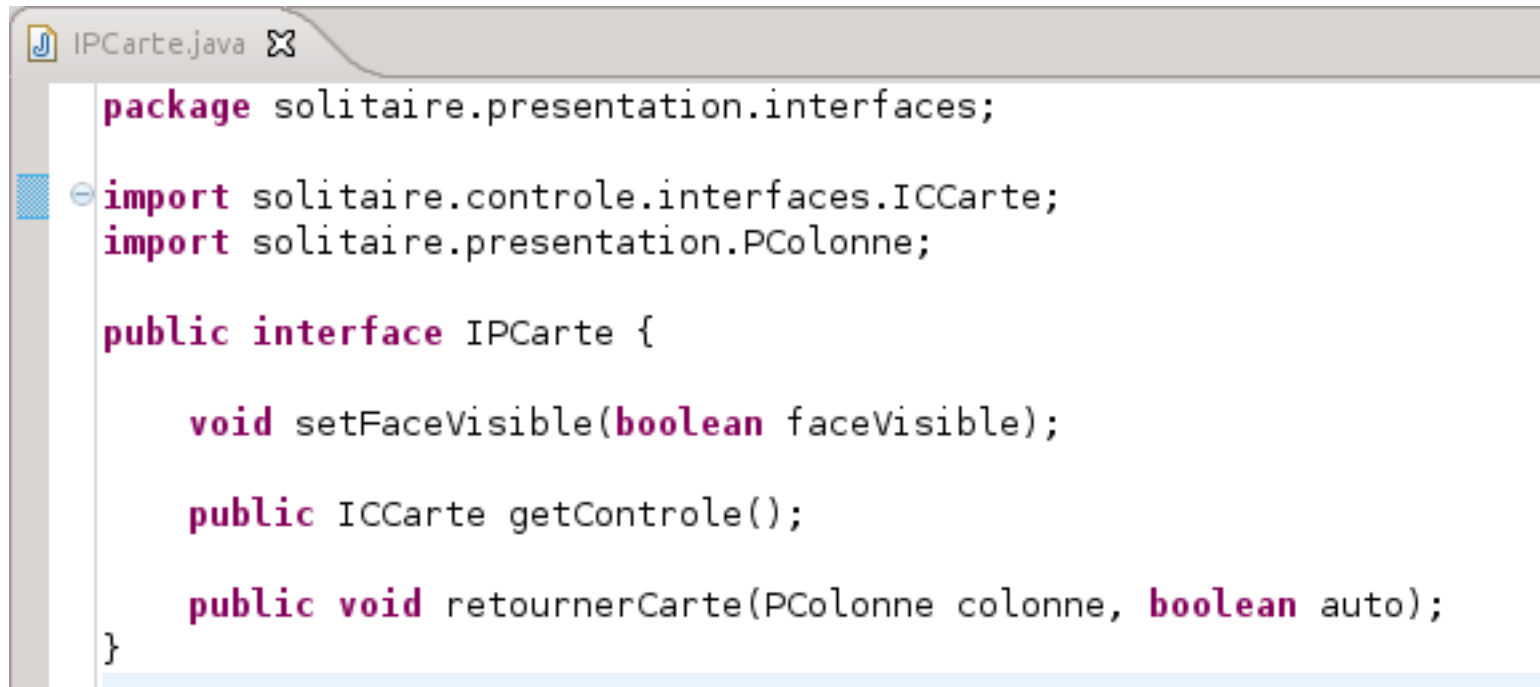
Le Controlleur

- CCarte.java: Le controlleur dérive du

```
package solitaire.controle;  
  
import java.io.Serializable;  
  
public class CCarte extends Carte implements ICCarte, Serializable {  
    IPCarte presentation;  
  
    public CCarte(int valeur, int couleur) {  
        super(Math.max(1, Math.min(valeur, 13)), Math.max(1, Math.min(couleur, 4)));  
        presentation = new PCarte(valeurs[getValeur() - 1] + couleurs[getCouleur() - 1], this);  
    }  
  
    @Override  
    public IPCarte getPresentation() {  
        return presentation;  
    }  
  
    @Override  
    public void setFaceVisible(boolean b) {  
        super.setFaceVisible(b);  
        presentation.setFaceVisible(b);  
    }  
}
```

L'interface de présentation

- IPCarte.java



```
package solitaire.presentation.interfaces;

import solitaire.controle.interfaces.ICCarte;
import solitaire.presentation.PColonne;

public interface IPCarte {

    void setFaceVisible(boolean faceVisible);

    public ICCarte getControle();

    public void retournerCarte(PColonne colonne, boolean auto);
}
```

E

Présentation (PCarte.java)

```

package solitaire.presentation;

import java.awt.Color;

public final class PCarte extends JPanel implements IPCarte, Transferable, Serializable {

    private static final long serialVersionUID = 1L;
    protected ICarte controle; // contrôleur associé
    protected JLabel face, dos;
    protected ImageIcon icone; // image de la face
    protected final static String SEP = System.getProperty("file.separator");
    protected static ImageIcon iconeDos = new ImageIcon(ClassLoader.getResource(
    public static final int largeur = iconeDos.getIconWidth() + 10;
    public static final int hauteur = iconeDos.getIconHeight() + 10;

    /**
     * initialiser une carte
     * @param chaine : nom de la carte (exemple "3H" = 3 Heart)
     */
    public PCarte(final String chaine, final ICarte controle) {
        this.controle = controle;
        // image de la face
        icone = new ImageIcon(ClassLoader.getResource("resources"
            + SEP + "cartesCSHD" + SEP + chaine + ".gif"));
        face = new JLabel(icone);
        add(face);
        face.setLocation(0, 0);
        face.setSize(largeur, hauteur);
        // image du dos
        dos = new JLabel(iconeDos);
        add(dos);
        dos.setLocation(0, 0);
        dos.setSize(largeur, hauteur);
        setLayout(null);
        setBackground(Color.white);
        setOpaque(true);
        setSize(face.getSize());
        setPreferredSize(getSize());
        setFaceVisible(false);
    }

    @Override
    public void setFaceVisible(boolean faceVisible) {
        face.setVisible(faceVisible);
        dos.setVisible(!faceVisible);
    }
}

```

Constructor Summary

[TasDeCartes](#)(java.lang.String nom, solitaire.application.Usine usine)
construction d'un tas de cartes :

Method Summary

void	depiler () dépile la carte au sommet du tas de cartes
void	empiler (solitaire.application.Carte carte) empile une carte sur le tas de cartes
void	empiler (solitaire.application.Tas tas) empile un tas de cartes sur le tas de cartes
void	empiler (solitaire.application.Tas tas, int nbCartes) empile les n cartes au sommet d'un tas de cartes sur le tas de cartes
solitaire.application.Carte	getBase () donne la carte à la base du tas de cartes
solitaire.application.Carte	getCarte (int n) donne la carte en position n du tas de cartes
java.lang.String	getNom () indique le nom du tas de cartes
int	getNombre () indique le nombre de cartes du tas de cartes
solitaire.application.Carte	getSommet () donne la carte au sommet du tas de cartes
boolean	isAlterne () indique si le tas de cartes est un tas de cartes alternées
boolean	isEmpilable (solitaire.application.Carte carte) indique si une carte est empilable sur le tas de cartes
boolean	isEmpilable (solitaire.application.Tas tas) indique si un tas de cartes est empilable sur le tas de cartes
boolean	isEmpilable (solitaire.application.Tas tas, int nbCartes) indique si les n cartes au sommet d'un tas de cartes sont empilables sur le tas de cartes
boolean	isVide () indique si un tas de cartes est vide
static void	main (java.lang.String[] args)

Modèle:
TasDeCartes.java

L'interface de controleur

ICTasDeCartes.j

```
package solitaire.controle.interfaces;

import solitaire.application.Carte;

public interface ICTasDeCartes {

    public void empiler(Carte c);

    public void depiler() throws Exception;

    public IPTasDeCartes getPresentation();

    /**
     * renvoie la carte au sommet du tas
     * @return
     * @throws Exception
     */
    public Carte getSommet() throws Exception;

    /**
     * indique si le tas est vide
     * @return
     */
    public boolean isVide();

    /**
     * retourne le nombre de cartes presentes dans le tas
     * @return
     */
    public int getNombre();
}
```

Le Controlleur

- CTasDeCartes.java: bien noter, ici, qu'il dérive directement du modèle (TasDeCartes)

```
package solitaire.controle;

import solitaire.application.Carte;

public class CTasDeCartes extends TasDeCartes implements ICTasDeCartes {

    PTasDeCartes presentation;

    public CTasDeCartes(String nom, CUsine usine) {
        super(nom, usine);
        presentation = new PTasDeCartes(this);
    }

    @Override
    public void empiler(Carte c) {
        super.empiler(c);
        presentation.empiler(((PCarte) ((CCarte) c).getPresentation()));
    }

    @Override
    public PTasDeCartes getPresentation() {
        return presentation;
    }
}
```

Dans empiler(Carte c):

- **Downcast** de la Carte vers une CCarte
- Récupération de la présentation
- **Downcast** de la présentation vers une PCarte

L'interface de présentation

- IPTasDeCartes.java

```
package solitaire.presentation.interfaces;  
  
+ import solitaire.controle.interfaces.ICTasDeCartes;  
  
public interface IPTasDeCartes {  
    public void empiler(PCarte c);  
    public void setType(TypeTas offset);  
    public ICTasDeCartes getControle();  
}
```

Présentation tas de cartes

(PTasDeCartes.java)

```
package solitaire.presentation;

import java.awt.Color;

public class PTasDeCartes extends JPanel implements IPTasDeCartes, Transferable,

    protected ICTasDeCartes controle;
    private TypeTas type;
    public static final int valOffset = 20;

    public PTasDeCartes(final ICTasDeCartes c) {
        this.setLayout(null);
        type = TypeTas.drag;
        controle = c;
        setSize(PCarte.largeur, PCarte.hauteur);
        setOpaque(false);
    }

    @Override
    public void empiler(final PCarte c) {
        c.setBackground(Color.white);
        int nb = getComponentCount();
        int x = 0;
        int y = 0;
        y = nb * valOffset;
        setSize(PCarte.largeur, PCarte.hauteur + nb * valOffset);

        // ajoute la carte comme fils de ce composant
        add(c, 0);
        // la place sur le dessus
        setComponentZOrder(c, 0);
        // a la bonne position
        c.setLocation(x, y);
        // redessine l'ensemble
        CSolitaire.getInstance().getPresentation().redraw();
    }
}
```

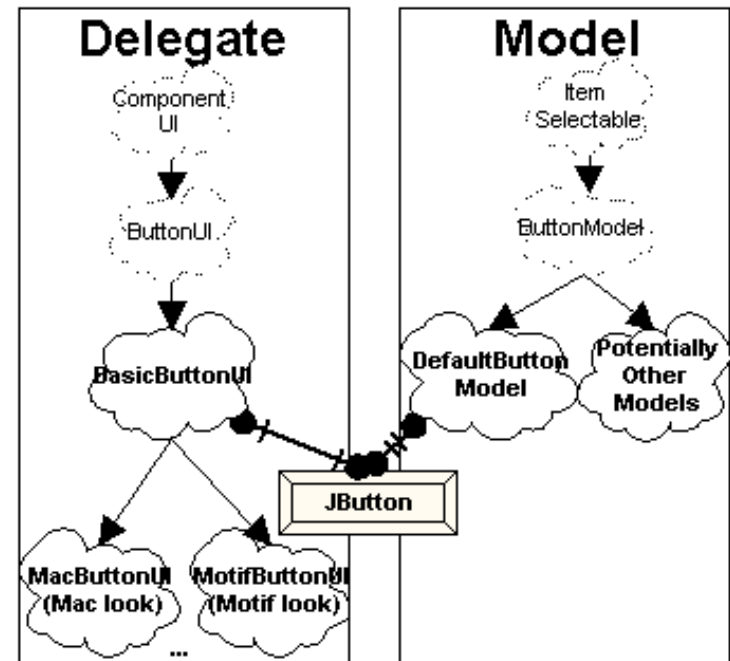
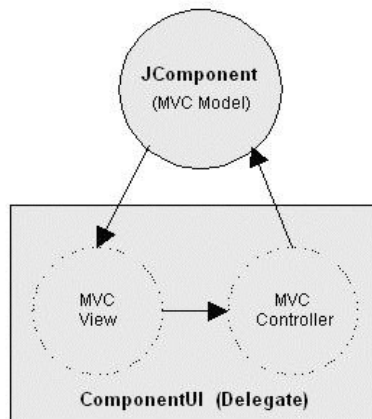
Bilan exemple pratique

- Le noyau fonctionnel **n'a pas été modifié**
- Le modèle proposé est flexible
 - modification du noyau, ou modification de la librairie graphique sans toucher à l'architecture
- Pas de notification dans le modèle (celui-ci se fait dans le contrôleur)

MVC dans Swing

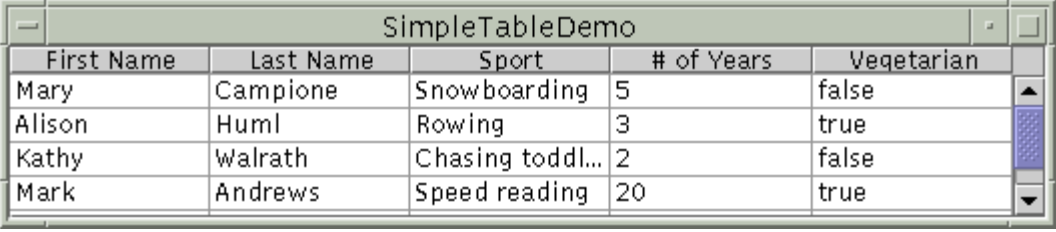
- Observer/Observable dans java.util
- MVC dans Swing :
 - Vue et Contrôleur sont remplacés par un objet **delegate**
 - JComponent = model + delegate
 - setUI() / setModel() permettent de modifier model/delegate

Swing Look & Feel



MVC et Swing

- Un composant Swing flexible: **JTable**



First Name	Last Name	Sport	# of Years	Vegetarian
Mary	Campione	Snowboarding	5	false
Alison	Huml	Rowing	3	true
Kathy	Walrath	Chasing toddl...	2	false
Mark	Andrews	Speed reading	20	true

- **JTable** sans modèle (peu d'intérêt)

```
Object[][] data = {
    {"Mary", "Campione", "Snowboarding", new Integer(5), new Boolean(false)},
    {"Alison", "Huml", "Rowing", new Integer(3), new Boolean(true)},
    {"Kathy", "Walrath", "Chasing toddlers", new Integer(2), new Boolean(false)},
    {"Mark", "Andrews", "Speed reading", new Integer(20), new Boolean(true)},
    {"Angela", "Lih", "Teaching high school", new Integer(4), new Boolean(false)}
};

String[] columnNames = {
    "First Name", "Last Name", "Sport", "# of Years", "Vegetarian"
};

JTable table = new JTable(data, columnNames);
```

JTable et TableModel

JTable avec modèle:

- un modèle implémente l'interface **TableModel**
- Une interface et une implémentation existent pour faciliter l'utilisation du modèle **TableModel** (*proposent des méthodes supplémentaires*)
 - une classe abstraite **AbstractTableModel**
 - une classe **DefaultTableModel**
- Implémentation des méthodes du modèle:
 - *int getRowCount() : retourne le nombre de ligne*
 - *int getColumnCount() : retourne le nom de colonne*
 - *Object getValueAt(int row, int column) : retourne l'objet pour la cellule*
 - *setValueAt(Object, int row, int co) : nouvelle valeur de la cellule*

JTable et TableModel

- *Class getColumnClass(int cc) : retourne le type de la colonne*
- *boolean isCellEditable(int r, int l) : indique si la cellule est modifiable*
- *String getColumnName(int coln) : retourne le nom de la colonne pour coln*
- *addTableModelListener(TableModelListener) : ajoute un écouteur qui est notifié à chaque fois que le modèle est modifié*
(notification du modèle vers la vue et/ou le contrôleur)
- Des méthodes (issues des sous-classes de **TableModel**) peuvent être utilisées directement:
 - *fireTableDataChanged() : notifie toutes les vues de modifications*
 - *fireTableRowsUpdated(int fr, int lr) : **notifie entre les lignes fr et lr (pour éviter un rafraichissement couteux)***
- L'interface **TableModelListener** contient une méthode
 - *tableChanged(TableModelEvent e) : appelée à la suite de chaque modification du modèle*

Abstract vs. DefaultTableModel

- **AbstractTableModel:**

- Implémentation partielle de **TableModel**, seules demeurent **abstraites** :
 - int getColumnCount()
 - int getRowCount()
 - Object getValueAt(int rowIndex, int columnIndex)
- Pour une application nécessitant une table non éditable il suffit de donner une implémentation à ces méthodes
- **Représente un Adapteur vers votre modèle (indépendamment de la façon dont les données sont représentées)**

- **DefaultTableModel:**

- Implémentation « simple » de **TableModel** lorsque les données sont exprimées sous la forme de tableaux (ou vecteurs) de lignes, chaque ligne étant elle-même un tableau (ou un vecteur) d'objets.
- Les constructeurs **JTable(Object[][] rowData, Object[] columnNames)** et **JTable(Vector rowData, Vector columnNames)** initialisent et utilisent un objet **DefaultTableModel**

Un exemple de AbstractTableModel

- A base d'un simple tableau

```
public class MyTableModel extends AbstractTableModel {
    private String[] columnNames = ...
    private Object[][] data = ...
    public int getColumnCount() {return columnNames.length;}
    public int getRowCount() {return data.length;}
    public String getColumnName(int col) {return columnNames[col];}
    public Object getValueAt(int row, int col) {return data[row][col];}
    public Class getColumnClass(int c) {return getValueAt(0,c).getClass();}
    public boolean isCellEditable(int row, int col) {return (col < 2);}
    public void setValueAt(Object value, int row, int col) {
        data[row][col] = value;
        fireTableCellUpdated(row,col)
    }
}
```

- Et la création du controleur

```
public class MyTableDemo implements TableModelListener {
    public MyTableDemo() {
        JTable myTable = new JTable(new MyTableModel());
        myTable.getModel().addTableModelListener(this);
        ...
    }
    public void tableChanged(TableModelEvent e) {
        System.out.println("Coucou");
    }
}
```

JTable: Editors et Renderers

- Une **JTable** (**JTree** et **JGraph** fonctionnent sur le même principe) dispose de classes spécifiques pour effectuer :
 - un rendu des cellules appelée classe « **Renderer** »
 - une édition des données des cellules appelée classe « **Editor** »
- Un objet « **Editor** » est appelé quand l'utilisateur effectue une action sur la cellule (sous condition que la cellule soit éditable)
- Un objet « **Renderer** » est une vue sans réaction directe qui retourne une visualisation de la données de la cellule



First Name	Last Name	Sport	# of Years	Vegetarian
Mary	Campione	Snowboarding	5	<input type="checkbox"/>
Alison	Huml	Snowboarding	3	<input checked="" type="checkbox"/>
Kathy	Walrath	Rowing	2	<input type="checkbox"/>
Mark	Andrews	Chasing toddlers	20	<input checked="" type="checkbox"/>
		Speed reading		

JTable: Editors et Renderers



First Name	Last Name	Sport	# of Years	Vegetarian
Mary	Campione	Snowboarding	5	<input type="checkbox"/>
Alison	Huml	Snowboarding	3	<input checked="" type="checkbox"/>
Kathy	Walrath	Rowing	2	<input type="checkbox"/>
Mark	Andrews	Chasing toddlers	20	<input checked="" type="checkbox"/>

1. Les objets **Renderer** affichent les données du modèle
« `getValueAt(...)` »
2. Lors d'un clic, les objets **Editor** affichent l'édition de la cellule
et retournent le nouvelle valeur
3. La nouvelle valeur est transmise au modèle via
« `setValueAt(...)` »

JTable: Editors et Renderers

- Il existe par défaut des objets « **Editor** » et « **Renderer** » définis dans la *JTable*
 - *getColumnClass()* retourne le type de l'objet de la cellule
- Selon le type de données se trouvant dans le modèle (indiqué par *getColumnClass()*) les objets « **Editor** » et « **Renderer** » retournent des composants prédéfinis
- Composant retourné par l'objet « **Renderer** »
 - *Boolean* : *JCheckBox*
 - *Number, Double et Float* : *JLabel* aligné à droite
 - *ImageIcon* : *JLabel* aligné au centre
- Composant retourné par l'objet « **Editor** »
 - *Boolean* : *JCheckBox*
 - Autre : *TextField*

JTable: Editors et Renderers

- Possibilité de modifier les objets « Renderer » et « Editor » de manière à paramétrer la visualisation et l'édition
- La *JTable* propose plusieurs manières de modifier ces objets
 - Un « Renderer » et/ou « Editor » par type d'objet (Class)
 - Un « Renderer » et/ou « Editor » par colonne
- Si vous souhaitez associer un « Renderer » et un « Editor » pour toutes les colonnes qui ont le même type utilisez la première solution
- Si vous souhaitez effectuer un « Renderer » et un « Editor » ponctuel pour une colonne utilisez la seconde solution
- Quelle que soit la solution les « Renderer/Editor » sont les mêmes, seule la liaison entre les « Renderer/Editor » et la *JTable* change

JTable: Editors et Renderers

- Première solution : un « Renderer/Editor » par type d'objet
 - ***setDefaultRenderer(Class, TableCellRenderer)*** : associe un « **Renderer** » au type spécifié par le paramètre *Class*
 - ***setDefaultEditor(Class, TableCellEditor)*** : associe un « **Editor** » au type spécifié par le paramètre *Class*
 - Permet d'associer un **Renderer** (ou **Editor**) à des données mêmes complexes (vos classes de données)
- Deuxième solution : un « Renderer/Editor » par colonne
 - Extraire le modèle de colonne ***TableColumnModel***
getColumnModel()
 - A partir du ***TableColumnModel*** choisir la colonne qui vous intéresse à paramétrer : ***TableColumn getColumn(int)***
 - Modifier le « **Renderer/Editor** »
 - *setCellEditor(TableCellEditor)* : associe l'« **Editor** » à la colonne
 - *setCellRenderer(TableCellRenderer)* : lie le « **Renderer** » à la colonne

JTable: Editors et Renderers

- Exemple : associer les «
Renderer/Editor » à une *JTable*

```
public class JTableDirectRenderer1 extends JFrame {
    public MyTableDemo() {
        ...
        JTable myTable = new JTable(new MyDefaultTableModel());
        myTable.setDefaultEditor(Integer.class, new MyAbstractCellEditor());
        myTable.setDefaultEditor(Boolean.class, new MyAbstractCellEditorBoolean());
        myTable.setDefaultRenderer(Boolean.class, new MyAbstractCellRenderer());

        TableColumnModel myColumnModel = myTable.getColumnModel();
        TableColumn myTableColumn = myColumnModel.getColumn(0);

        myTableColumn.setCellEditor(new MySecondAbstractCellEditor());
        myTableColumn.setCellRenderer(new TableCellRenderer() {
            public Component getTableCellRendererComponent(JTable, ...) {
                return new JButton((String)arg1);
            }
        });
    }
}
```

JTable: Renderer

- Un « **Renderer** » doit implémenter l'interface **TableCellRenderer**
- Cette interface possède une méthode unique qui retourne le composant graphique à afficher dans la cellule
 - **Component** *getTableCellRendererComponent(JTable table, Object v, boolean isSelected, boolean hasFocus, int row, int column)*
 - *JTable table* : référence de la JTable associé au « Renderer »
 - *Object v* : valeur à afficher issue du modèle et donnée par *getValueAt(...)*
 - *boolean isSelected* : la cellule est-elle sélectionnée ?
 - *boolean hasFocus* : la cellule a-t-elle le focus ?
 - *int row* : numéro de ligne de la cellule considérée
 - *int column* : numéro de colonne de la cellule considérée
- Il faut donc retourner un objet graphique **Component** qui sera utilisé pour effectuer le rendu de la cellule

JTable: Renderer

- Exemple : un « Renderer » personnalisé pour les types *Boolean*

```
public class MyAbstractCellRenderer implements TableCellRenderer {
    private JToggleButton vraie = new JToggleButton();
    public Component getTableCellRendererComponent(JTable arg0, Object arg1,
        boolean arg2, boolean arg3, int arg4, int arg5) {
        Boolean myBool = (Boolean )arg1;
        vraie.setSelected(myBool.booleanValue());
        return vraie;
    }
}
```

JTable: Editor

- Un « **Editor** » doit implémenter l'interface **TableCellEditor** qui hérite également de l'interface **CellEditor** utilisée par tous les objets ayant un « **Editor** » : *JTree, JTable, JList*, ...
- Cette interface possède entre autres une méthode qui retourne le composant à afficher pour l'édition
 - **Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int row, int column)**
 - *JTable table* : référence de la *JTable* associé à l'« Editor »
 - *Object value* : valeur de la cellule
 - *boolean isSelected* : la cellule est-elle sélectionnée?
 - *int row* : numéro de ligne de la cellule
 - *int column* : numéro de la colonne
- Pour éviter d'implémenter toutes les méthodes de l'interface **CellEditor** vous pouvez utiliser la classe **AbstractCellEditor**

JTable: Editor

- Si la classe **AbstractCellEditor** est utilisée en place de **CellEditor** certaines méthodes ont un comportement prédéfini
 - *stopCellEditing()* : déclenche l'arrêt de l'« Editor »
 - *cancelCellEditing()* : déclenche l'annulation de l'« Editor »
 - *addCellEditorListener(CellEditorListener)* : ajoute un « écouteur » d'édition
 - ...
- Cependant vous devrez fournir obligatoirement une implémentation de la méthode
 - **Object getCellEditorValue()** : retourne la valeur que l'utilisateur vient de saisir (la méthode *setValueAt(...)* du modèle est alors appelée)
- Principe de codage
 - Implémentez *getTableCellEditorComponent(...)* pour retourner le composant d'édition
 - La validation du composant d'édition doit appeler **stopCellEditing()**
 - Implémentez **getCellEditorValue()**

JTable: Editor

- Exemple : un « Editor » personnalisé pour les types *Boolean*

```
public class MyAbstractCellEditorBoolean extends AbstractCellEditor implements
TableCellEditor {
    private JToggleButton myButton = new JToggleButton();
    public MyAbstractCellEditorBoolean() {
        myButton.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                MyAbstractCellEditorBoolean.this.fireEditingStopped();
            }
        });
    }

    public Object getCellEditorValue() {
        return new Boolean(myButton.isSelected());
    }

    public Component getTableCellEditorComponent(JTable arg0, Object arg1,
        boolean arg2, int arg3, int arg4) {
        Boolean myBool = (Boolean )arg1;
        myButton.setSelected(myBool.booleanValue());
        return myButton;
    }
}
```

JTable: modèle de sélection

- Le modèle de sélection permet de gérer les aspects liés à la sélection de lignes ou de colonnes (une ou plusieurs ...)
- Un modèle de sélection doit implémenter l'interface appelée **ListSelectionModel**
- Possibilité de modifier un modèle de sélection pour toutes les lignes et un modèle de sélection pour toutes les colonnes
 - *JTable#setSelectionModel(ListSelectionModel)* : modifie le modèle de sélection pour les lignes
 - *TableColumnModel#setSelectionModel(ListSelectionModel)* : modifie le modèle de sélection pour toutes les colonnes
- Pour écouter le changement de sélection de lignes ou de colonnes ajoutez un écouteur sur le modèle de sélection
 - *ListSelectionModel#addListSelectionListener(ListSelectionListener)* : ajoute un écouteur de sélection

JTable: modèle de sélection

- La définition explicite du modèle de sélection n'est utile que si vous souhaitez effectuer des traitements pointus sur les sélections
- Pour le reste des aspects liés à la sélection utilisez les méthodes de la classe *JTable* qui pointent directement sur le modèle de sélection
 - *setSelectionMode(int)* : choix du modèle de sélection
 - *SINGLE_SELECTION* : une seule cellule peut-être sélectionnée
 - *MULTIPLE_INTERVAL_SELECTION* : plusieurs cellules sélectionnées à des moments différents
 - *SINGLE_INTERVAL_SELECTION* : plusieurs cellules sélectionnées en même temps
 - *int getSelectedColumn()* : retourne l'indice de la colonne sélectionnée
 - *int getSelectedRow()* : retourne l'indice de la ligne sélectionnée
 - *clearSelection()* : supprime la sélection actuelle

JTable: modèle de sélection

- Exemple : accès aux informations de la sélection courante

```
public class JTableDirectRenderer2 extends JFrame {  
    ...  
    public JTableExemple() {  
        ...  
        myTable.addMouseListener(new MouseAdapter() {  
            public void mouseClicked(MouseEvent e) {  
                System.out.println("Ligne:" + myTable.getSelectedRow());  
                System.out.println("Colonne:" + myTable.getSelectedColumn());  
            }  
        });  
        ...  
    }  
}
```

JTable: ajout et suppression

- L'ajout et la suppression dynamique de ligne se fait en ajoutant ou en supprimant des éléments au contenu sémantique de la **JTable**
- Rappelons que la méthode *getRowCount()* et *getValueAt(...)* sont appelées pour « peupler » la table
- Une solution est d'ajouter au modèle deux méthodes
 - *addRow(Object p, int pRow)* : ajoute l'objet *p* à la ligne *pRow*
 - *removeRow(int pRow)* : supprime la ligne *pRow*
- Le modèle n'est pas à même de connaître la ligne d'ajout ou la ligne à supprimer tout dépend de la ligne en cours de sélection
 - Ajout : ligne suivante de celle sélectionnée ou dernière ligne
 - Suppression : ligne en cours de sélection
- A chaque ajout ou suppression d'une ligne n'oubliez pas de notifier la vue des modifications apportées au modèle

JTable: ajout et suppression

- Exemple : ajout et suppression de ligne dans une *JTable*

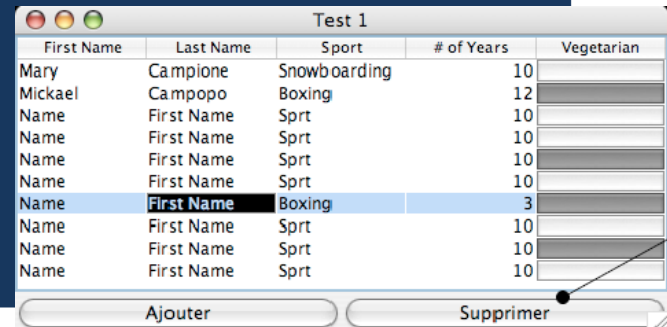
```
public class MyDefaultSecondTableModel extends AbstractTableModel {
    private ArrayList colonneName = new ArrayList();
    private ArrayList colonneLastName = new ArrayList();
    ...
    public void addNewRow(int ligne) {
        int addline = getRowCount();
        if (ligne != -1)
            addline = ligne + 1;
        colonneName.add(addline, "Name");
        colonneLastName.add(addline, "First Name");
        ...
        this.fireTableStructureChanged();
    }

    public void removeNewRow(int ligne) {
        if (getRowCount() == 0 || ligne < 0)
            return;
        colonneName.remove(ligne);
        colonneLastName.remove(ligne);
        ...
        this.fireTableStructureChanged();
    }
    ...
}
```

JTable: ajout et suppression

- Exemple : ajout et suppression de ligne dans une *JTable*

```
public class JTableDirectRenderer3 extends JFrame {  
    ...  
    public JTableExemple() {  
        JButton myAddButton = new JButton("Ajouter");  
        myAddButton.addActionListener(new ActionListener(){  
            public void actionPerformed(ActionEvent e) {  
                MyDefaultTableModel refModel = (MyDefaultTableModel)myTable.getModel();  
                refModel.addRow(myTable.getSelectedRow());  
            }  
        });  
  
        JButton myRemoveButton = new JButton("Supprimer");  
        myRemoveButton.addActionListener(new ActionListener(){  
            public void actionPerformed(ActionEvent e) {  
                MyDefaultTableModel refModel = (MyDefaultTableModel)myTable.getModel();  
                refModel.removeRow(myTable.getSelectedRow());  
            }  
        });  
    }  
}
```

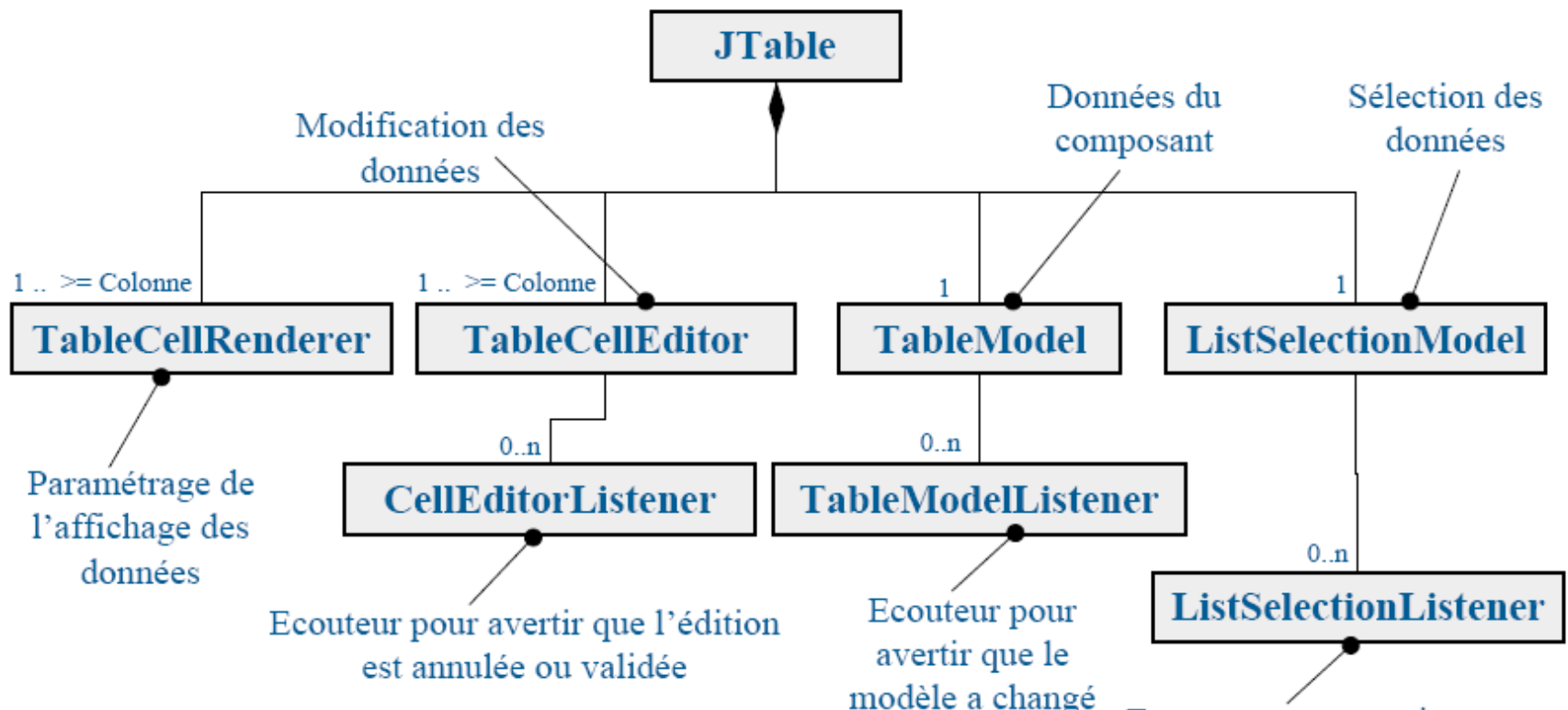


First Name	Last Name	Sport	# of Years	Vegetarian
Mary	Campione	Snowboarding	10	
Mickael	Campopo	Boxing	12	
Name	First Name	Sprt	10	
Name	First Name	Sprt	10	
Name	First Name	Sprt	10	
Name	First Name	Sprt	10	
Name	First Name	Boxing	3	
Name	First Name	Sprt	10	
Name	First Name	Sprt	10	
Name	First Name	Sprt	10	

Ajouter Supprimer

JTable: bilan

Un composant *JTable* est une vue paramétrable qui permet d'afficher et modifier des données d'une table



Architecture PAC

- PAC : Présentation, Abstraction et Contrôle
- (un modèle Franco-français) proposé par Joëlle Coutaz (1987)
- une architecture logicielle basée agents
- une approche hiérarchique
- 3 composants
 - Présentation
 - Abstraction
 - Contrôle

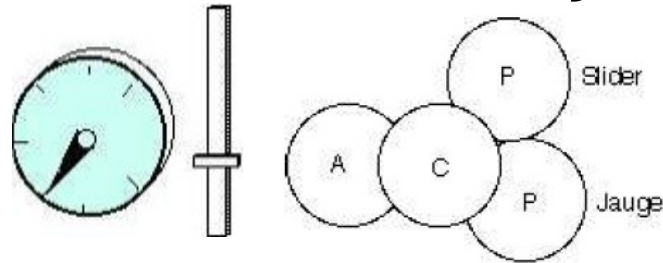
PAC

Le modèle repose sur un agent PAC, composé de

- **Présentation : la Présentation définit l'image du système**, c'est-à-dire son comportement en entrée comme en sortie vis-à-vis de l'utilisateur :
 - visualise l'état interne du composant Abstraction.
 - est en contact avec l'utilisateur
- **Abstraction : l'Abstraction désigne les concepts et les fonctions** du système
 - partie abstraite du composant manipulé
 - détaché de toute représentation graphique
- **Contrôle : le Contrôle maintient la cohérence** entre les facettes Présentation et Abstraction
 - cimente entre la Présentation et l'Abstraction
 - maj. de l'abstraction -> contrôle -> présentation
 - maj. de la présentation -> contrôle -> abstraction

Exemple

- Représentation d'une jauge



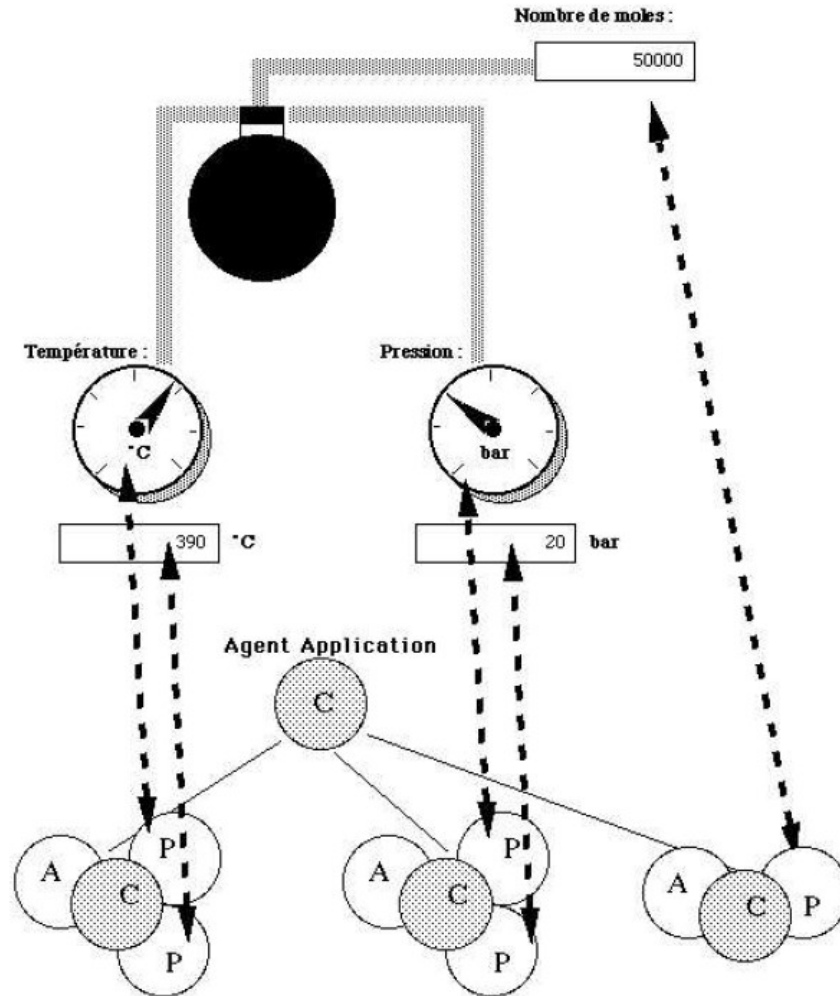
- => Une modification du slider entraine la modification de la jauge (analogique) et la mise à jour de l'abstraction :
- l'évènement « déplacer le curseur. » est notifié au contrôle
 - le contrôle calcule la nouvelle valeur de l'abstraction et notifie l'abstraction de ce changement
 - le contrôle notifie les présentations de la modification

=> **Toute la communication** passe par le **contrôleur**

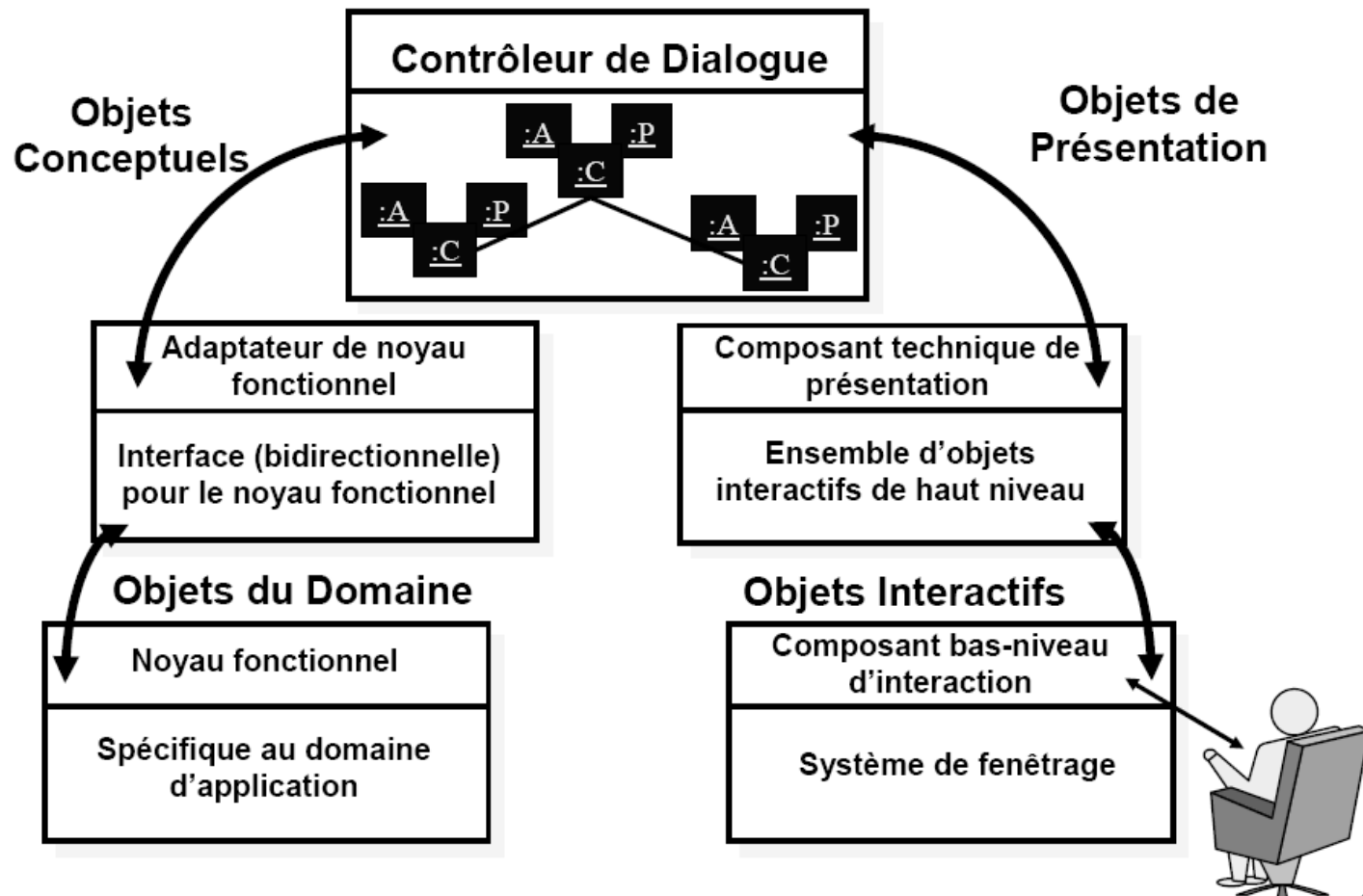
Liens entre agents

- les agents PAC sont organisés en hiérarchie arborescente
 - le PAC parent a la responsabilité (et le savoir) des agents fils
 - relation de composition et non d'héritage
 - le noeud père représente l'application
- le contrôle assure une liaison par relais (indirection) entre deux formalismes indépendants, celui du monde de la présentation et du monde abstrait.
- le contrôle assure aussi **une fonction d'arbitrage** (synchronisation, vérification des contraintes, gestion d'informations contextuelles) entre enfants et parents.
- un agent ne **possède pas nécessairement** de présentation ou d'abstraction (e.g. contrôle pur pour synchroniser d'autres contrôleurs)

Une approche hiérarchique



PAC-Amodeus (Nigay 1993)

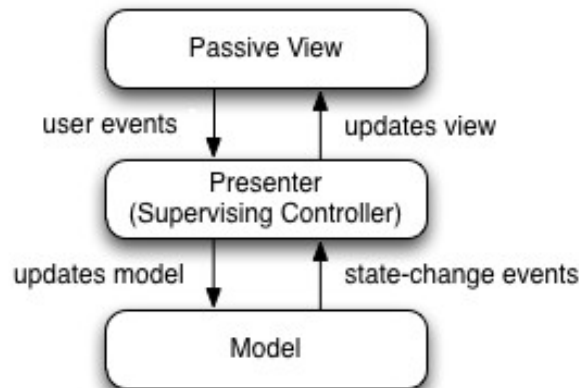


PAC-Amodeus

- Les objets initiaux forment le **noyau fonctionnel**
 - les objets peuvent être adaptés au niveau de l'ANF
 - les objets peuvent communiquer avec une Abstraction d'un agent PAC du contrôleur de dialogue
- Avantages/inconvénients
 - les objets initiaux restent des composants logiciels indépendants
 - la hiérarchie PAC est indépendante de la structure des objets initiaux
 - un agent PAC du Contrôleur de Dialogue est à l'écoute :
 - de l'utilisateur par sa facette présentation
 - du Noyau Fonctionnel par sa facette abstraction
 - complexité sur des projets de faible envergure

MVP – Modèle Vue Présentation

- Pattern d'architecture dérivé de MVC
 - **la Présentation** joue le rôle de contrôleur
 - Conçu pour faciliter le test d'interfaces (car **toutes les communications** passent par le contrôleur)



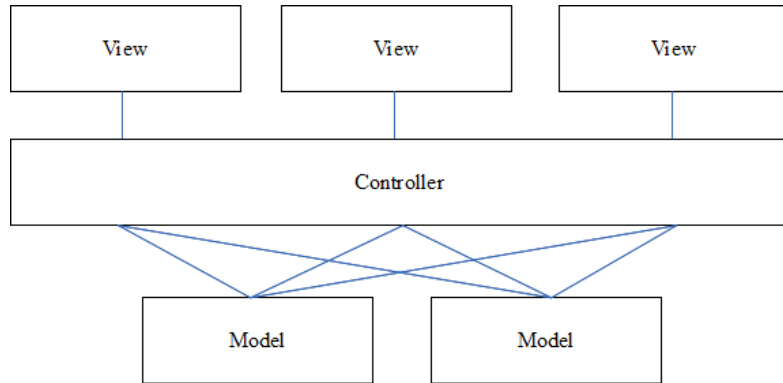
MVP – Modèle Vue Présentation

- **La Présentation** assure la **logique** entre la vue et le modèle
 - agit sur le Modèle et la Vue
 - adapte les données du Modèle à la Vue
 - adapte les données de la Vue au Modèle
 - la présentation parle généralement à une **Interface** de la Vue
 - **Le Modèle** est une interface vers les données
 - **La Vue** affiche les données et dirige les événements utilisateur vers la Présentation. Toute action sur la vue est déléguée directement à la Présentation.
- ⇒ Un pattern adapté aux architectures web (Vue = page web)
- Implémentable en .NET, Java, GWT via différentes librairies (Claymore, Nucleo.NET, Echo2, Biscotti)

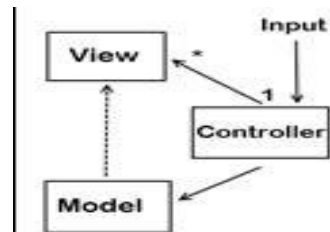
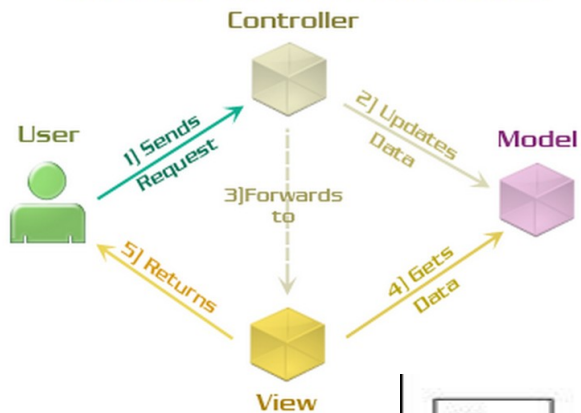
MVP – Modèle Vue Présentation

- Vue passive:
 - aucune logique dans la vue, toute communication se fait via la **Présentation**. **Vue** et **Modele** ne se parlent pas. Le modèle notifie la Présentation, la présentation met la **Vue** à jour.
 - Séparation claire, mise en place aisée de tests
 - Mais le binding de données est fait manuellement

MVC



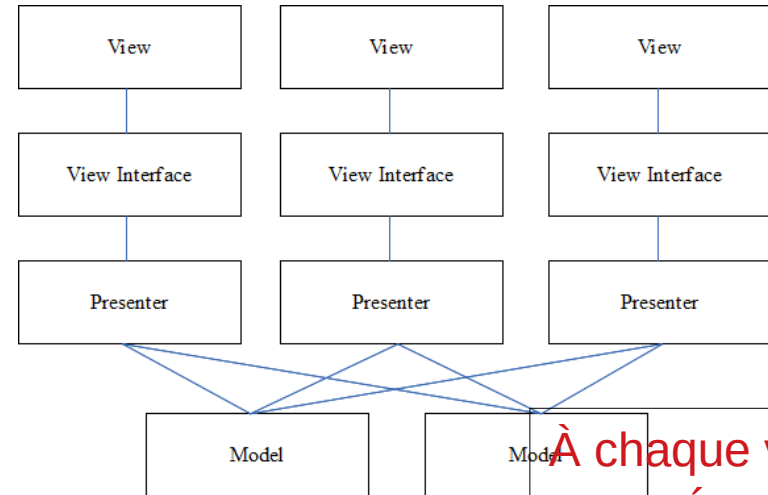
Model View Controller



MVC

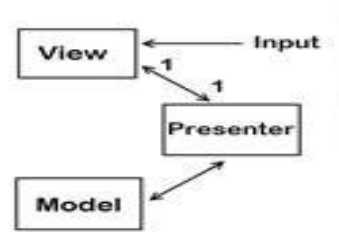
VS

MVP



À chaque vue son présenter!

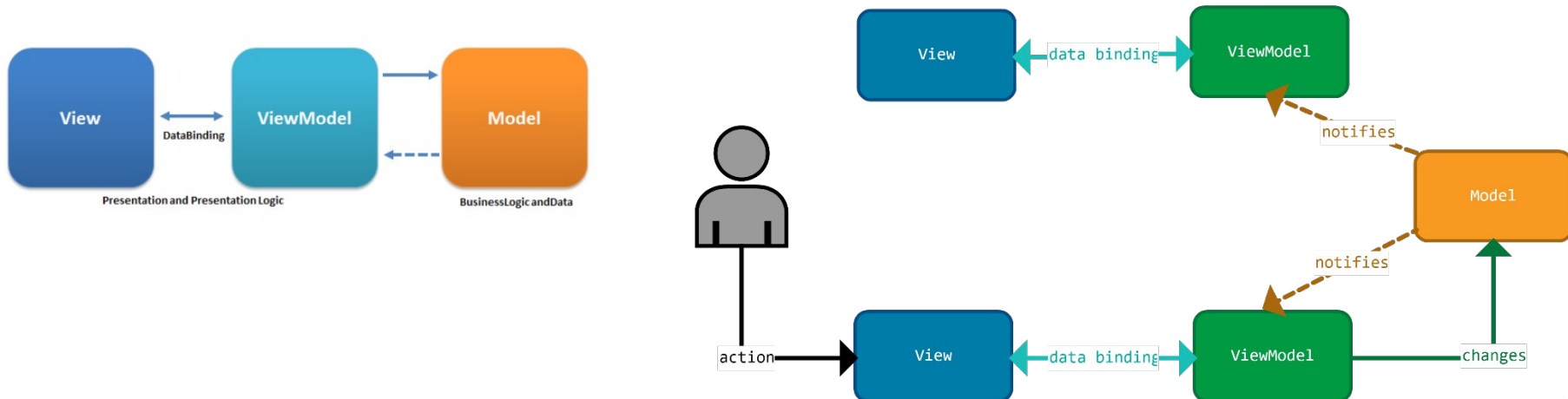
Model View Presenter



MVP

MVVM – Model/View/Viewmodel

- Pattern d'architecture dérivé de MVC, fondé sur la notion et les mécanismes de **Databinding**
 - **ViewModel: abstraction de la vue**, et un convertisseur qui expose/formate les objets du modèle dans des objets adaptés à la vue, cf Adapter
 - **Databinder**: mécanisme de synchronisation entre la vue et le viewmodel

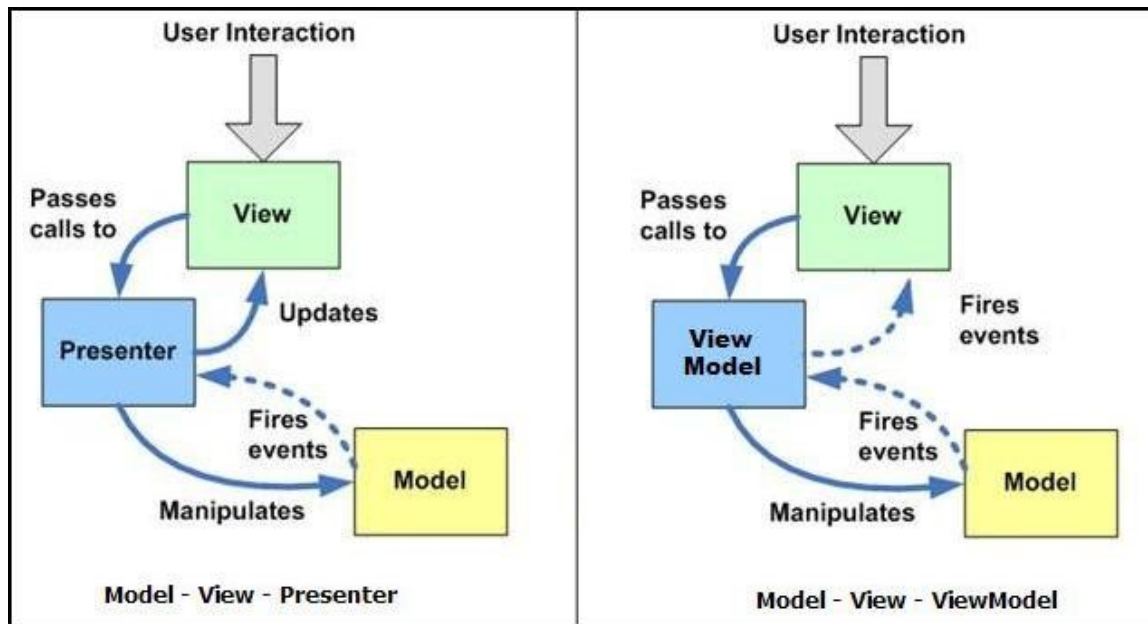


MVVM – Model/View/Viewmodel

- Illustration MVVM
 - MVVM au sein du framework Zk de zkoss (<http://www.zkoss.org>)
 - [https://
www.zkoss.org/wiki/Small_Talks/2011/October/Hello_ZK_MVVM](https://www.zkoss.org/wiki/Small_Talks/2011/October/Hello_ZK_MVVM)
 - Principe du data-binding [https://
www.zkoss.org/zkdemo/grid/data_binding](https://www.zkoss.org/zkdemo/grid/data_binding)

MVP - MVVM

- MVVM s'appuie sur un mécanisme de bi-directionnel) qui évite de maintenir la cohérence manuellement (eg, via des notifications ou des updates)



Bilan croisé

	MVC	MVP	PAC	MVVM
MVC				
MVP				
PAC				
MVVM				

Mais...

- Comment ça m'aide à choisir une architecture?

Bilan

- Différences MVC-PAC-MVP
 - dans PAC, toute communication passe par le controleur
 - dans MVP aussi, mais MVP n'intègre pas la dimension hiérarchique
 - dans PAC, la présentation est en interaction avec l'utilisateur (pas dans MVC)
 - dans MVP, c'est la vue qui est en interaction
 - MVVM est similaire à MVP, mais un mécanisme de DataBinding établit le lien VueModel/Vue
 - Dans MVP, il faut maintenir la cohérence Model <-> Vue manuellement (dans MVVM c'est réalisé par le DataBinding)
 - http://www.infragistics.com/community/blogs/todd_snyder/archive/2007/10/17/mvc-or-mvp-pattern-whats-the-difference.aspx
- Le pattern MVC est largement utilisé sur les clients lourds
 - convient aux cadres objets ou fonctionnels
 - une adaptation est nécessaire pour faciliter la modularité (via des interfaces/proxy/adapteur) – à réserver aux gros projets pour lesquels le risque de changement est présent
- Le pattern MVP est utilisé sur les client lourds (pour faire du test), et sur les architectures web
- L'approche agent (MVC, PAC, MVP, MVVM) permet une meilleure réutilisation qu'un modèle en couches
 - facilité d'extraire un composant (modèle, vue et controleur) pour le réutiliser sans tout ré-écrire

Morale

- => on n'a pas toujours le choix: les Toolkits graphiques proposent souvent des AL pour l'interaction par défaut
- => les toolkits limitent souvent les possibilités d'architectures évoluées/propres (voir Android)