POO / IHM
Architecture Logicielle

Anne-Marie Dery (pinna@polytech.unice.fr)
Audrey Occello (occello@polytech.unice.fr)

Architecture Logicielle et IHM : Pourquoi ?

1. Organiser le code (rangement)
2. Simplifier (diviser régner)
3. Organiser le travail
4. Modifier (une partie)
5. Ré-utiliser

modularité, évolutivité, flexibilité

Objectif : éviter de tout modifier si on change la partie fonctionnelle ou la partie IHM

Séparation possible

–Code pour IHM

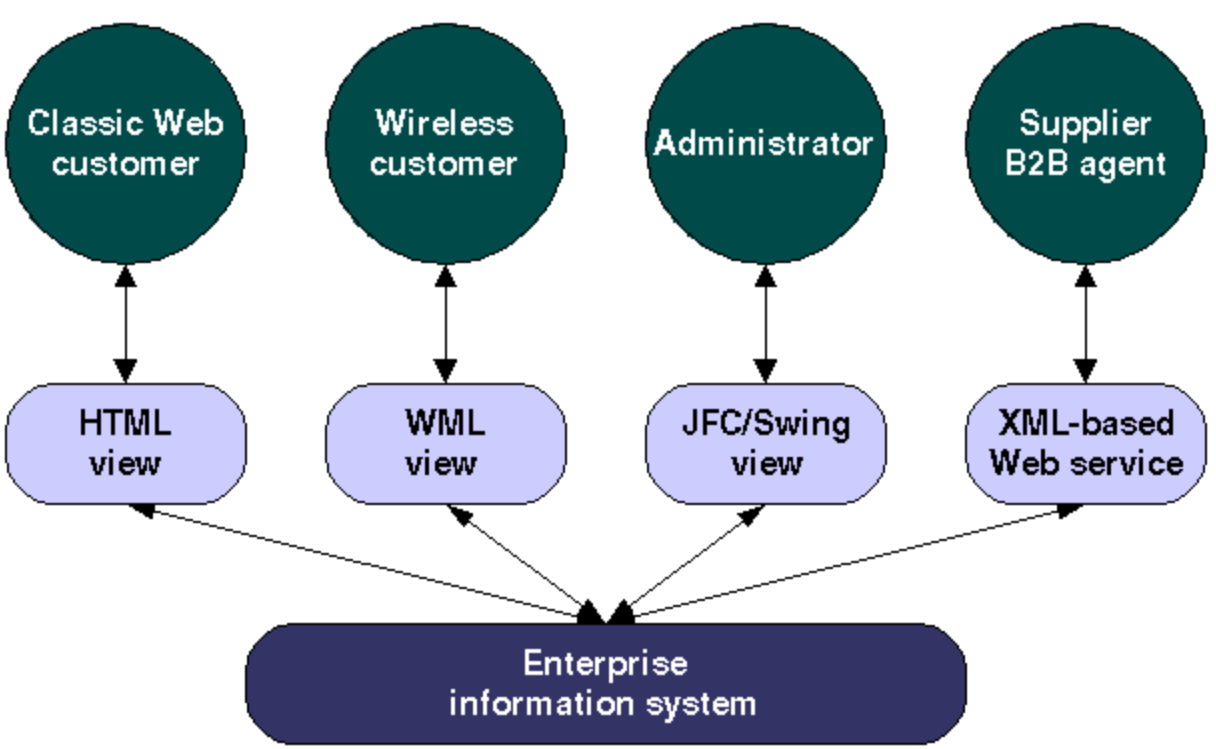
–Code «Métier»

•*Exemple*

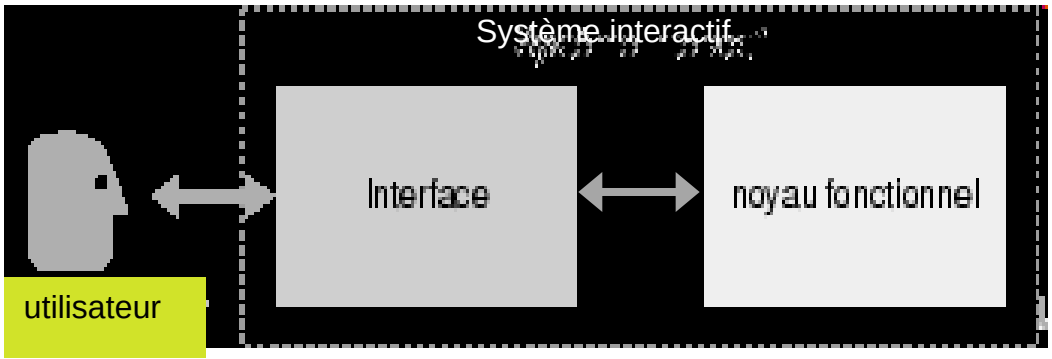
–IHM différente pour une Gestion d'un stock de chaussures ou de bibelots ?

–Linux sous gnome ou kde, le système change-t-il ?

Un problème classique ancien



Systèmes interactifs



Tous les modèles partent du **principe :**
un **système interactif** comporte une partie **interface** et une partie **application pure**
Cette dernière est souvent appelée *noyau fonctionnel*.

http://tel.archives-ouvertes.fr/docs/00/04/82/79/HTML/2_2modelesinterface_referen.html

Architectures logicielles

La plupart des modèles identifient en général trois types d'éléments :

un ``coté utilisateur'' (*présentations*, *vues*),

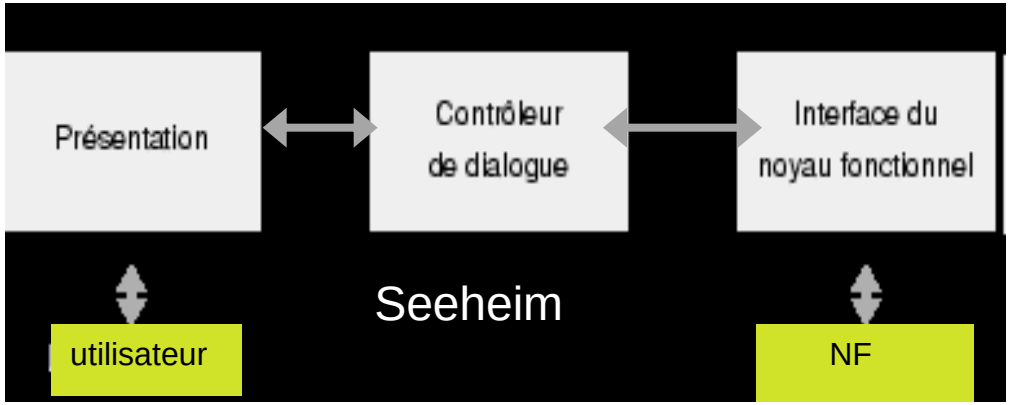
un ``côté noyau fonctionnel'' (*interfaces du noyau fonctionnel*, *abstractions*, *modèles*),

et des éléments articulatoires (*contrôleurs*, *adaptateurs*).

Modèles à couches er Modèles Acteurs

http://tel.archives-ouvertes.fr/docs/00/04/82/79/HTML/2_2modelesinterface_referen.html

Modèles à couches : Seeheim



Premier modèle (groupe de travail à Seeheim en 1985) - destiné au traitement lexical des entrées et sorties dans les interfaces textuelles – a servi de base à beaucoup d'autres modèles.

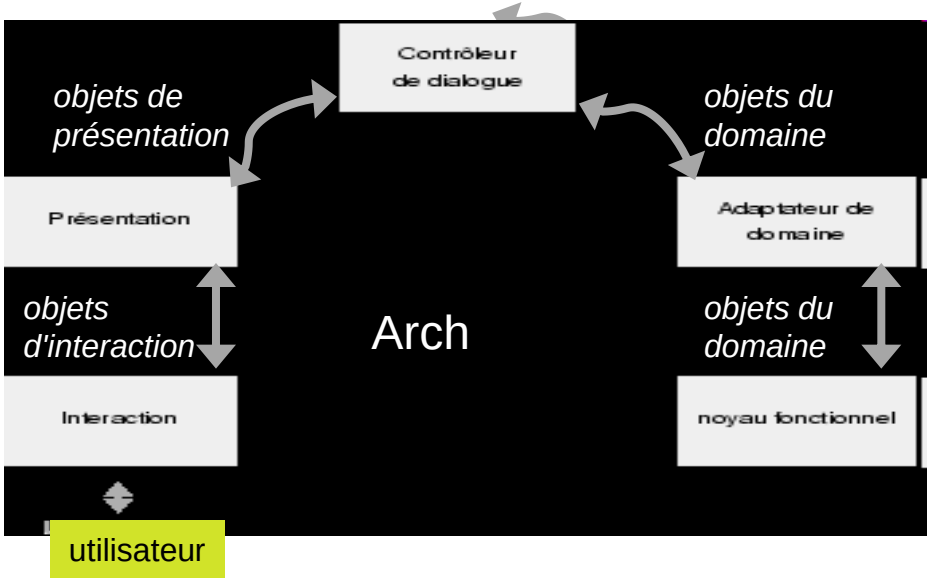
Seeheim : 3 composants

La présentation est la couche en contact direct avec les entrées et sorties (interprétation des actions utilisateur + génération des sorties au niveau lexical)

Le contrôleur de dialogue gère le séquencement de l'interaction

L'interface du noyau fonctionnel convertit les entrées en appels du noyau fonctionnel et les données abstraites de l'application en des éléments présentables à l'utilisateur

Modèles à couches : Arch



Le modèle **Arch** [1992]
5 composants et 3 types de données

Arch : 5 composants

Composant d'interaction - ensemble des widgets + communication avec les périphériques

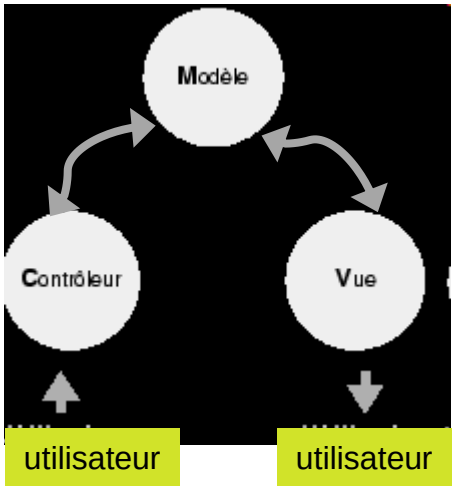
Composant de présentation - représentation logique des widgets indépendante de la boite à outils

Contrôleur de dialogue - responsable du séquencement des tâches et du maintien de la consistance entre les vues multiples.

Adaptateur du domaine - responsable des tâches dépendantes du domaine qui ne font pas partie du noyau fonctionnel mais qui sont nécessaires à sa manipulation par l'utilisateur.

Noyau fonctionnel *représente la partie non interactive de l'application.*

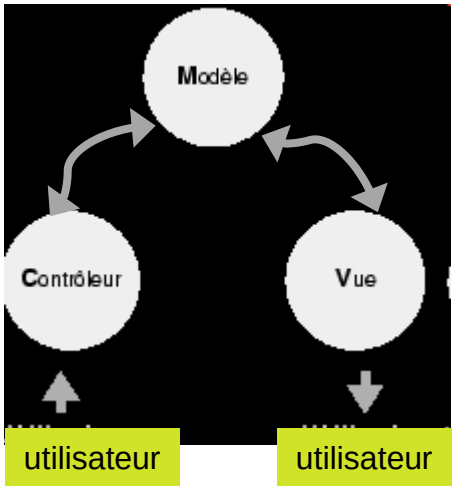
Modèles à agents : MVC



Le modèle **MVC** (Modèle, Vue, Contrôleur) Smalltalk [Goldberg and Robson, 1981].

Chacun des trois composantes de la triade MVC est un objet à part entière.

Modèles à agents : MVC



les systèmes interactifs = une hiérarchie d'agents mais la hiérarchie elle-même n'est pas explicite

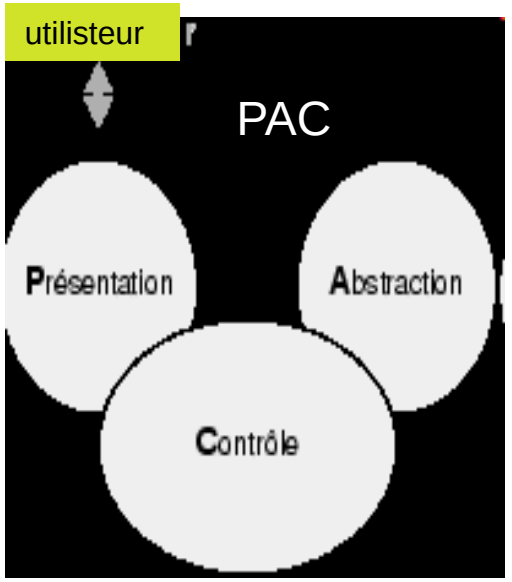
Un agent MVC = un *modèle,* une ou plusieurs *vues*, et un ou plusieurs *contrôleurs*

Le ***modèle*** = noyau fonctionnel de l'agent, notifie les vues à chaque fois que son état est modifié par le noyau ou par ses contrôleurs.
*La **vue*** = maintient une représentation du modèle pour l'utilisateur, mise à jour à chaque notification d'état.
*Le **contrôleur*** reçoit et interprète les événements utilisateur, les répercutant sur le modèle (modification de son état) ou sur la vue (retour instantané).

Modèles à agents : PAC

Présentation = structure et comportement en entrée et en sortie de l'agent pour l'utilisateur (fusion des composants vue et contrôleur de MVC)
Abstraction = la partie sémantique de l'agent (équivalent modèle de MVC)
Contrôle = consistance entre la présentation et l'abstraction, navigation dans la hiérarchie

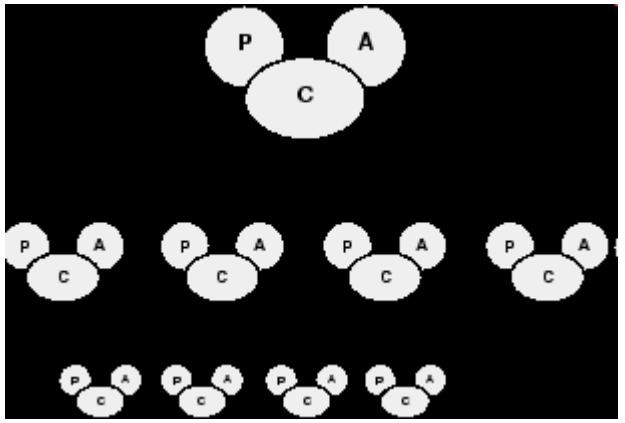
Le composant contrôle n'a pas d'existence explicite dans le modèle MVC.



Modèle à agents **PAC** [Coutaz, 1987]
hiérarchie d'agents explicite

Modèles à agents : PAC

le modèle PAC peut être appliqué à plusieurs niveaux d'un système interactif. Une application interactive peut ainsi être décrite comme une hiérarchie d'agents disposés sur plusieurs couches d'abstractions



Pourquoi autant de modèles ?

De nouveaux modèles inspirés de Arch pour gérer le Device Independence

Des modèles MVC selon les applications :

- quid des applications mobiles
- quid du Web

Que doit on faire ?

Identifier : A quoi sert le modèle ?

Spécificités des jeux vidéos

Des éléments spécifiques :

1. une intelligence artificielle
2. les règles du jeu, la gestion de la santé, de l'apparition et l'orchestration des éléments et autres..
3. un moteur qui permet l'orchestration des éléments du jeu un module 2D/3D qui affiche les images à l'écran ;
4. Un module qui gère les sons, la musique selon les événements du jeu
5. Un module réseau permettant de faire des jeux multijoueurs ;
6. Une interface utilisateur - les menus, l'écran de pause, …

Des patterns pour les Jeux

- Pour les jeux vidéos : Game Programming Patterns

  1. Simplification des rôles MVC avec des contrôleurs intégrés

     http://gameprogrammingpatterns.com/game-loop.html

  2. State Pattern : gestion des états du jeu

     http://gamedev.dreamnoid.com/2009/01/06/game-state-pattern/ éléments

  3. MVC Simplifie

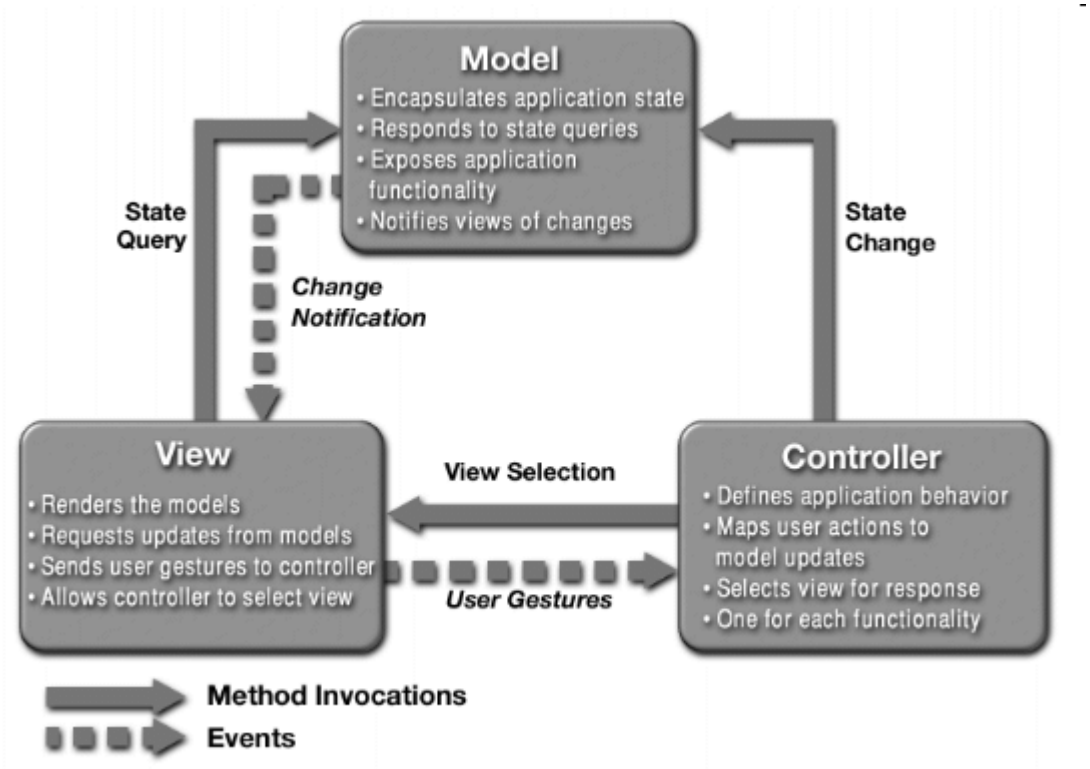
     http://blog.emmanueldeloget.com/index.php?post/2007/06/21/82-quelques-informations-sur-l-architecture-des-jeux-video

Spécificités des jeux DEVINT

- Je vous écoute….

Zoom : Architecture MVC

–**Model** : modélisation (données et comportement)
–**View:** représentation manipulable de l'objet
–**Control** : interprétation des entrées

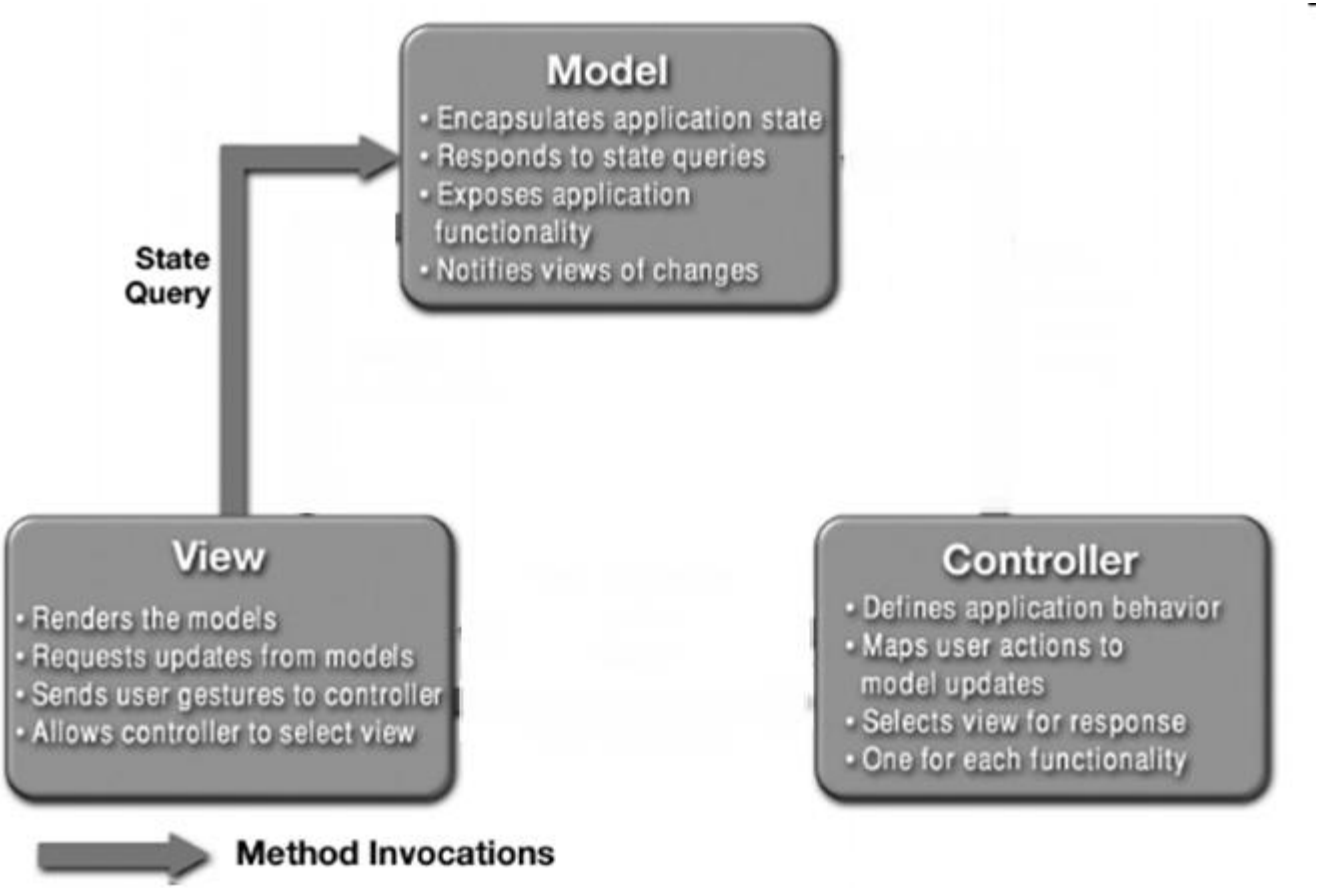


Séparer dans le code
–les données (le Modèle),
–La ou les Vues,
–Le Contrôle

•V **s'abonne** à M
•C **s'abonne** à V
•C **modifie** M et V

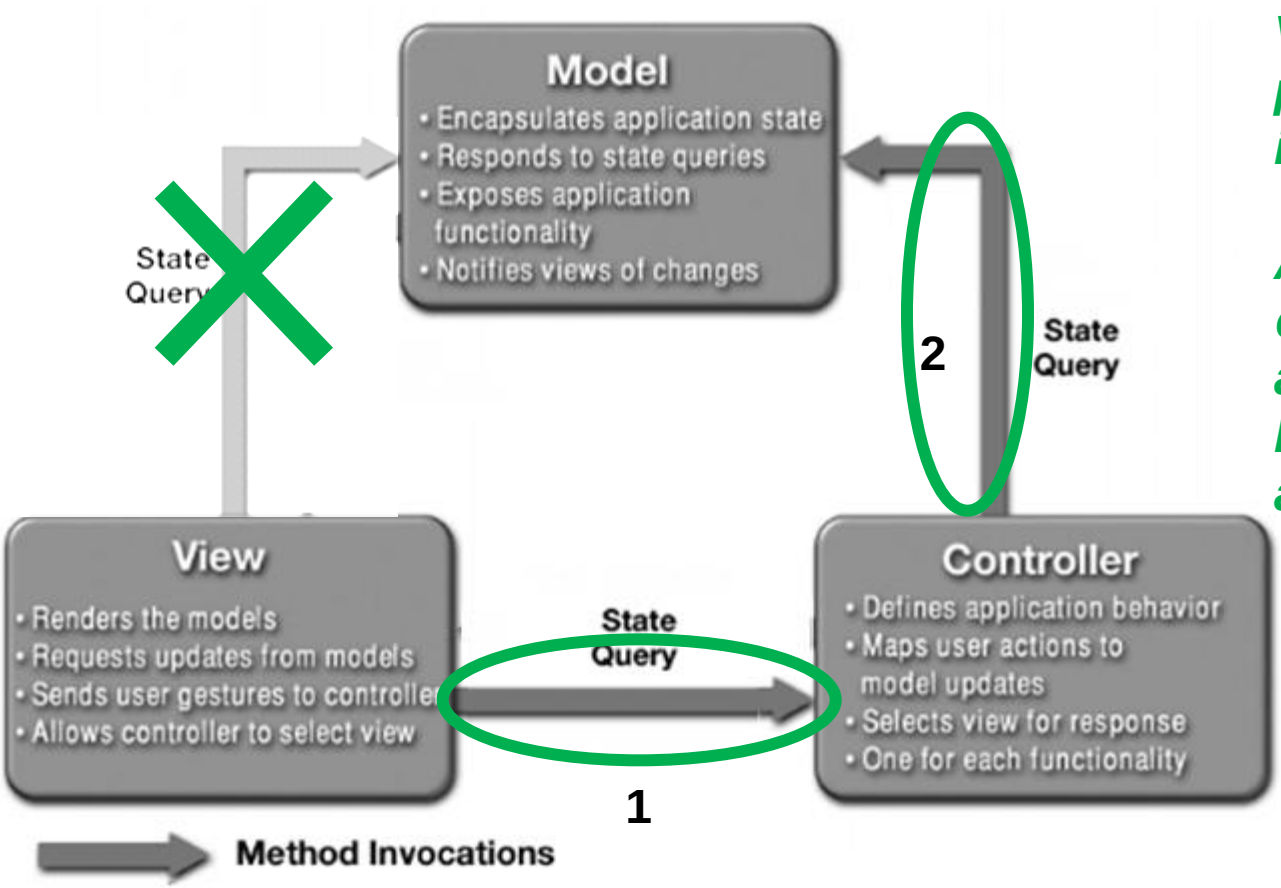
MVC – accès en lecture

Accès à des fonctionnalités qui ne modifient pas l'état du modèle
La vue interroge le modèle pour se mettre à jour (typiquement à son initialisation)
Obligation de passer par une interface d'abstraction



MVC – accès en lecture

Accès à des fonctionnalités qui ne modifient pas l'état du modèle
La vue interroge le modèle pour se mettre à jour (typiquement à son initialisation)
Obligation de passer par une interface d'abstraction

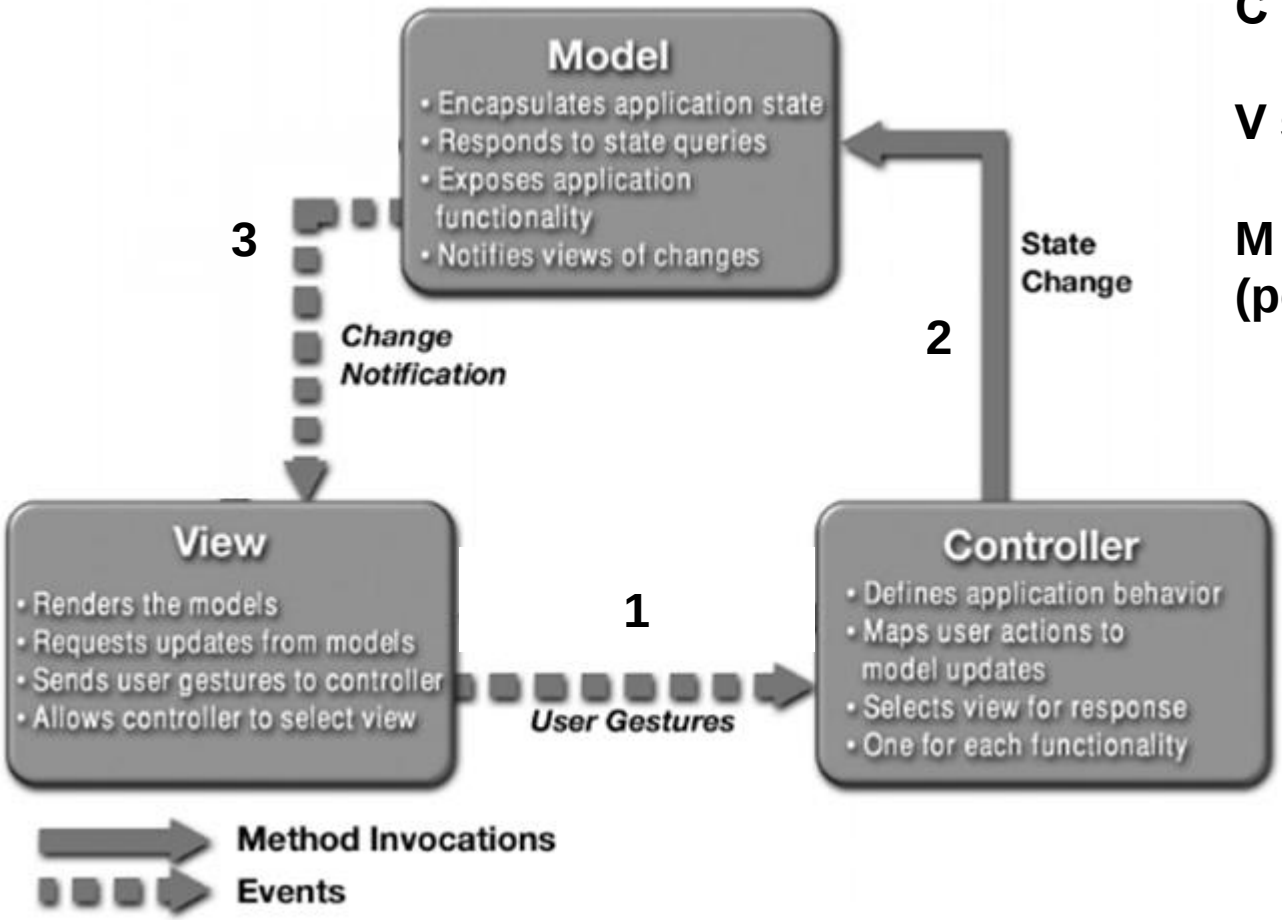


Variante : la vue passe par le contrôleur pour interroger le modèle

Avantage : pas besoin de gérer l'inferçage abstrait avec le modèle, le contrôleur lui peut y accéder + simplement

MVC – accès en écriture

Accès à des fonctionnalités qui modifient l'état du modèle
La vue déclenche des évènements système écoutés par le contrôleur
Obligation de laisser la main au modèle – MAJ de la vue dépendante de l'état du modèle



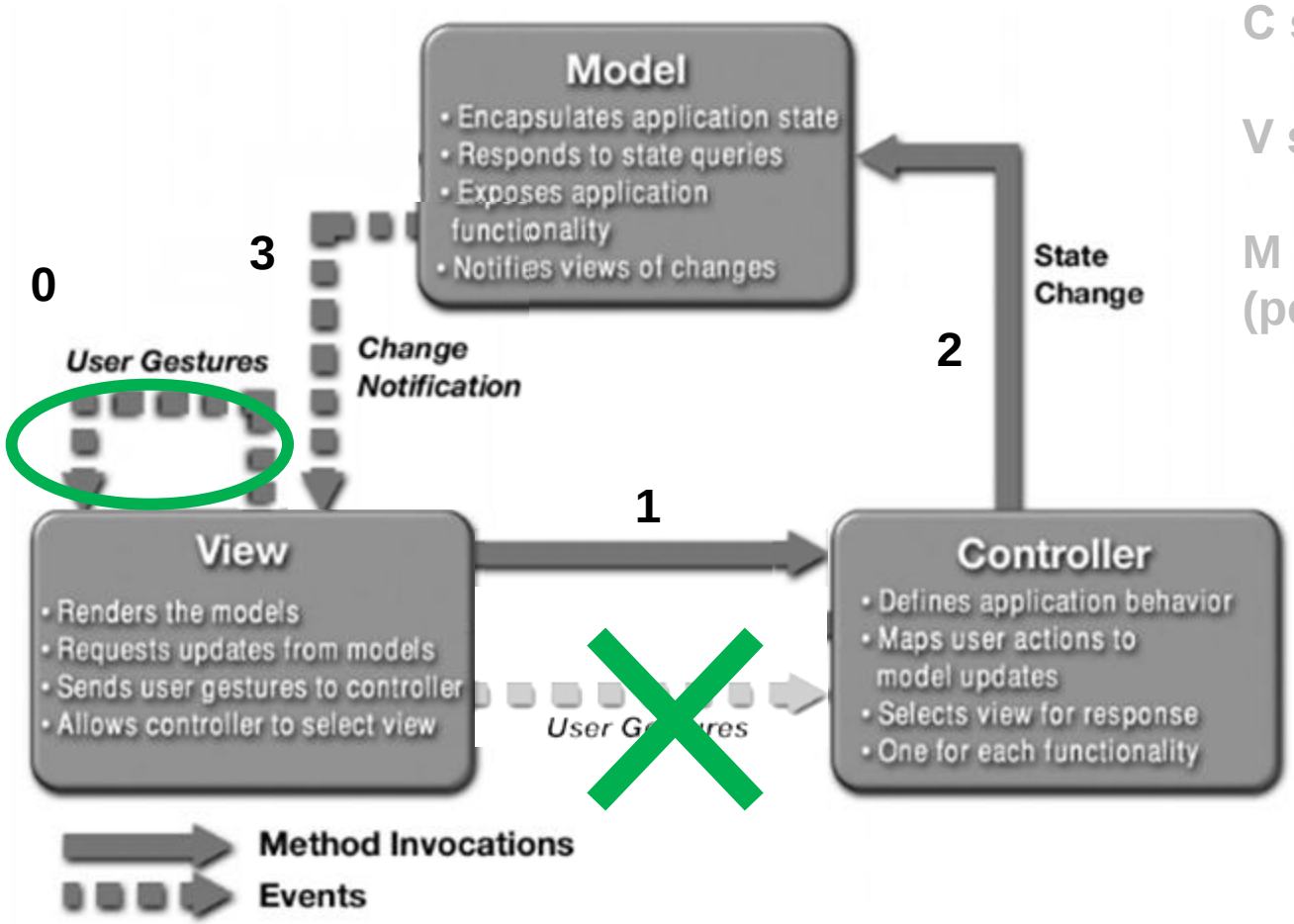
C s'abonne à V

V s'abonne à M

M modifie V implicitement (polymorphisme)

MVC – accès en écriture

Accès à des fonctionnalités qui modifient l'état du modèle
La vue déclenche des évènements système écoutés par le contrôleur
Obligation de laisser la main au modèle – MAJ de la vue dépendante de l'état du modèle



C s'abonne à V

V s'abonne à M

M modifie V implicitement (polymorphisme)

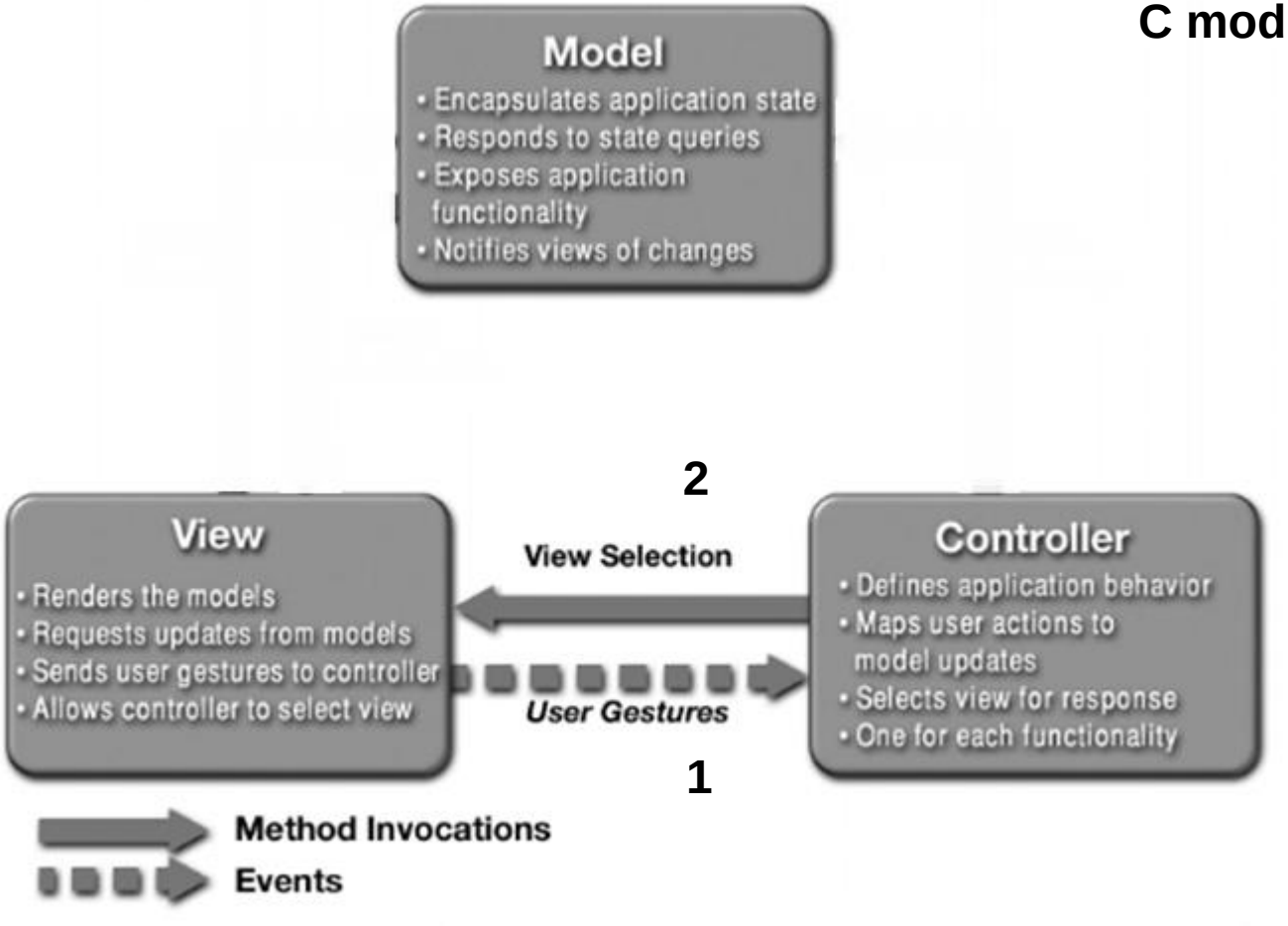
Variante : V écoute les évènements système

Avantage : V transfère les données impliquées à C, peut encapsuler les évènements en évènements de plus haut niveau

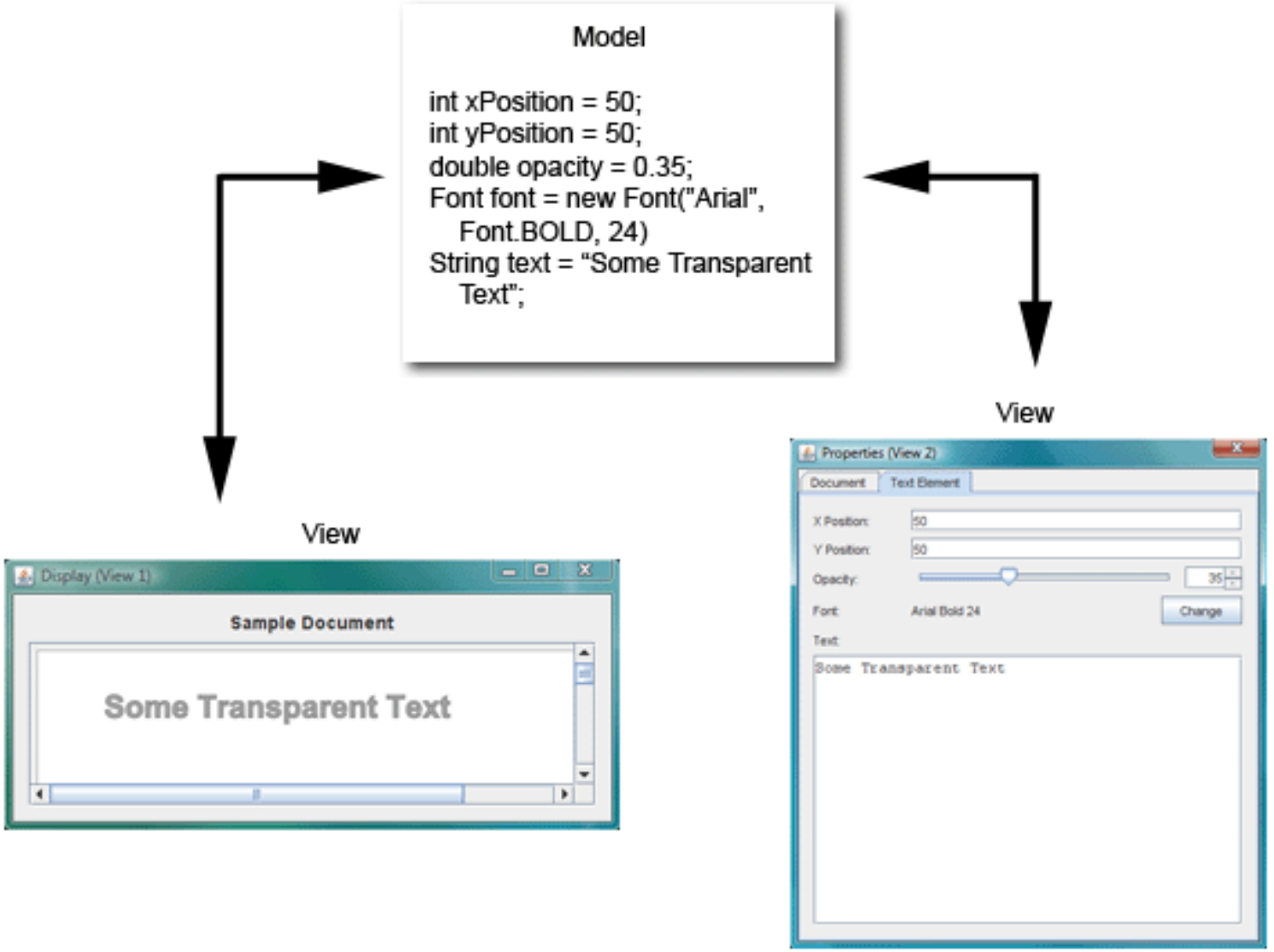
MVC – interaction pure

Interaction utilisateur nécessitant seulement un traitement du contrôleur
La vue déclenche des évènements système écoutés par le contrôleur
Pas d'accès au modèle – MAJ de la vue indépendante des données du modèle

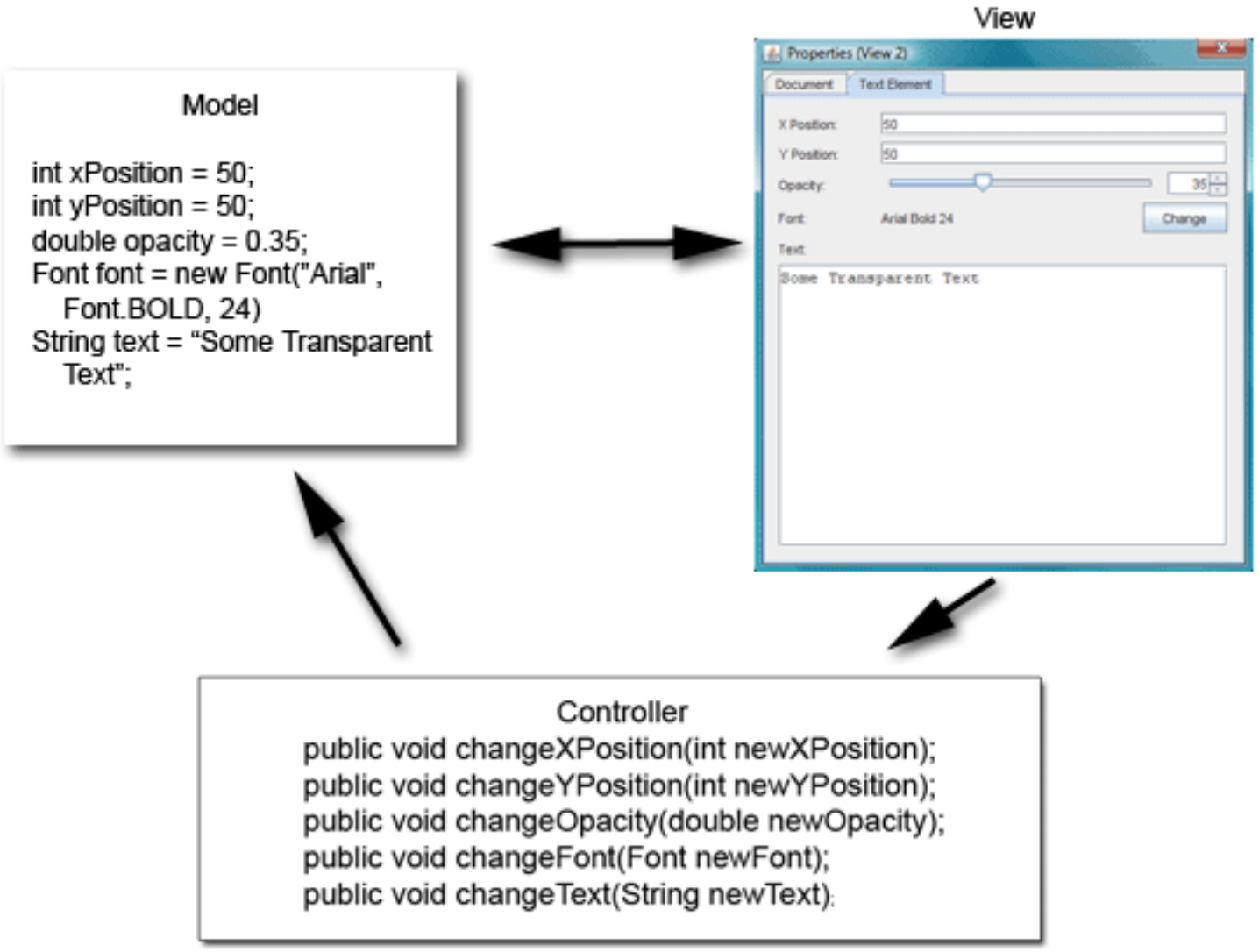
C s'abonne à V
C modifie V directement



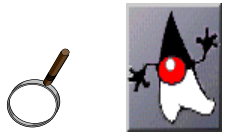
Plusieurs vues – 1 modèle



Exemple


View
Model
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
Font.BOLD, 24)
String text = "Some Transparent
Text";
Properties (View 2)
Document
Text Element
X Position:
50
Y Position:
50
Opacity:
35
Font:
Arial Bold 24
Change
Text:
Some Transparent Text
Controller
public void changeXPosition(int newXPosition);
public void changeYPosition(int newYPosition);
public void changeOpacity(double newOpacity);
public void changeFont(Font newFont);
public void changeText(String newText);

Pour voir le code complet



- http://java.sun.com/developer/technicalArticles/javase/mvc/

Exemple MVC pour un éditeur simple

0. initialisation de la vue avec les données du modèle

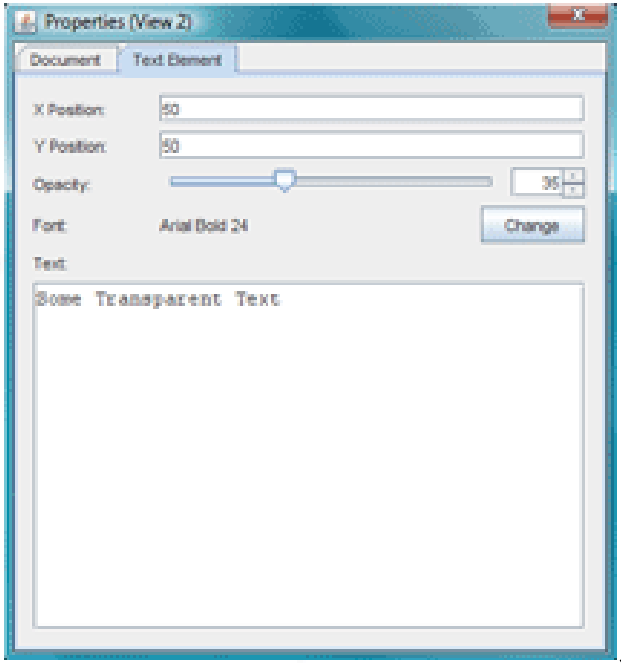
4. écoute des changements d'états du modèle

Model

```
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
    Font.BOLD, 24)
String text = "Some Transparent
    Text";
```

3. délégation au modèle

View



2. appel d'une des 5 méthodes du contrôleur avec la donnée à modifier

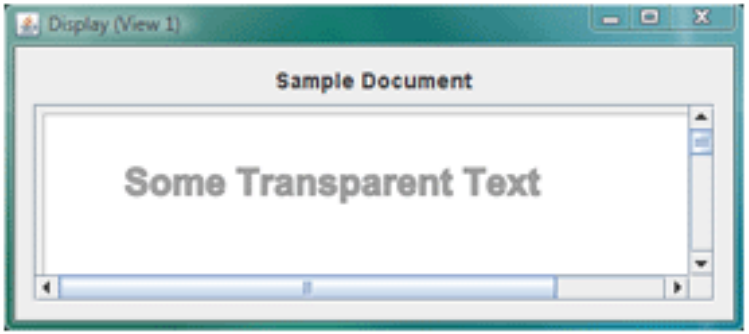
Controller

```
public void changeXPosition(int newXPosition);
public void changeYPosition(int newYPosition);
public void changeOpacity(double newOpacity);
public void changeFont(Font newFont);
public void changeText(String newText);
```

1. écoute de l'événement système lié à l'édition d'un des 5 paramètres

1 modèle – Plusieurs vues

View

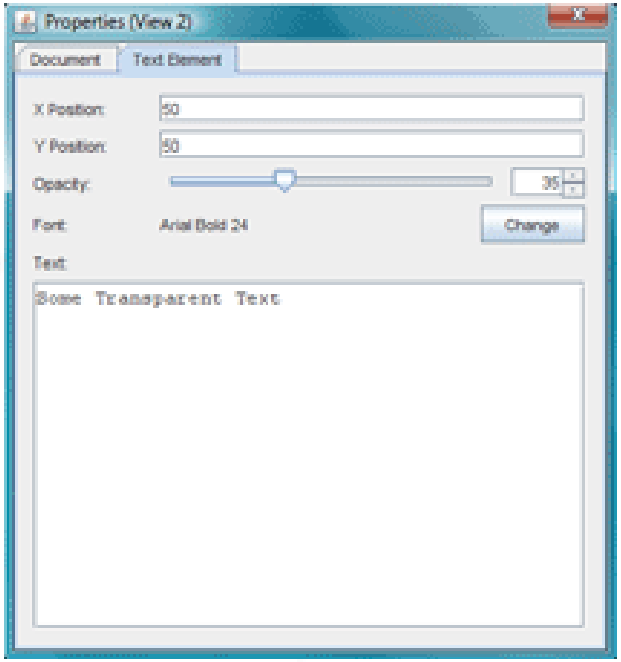


Un ou plusieurs contrôleurs ?

Model

```
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
    Font.BOLD, 24)
String text = "Some Transparent
    Text";
```

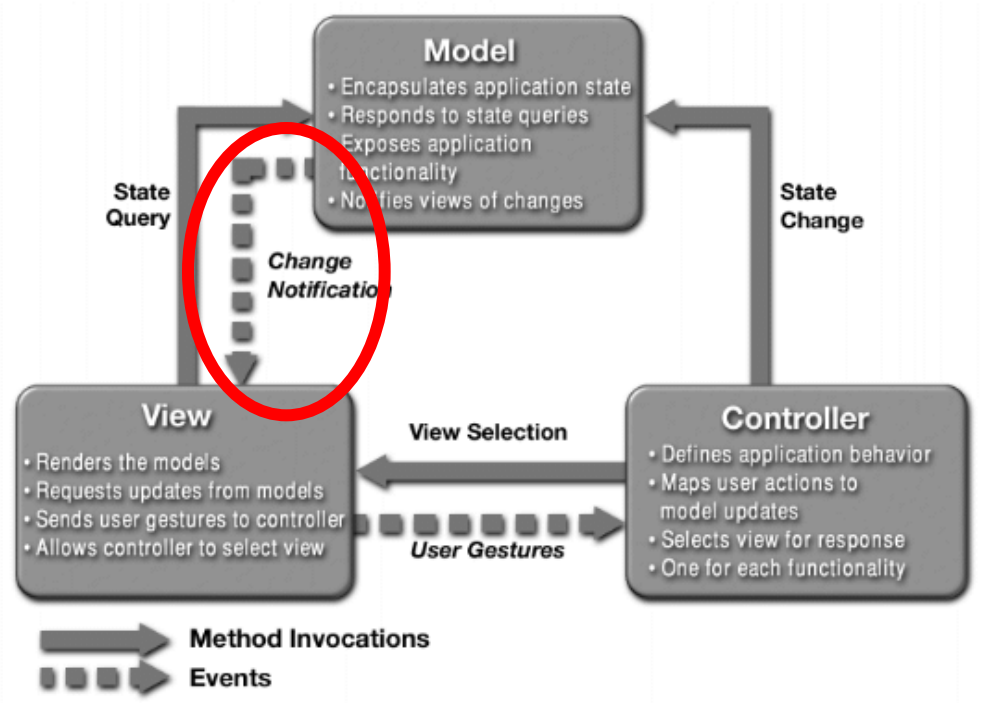
View



Synchronisation entre la vue et le modèle

- Se fait par l'utilisation du pattern Observer.
- Permet de générer des événements lors d'une modification du modèle et d'indiquer à la vue qu'il faut se mettre à jour

- Observable = le modèle
- Observers = les vues



Observer Observable

Motivation

- Un effet de bord fréquent de la partition d'un système en une collection de classes coopérantes est la nécessité de maintenir la consistance entre les objets reliés entre eux.

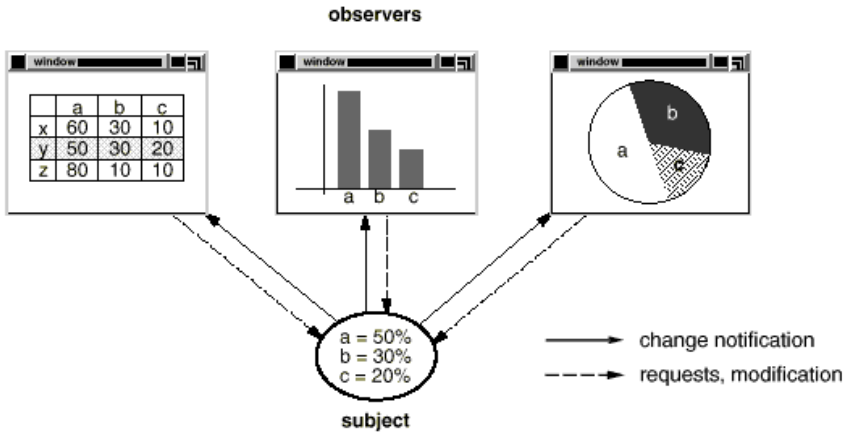
- On ne veut pas obtenir la consistance en liant étroitement les classes, parce que cela aurait comme effet de réduire leur réutilisabilité.

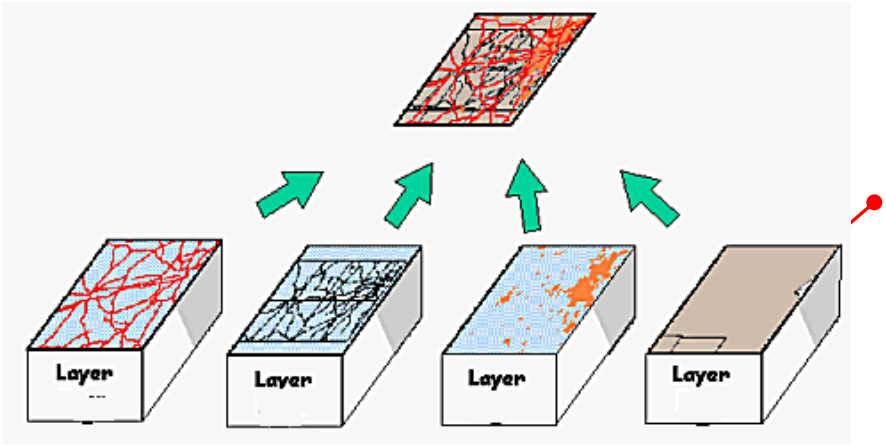
Moyen

Définir une dépendance de “1” à “n” entre des objets telle que

lorsque l’état d’un objet change,

tous ses dépendants sont informés et mis à jour automatiquement





Quand l'appliquer

- Lorsqu'une abstraction possède deux aspects dont l'un dépend de l'autre.
  - L'encapsulation de ces aspects dans des objets séparés permet de les varier et de les réutiliser indépendamment.
  - Exemple : Modèle-Vue-Contrôleur

- Lorsqu'une modification à un objet exige la modification des autres, et que l'on ne sait pas a priori combien d'objets devront être modifiés.

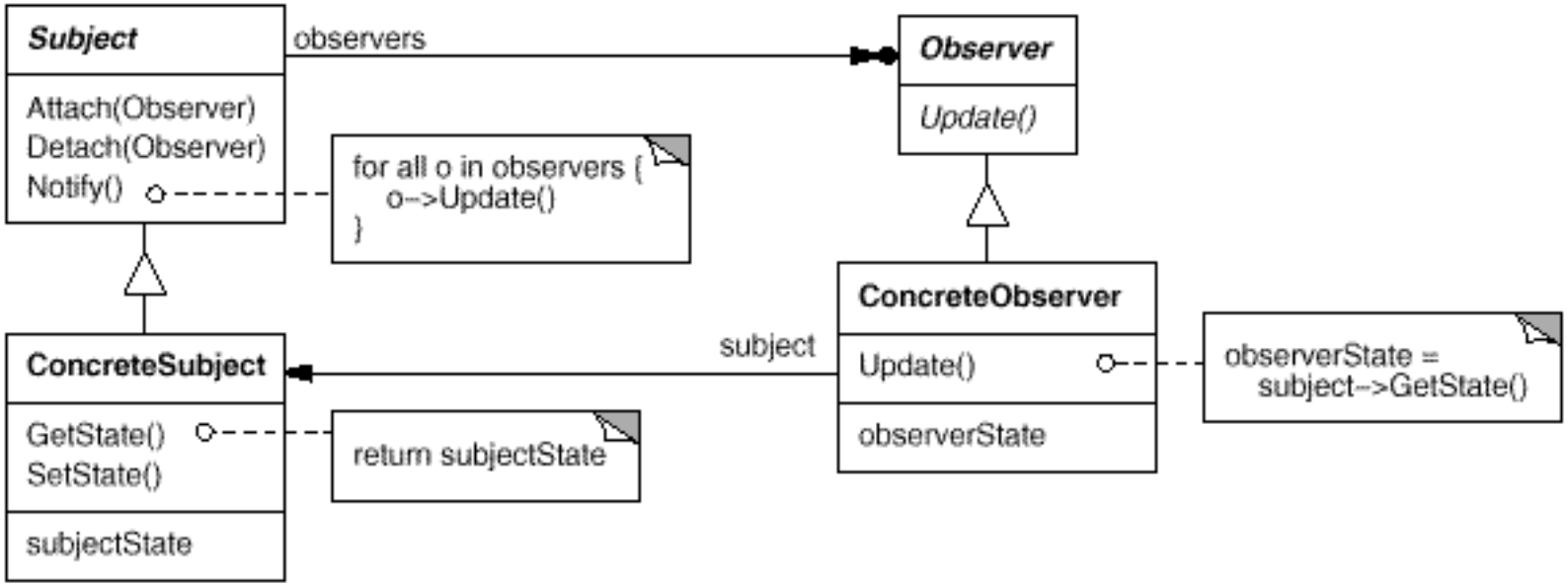
- Lorsqu'un objet devrait être capable d'informer les autres objets sans faire d'hypothèses sur ce que sont ces objets,

Besoin d'événements

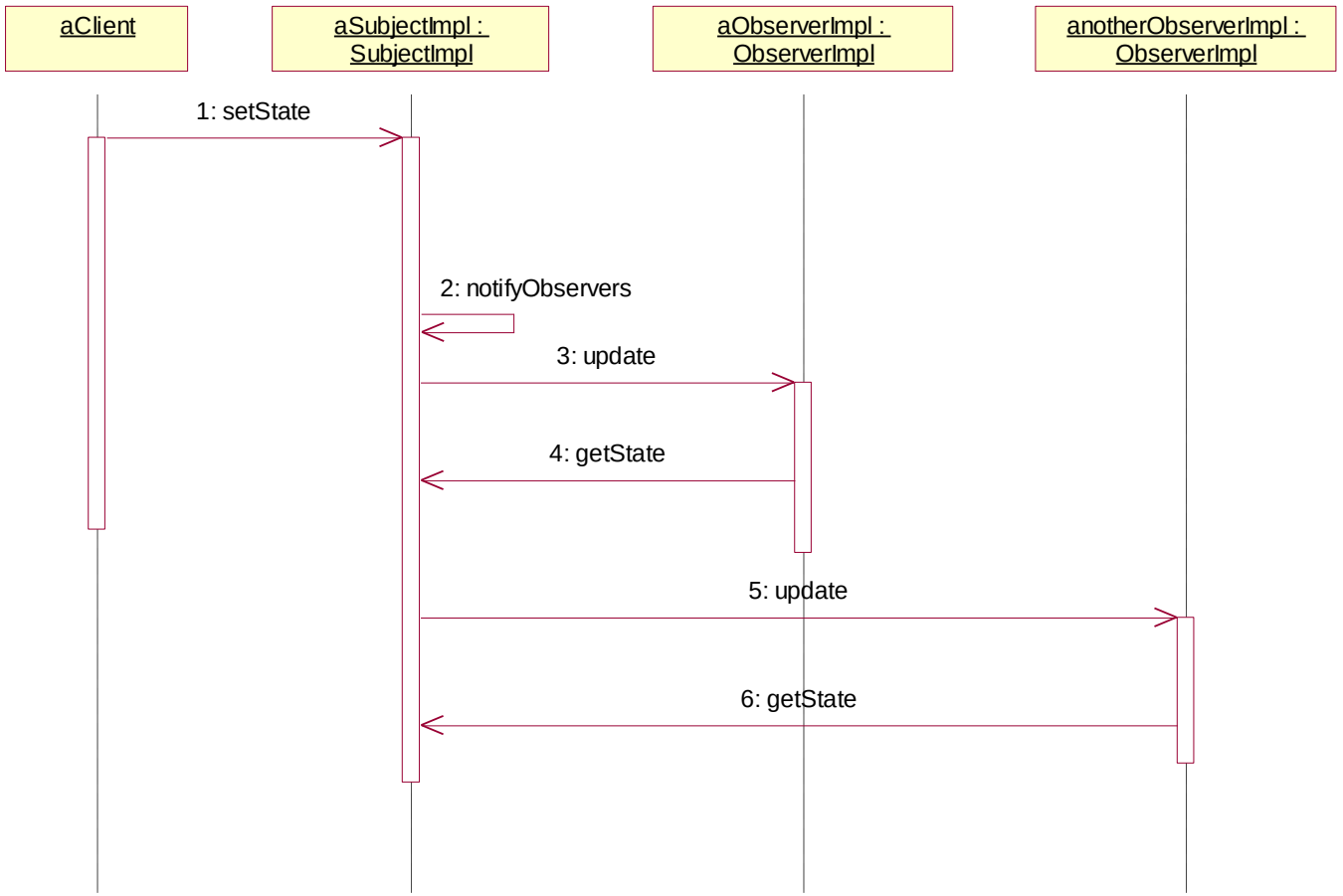
- Le pattern "Observer" décrit
  - comment établir les relations entre les objets dépendants.

- Les objets-clés sont
  - la **source**
    - Peut avoir n'importe quel nombre d'observateurs dépendants
    - Tous les observateurs sont informés lorsque l'état de la source change
  - l'**observateur**.
    - Chaque observateur demande à la source son état afin de se synchroniser

Structure


Subject
observers
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState

Collaborations



Bénéfices

- Utilisation indépendante des sources et des observateurs.
  - On peut réutiliser les sources sans réutiliser les observateurs et vice-versa.
  - On peut ajouter des observateurs sans modifier la source et les autres observateurs.
- Support pour la communication “broadcast”
  - La source ne se préoccupe pas du nombre d’observateurs.

Implémentations Java du pattern

Une classe et une interface : class Observable {... } et
interface Observer

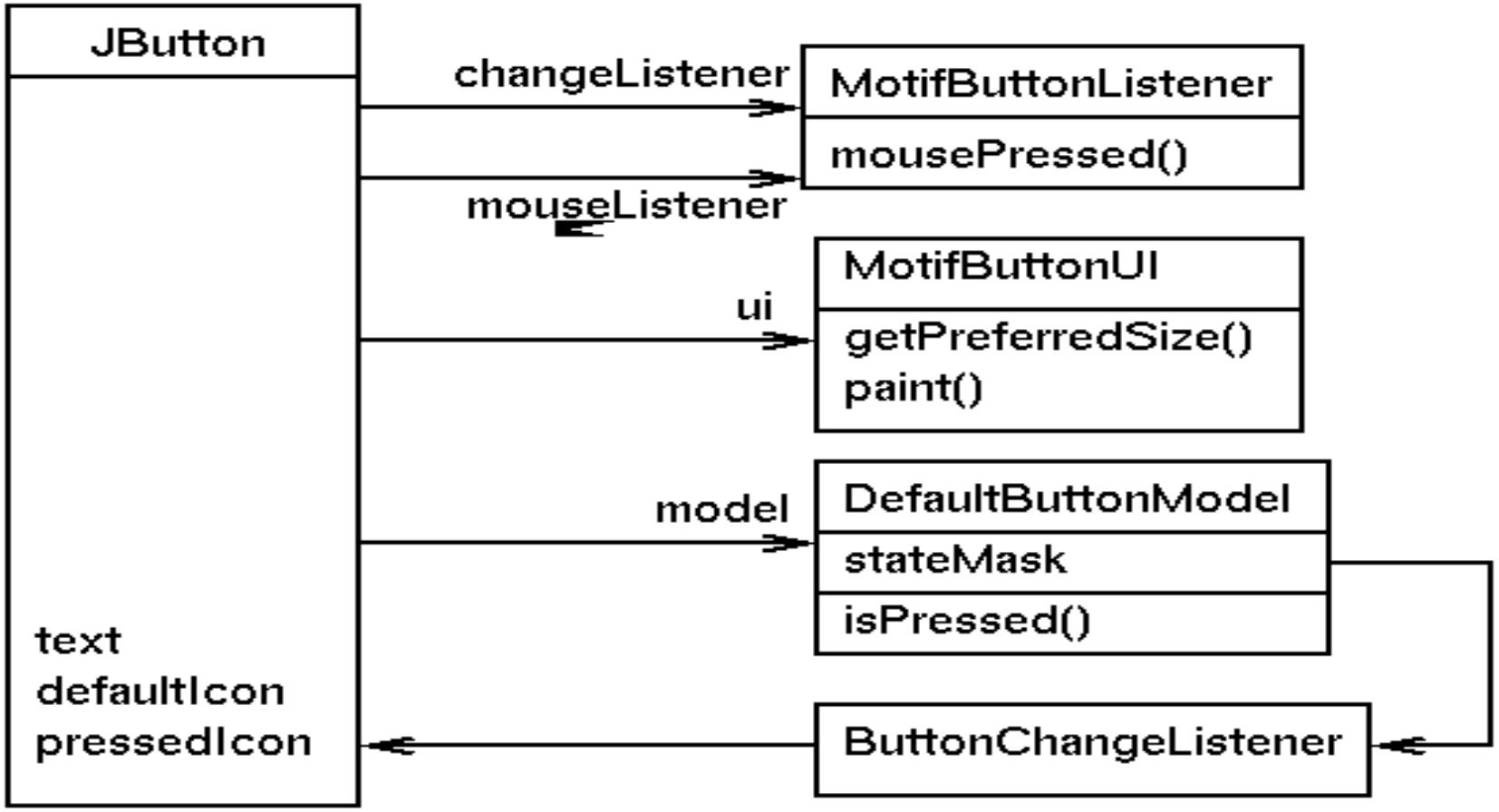
Un objet Observable doit être une instance de la classe qui dérive de la classe Observable

Un objet observer doit être instance d'une classe qui implémente l'interface Observer

void update(Observable o, Object arg);

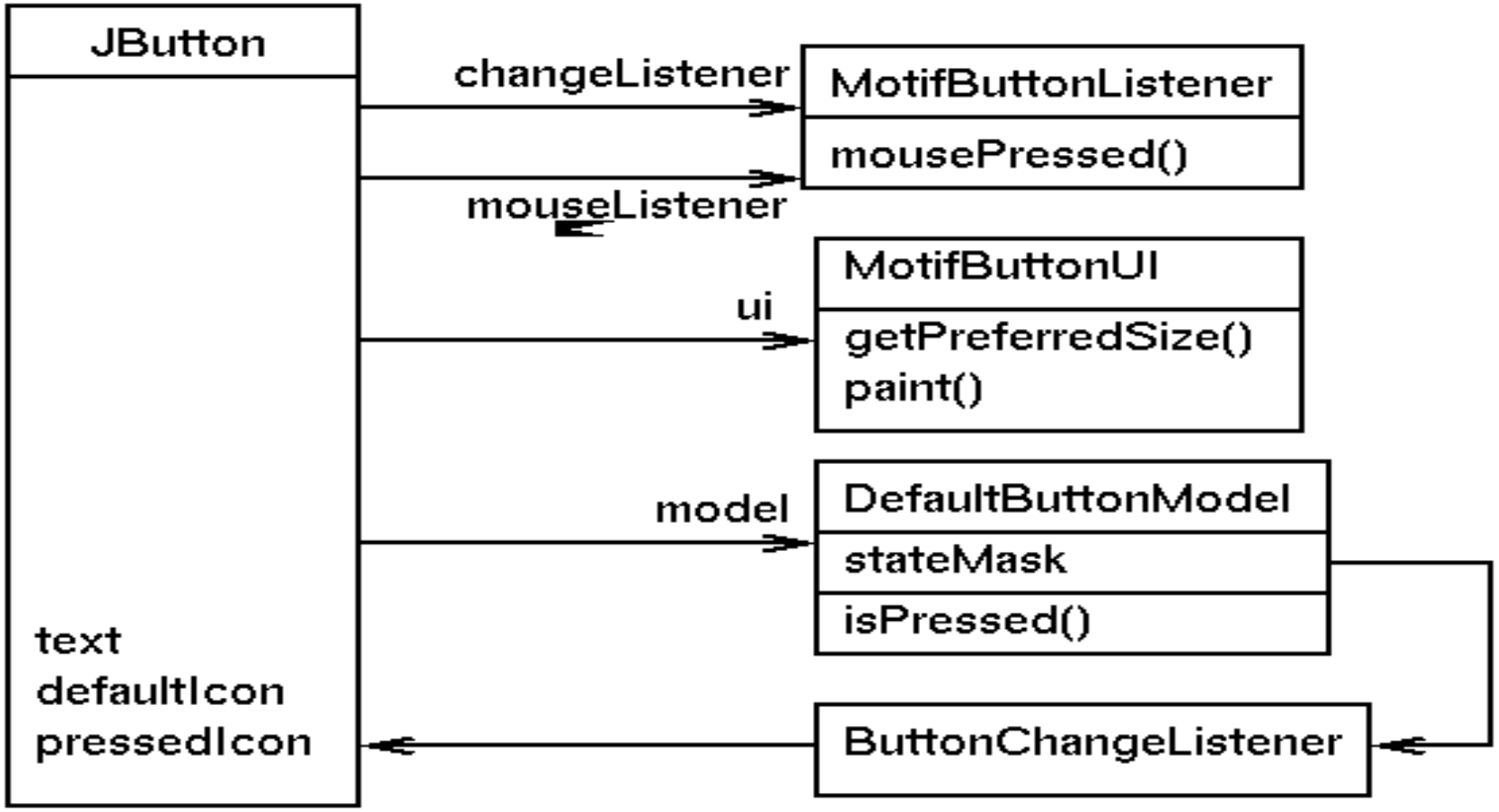
Des listeners : ajouter des listerners, notifier les listeners avec des événements, réagir aux événements

Exemple de Listener



jan.newmarch.name/java/replayJava/paper.htm

Exemple de Listener

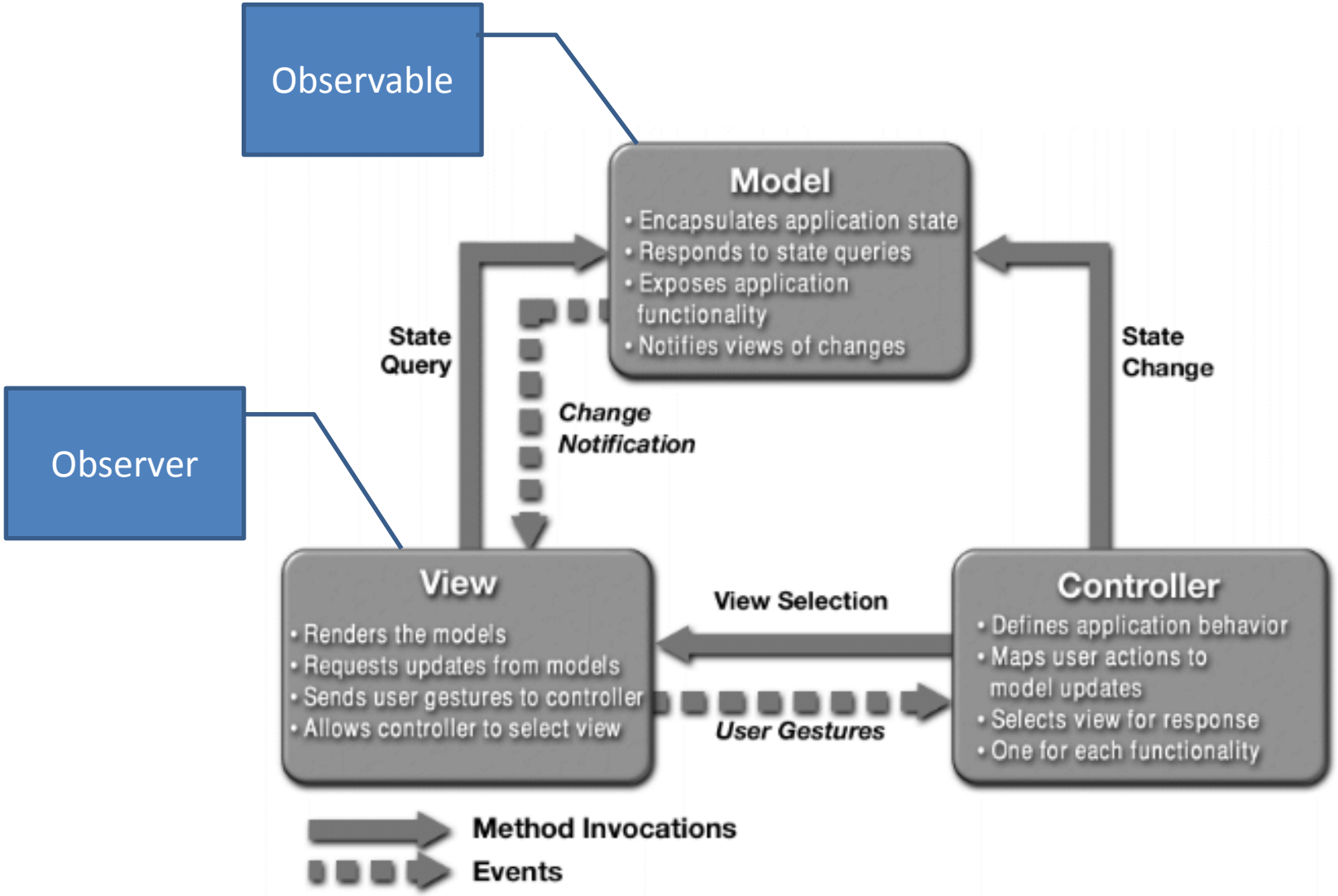


jan.newmarch.name/java/replayJava/paper.htm

Listeners Supported by Swing Components

- http://java.sun.com/docs/books/tutorial/uiswing/events/intro.html

Le pattern Observer et MVC



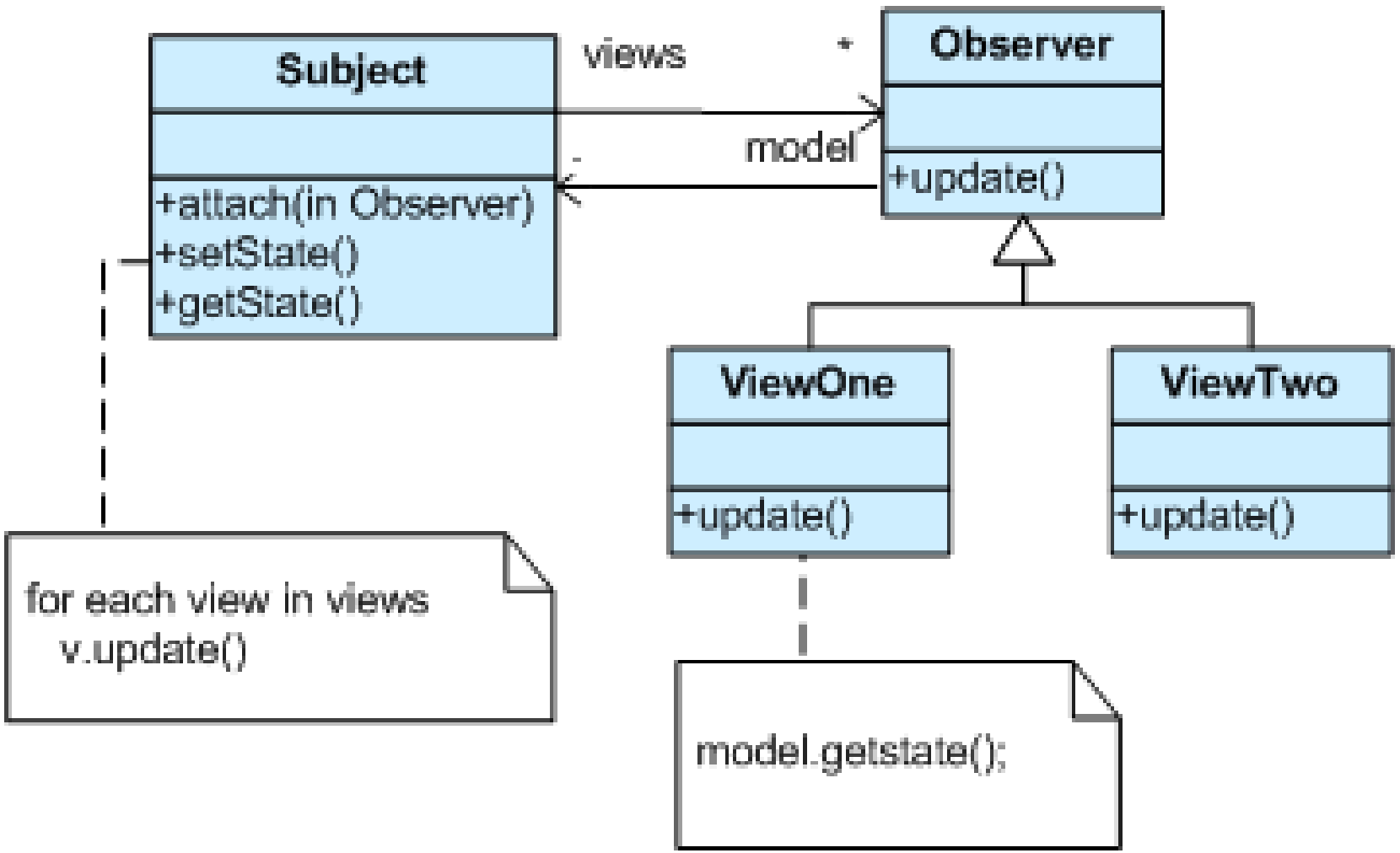
Rappel sur le patron MVC

- Impose une séparation en 3 couches :
  - M : représente les données de l'application. Définit interaction avec la base de données et le traitement des données
  - V : représente l'interface utilisateur. Effectue aucun traitement, ne fait que l'affichage des données (fournit par M). Possibilité d'avoir plusieurs vues pour un même M
  - C : gère l'interface entre le modèle et le client. Interprète la requête de ce dernier pour lui envoyer la vue correspondante. Effectue la synchronisation entre le modèle et les vues

Synchronisation entre la vue et le modèle

- Se fait par l’utilisation du pattern Observer.
- Permet de générer des événements lors d’une modification du modèle et d’indiquer à la vue qu’il faut se mettre à jour

Synchronisation entre la vue et le modèle



Mise en place

- Exemple assez simple d'une application qui permet de modifier un volume.

- Plusieurs vues pour représenter le volume et après toutes modifications, toutes les vues devront être synchronisées.

- Conseils : dans une grosse application, il est important de faire différents packages pour les différentes préoccupations

Le modèle : la base

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

Pour la notification

- Pour permettre la notification de changement de volume, on utilise des *listeners*

- On crée donc un nouveau *listener* (*VolumeListener*) et un nouvel événement (*VolumeChangeEvent*)

Le listener et l'event

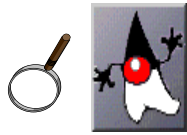
```
import java.util.EventListener;
public interface VolumeListener extends EventListener {
    public void volumeChanged(VolumeChangedEvent event);
}
```

```
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;
    }
}
```

Implémentation du système d'écouteurs dans le modèle (1/2)

```
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;
    }
    public void setVolume(int volume) {
        this.volume = volume;    fireVolumeChanged();   }
```

Implémentation du système d'écouteurs dans le modèle (2/2)



```
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(VolumeListener.class);
        for(VolumeListener listener : listenerList){
            listener.volumeChanged(
                    new VolumeChangedEvent(this, getVolume()));
        }
    }
}
```

Ce que l'on a maintenant

- Le modèle est maintenant capable d'avertir tous ses écouteurs à chaque changement de volume

- En fonction de l'application, il est possible d'imaginer plusieurs *listeners* par modèles et d'autres événements dans les *listeners* (par exemple quand le volume dépasse certains seuils)

- Remarque : le modèle peut devenir très vite conséquent

Pré-requis pour le contrôleur

- Pour éviter d'être dépendant de *Swing*, on va créer une classe abstraite représentant une vue de volume

```
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

- Manipulera des objets de type *View* et non plus de type *Swing*

- Un seul contrôleur sera créé dans un soucis de simplicité puisque les 3 vues font la même chose.

- Dans le cas de vues complètement différentes, il est fortement conseillé d'utiliser plusieurs contrôleurs

Les vues

- Nous supposons 3 vues (dans le cadre du cours, nous n'en montrerons qu' une) :
  - Une vue permettant de modifier le volume avec un champ de texte et valider par un bouton
  - Une vue permettant de modifier le volume à l'aide d'une jauge + bouton pour valider
  - Une vue listant les différents volumes

- Toutes ces vues sont représentées par une JFrame

Contrôleur (1/2)

```
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();
    }
    private void addListenersToModel() {
        model.addVolumeListener(fieldView);
        model.addVolumeListener(spinnerView);
        model.addVolumeListener(listView);
    }
```

Contrôleur (2/2)

```
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

- Nous allons faire 3 vues (dans le cadre du cours, nous n'en montrerons qu' une) :
  - Une vue permettant de modifier le volume avec un champ de texte et valider par un bouton
  - Une vue permettant de modifier le volume à l'aide d'une jauge + bouton pour valider
  - Une vue listant les différents volume

- Toutes ces vues sont représentées par une JFrame

JFrameField (1/3)

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.NumberFormat;
import javax.swing.*;
import javax.swing.text.DefaultFormatter;
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);
    }
```

JFrameField (2/3)

```
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()).setAllowsInvalid(false);
    contentPane.add(field);
    button = new JButton("Mettre à jour");
    button.addActionListener(this);
    contentPane.add(button);
    frame.setContentPane(contentPane);
    frame.setTitle("JFrameSpinnerVolume");
    frame.pack();
}
```

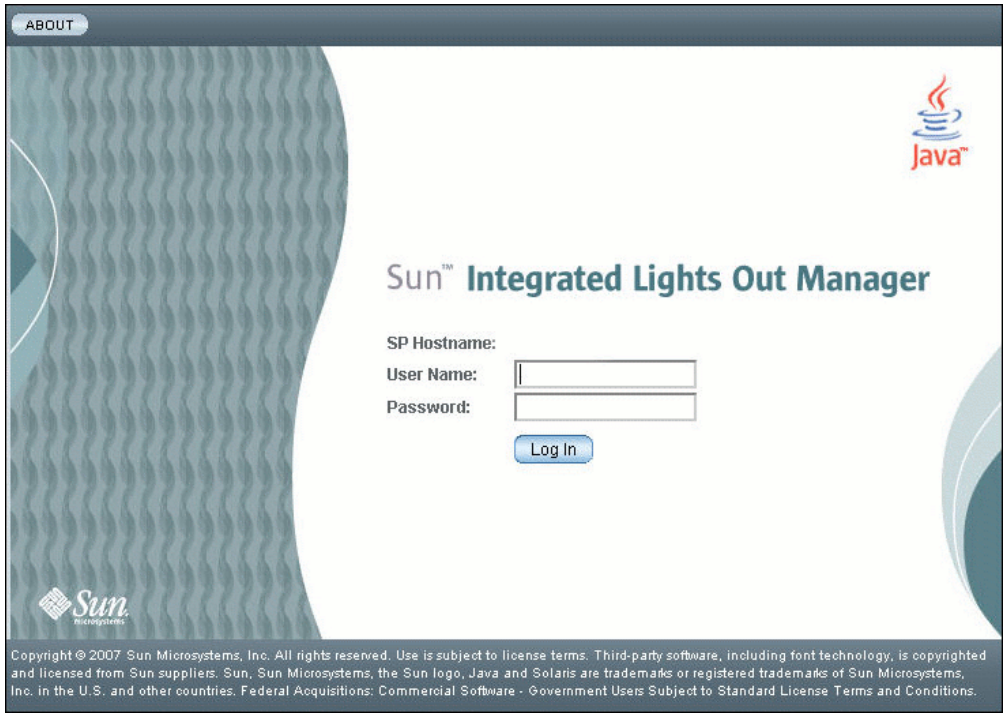
JFrameField (3/3)

```
    @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.parseInt(field.getValue().toString()))
    }
}
```

Avez vous compris MVC ?

L'exemple du login

L’exemple du login



IHM en Anglais

2 vues parmi d’autres

IHM en français

Un modèle

- Des variables d'instance
  - *login* (identifiant du compte)
  - *password* (mot de passe de connexion)
  - *connected* (vrai si on a pu se connecter)

- Des méthodes
  - *boolean loginCheck(Sting) :* vérification du paramètre par rapport à *login*
  - *boolean passwordCheck(String) :* vérification du paramètre par rapport à *password*
  - *void connect(String, String) :* fait aussi toutes les vérifications
  - *void disconnect() : déconnexion du compte*
  - *boolean isConnected() : indique si l'utilisateur est connecté ou pas*

Cas d’interaction

Cas no 1

Les données peuvent être modifiées jusqu’au moment de l’utilisation du bouton

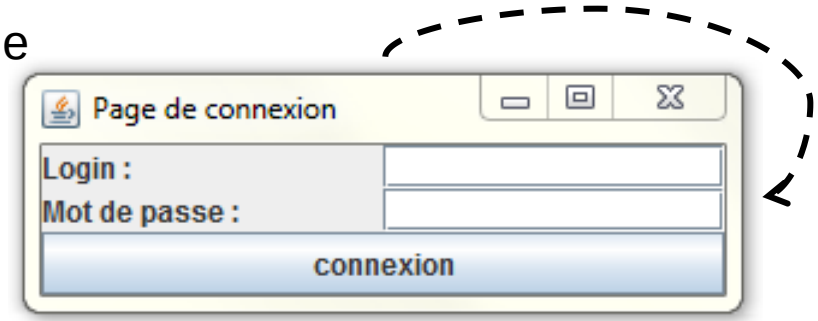
Actions : tentative de connexion + affichage d’une autre fenêtre avec message de connexion ou d’erreur après click sur le bouton

Cas d'interaction no 1

1. écoute de l'événement système lié à l'activation du bouton de connexion

Le Modèle

4. écoute événement de MAJ du modèle



2. appel de méthode sur le contrôleur avec transmission des données login/pwd

3. appel de la méthode connect du modèle

Contrôleur no 1

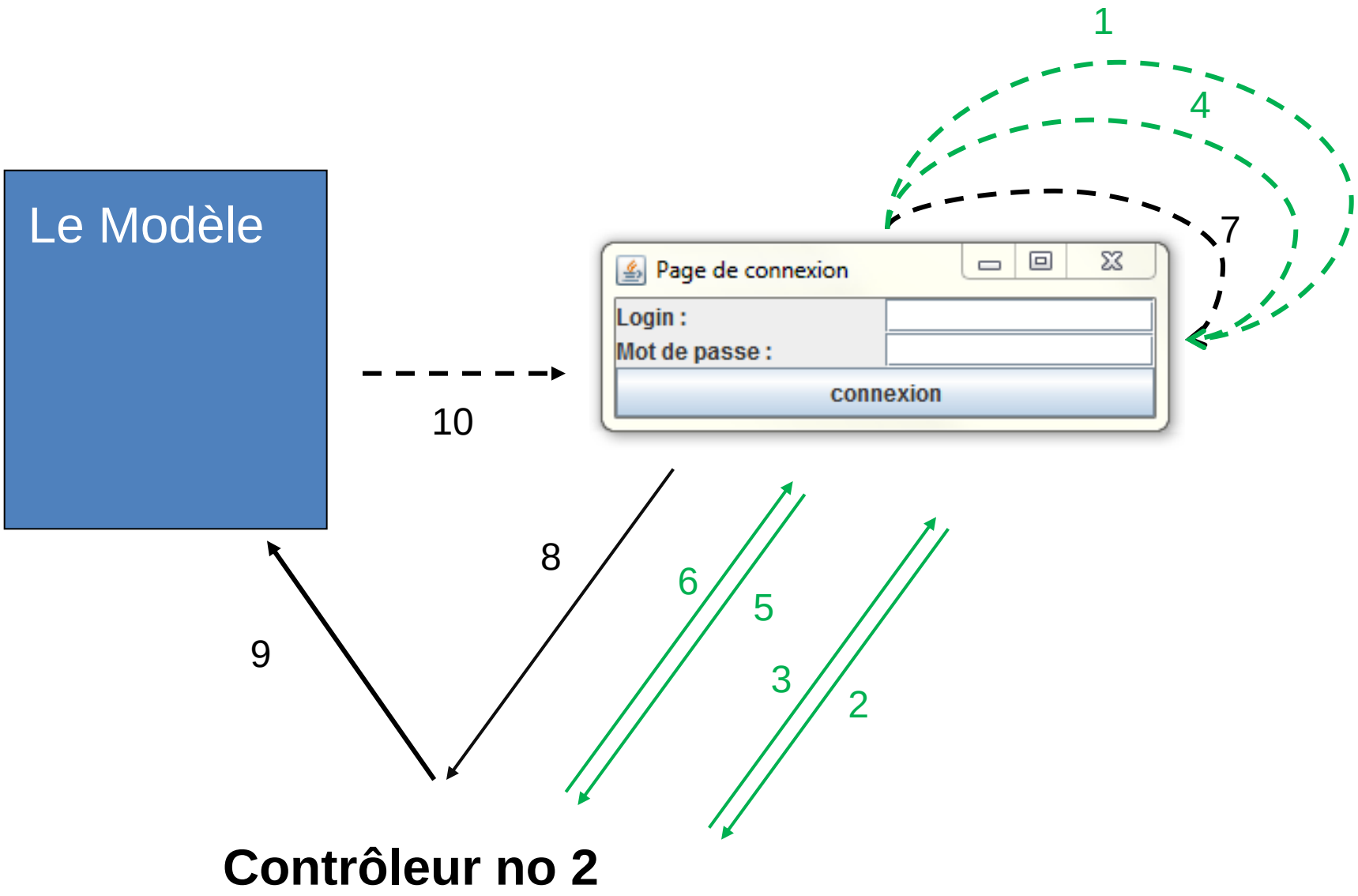
Cas d’interaction

Cas no 2

Les données peuvent être modifiées jusqu’au moment de leur « saisie » par un retour charriot

Actions : tentative de connexion + affichage d’une autre fenêtre avec message de connexion ou d’erreur après click sur le bouton

Cas d'interaction no 2



Cas d’interaction

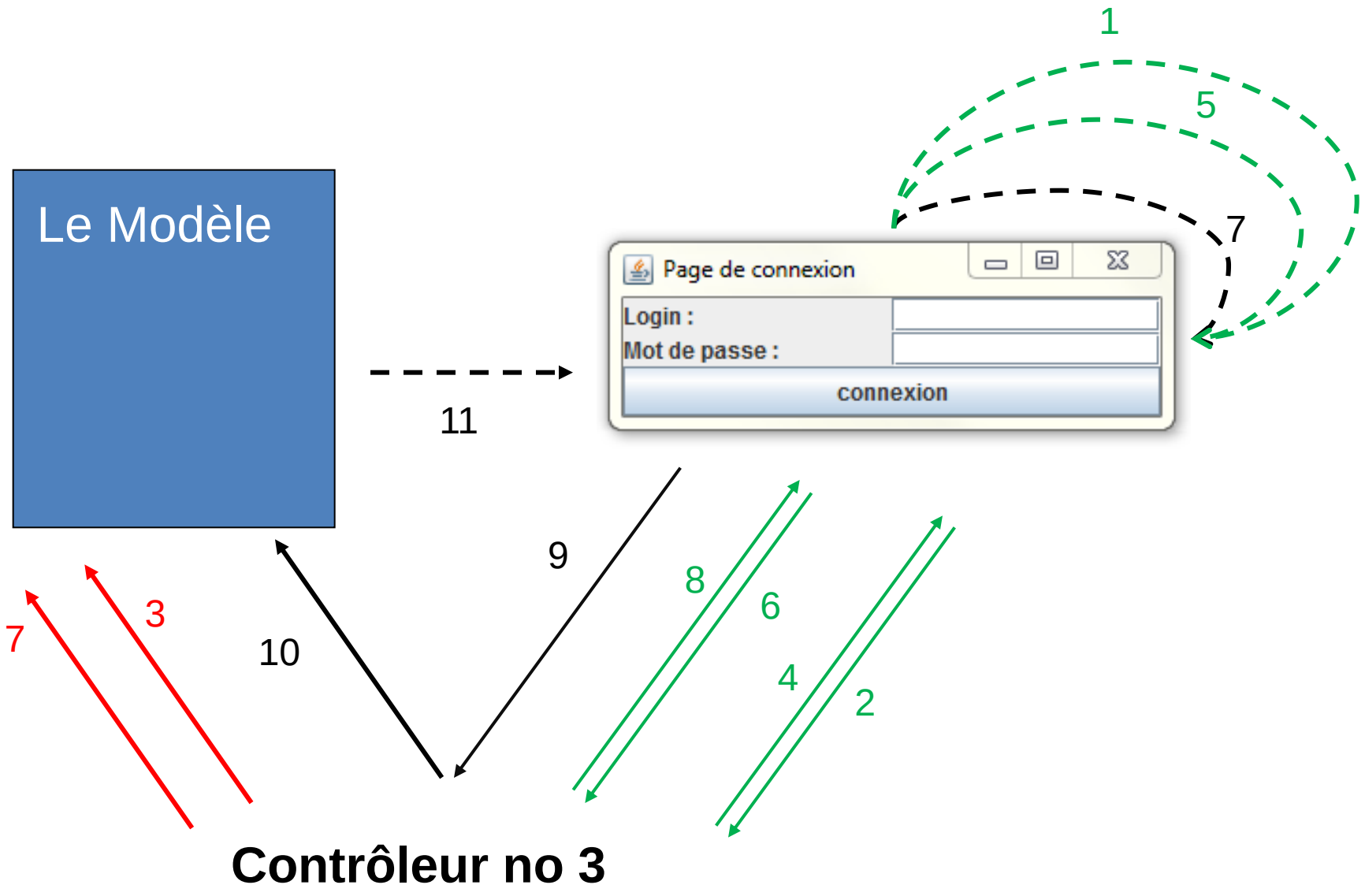
Cas no 3 :

Les données peuvent être modifiées jusqu’au moment de leur « saisie » par un retour charriot

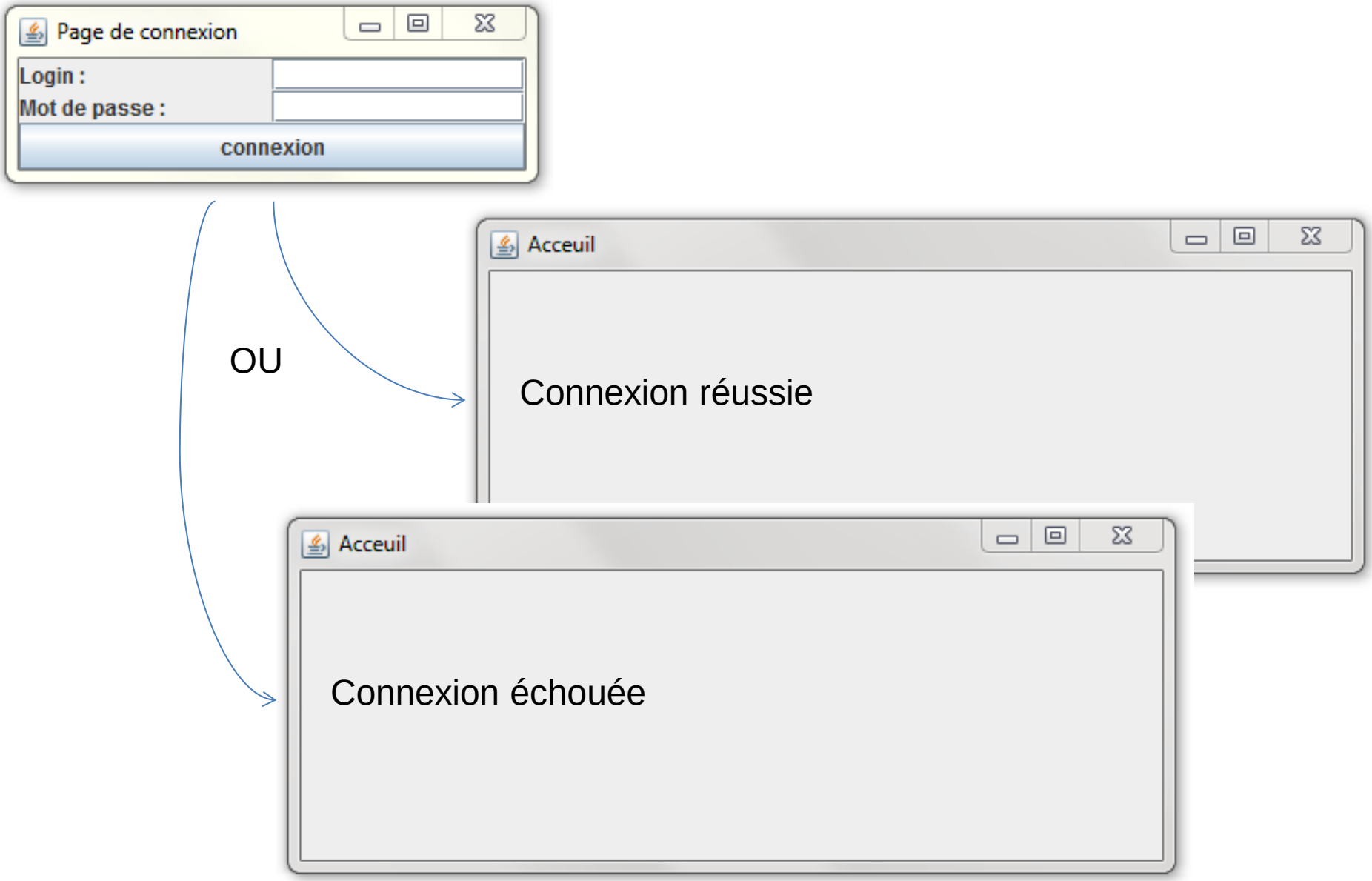
Actions : vérification des données faite au fur et à mesure des retours chariots, ce qui peut entrainer un message d’erreur

connexion ou non + affichage d’une autre fenêtre avec message de connexion ou d’erreur après click sur le bouton

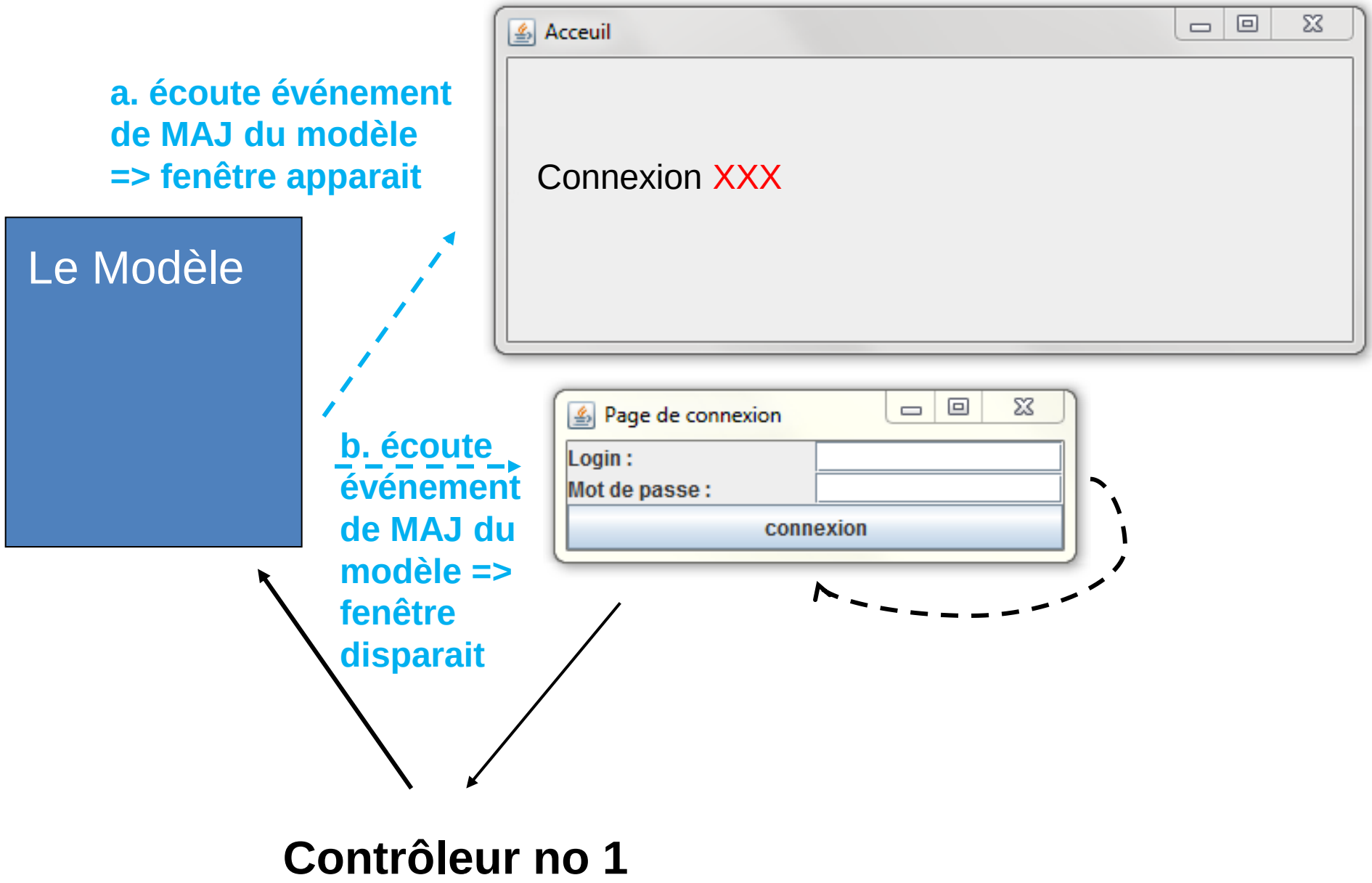
Cas d'interaction no 3



Passage d’une vue à l’autre



Passage d'une vue à l'autre (pour le cas n°1)



Questions

- Avez-vous d'autres exemples d'interactions ?
- Avez-vous d'autres exemples de vues ?
- Quelle est la différence entre vue et interaction ?
- Peut on changer la façon d'interagir sans modifier les vues (ou les modèles) ?
- Plusieurs vues peuvent elles interagir de la même façon ?
- Peut on interagir de plusieurs façons avec une vue ?
- Peut on changer de modèle sans modifier ses vues ?

Conclusions sur MVC

- MVC permet de :
  - Changer une couche sans altérer les autres
  - Synchroniser les vues. Toutes les vues qui montrent la même chose sont synchronisées
- Pour s'assurer une bonne séparation des couches, vérifier que :
  - Les vues ne connaissent pas le type concret des données des modèles
  - Les contrôleurs et les modèles ne connaissent pas le type concret des composants d'interaction des vues