POO / IHM
Architecture Logicielle

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

Exemple MVC pour un éditeur simple

0. initialisation de la vue avec les données du modèle

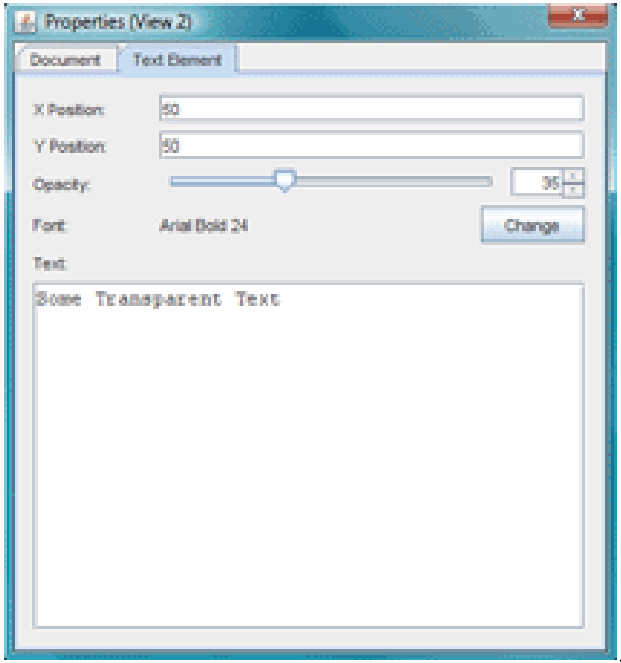
4. écoute des changements d'états du modèle

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

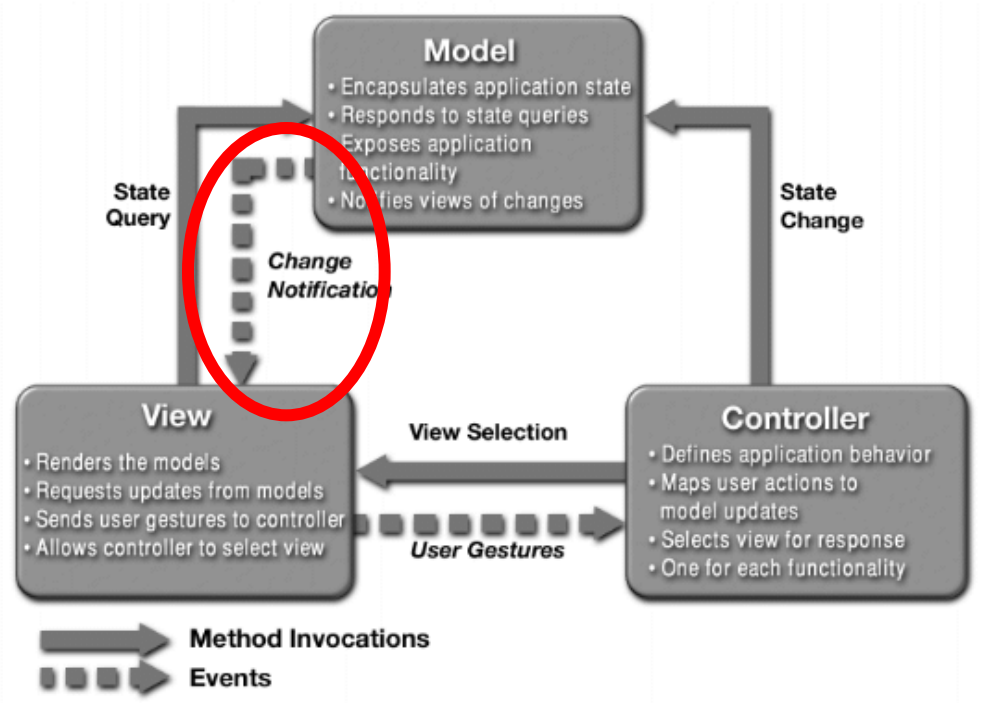
```
public void changeXPosition(int newXPosition);
public void changeYPosition(int newYPosition);
public void changeOpacity(double newOpacity);
public void changeFont(Font newFont);
public void changeText(String newText);
```

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

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



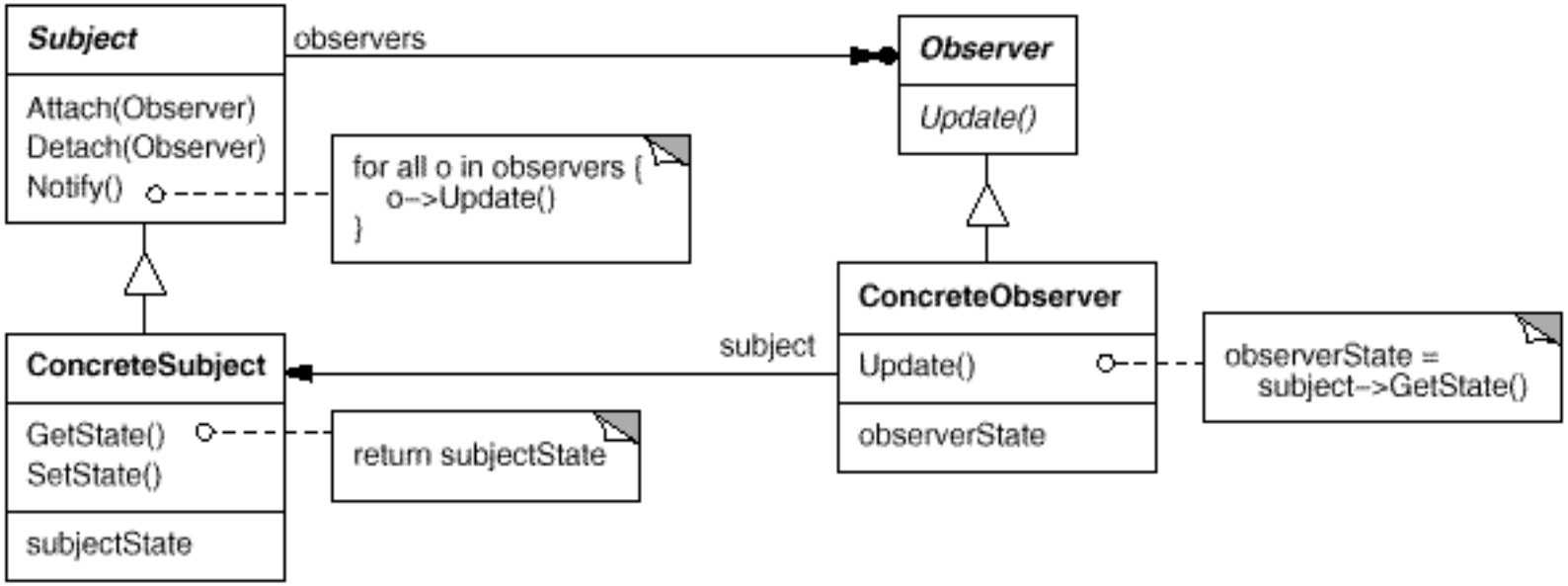
Observer Observable

Besoin d'événements

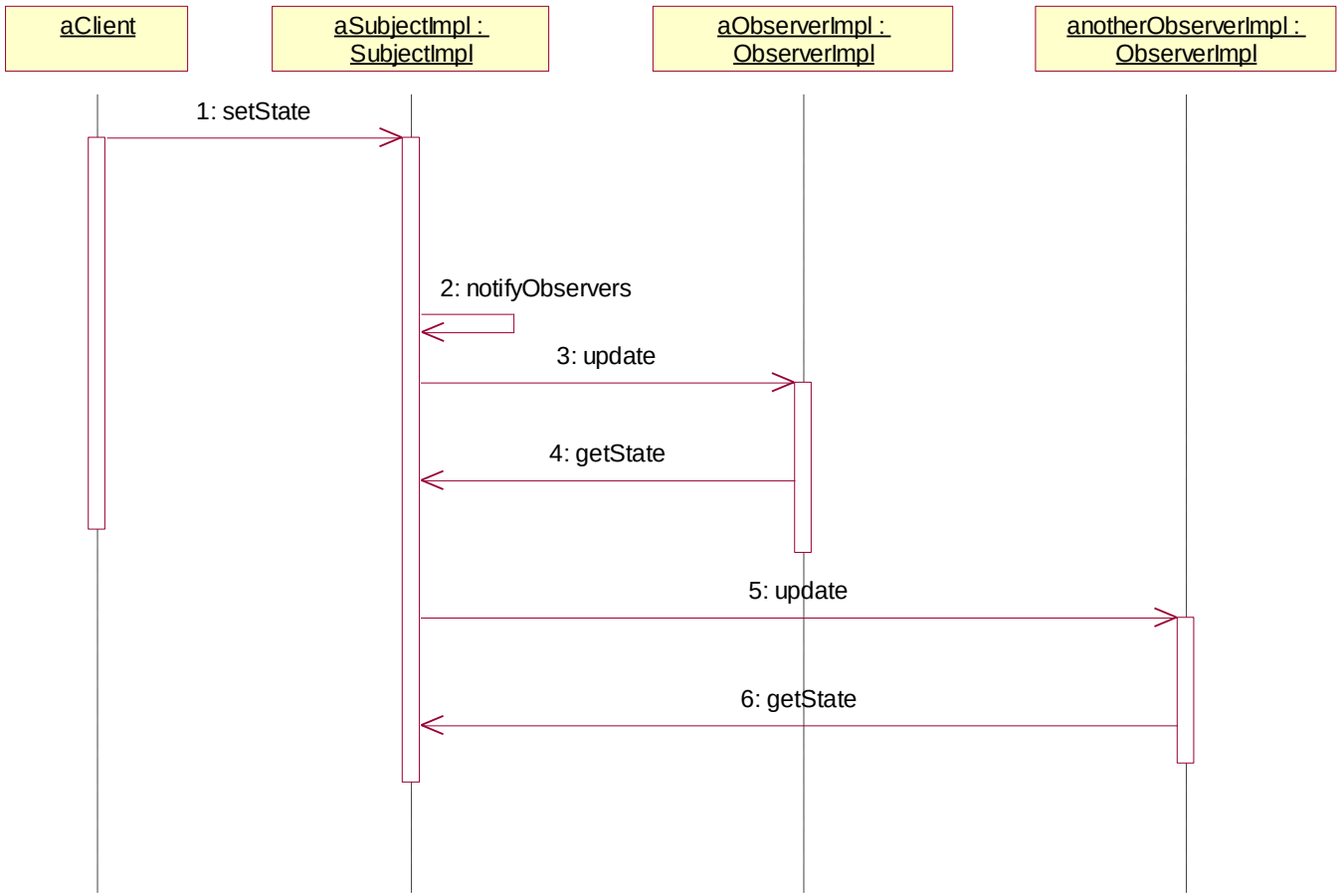
- Le pattern "Observer" décrit
  - comment établir les relations entre les objets dépendants.

- 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

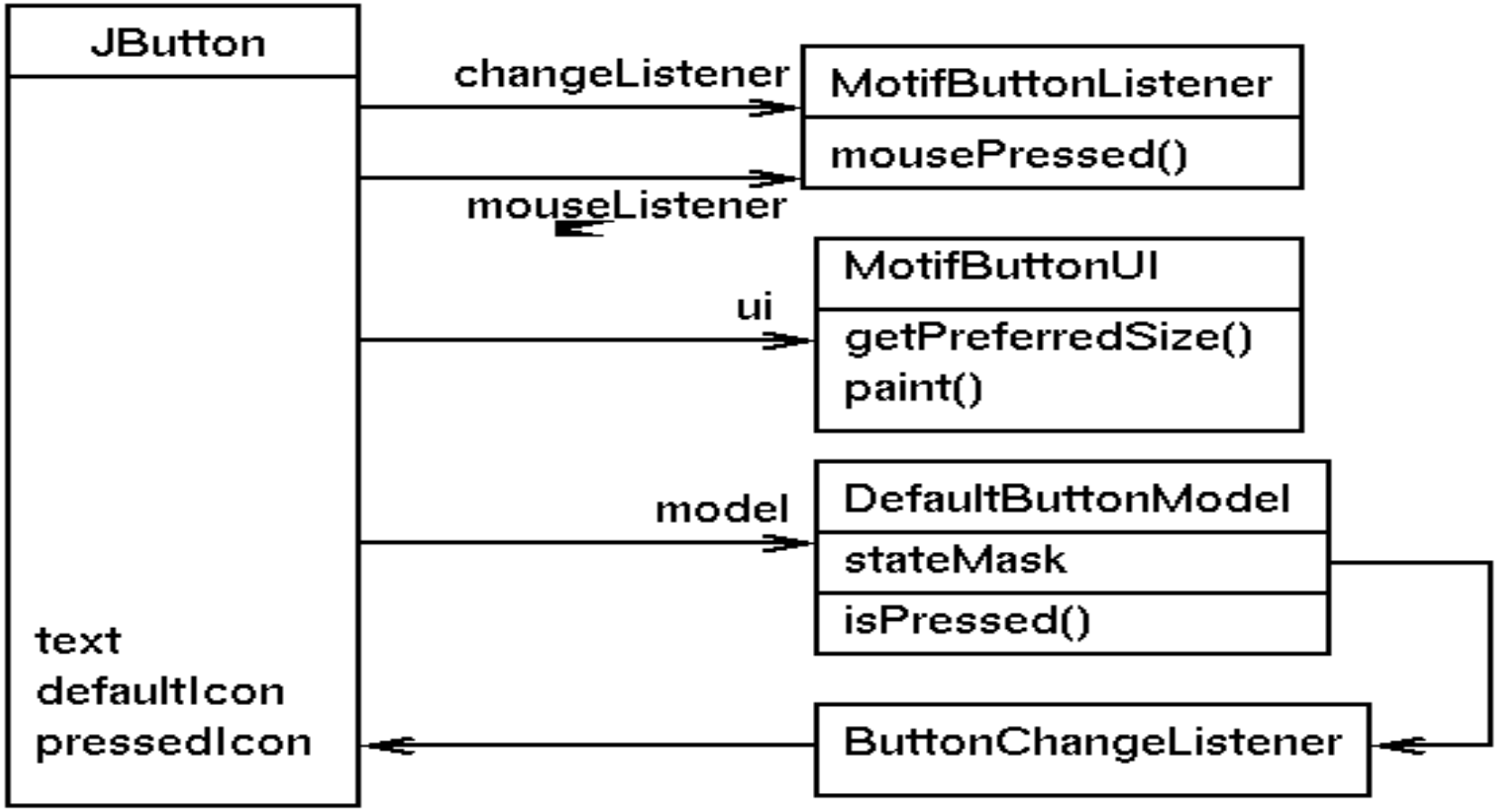
Un objet Observable doit être une instance de la classe qui dérive de la classe Observable

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

void update(Observable o, Object arg);

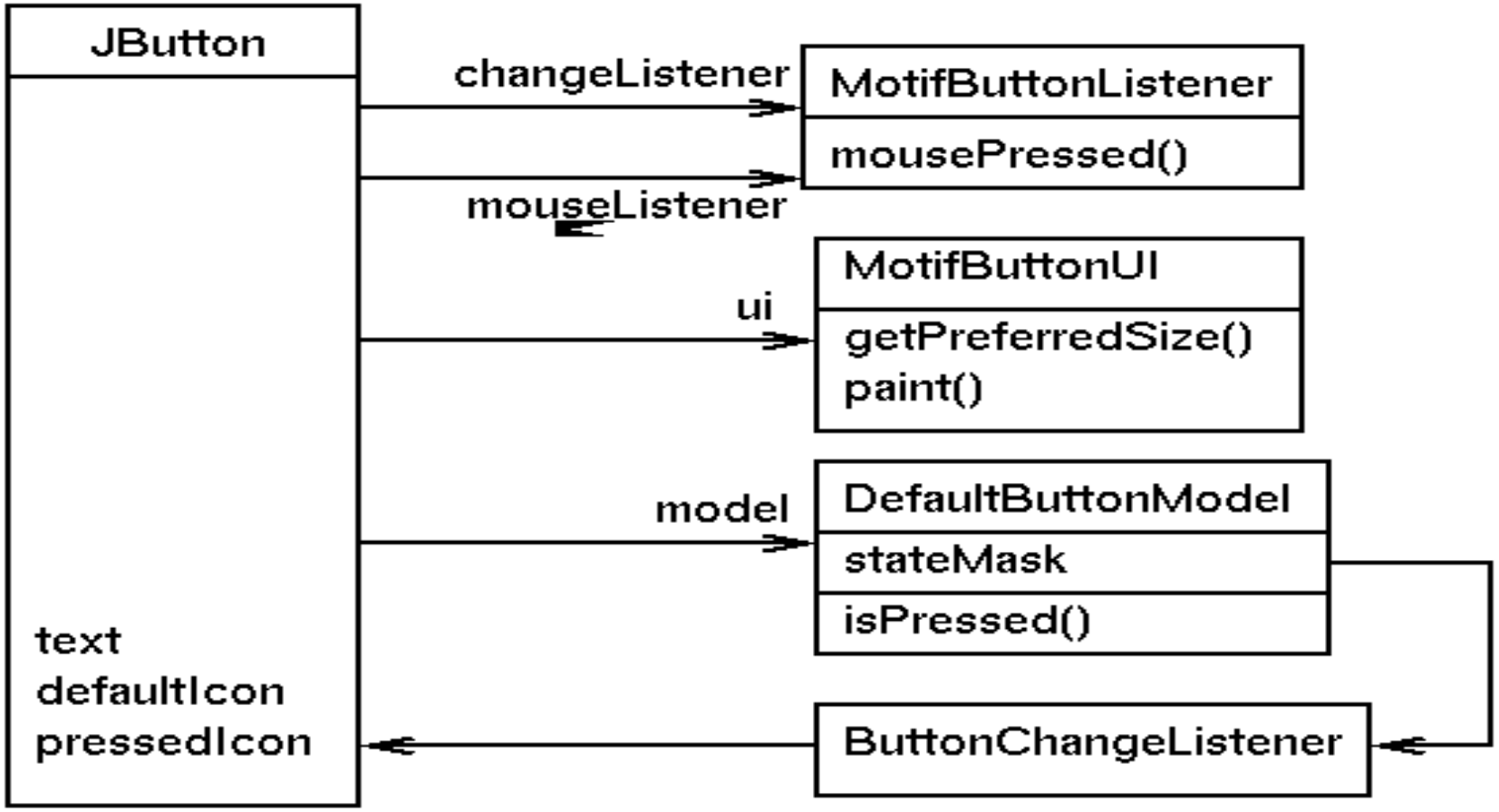
Des listeners : ajouter des listerners, notifier les listeners avec des événements, réagir aux événements

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 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

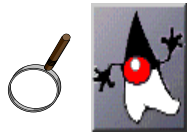
```
import java.util.EventListener;
public interface VolumeListener extends EventListener {
    public void volumeChanged(VolumeChangedEvent event);
}
```

```
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()))
    }
}
```

Un peu plus sur les événements

Partie comportementale

Audrey Occello et AM Dery-Pinna

Les événements

- Les composants Swing peuvent avertir d'une action utilisateur effectuée sur eux-mêmes

- Sans gestion des événements, l'IHM n'a aucun comportement …

- Pour la plupart, ils se trouvent dans les packages java.awt.event et javax.swing.event
- Ils se nomment XXXEvent avec XXX pouvant se décliner en :
  - Key (clavier)
  - Mouse (clic et mouvement entrant ou sortant)
  - Focus (activation ou désactivation du focus du clavier)
  - Window (seulement JDialog et JFrame – fermeture, iconification, ...)
  - …

Les événements

- Les composants Swing peuvent avertir d'u action utilisateur effectuée sur eux-mêmes
- Sans gestion nportement …
- Pou a.awt.event et javax.sw

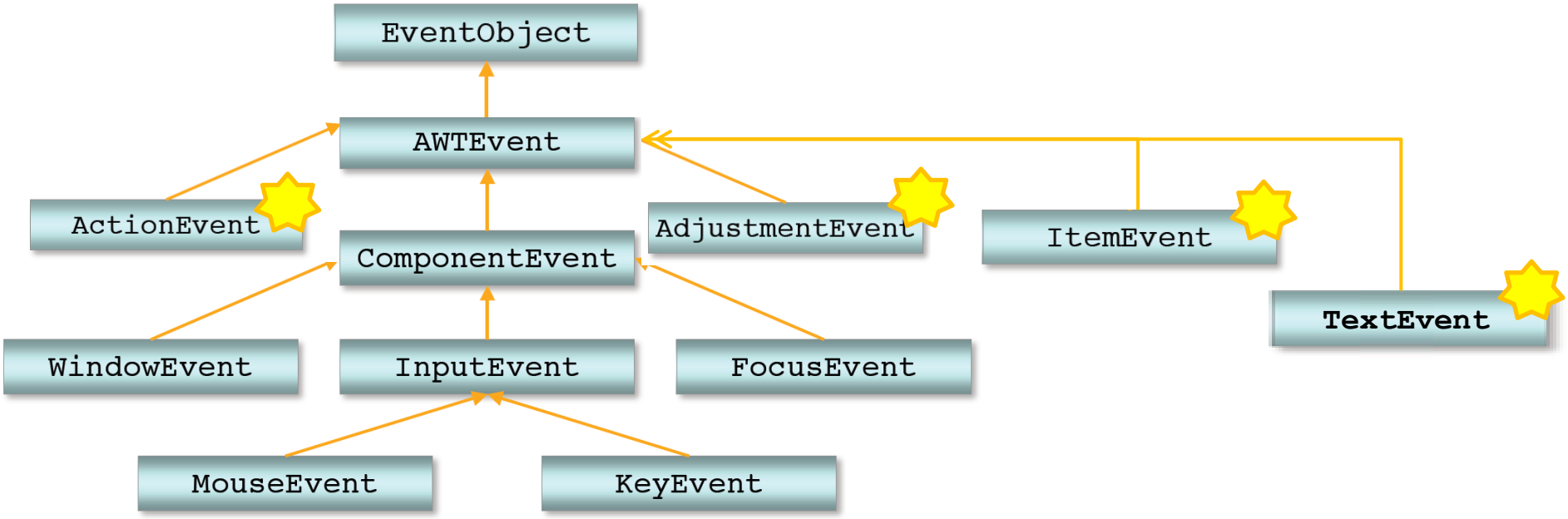
liner en :

  - K
  - M
  - focus du clavier)
  - Window (s ent JDi g et JFrame – fermeture, iconification, ...)
  - …

Il s'agit ici d'événements dits « systèmes ». On ne contrôle pas leur déclenchement. Nous verrons plus tard une autre forme d'événements.

AWTEvent

- dérive de *java.util.EventObject*
- contient par héritage
  - *getSource()* qui renvoie la source de l'événement (objet où l'événement s'est produit)



EVÉNEMENTS *ACTIONEVENT*

- une action sur un composant réactif :
  - Clic sur un item de menu,
  - Clic sur un bouton,
  - …
- Sont émis par les objets de type :
  - Boutons : *JButton, JToggleButton, JCheckBox*
  - Menus : *JMenu, JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem …*
  - Texte : *JTextField*

EVÉNEMENTS : *ITEMEVENT*

- composant est sélectionné ou désélectionné
- émis par les objets de type :
  - Boutons : *JButton, JToggleButton, JCheckBox*
  - Menus : *JMenu, JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem*
  - Mais aussi *JComboBox, JList*

EVÉNEMENTS *ADJUSTMENTEVENT*

- Se produisent quand un élément ajustable comme une *JScrollBar* sont ajustés

- Sont émis par *ScrollBar*, *JScrollBar*

EVÉNEMENTS *TEXTEVENT*

- Se produisent quand le contenu d'un élément contenant du texte est modifié

- Sont émis par *TextField, TextArea, JTextField*, *JTextArea, JPasswordField*

Autres événements en bref

- Les sous-classes de *AbstractButton* génèrent des **ChangeEvent** quand on modifie l'état d'un bouton
- Les sous-classes de *JMenuItem* génèrent des ***MenuDragMouseEvent*** et des ***MenuKeyEvent***
- Une *JList* génère des ***ListSelectionEvent*** quand on sélectionne des elements de la liste

- ***Evénements non graphiques :*** Les modèles de données associés aux listes et aux tables génèrent des ***ListDataEvent*** et des ***TableModelEvent*** quand des changements se produisent sur les données

Les écouteurs

- Pour être averti, il faut écrire un objet capable d'être averti !
- Depuis (JAVA 1.1), les interfaces écouteurs
  - correspondent aux événements principaux
  - définissent une méthode par action utilisateur élémentaire
  - extensions de *java.util.EventListener*

- Un objet implémente l'interface "d'écoute" correspondant à l'événement à surveiller
  - Par exemple, un *JButton* avertit les objets implémentant l'interface *ActionListener* des clics

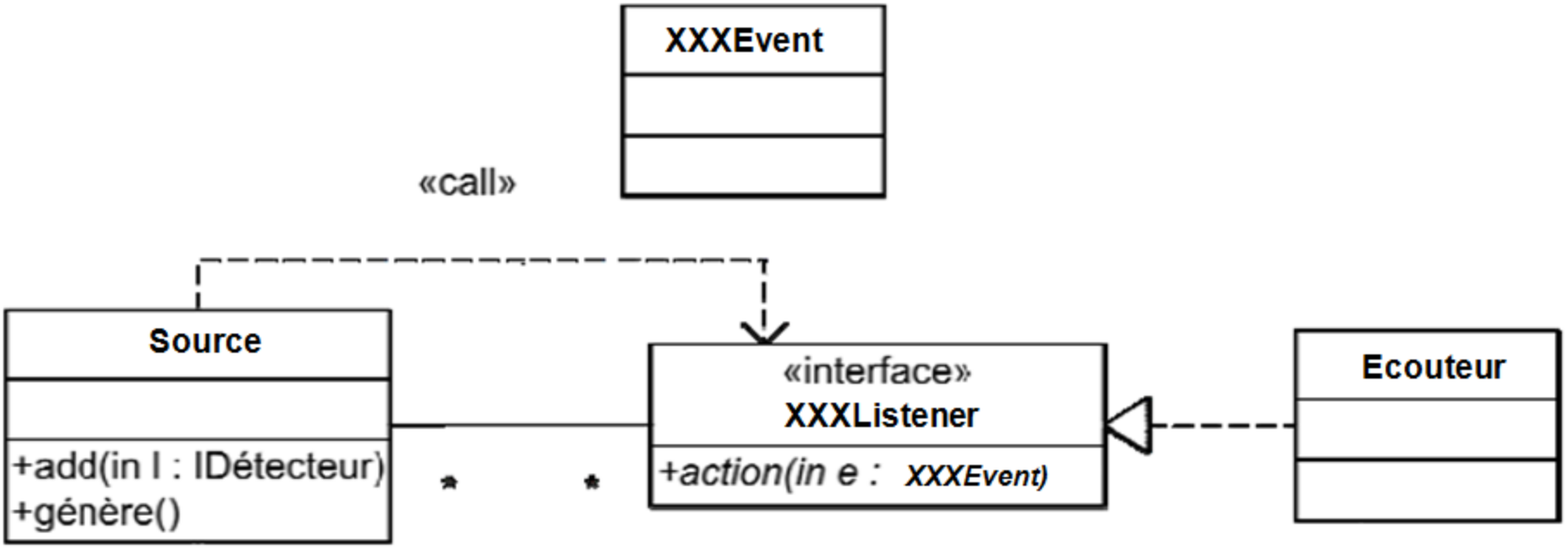
Principes des écouteurs

- Programmation par événements :
  1. Un écouteur s'abonne à une source
  2. La source déclenche l'événement : propagation à toute la hiérarchie de composition (le composant directement concerné puis tous ceux qui le contiennent) + aux abonnés
  3. Les abonnés réagissent

- Les composants graphiques sont des sources

- Les événements déclenchés par une action utilisateur englobent des informations que l'on peut récupérer pour réagir en fonction

Principes des écouteurs



Comment connaître les listeners associés à un composant donné ?

- avec une liste exhaustive :

http://download.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html

- en repérant dans la documentation les méthodes "addXXXListener" XXX étant les mêmes que pour "XXXEvent"

- Attention toutes les instances d'un même type de composant graphique (objets de même classe) émettent les mêmes événements !
  - Par exemple, on peut vouloir écouter les clicks émis par deux boutons
- Comment réagir différemment dans ce cas ?
  => il faut regarder qui est la source pour différencier deux instances d'événements (*getSource()*)

Contrôle des Fenêtres

- WINDOWSLISTENER
  - Concerne tout ce qui est en rapport avec la fenêtre : Ouvrir, Fermer, Réduire, Agrandir, …

windowActivated*(WindowEvent e)* : fenêtre active par clic
windowDeactivated*(WindowEvent e)* : fenêtre désactivée

windowClosed*(WindowEvent e)* : fenêtre fermée
windowOpened*(WindowEvent e)* : fenêtre visible 1ere fois
windowClosing*(WindowEvent e)* : fermeture par menu fermé du système

windowDeiconified*(WindowEvent e)* : fenêtre restaurée
windowIconified(*WindowEvent e)* : fenêtre réduite

Contrôle de la souris

- MOUSELISTENER, MOUSEMOTIONLISTENER et MOUSEINPUTLISTENER (qui inclut les 2 premiers)
  - tout ce qui est en rapport avec la souris : Clics boutons souris et position/déplacements du curseur

- MOUSELISTENER permet de gérer :
  - Les différents états du clic souris : Clic, Pressé, Relâché
  - Et également l'entrée/sortie sur un composant
- MOUSEMOTIONLISTENER permet de gérer les mouvements de la souris sur un composant avec bouton appuyé ou relâché

MOUSELISTENER

- ***mouseClicked****(MouseEvent e)* : clic souris sur un composant (pressed'N'released)

- ***mouseEntered****(MouseEvent e)* : curseur de la souris rentre sur un composant
- ***mouseExited(****MouseEvent e)* : curseur de la souris sort d'un composant

- ***mousePressed****(MouseEvent e)* : bouton de la souris pressé sur un composant
- ***mouseReleased****(MouseEvent e)* : bouton relâché

MOUSEMOTIONLISTENER

- ***mouseDragged****(MouseEvent e)* : bouton de la souris appuyé sur un composant et la souris est déplacée

- ***mouseMoved(MouseEvent e)*** : souris déplacée sur un composant sans bouton enfoncé

MOUSEINPUTLISTENER

- interface écouteur de *javax.swing.event*.
  - Implémente
    - un *MouseListener* et un *MouseMotionListener*

ATTENTION : il n'existe pas de méthode *addMouseInputListener* il faut enregistrer l'écouteur deux fois :

- avec *addMouseListener*
- et avec *addMouseMotionListener*

Contrôle du clavier

- KeyListener, comme pour la souris, on va retrouver la notion de : Pressé/Relâché/Tapé
  - *keyPressed(KeyEvent e)* : touche du clavier pressée
  - *keyReleased(KeyEvent e)* : touche du clavier relâchée
  - *keyTyped(KeyEvent e)* : touche du clavier tapée

- FocusListener permet de gérer le *focus* et donc de savoir si un composant a obtenu le *focus* ou s'il la perdu
  - *focusGained(FocusEvent e)* :

  un composant obtient le focus
  - *focusLost(FocusEvent e)* : un composant perd le focus

Contrôle des composants

- COMPONENTLISTENER permet de gérer :
    - L'apparition/disparition d'un composant
    - Le déplacement d'un composant
    - Le redimensionnement d'un composant

componentHidden*(ComponentEvent e)* : le composant est rendu invisible
componentShown*(ComponentEvent e)* : le composant est rendu visible

componentMoved*(ComponentEvent e)* : la position du composant a changé
componentResized*(ComponentEvent e)* : les dimensions du composant ont changé

Autres contrôles

- Exemples d'écouteurs d'événements associés à des composants graphiques spécifiques
  - *ActionListener : void* ***actionPerformed(****ActionEvent e)*
  - *ItemListener : void* ***itemStateChanged(****ItemEvent e)*
  - *AdjustementListener : void* ***adjustementValueChanged*** *(AdjustementEvent e)*

Comment s'abonner ?

- Il faut ajouter un écouteur (objet implémentant une interface d'écoute) à la source à surveiller
  - on utilise la méthode *addXXXListener (XXXListener l)* sur le composant désiré

- Il suffit alors de remplacer les XXX parce que l'on souhaite avoir

Comment implémenter les écouteurs ?

- Il existe plusieurs méthodes
  - Définir des classes anonymes au niveau de l'abonnement au composant source
  - Définir des classes internes à la classe de vue
  - Implémenter une interface Listener :
    - revient à séparer le code comportemental de l'IHM du code structurel …
    - … sauf si c'est la classe de vue elle-même qui implémente l'interface

Définir une classe anonyme

```
 // lignes de code dans le constructeur de la vue
button = new JButton("test");
button.addMouseListener(new MouseAdapter() {
        @Override
        public void mouseClicked (MouseEvent e) {
            // code que l'on souhaite effectuer
        }
   });
```

Définir une classe interne

```
// ligne de code du constructeur de la vue
window.addWindowListener(new WindowHandler());

// classe interne WindowHandler
// pour les événements de fermeture
class WindowHandler extends WindowAdapter {
    public void windowClosing( WindowEvent e ) {
        window.dispose();
        System.exit(0);
    }
}
```

Implémenter une interface

```
public class MaClass extends MouseListener {
    // dans le constructeur
    maVue.addMouseListener(this);

    void mouseClicked(MouseEvent e)  {}
    void mouseEntered(MouseEvent e)  {}
    void mouseExited(MouseEvent e)  {}
    void mousePressed(MouseEvent e)  {}
    void mouseReleased(MouseEvent e)  {}
}
```

Attention il faudra connaitre l'objet maVue !
maVue peut aussi valoir "this" (la vue écoute elle-même les événements émis par ses composants)

Quid de l'utilisation

- Il faut implémenter toutes les méthodes de l'interface, y compris celles dont on ne se sert pas !
- Alternative pour les paresseux : Utiliser les classes Adapter
  - Elles implémentent une interface (ici écouteur) mais certaines méthodes n'ont pas de code
  - Etendre la classe *Adapter* choisie en redéfinissant uniquement les méthodes que l'on souhaite utiliser

ListenerInterface	AdapterClass	Methods
ActionListener	none	actionPerformed
AdjustmentListener	none	adjustmentValueChanged
ComponentListener	ComponentAdapter	componentHidden, componentMoved, componentResized, componentShown
ContainerListener	ContainerAdapter	componentAdded, componentRemoved
FocusListener	FocusAdapter	focusGained, focusLost
ItemListener	none	itemStateChanged
KeyListener	KeyAdapter	keyPressed, keyReleased, keyTyped
MouseListener	MouseAdapter	mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased
MouseMotionListener	MouseMotionAdapter	mouseDragged, mouseMoved
TextListener	none	textValueChanged
WindowListener	WindowAdapter	windowActivated, windowClosed, windowClosing, windowDeactivated, windowDeiconified, windowIconified, windowOpened

Etendre une classe Adapter

```
public class MaClass extends MouseAdapter {
    ...
    unObject.addMouseListener(this);
    ...
    public void mouseClicked(MouseEvent e) {
        ...
        // l'implementation de la méthode
        // associée à l'événement vient ici ...
    }
}
```

CREATION D'UN PANNEAU

```
class ButtonPanel extends JPanel
   implements ActionListener    // interface écouteur d'événements  {
   private JButton Boutonjaune;
   private JButton BoutonBleu;
   private JButton BoutonRouge;

PLACER DES COMPOSANTS DANS LE PANNEAU
public ButtonPanel() // constructeur de la classe ButtonPanel  {
      Boutonjaune = new JButton("Jaune");
      BoutonBleu = new JButton("Bleu");
      BoutonRouge = new JButton("Rouge");
      // Insertion des trois boutons dans l'objet ButtonPanel
      add(Boutonjaune);
      add(BoutonBleu);
      add(BoutonRouge);
ASSOCIER DES EVENEMENTS AUX COMPOSANTS
      // Les sources d'événements sont déclarées à l'écouteur
      Boutonjaune.addActionListener(this);
      BoutonBleu.addActionListener(this);
      BoutonRouge.addActionListener(this);
   }
```

TRAITEMENT DES EVENEMENTS

```
public void actionPerformed(ActionEvent evt)  {
   // Permet de traiter l'événement en fonction de l'objet source
      Object source = evt.getSource();
      Color color = getBackground();
      if (source == Boutonjaune) color = Color.yellow;
      else if (source == BoutonBleu) color = Color.blue;
      else if (source == BoutonRouge) color = Color.red;
      setBackground(color);
      repaint();
}
```

CREATION DE LA FENETRE ET PLACEMENT DU PANNEAU

```
class ButtonFrame extends JFrame  {
   public ButtonFrame()  {
      setTitle("ButtonTest");
      setSize(300, 200);
      addWindowListener(new WindowAdapter()  {
         public void windowClosing(WindowEvent e)  {
            System.exit(0);
         }
      } );
      Container contentPane = getContentPane();
      contentPane.add(new ButtonPanel());
   }
}

public class ButtonTest  {
  public static void main(String[] args)  {
      JFrame frame = new ButtonFrame();
      frame.show();
   }
 }
```

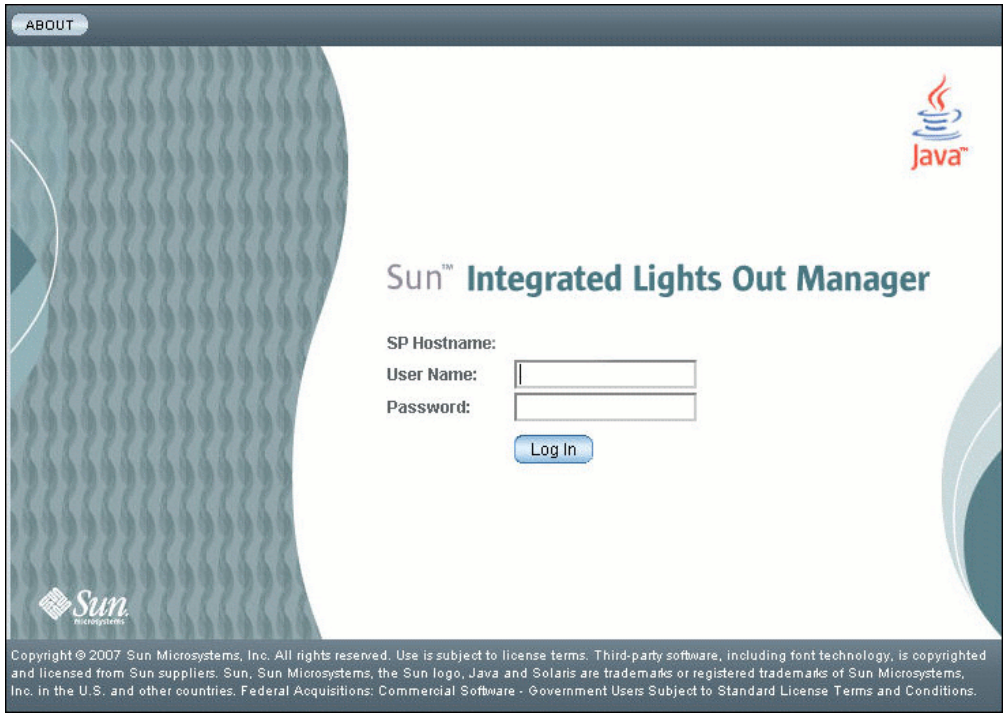
```
public class ResizeImg implements ComponentListener {
  private JLabel label;
  private ImageIcon image;
  public ResizeImg() throws MalformedURLException {
    JFrame frame = new JFrame("Exemple");
    URL url = new URL("http://www.google.com/intl/en_ALL/images/logo.gif");
    image = new ImageIcon(url); label = new JLabel(image);
    frame.getContentPane().add(label);
    frame.getContentPane().addComponentListener(this);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
  }
  public void componentHidden(ComponentEvent arg0) {}
  public void componentMoved(ComponentEvent arg0) {}
  public void componentShown(ComponentEvent arg0) {}
  public void componentResized(ComponentEvent arg0) {
    Component c = arg0.getComponent();
    Image img = image.getImage() ;
    Image newi = img.getScaledInstance(c.getWidth(), c.getHeight(),
                                       java.awt.Image.SCALE_SMOOTH);
    image.setImage(newi);
 }
  public static void main(String[] args) throws MalformedURLException {
    new ResizeImg();
  }
}
```

```
public class IllustrationDrag implements MouseInputListener {
    private Point p ; private Color c;
    public IllustrationDrag() {
         JFrame f = new JFrame(“D & D");
         f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         ImageIcon image = new ImageIcon("my_work.png");
         JLabel lbl = new JLabel(image);
         lbl.setOpaque(true); lbl.setSize(lbl.getPreferredSize());
         lbl.addMouseListener(this); lbl.addMouseMotionListener(this);
         JPanel p = new JPanel(null); p.add(lbl);
         f.setContentPane(p); f.setSize(300,300); f.setVisible(true);
    }
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {
         Component source = (Component) e.getSource();
         c = source.getBackground(); source.setBackground(Color.yellow);
    }
    public void mouseExited(MouseEvent e) {
         Component source = (Component) e.getSource(); source.setBackground(c);
    }
    public void mousePressed(MouseEvent e) { p = e.getPoint(); }
    public void mouseReleased(MouseEvent e) {}
    public void mouseDragged(MouseEvent e) {
         Point newP = e.getPoint();
         Component source = (Component) e.getSource();
         source.setLocation(source.getX()+newP.x-p.x, source.getY()+newP.y-p.y);
    }
    public void mouseMoved(MouseEvent e) {}
    public static void main(String[] args) {
         IllustrationDrag dd = new IllustrationDrag();
    }
}
```

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

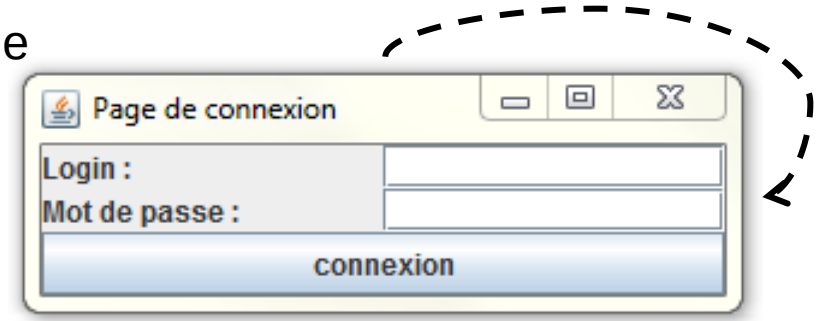
Actions : tentative de connexion + affichage d’une autre fenêtre avec message de connexion ou d’erreur après click sur le bouton

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

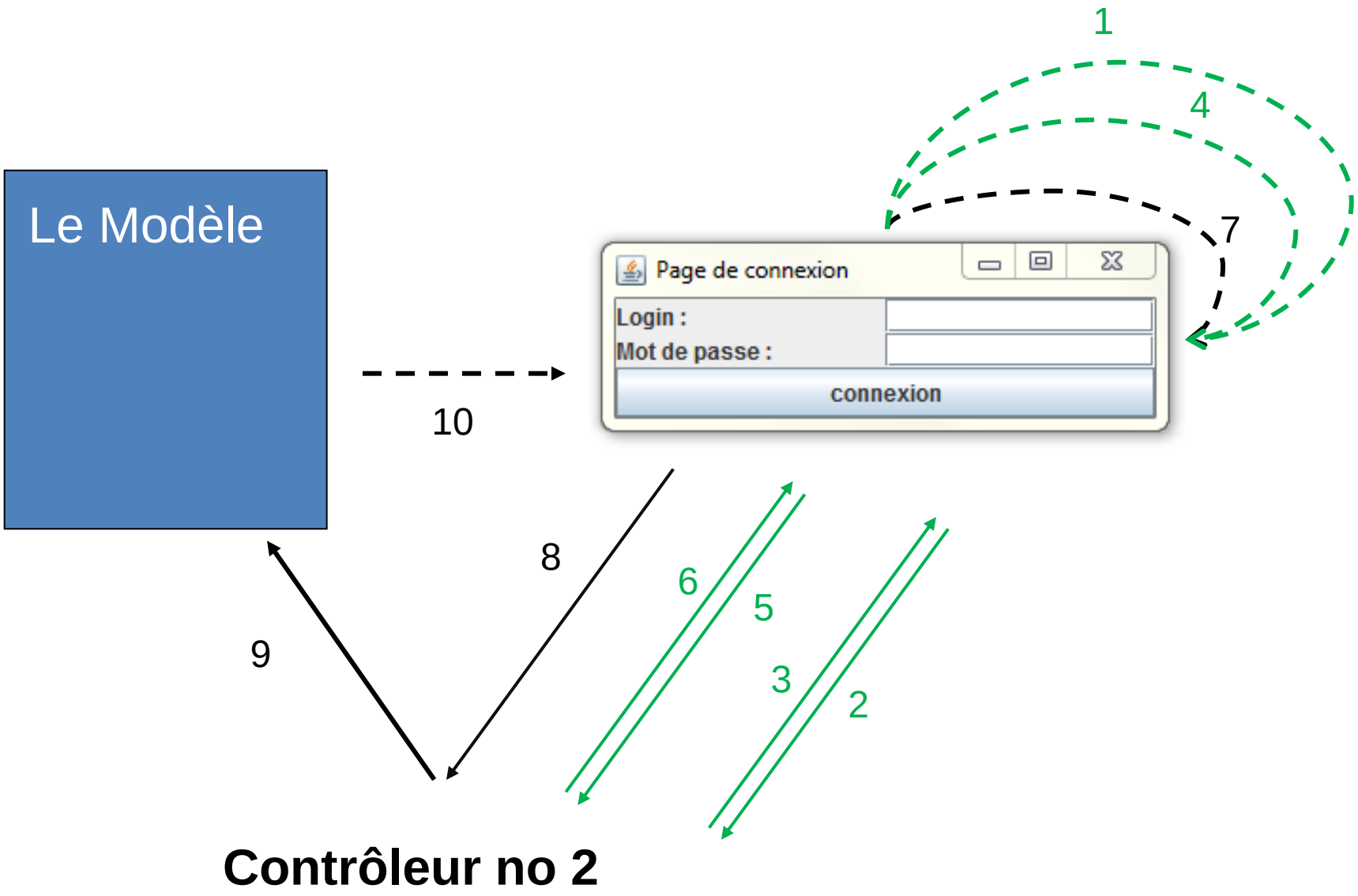
Cas d’interaction

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

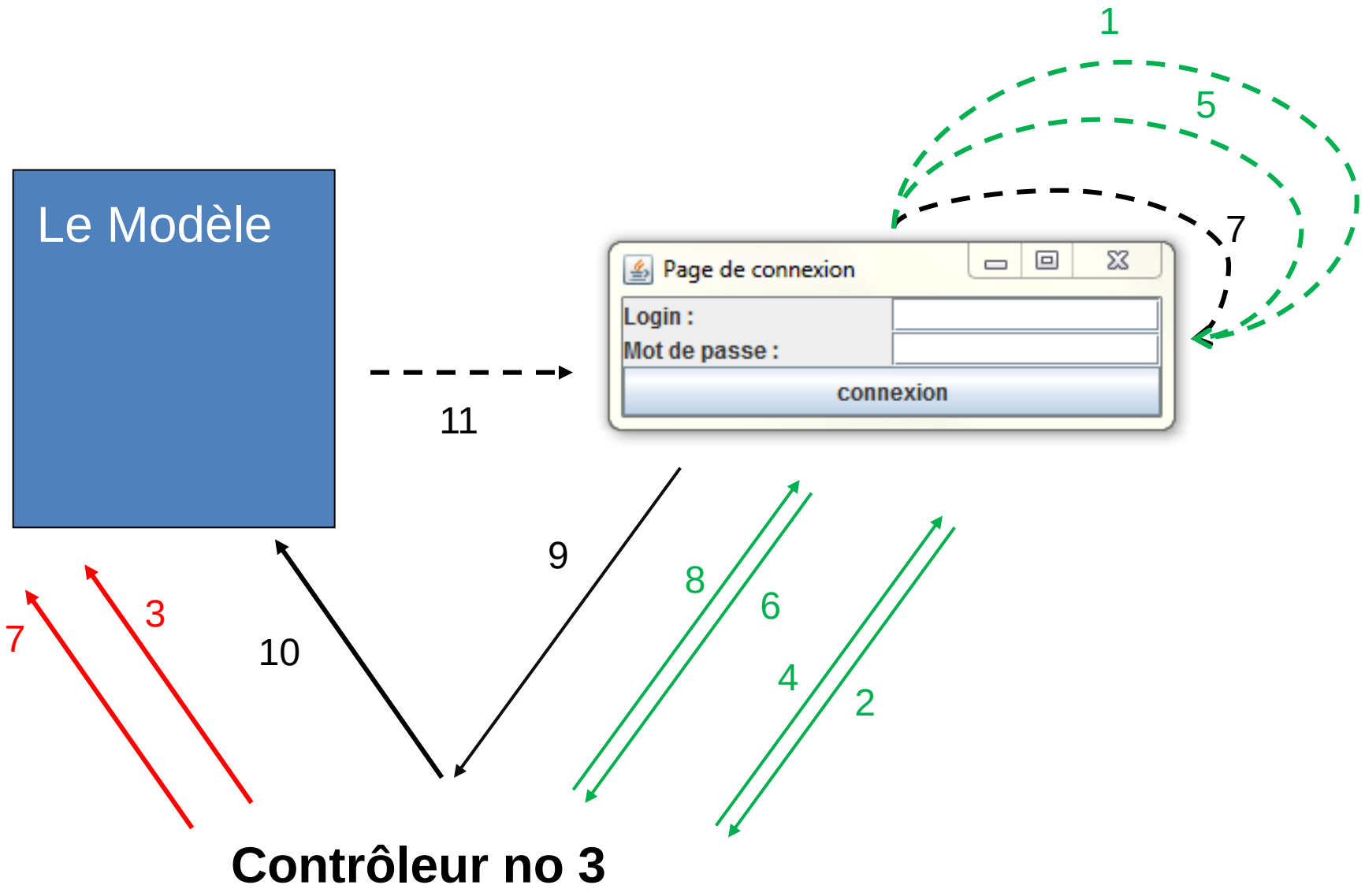
Cas no 3 :

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

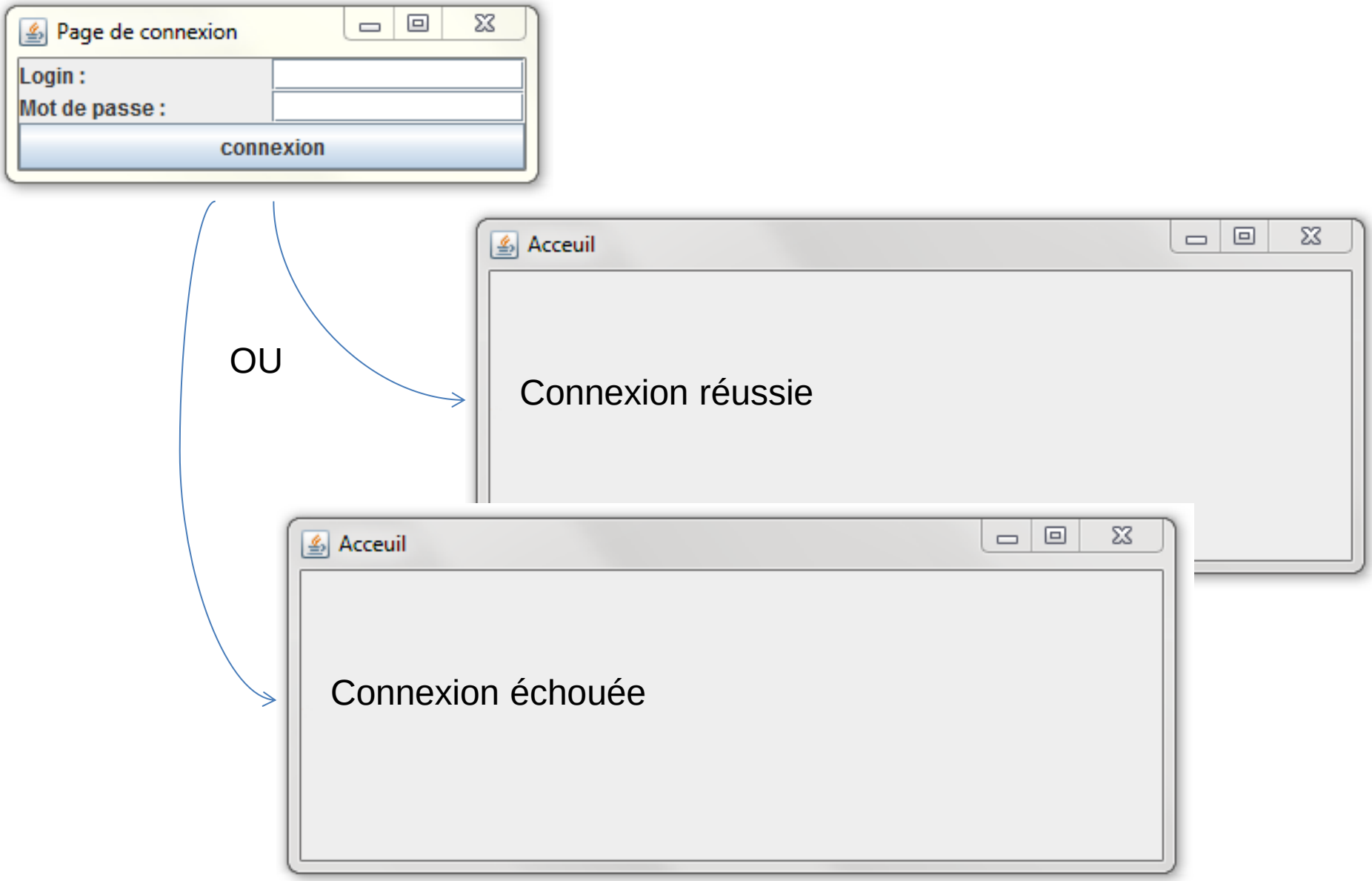
Actions : vérification des données faite au fur et à mesure des retours chariots, ce qui peut entrainer un message d’erreur

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)