

# BUT Informatique 1<sup>ère</sup> année

## R2.02: Développement d'applications avec IHM

Cours 2 : Programmation événementielle  
et MVC, autres composants graphiques

Sébastien Lefèvre  
[sebastien.lefevre@univ-ubs.fr](mailto:sebastien.lefevre@univ-ubs.fr)

# : Programmation événementielle

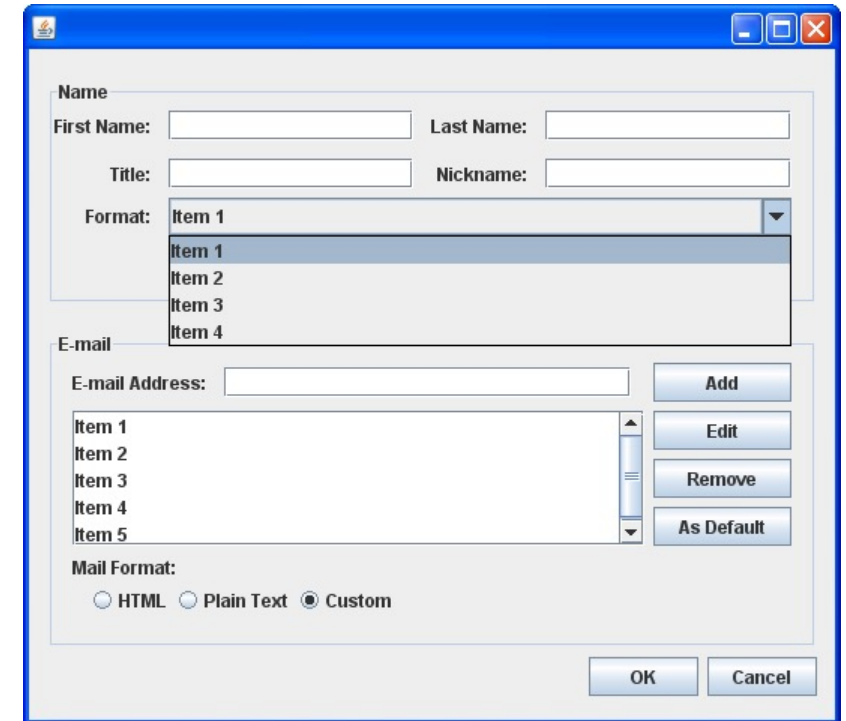
# Un programme graphique

Un programme = une suite d'instructions...

Le comportement d'une IHM dépend de ce que veut l'utilisateur...

Mais on ne peut pas anticiper les actions de l'utilisateur !

Comment faire ?



The image shows a screenshot of a graphical user interface (GUI) window, likely for user registration or profile management. The window has a blue title bar with standard Windows window controls (minimize, maximize, close). The main content area is divided into several sections:

- Name:** This section contains four text input fields: "First Name:", "Last Name:", "Title:", and "Nickname:". Below these is a "Format:" dropdown menu currently showing "Item 1". A list box is open below the dropdown, showing "Item 1", "Item 2", "Item 3", and "Item 4".
- E-mail:** This section contains an "E-mail Address:" text input field. To its right are four buttons: "Add", "Edit", "Remove", and "As Default". Below the input field is a list box containing "Item 1", "Item 2", "Item 3", "Item 4", and "Item 5".
- Mail Format:** This section contains three radio buttons: "HTML", "Plain Text", and "Custom". The "Custom" radio button is selected.

At the bottom right of the window are two buttons: "OK" and "Cancel".

# Un programme graphique

Une IHM s'appuie sur le paradigme de la programmation événementielle :

- On va anticiper toutes les actions possibles de l'utilisateur
- On décrit (par du code) les réactions que le programme doit avoir à ces différentes actions

# Programmation événementielle

L'action de l'utilisateur sur l'interface graphique provoque un événement (event)

Si l'interface graphique (listener) est à l'écoute de cet événement

Alors le programme peut réagir par l'intermédiaire d'un module de réaction

# Programmation événementielle

3 éléments nécessaires :

- Un événement XXXEvent
- Un widget à l'écoute de XXXEvent
- Une méthode de réaction à XXXEvent

```
import java.awt.*; import javax.swing.*;

public class Beeper extends JPanel implements ActionListener {

    JButton button;

    public Beeper() {

        super(new BorderLayout());

        button = new JButton("Click Me");

        button.setPreferredSize(new Dimension(200, 80));

        add(button, BorderLayout.CENTER);

        button.addActionListener(this);

    }

    public void actionPerformed(ActionEvent e) {

        Toolkit.getDefaultToolkit().beep();

    }

    private static void createAndShowGUI() {

        //Create and set up the window.

        JFrame frame = new JFrame("Beeper");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JComponent newContentPane = new Beeper();

        newContentPane.setOpaque(true); //content panes must be opaque
```

```
        frame.setContentPane(newContentPane);

        frame.pack();

        frame.setVisible(true);

    }

    public static void main(String[] args) {

        javax.swing.SwingUtilities.invokeLater(new Runnable() {

            public void run() {

                createAndShowGUI();

            }

        });

    }

}
```

démo

# : Les types d'événement



# java.awt.event

Tous les évènements qu'une IHM Java peut détecter se trouvent dans java.awt.event

- KeyEvent : clavier
- MouseEvent : souris
- WindowEvent : fenêtre
- ActionEvent : action
- etc

# Événements Java

Tous les événements précédents sont

- des classes Java
- qui héritent de `java.util.EventObject`

La classe `EventObject` fournit la méthode

- `Object getSource()`

qui permet de savoir sur quel composant l'utilisateur à agit

Cela permet une seule méthode de réaction pour un même type d'événement en provenance de plusieurs widgets différents.

# Question 1

Est-ce que tous les événements peuvent être capturés par tous les widgets ?

# Question 1

Est-ce que tous les événements peuvent être capturés par tous les widgets ?

Événements	Widgets
ActionEvent	JButton JTextField JMenu JMenuItem List
MouseEvent	(J)Component JList
KeyEvent	(J)Component
FocusEvent	(J)Component
InputMethodEvent	JTextComponent JTextField JTextArea
ItemEvent	JCheckBox Choice List
WindowEvent	JWindow

# Question 2

Que se passe-t'il lorsqu'un widget subit une action de l'utilisateur ?

## Question 2

Que se passe-t'il lorsqu'un widget subit une action de l'utilisateur ?

Réponse :

Un objet xxxEvent est automatiquement instancié et envoyé à tous les objets (de réaction) qui sont à l'écoute de xxxEvent



# Comment mettre un widget à l'écoute ?

Ce processus n'est pas automatique.

Si on veut qu'un widget réagisse à un événement autorisé, il faut lui attacher explicitement un objet écouteur (listener) de ce type d'événement (XXXListener)... sinon rien ne se passe.

Pour chaque widget concerné par XXXEvent, il existe une méthode `addXXXListener (XXXListener o)`

Le XXXListener va se mettre à l'écoute de l'événement et y réagir grâce à ses méthodes de réaction



```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Beeper extends JPanel {

    JButton button;

    public Beeper() {

        super(new BorderLayout());

        button = new JButton("Click Me");

        button.setPreferredSize(new Dimension(200, 80));

        add(button, BorderLayout.CENTER);

        button.addActionListener(new Ecouteur());

    }

    private static void createAndShowGUI() {

        //Create and set up the window.

        JFrame frame = new JFrame("Beeper");

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Create and set up the content pane.

        JComponent newContentPane = new Beeper();

        newContentPane.setOpaque(true); //content panes must be opaque

        frame.setContentPane(newContentPane);

        //Display the window.

        frame.pack();

        frame.setVisible(true);

    }

    public static void main(String[] args) {

        //Schedule a job for the event-dispatching thread:

        //creating and showing this application's GUI.

        javax.swing.SwingUtilities.invokeLater(new Runnable() {

            public void run() { createAndShowGUI(); });

        }

    }
}

```

## classe Ecouteur ? de type ActionListener

```

public class Ecouteur implements ActionListener {

    public Ecouteur() { }

    public void actionPerformed(ActionEvent e) {

        Toolkit.getDefaultToolkit().beep();

    }

}

```

démo

# : Les réactions

# Interfaces pour les réactions

`addXXXListener (objetXXXListener);`

Le type XXXListener est une interface

➔ Il faut donc créer une nouvelle classe qui implémente l'interface XXXListener

ActionListener : 1 méthode

MouseListener : 5 méthodes

➔ Comment faire ?

# MouseListener

Il faut implémenter les 5 méthodes :

- mouseClicked
- mouseEntered
- mouseExited
- mousePressed
- mouseReleased

Même si on n'a pas besoin de gérer tous ces événements !

# Exercice

Comment faire pour que l'écouteur puisse modifier l'interface ?

Par exemple : le clic sur le bouton change l'apparence du bouton  
(+1)

# Solution

L'écouteur possède parmi ses attributs une référence vers le bouton.

Cet attribut est initialisé lors de l'appel au constructeur :  
Ecouteur(b)

démo

# Comment éviter les multiples implémentations vides ?

Si on n'a besoin de spécifier le comportement que pour certains des événements définis dans une interface XXXListener, comment faire ?

Passer par des classes abstraites XXXAdapter qui implémentent XXXListener :

- Le XXXAdapter offre une implémentation de toutes les méthodes définies par l'interface
- On se limite à une surcharge des méthodes intéressantes
- Attention, Adapter est une classe  
➔ ici, Ecouteur extends XXXAdapter au lieu de Ecouter implements XXXListener

# Conclusion

On veut créer la réaction à un événement agissant sur un widget :

1. Quels sont les événements que peut capturer mon widget ?  
cf tableau ou Javadoc
2. Créer un objet à l'écoute de cet événement (objetEcouleur), c'est-à-dire une instance de la classe qui soit implémente toutes les méthodes de l'interface XXXListener, soit surcharge les méthodes de la classe abstraite XXXAdapter
3. Attacher l'objet (objetEcouleur) au widget à l'aide de la méthode addXXXListener(objetEcouleur)
4. Le code qui réalise la réaction à l'événement XXXEvent doit être écrit à l'intérieur d'une ou plusieurs méthodes de la classe qui implémente XXXListener / hérite de XXXAdapter



Événements	Ecouteurs	Méthodes	Widgets
ActionEvent	ActionListener	actionPerformed	JButton JTextField JMenu JMenuItem List
MouseEvent	MouseListener	mouseClicked mouseEntered mouseExited mousePressed mouseReleased	(J)Component JList
KeyEvent	KeyListener	keyPressed keyTyped keyReleased	(J)Component
FocusEvent	FocusListener	focusGained focusLost	(J)Component
InputMethodEvent	InputMethodListener	inputMethodTextChanged	JTextComponent JTextField JTextArea
ItemEvent	ItemListener	itemStateChanged	JCheckBox Choice List
WindowEvent	WindowListener	windowActivated windowClosing	JWindow

# Les différentes formes de classes d'écouteurs

# Solution 1

Les évènements sont gérés dans des classes d'écouteur dédiées



**++ bon découpage du code (modèle MVC)**

- il faut passer au constructeur de la classe externe une référence sur la classe IG
- il faut écrire beaucoup d'accesseurs dans la classe IG pour permettre l'accès à ses composants
- chaque écouteur d'évènements sur un widget nécessite l'écriture d'une nouvelle classe (beaucoup de classes, de fichiers)

**MAIS**

**Cela reste la meilleure approche en terme de conception !**

**➔ Il faut néanmoins essayer de limiter le nombre de classes de réaction**

# Solution 2

La classe d'IG est également un écouteur

++ pas d'accessor à écrire

- la classe d'IG contient beaucoup de code
- toutes les réactions sur tous les widgets sont dans la classe d'IG
- la classe d'IG doit implémenter toutes les interfaces de réaction
- l'utilisation de multiples Adapter n'est pas possible (pas d'héritage multiple en Java)

## ET SURTOUT

-- Mélange des aspects graphiques (vue) et réaction (contrôleur)

➔ A EVITER !



# Solution 3

La classe d'écouteur est définie comme une classe interne de l'IG  
Classe interne = classe définie dans une autre classe



- ++ pas d'accessor à écrire (accessibilité de droit entre les 2 classes)
- ++ les classes internes peuvent hériter des Adapter

- toutes les réactions sur tous les widgets sont dans le fichier définissant la classe d'IG

**ET SURTOUT**

- **Mélange des aspects graphiques (vue) et réaction (contrôleur), dans le même fichier !**

**➔ A EVITER !**

# Solution 4

La classe d'écouteur est définie comme une classe interne **anonyme** de l'IG  
Classe anonyme = classe sans nom définie dans une autre classe



++ classe anonyme = classe interne (donc mêmes avantages que solution 3)

## MAIS

- Mélange des aspects graphiques (vue) et réaction (contrôleur), dans le même fichier !
- Code illisible !!!
- ➔ A EVITER !

## Alors pourquoi ?

Solution très souvent utilisée dans les IDE pour faire de la génération automatique de code

# Conclusion



Privilégier la solution 1

➔ Il faut néanmoins essayer de limiter le nombre de classes de réaction

Comment faire ? **Mutualiser !**

- Une même classe pour écouter les évènements sur plusieurs widgets
- Un même objet

➔ accès au widget à l'origine de l'événement par la méthode *Object getSource()* de la classe d'événement

Exemple :

une application compteur avec un bouton + et un bouton RAZ

# : Le patron de conception MVC



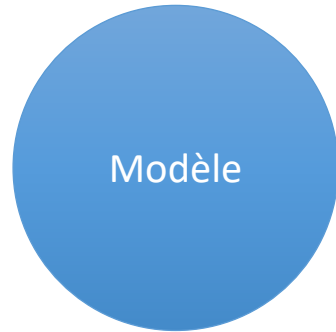
# MVC : historique

L'architecture MVC (Model-View-Controller) a pour objectif d'organiser une application interactive en séparant clairement :

- un module chargé exclusivement de la gestion des données (Modèle)
- un module chargé exclusivement de la représentation des données (l'interface graphique, Vue)
- un module chargé exclusivement de traiter les actions/réactions avec l'utilisateur (Contrôleur)

Idée de la société Rank Xerox en 1970 appliquée au GUI (Graphical User Interface) dès 1980.

# MVC : organisation



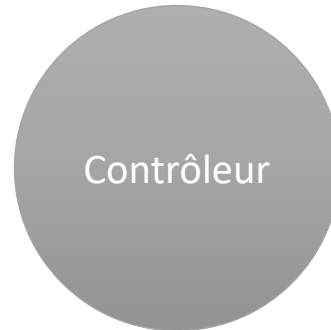
Modèle (base de données)

- Structure de données
- Opérations sur les données



Vue (présentation)

- Interface graphique
- Composée des widgets



Contrôleur (chef d'orchestre)

- Gère l'interaction avec l'utilisateur
- Interroge le modèle
- Change la vue

# MVC : avantages

- Amène un bon découpage du code en
  - Structure orientée objet
  - 3 modules (presque) indépendants
- Facilite la maintenance : la modification d'un module n'influence pas les autres
- Le module Vue qui demande le plus de travail peut avoir plusieurs versions sans perturber les autres modules
- Facilite le développement en équipe
- Concentration accrue sur l'ergonomie

# MVC : organisation en Java

1 module = 1 paquetage

- Package control
- Package view
- Package data (ou model)

# View

Contient TOUTES les classes Java qui mettent en place le décor (GUI)

Mais ne contient PAS :

- Les classes de réaction car elles sont à l'écoute des évènements utilisateur (partie contrôle)
- Les classes qui mémorisent et agissent sur les données (partie modèle)

# Model

Contient TOUTES les classes Java qui agissent sur les données :

- Ajout, suppression, modification
- Recherche
- Ecriture/lecture pour la persistance

Ne contient AUCUNE déclaration ou importation de classes du package view

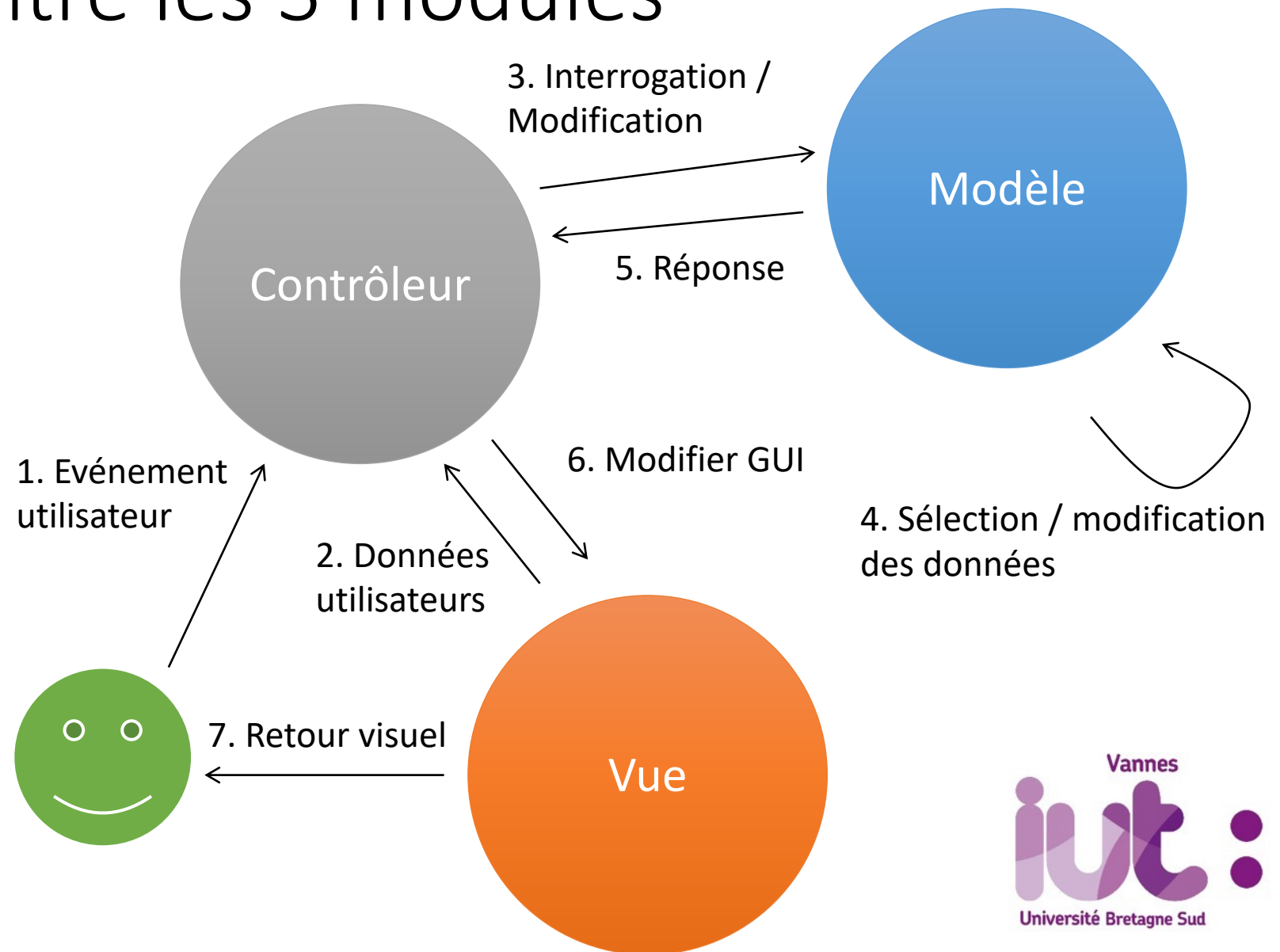
# Control

Contient TOUTES les classes Java qui sont à l'écoute des événements utilisateur (= les classes de réaction aux événements)

En réaction aux événements :

- Elles agissent sur les données
- Elles modifient l'IG (interaction utilisateur)

# Dialogue entre les 3 modules





: Encore un peu de Swing?

# Lancer une application graphique

```
public static void main(String[] args) {  
    javax.swing.SwingUtilities.invokeLater(  
        new Runnable() {  
            public void run() {  
                new InterfaceGraphique();  
            }  
        }  
    );  
}
```

ou juste

```
public static void main(String[] args) {  
    new InterfaceGraphique();  
}
```

Exemple

# D'autres gestionnaires de placement

- CardLayout
- BoxLayout
- GridBagLayout
- ...

Exemple

# Menus

- Barre de menu : JMenuBar  
(méthode setJMenuBar dans une fenêtre)

- Menu : JMenu  
(à ajouter à la barre)

- Item : JMenuItem  
(à ajouter aux menus)

Réactions : ActionEvent

- getSource()
- getActionCommand()

Exemple

# Fenêtres de dialogue prédéfinies

## JOptionPane

- showMessageDialog
- showConfirmDialog
- showInputDialog
- showOptionDialog

## JFileChooser

Exemple