

Interfaces Graphiques

Interactivité : événements et actions

Jean-Baptiste.Yunes@univ-paris-diderot.fr

Université Paris Diderot

©2015

- Les composants Swing permettent (en général) à l'utilisateur d'agir sur l'application
- pour cela ils effectuent une série de **gestes** depuis les périphériques, lesquels sont interprétés comme **actions** logiques
 - **gestes** : événements bruts, bas-niveau
 - **actions** : événements logiques, sémantiques



Extrait du tutoriel Swing

- Une sollicitation depuis un composant (*source*)
 - provoque la création d'un événement (*event*)
 - lequel est distribué aux parties fonctionnelles de traitement (*listener*)
- découplage entre interface et métier

- Ceci nécessite qu'un `Listener` s'enregistre auprès d'une source
- mécanisme de rappel (*callback*)
 - « Tiens voilà mon numéro de téléphone, comme ça tu pourras me prévenir plus tard »
- Attention : lors du rappel, soyez rapide!
 - ne faites pas traîner la conversation car le rappel est effectué dans le même `Thread` que celui qui gère les événements
 - sinon `SwingWorker` (voir plus tard...)

- Chaque composant Swing peut générer des événements
- deux catégories :
 - les événements génériques aux composants Swing
 - les événements spécifiques (relatifs à la sémantique de l'objet graphique)

- Les événements génériques (principaux) :
 - `ComponentEvent` (taille, position, visibilité)
 - `FocusEvent` (*capture* le clavier ou non)
 - `KeyEvent` (frappe clavier)
 - `MouseEvent` (action souris)
 - `MouseEvent` (action souris)

- JButton, JMenuItem, JToggleButton, JRadioButton, JTextField, JPasswordField, JComboBox, JCheckBox, JFormattedTextField
- `ActionEvent`
 - représente l'action logique (clic sur le bouton, choix de l'option, saisie du champ, etc)
 - attention : action logique
 - *i.e.* : le clic peut-être obtenu par un raccourci par exemple...

- pour qu'un objet reçoive un `ActionEvent`
- il faut qu'il implémente l'interface
 - `ActionListener`
 - `public void`
`actionPerformed(ActionEvent)`
 - méthode appelée si l'objet est enregistré auprès d'une source possible
 - `addActionListener(ActionListener)`
 - et lorsqu'on déclenche par actions physiques l'action logique (clic sur un bouton par ex.)


```
public class MaClasseDEcouteur implements ActionListener {
...
    public void actionPerformed(ActionEvent e) {
        // code exécuté si l'instance est enregistrée
        // et si l'utilisateur déclenche l'Action
    }
...
}

JButton monComposant = new JButton("un bouton");
...
instanceDeMaClasseEcouteur = new MaClasseDEcouteur();
// enregistrement d'un écouteur sur un composant
// susceptible de générer une Action
monComposant.addActionListener(instanceDeMaClasseDEcouteur);
...
```



- puisqu'un `Listener` peut recevoir un événement provenant de différentes Sources
- comment distinguer la source ?
 - la classe `EventObject` (super-classe des événements) fournit la méthode
 - `Object getSource()`

```
public class MaClasseDEcouteur implements ActionListener {
...
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == monComposant) { ... }
        if (e.getSource() == monAutreComposant) { ... }
    }
...
}

...
instanceDeMaClasseEcouteur = new MaClasseDEcouteur();
// enregistrement d'un écouteur sur un composant
// susceptible de générer une Action
monComposant.addActionListener(instanceDeMaClasseDEcouteur);
monAutreComposant.addActionListener(instanceDeMaClasseDEcouteur);
...
}
```

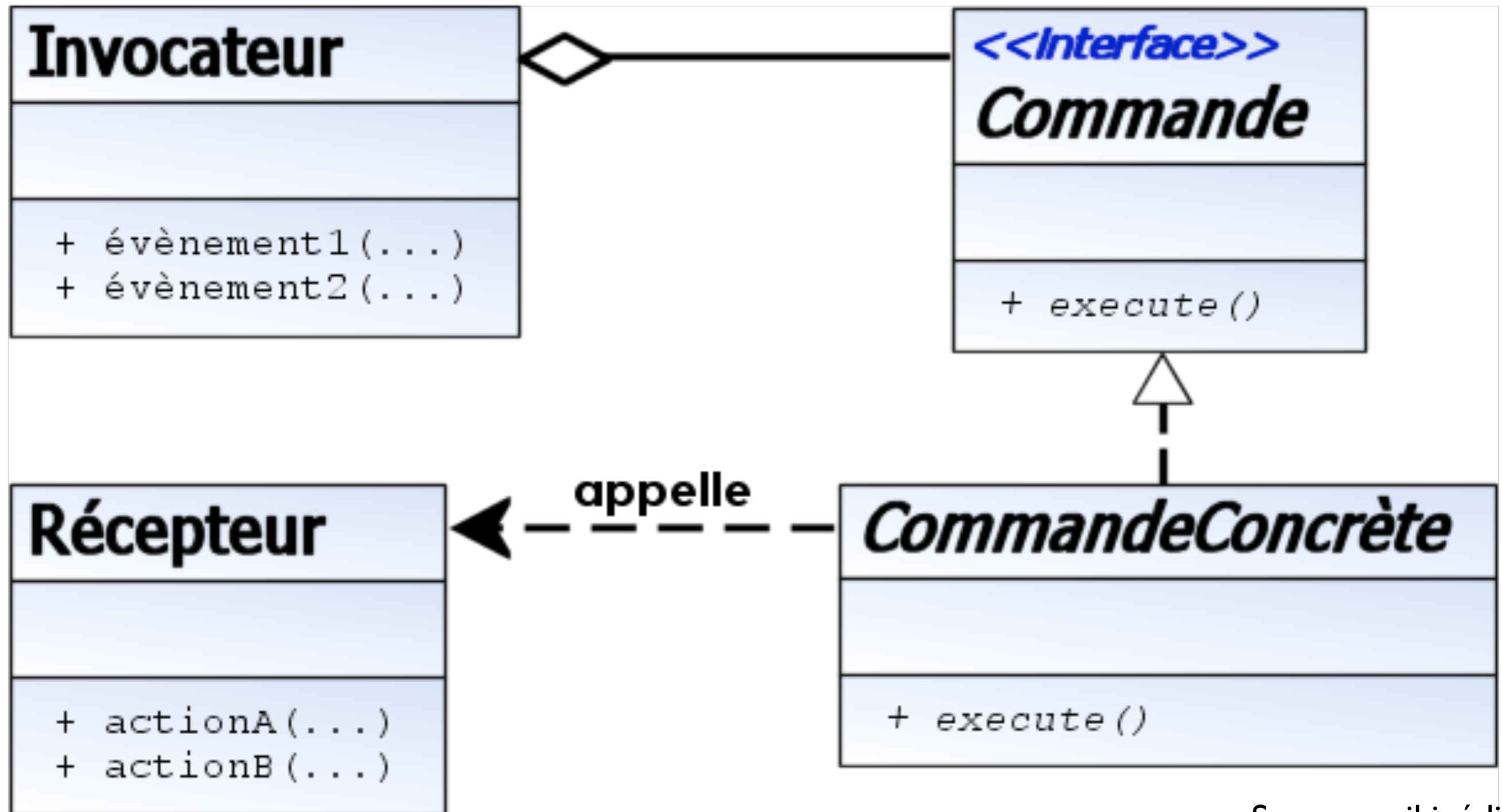
- utiliser une **commande associée** :
- une chaîne de caractère portée par la Source et qui permet au `Listener` de réaliser des variantes de traitement
 - avantages
 - c'est la Source qui décide quelle variante elle veut obtenir
 - une Source peut facilement changer de rôle
- `String getActionCommand()`
- `setActionCommand(String)`

Source / Listener

```
public class MaClasseEcouleur implements ActionListener {
    private void save() { ... }
    private void quit() { ... }
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("save and quit")) {
            save();
            quit();
        }
        if (e.getActionCommand().equals("quit")) {
            quit();
        }
    }
}

...
instanceDeMaClasseEcouleur = new MaClasseEcouleur();
JButton b = new JButton("Quitter");
b.setActionCommand("quit");
b.addActionListener(instanceDeMaClasseEcouleur);
...
```

- si plusieurs composants servent à déclencher une même action
- comment les paramétrer facilement ?
 - réifier totalement la notion d'Action
 - interface `Action` / classe `AbstractAction`
 - c'est une implémentation du design pattern `Command`



Source : wikipédia

- une `Action` (sous-interface de `ActionListener`) doit répondre à différentes méthodes
 - `actionPerformed` évidemment
 - en particulier doit permettre d'y associer/ retrouver des valeurs associées à des clés
 - c'est une **mémoire associative**
 - les composants concernés peuvent y retrouver ce dont ils ont besoin (icône, texte, etc)

- Clés
 - ACCELERATOR_KEY
 - ACTION_COMMAND_KEY
 - DEFAULT
 - DISPLAYED_MNEMONIC_INDEX_KEY
 - LARGE_ICON_KEY
 - LONG_DESCRIPTION
 - MNEMONIC_KEY
 - NAME
 - SELECTED_KEY
 - SHORT_DESCRIPTION
 - SMALL_ICON

Source / Listener

```
public class Quitter extends AbstractAction {
    public Quitter() {
        putValue(Action.SHORT_DESCRIPTION,
            "Quitter l'application");
        putValue(Action.NAME, "Quitter");
    }
    public void actionPerformed(ActionEvent e) {
        // fait quelque chose pour quitter l'application
        System.exit(0);
    }
}
```

...

```
// instantiation d'une action pour quitter
Quitte q = new Quitte();
```

```
// un bouton qui occasionnera l'action de quitter
JButton b = new JButton(q);
```

...

- Il peut être utile de recevoir des événements en provenance du clavier
- interface `KeyListener`
- Enregistrement d'un tel écouteur sur un composant
 - `addKeyListener(KeyListener)`

- deux types d'événements :
 - la saisie d'un caractère (Unicode)
 - `keyTyped (KeyEvent)`
 - l'appui et le relâchement d'une touche
 - `keyPressed (KeyEvent)`
 - `keyReleased (KeyEvent)`

- KeyEvent
 - `char getKeyChar()` pour récupérer un caractère Unicode ou `CHAR_UNDEFINED`
 - `int getKeyCode()` pour récupérer la *touche clavier*
 - renvoie un code `VK_*`
 - `VK_A`, `VK_ENTER`, `VK_END`, `VK_F1`,
`VK_PAGE_DOWN`...

- Les événements provenant de la souris sont classés en 3 catégories :
 - les clics
 - les plus ordinaires
 - les déplacements
 - nécessitent une attention particulière car ils peuvent être très nombreux
 - la rotation de la roue
 - dont la gestion est particulière

- Les clics
- interface `MouseListener`
 - `addMouseListener(MouseListener)` de la classe `Component`
- 5 méthodes

- `mouseClicked(MouseEvent)`
- `mousePressed(MouseEvent)`
- `mouseReleased(MouseEvent)`
- `mouseEntered(MouseEvent)`
 - lorsque la souris entre dans l'espace du composant
- `mouseExited(MouseEvent)`
 - lorsque la souris sort de l'espace du composant

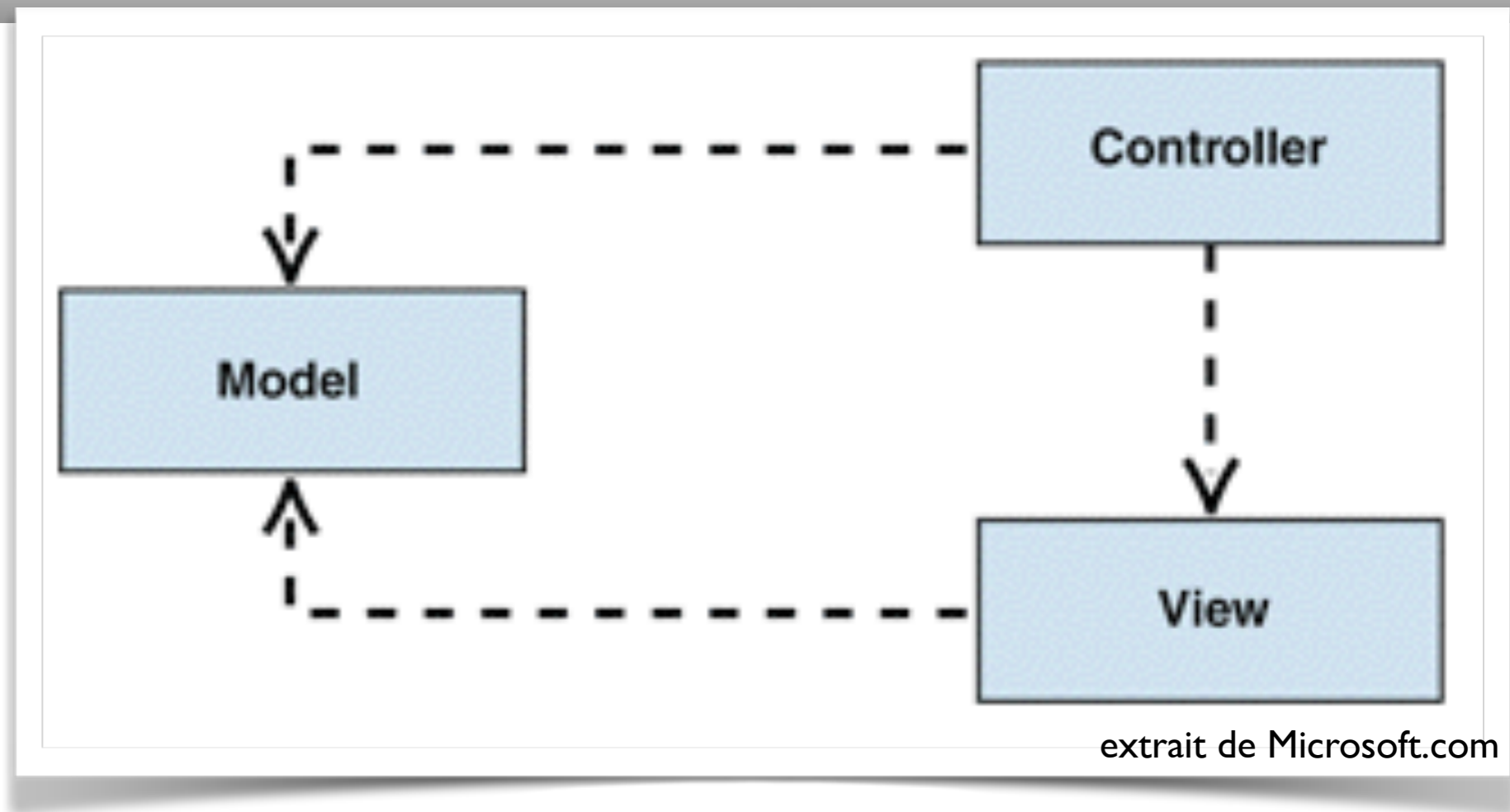
- `int getButton()`
 - `MouseEvent.BUTTON1`
 - `MouseEvent.BUTTON2`
 - `MouseEvent.BUTTON3`
- `int getClickCount()`
 - multi-clic
- `int getX(), int getY()`
 - guess what ?

- les déplacements de la souris
 - ne doivent être interceptés que lorsque c'est strictement utile
 - le nombre d'événements générés peut être important
 - appels très nombreux des callbacks...
- en général on attend l'appui sur un bouton pour démarrer la réception
- puis on arrête d'écouter en relâchant un bouton

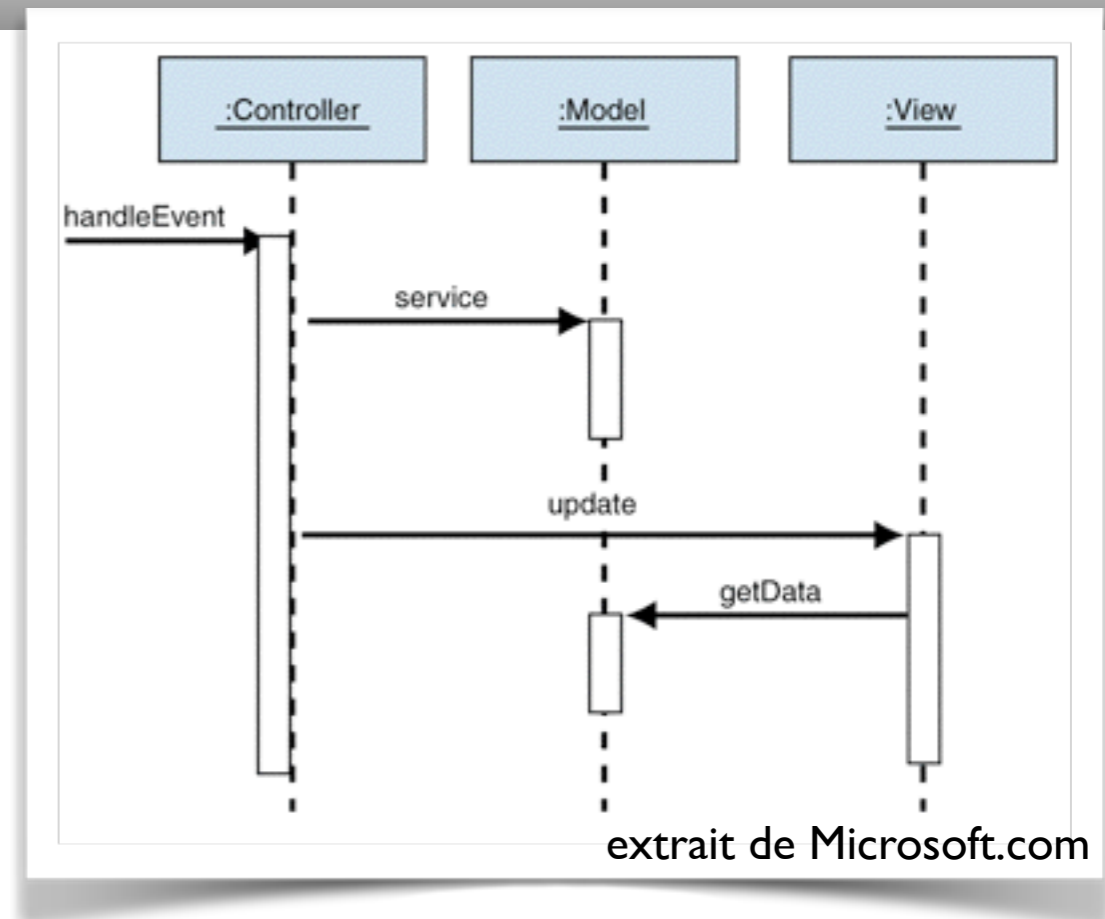
- les déplacements de la souris
 - interface `MouseMotionListener`
 - `addMouseMotionListener` dans les composants
 - avec bouton maintenu
 - `mouseDragged(MouseEvent)`
 - sans bouton maintenu
 - `mouseMoved(MouseEvent)`

- aucun moyen de savoir si une souris est équipée d'une roue
- interface `MouseWheelListener`
 - `addMouseWheelListener` des composants
- méthode `mouseWheelMoved(MouseWheelEvent)`
 - attention, événement particulier
`MouseWheelEvent`

- `int getScrollAmount()`
 - nombre d'unités par clic de roue
- `int getScrollType()`
 - deux types : `WHEEL_UNIT_SCROLL`,
`WHEEL_BLOCK_SCROLL`
- `int getUnitsToScroll()`
 - nombre d'unités totales à scroller
(`amount*rotation`)
- `int getWheelRotation()`
 - nombre de clics de la roue

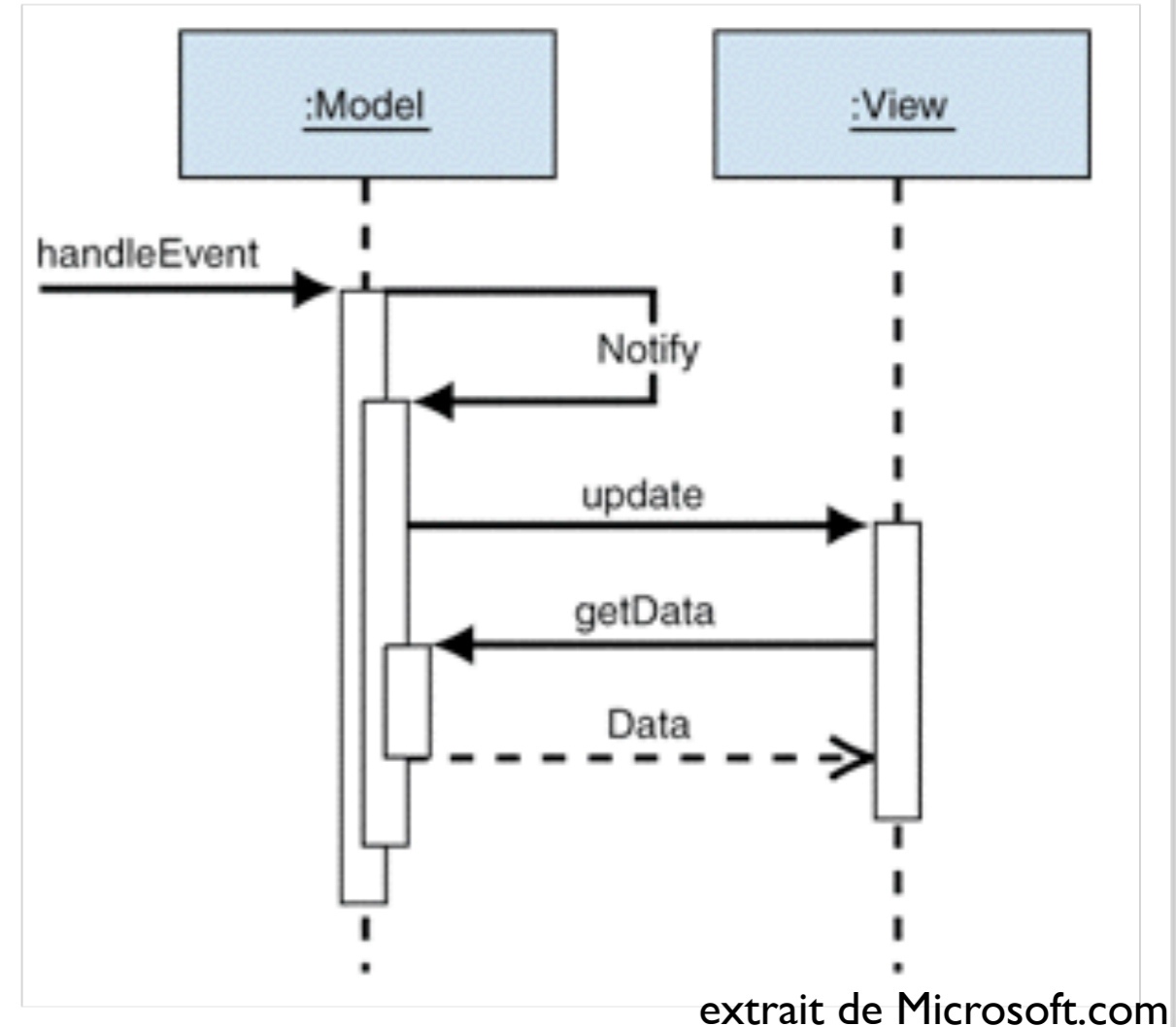
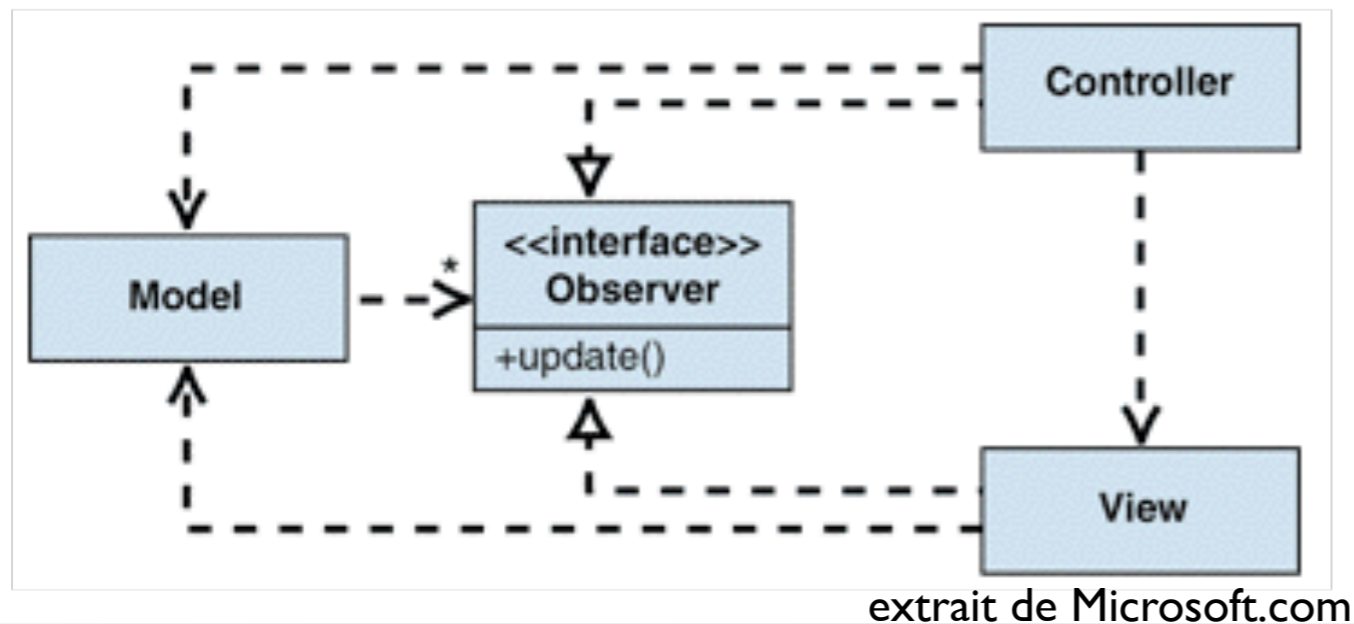


- Comment obtenir un bon découplage entre la partie métier et l'interface
 - le motif conceptuel MVC
 - Modèle / Vue / Contrôleur



- Le modèle est conçu indépendamment de l'interface
- Le contrôleur reçoit des événements et modifie en conséquence le modèle, puis prévient les vues de se remettre à jour
- La vue, sur réquisition du contrôleur, récupère les données intéressantes du modèle et les présente à l'interface

Le MVC actif



- Problème : le modèle peut se mettre à jour de lui-même ou par un autre biais
- On utilise alors le pattern Observer