Le comportement des interfaces

- Gestion des interactions avec l'utilisateur : souris, clavier, ...
- Gestion des changements d'états des composants ou de valeur des données affichées
- Il doit être possible de répondre à ces événements, qu'ils soient déclenchés par l'utilisateur ou par l'application

- Java propose la gestion des événements à travers un modèle *Evénement/Ecouteur*
  - à chaque type d'événement susceptible de requérir un traitement est associé un objet événement, créé quand l'événement survient
  - des objets écouteurs qui, quand l'événement survient, déclenche un traitement adapté
  - les objets qui créé des événements sont les sources d'événements

Les événements en Java

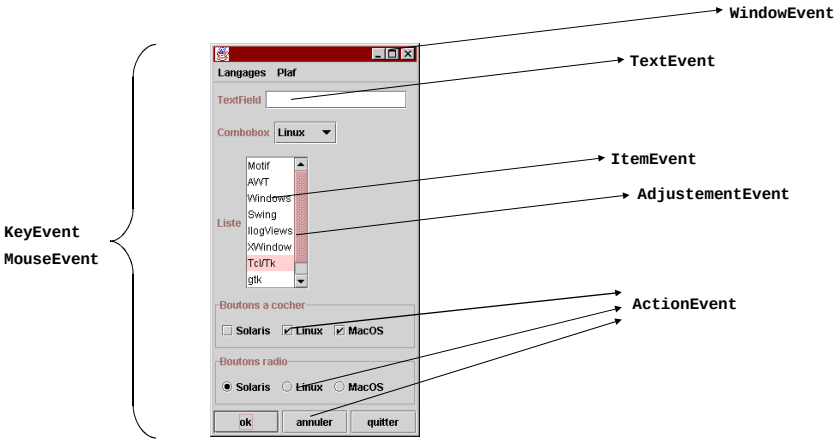
- Le comportement des composants (et potentiellement de n'importe quel objet Java!) est géré par l'émission et le traitement d'événements (class **`EventObject`**), créés par les composants suite à une action

- Un événement est un objet qui encapsule des informations sur la source de l'événement et sa nature

- *Exemple : un événement clavier décrit*
  - *la touche concernée*
  - *la nature de l'événement (appui, appui+relachement, relachement)*
  - *le composant actif lors de l'événement (source)*
  - *le fait que des touches de modifications (SHIFT, ALT, CTRL) soient appuyées ou pas au moment de l'événement.*

Evénements AWT (1/3)

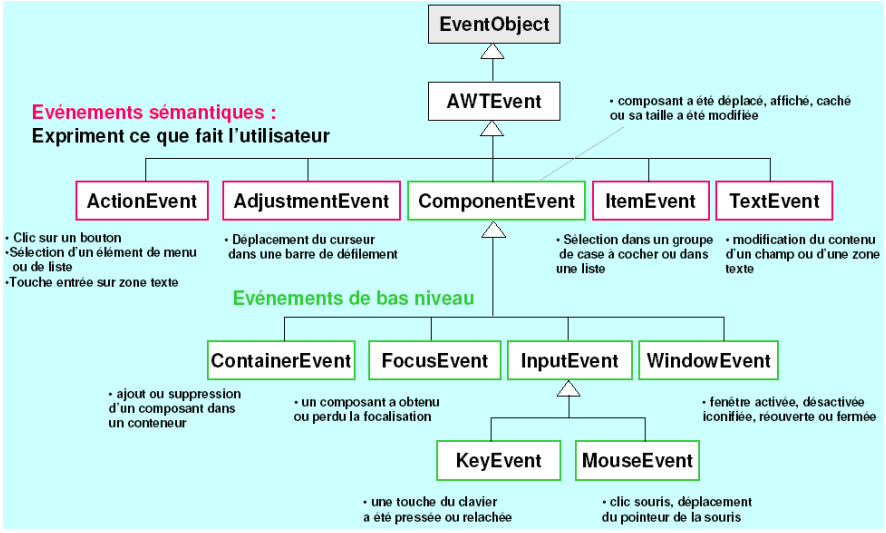
- Chaque composant est susceptible de générer certains événements. Un composant n'est pas concerné par tous les événements.



Evénements AWT (2/3)

- Evénements AWT :
  - Evénements souris sur un composant : **`MouseEvent`** (**`MOUSE_CLICKED, MOUSE_DRAGGED, MOUSE_MOVED, MOUSE_ENTERED`**, ...)
  - Evénements claviers sur un composant : **`KeyEvent`** (**`KEY_PRESSED, KEY_TYPED, VK_F10, VK_A`**, ...)
  - Evénements liés aux manipulation sur une fenêtre : **`WindowEvent`** (**`WINDOW_CLOSED, WINDOW_ICONIFIED`**, ...)
  - Evénements liés au focus : **`FocusEvent`** (**`FOCUS_GAINED, FOCUS_LOST`**, ...)
  - Evénements liés à des actions sur des composants : **`ActionEvent`** (presser un **`Button`**, ...)
  - Evenements liés à des composants textuels : **`TextEvent`** (**`TEXT_VALUE_CHANGED`**, ...)
  - ...

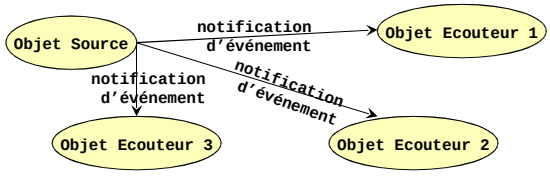
Evénements AWT (3/3)



Evénements Swing

`AncestorEvent`	ancestor added, moved, removed
`CaretEvent`	text caret changed
`ChangeEvent`	state change
`DocumentEvent`	document attributes change, content inserted, content removed (Note: *interface*)
`HyperlinkEvent`	hyperlink activated, entered, exited
`InternalFrameEvent`	frame activated, closed, closing, deactivated, deiconified, iconified, opened
`ListDataEvent`	contents changed, interval added or removed
`ListSelectionEvent`	list selection status change
`MenuDragMouseEvent`	menu drag mouse dragged, entered, exited, released
`MenuEvent`	menu selected/posted, deselected, canceled
`MenuKeyEvent`	menu key pressed, released, typed
`PopupMenuEvent`	popup menu selected/posted, becoming visible, becoming invisible
`TableColumnModelEvent`	table column added, resized, moved, removed, selection changed
`TableModelEvent`	table model change
`TreeExpansionEvent`	tree expanded or collapsed
`TreeModelEvent`	tree nodes changed, inserted, removed, or drastically changed
`TreeSelectionEvent`	tree selection status changed
`UndoableEditEvent`	undoable operation happened

Gestion des événements (1/3)



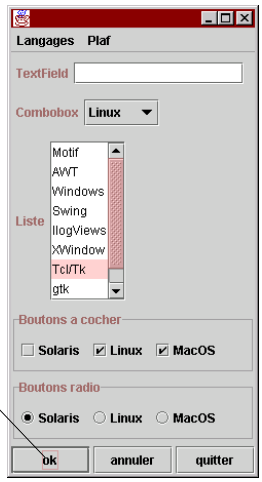
- Chaque source d'événement connait les objets qui l'écoutent et leur envoie les événements qu'elle produit
- Les objets qui écoutent, les écouteurs, sont enregistrés comme tels auprès de la source
- Tout composant peut être source d'événement (événements souris, événements clavier, changements d'état, ...)

Gestion des événements (2/3)

```
public class MyListener implements ActionListener{

   public void actionPerformed(ActionEvent e){
     System.out.println("clic sur bouton");
   }

}
```

Un écouteur est un objet qui traite un ou plusieurs événements

Un écouteur doit être attaché à une source d'événement qui fait appel à ses méthodes de traitement d'événements!



Gestion des événements (3/3)

- Au moment de la création de l'interface :
  - on crée les composants sources d'événements et les écouteurs
  - on lie les écouteurs aux sources
- Les méthodes permettant de lier des écouteurs à des composants correspondent à des interfaces (au sens Java) d'écouteurs

```
public class MonBoutonTest extends JFrame{

  public MonBoutonTest(){
    ...
    Button b = new Button("OK");
    MyListener ml = new MyListener();
    b.addActionListener(ml);
    ...
  }

  ...

}
```

- Une fois l'interface créée, les actions créent des événements et déclenchent l'appel des méthodes des écouteurs

Les écouteurs

- Les écouteurs (ou délégués) sont du type **`EventListener`** (interface)
  - **`KeyListener`**, **`MouseListener`**, ...
- Les écouteurs décrivent les actions à entreprendre en fonction des événements, dans des méthodes aux noms prédéfinis
- *Exemples* :
  - l'interface **`KeyListener`** comporte les méthodes **`keyPressed(KeyEvent e)`**, **`keyReleased(KeyEvent e)`** et **`keyTyped(KeyEvent e)`**
  - l'interface **`MouseListener`** comporte les méthodes **`mouseClicked(MouseEvent e)`**, **`mouseEntered(MouseEvent e)`** , **`mouseExited(MouseEvent e)`**, **`mousePressed(MousEvent e)`** et **`mouseReleased(MouseEvent e)`**
- Les écouteurs s'enregistrent auprès d'une source d'événements, leurs méthodes sont activées par les événements correspondants

Exemple d'utilisation d'écouteur

- Création d'un **`Listener`** particulier

```
public class MyListener implements MouseListener{

  public void mouseClicked(MouseEvent e){System.out.println(“mouse clicked”);}
  public void mouseEntered(MouseEvent e){System.out.println(“mouse entered”);}
  public void mouseExited(MouseEvent e){System.out.println(“mouse exited”);}
  public void mousePressed(MouseEvent e){System.out.println(“mouse pressed”);}
  public void mouseReleased(MouseEvent e){System.out.println(“mouse released”);}

}
```

- Utilisation du **`Listener`**

```
public class MonBoutonTest extends Frame{

  ...
  Button b = new Button("Mon Bouton");
  b.addMouseListener(new MyListener());
  ...

}
```

Evénements/écouteurs AWT (1/2)

Evénement généré par	**Interface d'écoute à implémenter**	**Signatures des méthodes abstraites (à redéfinir obligatoirement)**
Button, List, MenuItem, TexField	ActionListener	public void actionPerformed(ActionEvent e)
Scrollbar	AdjustmentListener	public void adjustmentValueChanged(AdjustmentEvent e)
Dialog, Frame	ComponentListener	public void componentHidden(ComponentEvent e) public void componentMoved(ComponentEvent e) public void componentResized(ComponentEvent e) public void componentShown(ComponentEvent e)
Container	ContainerListener	public void componentAdded(ContainerEvent e) public void componentRemoved(ContainerEvent e)
Component	FocusListener	public void focusGained(FocusEvent e) public void focusLost(FocusEvent e)
Checkbox, CheckboxMenuItem, Choice	ItemListener	public void itemStateChanged(ItemEvent e)

Evénements/écouteurs AWT (2/2)

Evénement généré par	Interface d'écoute à implémenter	Signatures des méthodes abstraites (à redéfinir obligatoirement)
Component	KeyListener	public void keyPressed(KeyEvent e) public void keyReleased(KeyEvent e) public void keyTyped(KeyEvent e)
Canvas, Dialog, Frame, Panel, Window	MouseListener	public void mouseClicked(MouseEvent e) public void mouseEntered(MouseEvent e) public void mouseExited(MouseEvent e) public void mousePressed(MouseEvent e) public void mouseReleased(MouseEvent e)
Canvas, Dialog, Frame, Panel, Window	MouseMotionListener	public void mouseDragged(MouseEvent e) public void mouseMoved(MouseEvent e)
TextField	TextListener	public void textValueChanged(TextEvent e)
Dialog, Frame	WindowListener	public void windowActivated(WindowEvent e) public void windowClosed(WindowEvent e) public void windowClosing(WindowEvent e) public void windowDeactivated(WindowEvent e) public void windowDeiconified(WindowEvent e) public void windowIconified(WindowEvent e) public void windowOpened(WindowEvent e)

Implémentation des écouteurs

- Tout objet peut être écouteur, il suffit qu'il implémente l'interface correspondante et qu'il s'enregistre auprès d'une source
- Un composant peut être un écouteur, en particulier une source peut s'écouter elle même!

ButtonListenerTest

```
public class ButtonListener extends JButton
   implements ActionListener{

  private boolean icon;

  public ButtonListener(String label){
    super(label);
    this.addActionListener(this);
    this.icon = false;
  }

  public void actionPerformed(ActionEvent e){
    if(!this.icon){
      this.setIcon(new ImageIcon("java.gif"));
      this.icon = true;
    }
    else{
      this.setIcon(null);
      this.icon = false;
    }
  }

}
```

Composants écouteurs

- La classe Component implémente une méthode **`processEvent(AWTEvent e)`** qui est appelée automatiquement dès qu'un événement AWT survient sur le composant. On peut donc traiter les événements au niveau du composant en redéfinissant cette méthode. Il est nécessaire de spécifier que le composant écoute le type d'événement considéré (méthode **`enableEvents`**).

- En pratique, des méthodes plus spécifiques sont utilisées
  - **`processActionEvent`**
  - **`processComponentEvent`**
  - **`processFocusEvent`**
  - **`processKeyEvent`**
  - **`processMouseEvent`**
  - **`processMouseMotionEvent`**
  - ...

```
public class ProcessEventTest extends Button{

  public ProcessEventTest(){
    super("ProcessEvent Bouton");
    this.enableEvents(AWTEvent.ACTION_EVENT_MASK);
  }

  protected void processActionEvent(ActionEvent e){
    System.out.println("event");
    super.processActionEvent(e);
  }

}
```

ProcessEventTest

Raccourcis clavier

- Il est possible d'implémenter des raccourcis clavier par la méthode **`registerKeyboardAction(ActionListener,String,KeyStroke,int)`** de **`JComponent`**
- Méthodes plus récemment introduites : **`setInputMap`** et **`setActionMap`**

```
public class HotKey extends JFrame implements ActionListener{

  public HotKey(){
    super("MyFrame");
    JButton b = new JButton("HotKey");
    JButton c = new JButton("not HotKey");
    b.registerKeyboardAction(this,"First Action",KeyStroke.getKeyStroke(KeyEvent.VK_F11,0),
                                                          JComponent.WHEN_FOCUSED);
    this.getRootPane().registerKeyboardAction(this,"Second Action",
                          KeyStroke.getKeyStroke(KeyEvent.VK_SPACE,InputEvent.CTRL_MASK),
                          JComponent.WHEN_IN_FOCUSED_WINDOW);
    ...
  }

  public void actionPerformed(ActionEvent event){
    if(event.getActionCommand().equals("First Action")) System.out.println("F11");
    if(event.getActionCommand().equals("Second Action")) System.out.println("Space");
  }

}
```

HotKeyTest

Manipulation des événements (1/2)

- Les objets événements permettent de connaitre le composant source par la méthode **`getSource()`** de **`EventObject`**
- On peut connaitre les paramètres de l'événement, par exemple pour savoir si une touche est enfoncée en même temps qu'on clique. Méthodes **`getModifiers()`**, **`isShitfDown()`**, **`isControlDown()`**, ... de **`InputEvent`**
- On peut aussi absorber l'événement, ce qui empêche qu'il subisse le traitement par défaut (valable uniquement pour les événements de **`InputEvent`**, ie les **`KeyEvent`** et **`MouseEvent`**) : méthode **`consume()`** de **`InputEvent`**

Consommation d'un événement

```
public class MyListener implements MouseListener{

  public void mouseClicked(MouseEvent e){System.out.println(" mouse clicked/Source : " +
   e.getSource().toString());System.out.println(e.isShiftDown());}
...
  public void mousePressed(MouseEvent e){System.out.println(" mouse pressed/Source : " +
   e.getSource().toString());System.out.println(e.isShiftDown());e.consume();}

}
```

EventConsumeTest

EventQueue

- La classe Toolkit de AWT offre de nombreuses méthodes pour faire un lien entre les classes indépendantes de la plateforme de AWT et celles qui ont une part d'implémentation native
- On peut récupérer une instance de Toolkit grace à la méthode Toolkit.getDefaultToolkit()
- On peut gérer la file d'événements créés par le système grace à getSystemEventQueue()
- Une EventQueue permet de gérer la file d'événements
  - envoyer directement des événements à la file
  - dispatcher des événements de la file
  - ...

Dialogue entre composants

- *Exemple* : deux checkbox s'écoutent l'une l'autre et modifient leur état en fonction des actions intervenues sur l'autre

```
public class MyCheckBox extends JCheckBox implements ActionListener{

  public void actionPerformed(ActionEvent e){
    MyCheckBox cb = (MyCheckBox) e.getSource();
    if(cb.isSelected()) this.setSelected(false);
    else this.setSelected(true);
  }

}
```

```
public class ComponentDialog extends JFrame{

  public ComponentDialog(){
    super("Component Dialog");
    MyCheckBox one = new MyCheckBox("ONE");
    MyCheckBox two = new MyCheckBox("TWO");
    one.addActionListener(two);
    two.addActionListener(one);
    ...
  }

}
```

ComponentDialogTest

Adapteurs

- Les classes **`adapter`** permettent d'éviter de réécrire toutes les méthodes d'une interface d'écouteur
- *Exemple* : **`MouseAdapter`**

```
public class MouseAdapter implements MouseListener{

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

}
```

- Utilisation de l'adapteur

```
public class MyListener2 extends MouseAdapter{

  public void mouseClicked(MouseEvent e){System.out.println("mouse clicked");}

}
```


Ecouteurs et Classe internes

- Pour éviter qu'un composant étende un composant préexistant et un adapteur (héritage multiple impossible en Java), et qu'on ne veut pas implémenter une interface trop lourde, on peut utiliser le mécanisme des classes internes anonymes (ces classes n'ont accès qu'aux variables locales **`final`**)

```
public class MyButton extends JButton extends MouseAdapter{

  ...
  public void mouseClicked(MouseEvent e){System.out.println("mouse clicked");}
  ...

}
```

```
public class MyButton extends JButton{

  public MyButton(){
    ...
    this.addMouseListener(new MouseAdapter(){
      public void mouseClicked(MousEvent e){System.out.println("mouse clicked");}
    });
    ...
  }

  ...

}
```

Classes internes

- Une classe interne est une classe définie à l'intérieur d'une autre classe. Il existe trois types de classes internes :
  - *classe membre* : définie au même niveau que les attributs et méthodes de la classe englobante

```
public class A{
  ...
  public class B{
    ...
  }
}
```

  - *classe locale* : définie à l'intérieur d'une méthode

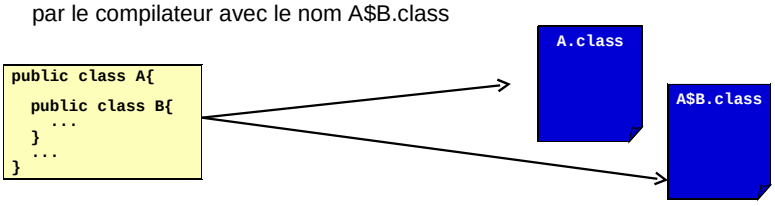
```
public void method(){
  public class B{
    ...
  }
  ...
}
```

  - *classe anonyme* : définie à l'intérieur d'une expression

```
Objet o = new B(){
...
}
```


Classe membre (1/3)

- Toute instance d'une classe membre est associée à une instance de la classe englobante
  - le compilateur insère automatiquement dans la classe membre un attribut privé qui référence l'instance englobante
  - le compilateur insère automatiquement un argument caché à tous les constructeurs de la classe membre
  - la JVM ne sait pas gérer les classes internes, elles sont donc interprétées comme des classes normales. Une classe B membre d'une classe A est créée par le compilateur avec le nom A$B.class



Classe membre (2/3)

- Une classe membre a accès aux attributs et méthodes de la classe englobante, même s'ils sont privés
- L'accès à l'instance de la classe englobante se fait dans la classe membre par **`Nom_Classe_Englobante.this`** (s'il n'y a pas d'ambiguité, le nom du membre de la classe englobante suffit)

```
public class A{

  private void methode(){...}

  public class B{
    A.this.methode();
    // A.this n'est pas this!
  }

}
```

- A l'extérieur de la classe englobante, l'instanciation de la classe membre se fait à travers la classe englobante

```
A.B instanceDeB = new A.B();
```


Classe membre (2/3)

- Si la classe membre est déclarée **static**, elle ne porte pas de référence à une instance de la classe englobante :
  - le mot clé this ne peut y être employé, donc pas d'accès aux attributs et méthodes non static de la classe englobante.
  - possibilité d'instancier la classe membre sans passer par une instance de la classe englobante (**`A.B instanceDeB = new A.B()`**)
- L'héritage est totalement indépendant de l'imbrication

```
public class A extends C{

  public class B extends D{
    ...
  }
  ...
}
```

- On peut imbriquer des classes sur autant de niveaux que l'on veut!!

Classe locale

- Une classe locale est aux classe membre ce que sont les variables locales aux attributs d'une classe
- Une classe locale n'est définie que dans le bloc de code où elle est définie. Elle ne peut être définie que **`static`** ou **`abstract`** (elle est forcément private)
- Une classe locale a accès aux attributs et méthodes de la classe englobante, même s'ils sont privés
- Une classe locale ne peut utiliser une variable locale que si elle est déclarée **`final`**
- *Compilation* : une classe B définie dans la classe A portera après compilation le nom AiB.class où i est l'ordre d'apparition de cette classe interne dans le code de A

Classes anonymes

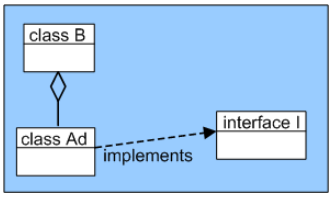
- Une classe anonyme est une classe locale sans nom, définie dans une instruction : on regroupe la définition et l'instanciation d'une classe locale
- Pas de constructeur : l'instance est créée avec le constructeur par défaut. Pas de typage de classe (une classe anonyme est toujours **`final`**)
- Utiliser une classe anonyme si une seule instance de cette classe doit être créée
- Utiliser une classe locale si plusieurs instances de cette classe doivent être créées et/ou si un constructeur est nécessaire
- *Compilation* : une classe anonyme définie dans la classe A portera après compilation le nom A$i.class où i est l'ordre d'apparition de cette classe interne dans le code de A

Où gérer les événements?

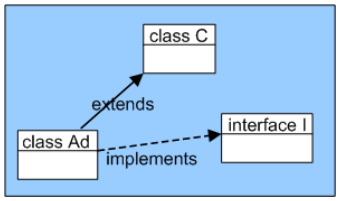
- *Philosophie objet* : découpler au maximum les morceaux du code pour faciliter sa réutilisabilité
- Si on lie un écouteur à un composant, en faisant implémenter l'écouteur par le composant, on peut ne pas pouvoir réutiliser le composant
  - Application *A1* : le composant s'écoute et déclenche en cas d'événement une action *Action1*
  - Application *A2* : le composant s'écoute toujours (réutilisation du code) et est écouté par un objet qui déclenche en cas d'événement une action *Action2* incompatible avec *Action1*
- Pour respecter la philosophie objet et garantir une réutilisabilité maximum du code, il faut séparer les composants et les écouteurs
- En pratique, utiliser les classes internes ou faire implémenter les écouteurs par les composants est courant

Pattern Adapter

- Le pattern **`Adapter`** est utilisé pour rendre une classe conforme à une autre en cas d'incompatibilité d'interfaces :
  - pour utiliser une interface dont on ne veut pas réécrire toutes les méthodes
  - pour utiliser une classe dont l'interface ne convient pas, par exemple conflits sur les signatures des méthodes
  - pour sous-classer plusieurs classes



Object Adapter



Class Adapter

Design Pattern (1/3)

- Un ***pattern*** (ou design pattern) décrit :
  - une situation constituant un problème souvent rencontré dans le développement d'applications
  - une (ou plusieurs) solution(s) type(s) à ce problème, de manière semi-formelle
  - les conséquences de l'usage de ce pattern
- Un pattern est indépendant des différents langages objets, c'est une solution abstraite de haut niveau, pas une brique logicielle
- Un pattern s'exprime souvent par des *interfaces* et des *classes abstraites*, dans un schéma UML
- Les patterns apportent aux concepteurs de logiciels :
  - un vocabulaire commun
  - un catalogue de solutions et une capitalisation de l'expérience
  - une gestion des problèmes à un niveau plus élevé d'abstraction

Design Pattern (2/3)

- Avantages des patterns :
  - vocabulaire commun
  - capitalisation de l'expérience : catalogue de solutions
  - niveau d'abstraction plus élevé améliorant la construction des logiciels
  - réduction de la complexité du développement de logiciel
- Inconvénients :
  - effort d'abstraction et de synthèse
  - apprentissage des patterns
- Règles pour qu'une solution à un problème de développement devienne un pattern :
  - être présent dans au moins 3 gros projets largement utilisés
  - se retrouver dans des projets écrits dans des langages objets différents
  - ne pas être décomposable en patterns plus simples
  - être documenté

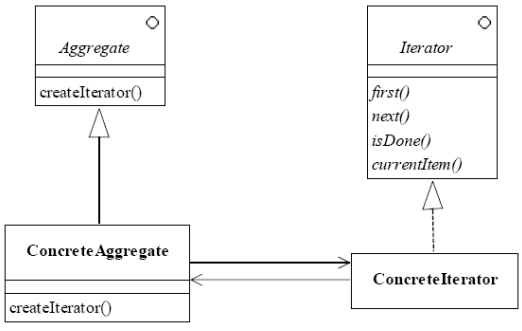
Design Pattern (3/3)

- Patterns reconnus à ce jour :

	Créateurs	Structuraux	Comportementaux
Class	Factory Method	Adapter(class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter(objet) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Respons. Command Iterator Mediator Memento Observer State Strategy Visitor

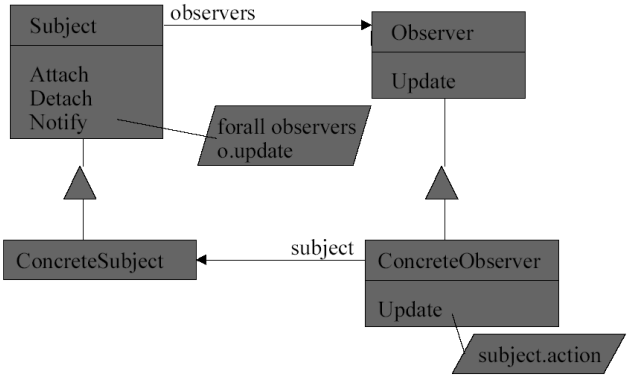
Pattern Iterator

- Le pattern Iterator permet d'accéder séquentiellement à un agrégat d'objets sans dévoiler la structure de l'agrégat



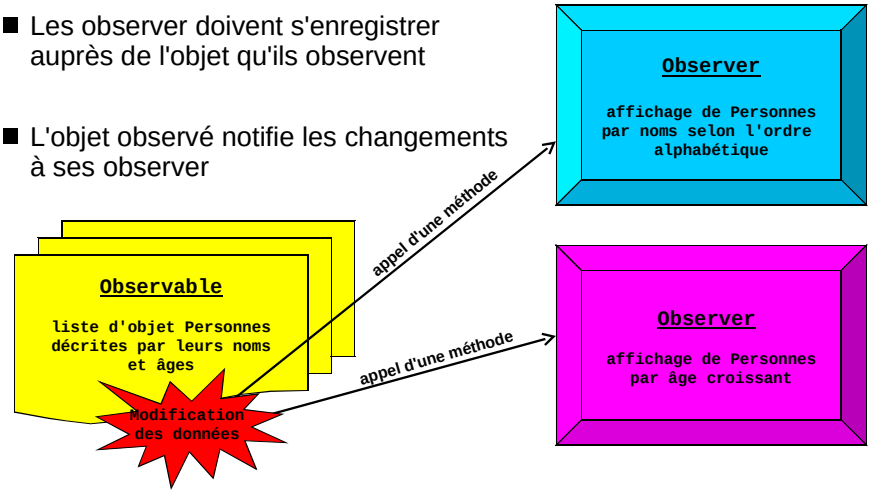
Pattern Observer (1/2)

- Le pattern *Observer* est utilisé quand :
  - un objet a plusieurs aspects, dépendants les uns des autres, et qu'on veut les implémenter de façon modulaire
  - la modification d'un objet se répercute sur d'autres
  - un objet doit notifier quelque chose à d'autres objets sans pour autant les connaitre



Pattern Observer (2/2)

- Les observer doivent s'enregistrer auprès de l'objet qu'ils observent
- L'objet observé notifie les changements à ses observer



Le pattern Observer en Java (1/2)

- L'interface Observer n'a qu'une méthode, appelée par les objets que l'Observer observe : **`void update(Observable o,Object arg)`**
- Un Observable (Subject) est un objet doté d'un ou plusieurs Observer, qui s'enregistrent auprès de l'Observable.
- *Exemple : observer un nombre entier*

```
public class Entier extends Observable{

  private int nb;

  public Entier(int nb){
    super();
    this.nb = nb;
  }

  public int getValue(){
    return this.nb;
  }

  ...
```

```
  ...

  public void incremente(){
    nb++;
    this.setChanged();
    this.notifyObservers();
  }

  public void decremente(){
    nb--;
    this.setChanged();
    this.notifyObservers();
  }

}
```

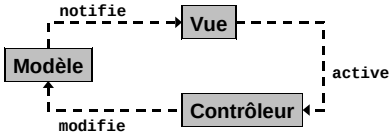

Le pattern Observer en Java (2/2)

```
public class ObserverEntier extends JButton implements Observer{

  private Entier e;

  public ObserverEntier(Entier e){
    super(Integer.toString(e.getValue()));
    this.e = e;
    e.addObserver(this);
  }

  public void update(Observable o,Object arg){
    this.setText(Integer.toString(e.getValue()));
  }

}
```

ObserverTest

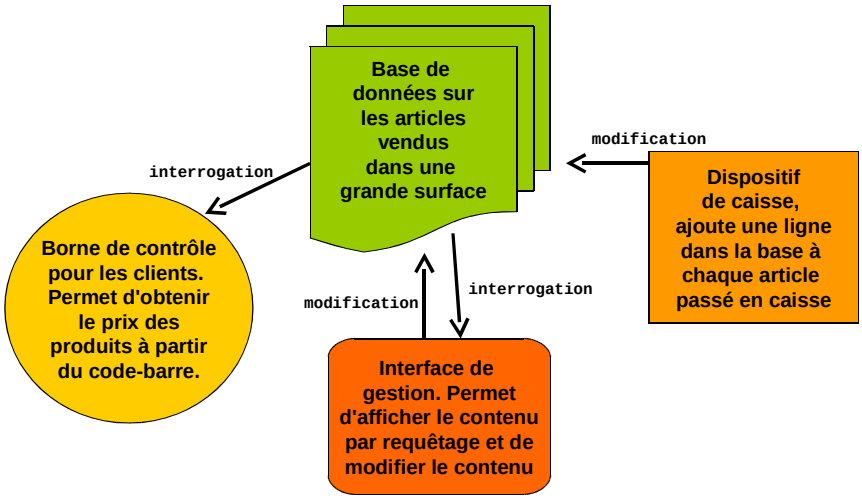
Observer et MVC

- Dans les interfaces graphiques, il est préférable de séparer les *données manipulées* et l'*affichage de ces données*
  - d'un coté les classes qui implémentent le modèle des données
  - de l'autre les classes qui gèrent l'affichage des données dans l'interface
  - le pattern Observer/Observable permet de lier les deux
- **Généralisation** : le MVC (Modèle Vue Contrôleur) : on ajoute aux parties modèle et vue la partie controleur qui gèrent les actions sur le modèle

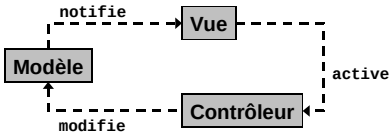


- MVC : pattern de conception créé dans les années 80 pour le développement d'interface en SmallTalk

Exemple de MVC



Modèle Vue Contrôleur (1/2)



- Le modèle ne dépend ni de la vue ni du controleur. Il peut donc être développé et testé indépendamment.
  - *Bémol : il faut parfois penser à la vue quand on développe le modèle, si par exemple le modèle va subir des mises à jour fréquentes, il faut que le modèle, au lieu d'innonder la vue de demande de mises à jour, groupe ses requêtes de maj*
- La vue et le contrôleur dépendent toujours du modèle
- Si modèle et vue respecte le pattern Observer, la vue est mise à jour par le modèle, le contrôleur n'agit que sur le modèle
  - permet d'optimiser le rafraichissment (le controleur ne sait pas en détail ce qui a changé)
  - plus simple s'il y a plusieurs vues et controleurs

Modèle Vue Contrôleur (2/2)

- Avantage du MVC :
  - un seul modèle peut avoir plusieurs vues et plusieurs controleurs (applications réparties)
  - possibilités de modifier le modèle indépendamment de ses vues, et de modifier les vues sans toucher au modèle, autonomie des composants (les interfaces changent plus rapidement que les données, connaissances métier)
- Inconvénients du MVC :
  - demande un travail de conception plus poussé
  - une application MVC est a priori plus complexe à développer mais aussi à débugger, du fait des interactions plus riches entre les 3 parties
- Variante du MVC : on fusionne la vue et le controleur, ce qui réduit la complexité (exemple du bouton qui affiche l'entier et le modifie aussi)

Exemple de MVC (1/4)

- Un modèle : la classe Entier

```
public class Entier extends Observable{

  private int nb;

  public Entier(int nb){
    super();
    this.nb = nb;
  }

  public int getValue(){
    return this.nb;
  }

  public void incremente(){
    nb++;
    this.setChanged();
    this.notifyObservers(new Integer(nb));
  }

  ...
```

```
  ...

  public void decremente(){
    nb--;
    this.setChanged();
    this.notifyObservers(new Integer(nb));
  }

  public void modifie(int modifieur){
    this.nb = this.nb + modifieur;
    this.setChanged();
    this.notifyObservers(new Integer(nb));
  }

}
```


Exemple de MVC (2/4)

- Une vue : affichage en chiffre

```
public class Vue1 extends JLabel implements Observer{

  private Entier e;

  public Vue1(Entier e){
    super(Integer.toString(e.getValue()));
    this.e = e;
    e.addObserver(this);
  }

  public void update(Observable o,Object arg){
    this.setText(((Integer) arg).toString());
  }

}
```

Exemple de MVC (3/4)

- Une autre vue : affichage en chiffre en binaire

```
public class Vue2 extends JLabel implements Observer{

  private Entier e;

  public Vue2(Entier e){
    super(Integer.toBinaryString(e.getValue()));
    this.e = e;
    e.addObserver(this);
  }

  public void update(Observable o,Object arg){
    this.setText(Integer.toBinaryString(((Integer) arg).intValue());
  }

}
```


Exemple de MVC (4/4)

- Un contrôleur

```
public class Controleur extends JButton{

  private Entier e;
  private int pas;

  public Controleur(Entier e,String action){
    super(action);
    this.e = e;
    if(action.equals("+")) this.pas = 1;
    if(action.equals("-")) this.pas = -1;
    this.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        action();
      }
    });
  }

  public void action(){
    this.e.modifie(pas);
  }

}
```


MVCTest

Action

- Les actions Swing offrent une manière de centraliser une action qui peut ensuite être implémentée par plusieurs composants. Une telle action est définie comme sous-classe de **`AbstractAction`** (classe abstraite)

- Une **`AbstractAction`** implémente l'interface **`Action`** qui hérite de **`ActionListener`**. **`AbstractAction`** :
  - contient une etiquette et une icone qui décrivent l'action, une description longue et une courte pour les bulles d'aide. On modifie ces attributs par la méthode **`putValue(String,Object)`**, on y accède par **`getValue(String)`**
  - gère le fait d'être activée ou pas par la méthode **`setEnabled(boolean)`**
  - gère l'action effectuée par la méthode par la méthode **`actionPerformed(ActionEvent)`**
  - gère les changements de propriétés de l'action en offrant la possibilité de lui attacher des écouteurs d'événements **`PropertyChangeEvent`**

Exemples d'Action

```
public class Action1 extends AbstractAction{

  public Action1(){
    super("Java",new ImageIcon("java.gif"));
    this.putValue(SHORT_DESCRIPTION,"Java Language");
  }

  public void actionPerformed(ActionEvent e){
    System.out.println("Java");
  }

}
```

```
public class Action2 extends AbstractAction{

  public Action2(){
    super("C++");
    this.putValue(SHORT_DESCRIPTION,"C++ Language");
  }

  public void actionPerformed(ActionEvent e){
    System.out.println("C++");
  }

}
```

Utilisation des Action

```
    ...
    Action1 a1 = new Action1();
    Action2 a2 = new Action2();
    JMenuBar jmb = new JMenuBar();
    JMenu jm = new JMenu("File");
    jm.add(a1);jm.add(a2);
    jmb.add(jm);
    this.setJMenuBar(jmb);
    JToolBar jtb = new JToolBar();
    jtb.add(a1);jtb.add(a2);
    JButton des = new JButton("Active/Desactive Java");
    des.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){desactive();}
    });
    jtb.add(des);
    this.getContentPane().add(jtb, BorderLayout.PAGE_START);
    this.getRootPane().getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW).put(
                                        KeyStroke.getKeyStroke(KeyEvent.VK_F11,0),"a1");
    this.getRootPane().getActionMap().put("a1",a1);
    this.getRootPane().getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW).put(
                                         KeyStroke.getKeyStroke(KeyEvent.VK_F6,0),"a2");
    this.getRootPane().getActionMap().put("a2",a2);
    ...
```


```
public void desactive(){
  if(a1.isEnabled()) this.a1.setEnabled(false);
  else this.a1.setEnabled(true);
}
```


Un peu d'IHM...

- Pour faire de bonnes interfaces, il ne suffit pas de savoir utiliser les primitives graphiques d'un langage, il faut aussi respecter certaines règles, certaines méthodes permettant d'offrir des interfaces conviviales, efficaces, ergonomiques, ...

  => c'est ce qu'on appelle l'IHM (Interaction Homme Machine, ex-Interface Homme Machine) qui étudie les dispositifs d'interface entre les machines et les utilisateurs, dans le but d'améliorer les interactions

- Les interfaces représentent près de la moitié du développement d'un logiciel d'où l'importance de l'IHM

...encore un peu...

- *IHMs* :
  - paradigme technologique : l'interface reproduit la structure sous-jacente à l'application. Demande une maitrise de cette application et de ses mécanismes internes
  - paradigme de la métaphore : l'interface mime le comportement d'un objet déjà connu par l'utilisateur (dossier pour les répertoire, documents papiers)
  - paradigme idiomatique : utilise des éléments d'interface simples et stéréotypés (fenêtre, ...). L'interface ne reproduit pas forcément les mécanismes internes de l'application
- MVC est un modèle permettant de bien développer ses interfaces, mais c'est un modèle interne, orienté machine
- *Exemples de règles orientées utilisateur* :
  - Cohérence : seules les actions possibles sont offertes à l'utilisateur (griser les boutons non pertinents et les désactiver)
  - Offrir à l'utilisateur plusieurs façons de mener une même action : par un item dans un menu, par un bouton de la barre d'outils, par un raccourci clavier

...et encore un peu

- Compatibilité : il faut que l'interface et son comportement soit compatible le plus possible avec ce que connait déjà l'utilisateur
  - le support papier
  - les autres logiciels
- Homogénéité : rendre la restitution des mêmes informations identique, améliorer la prévisibilité du comportement de l'interface, faciliter l'utilisation
- Concision : ne pas obliger l'utilisateur à mémoriser beaucoup de choses, limiter la charge de travail, minimiser le nombre d'opérations à réaliser, le nombre d'entrées (valeurs par défaut)
- Flexibilité : permet à l'interface de s'adapter à différents utilisateurs (raccourcis claviers,) L'interface s'adapte à l'utilisateur et non l'inverse
- FeedBack et guidage : l'utilisateur doit pouvoir à tout moment savoir quel est l'état du système, être informé du résultat d'une action, savoir comment poursuivre ses actions
- Controle explicite : l'utilisateur doit avoir le sentiment de controler le logiciel, les actions ne se font que sur intervention de l'utilisateur
- Gestion des erreurs : l'utilisateur doit pouvoir récupérer de ses erreurs (undo/redo), l'interface doit prévenir les erreurs (controle des entrées)