Introduction à l'Interaction Homme-Machine

TIS3
2009-2010

Renaud Blanch
IIHM - LIG - UJF
mailto:renaud.blanch@imag.fr
http://iihm.imag.fr/blanch/

Remerciements

Philippe Genoud
(INRIA - HELIX, UJF)

Jean Berstel
(Université de Marne la Vallée)

2. La boîte à outils de constructions d'interfaces Java/Swing

2.0 Les BàO de Java

Indépendance vis-à-vis du système hôte

"Write Once, Run Anywhere."

Slogan de Java,
difficile à mettre en œuvre,
en particulier pour les applications
ayant des **interfaces graphiques**.

Indépendance vis-à-vis du système hôte

exemple :
Une **étiquette** représentée par la classe ***Label***

```
class Label {
   public Label(String text) { ... }

   public void setText(String text) { ... }
   public String getText() { ... }

   ...

   public void paint(Graphics g) { ... }
};
```

Indépendance vis-à-vis du système hôte

Deux approches coexistent :
- **maximaliste**,
  i.e. déléguer le plus possible au système hôte ; et
- **minimaliste**,
  i.e. dépendre le moins possible du système hôte.

BàO : Approche maximaliste

déléguer le plus possible au système hôte

La classe *Label* a **des réalisations** différentes (une par hôte) **qui délèguent** le travail **à un composant natif**.

BàO : Approche maximaliste

avantages :
+ apparence et comportement conformes à ceux de l'hôte ;
+ réutilisation de l'existant.

inconvénients :
- seuls les services **présents sur tous les hôtes** peuvent être offerts ;
- comportements variables d'un système à l'autre, nécessité d'une **couche d'adaptation**.

BàO : Approche maximaliste

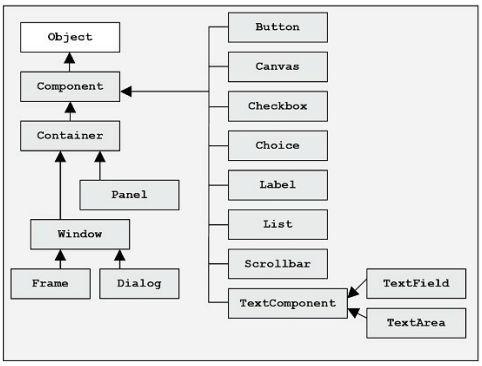
C'est l'approche suivie par la BàO historique de Java
Abstract Widget Toolkit (AWT),
fournie par les paquets `java.awt.*`,
tombée en désuétude.

BàO : Approche maximaliste

C'est l'approche suivie par la BàO historique de Java
Abstract Widget Toolkit (AWT),
fournie par les paquets `java.awt.*`,
tombée en désuétude.

C'est aussi l'approche de la BàO utilisée par *Eclipse*
Standard Widget Toolkit (SWT),
fournie par les paquets `org.eclipse.swt.*`,
de plus en plus utilisée.

AWT : classes de *widgets*



BàO : Approche minimaliste

dépendre le moins possible du système hôte

Le système est abstrait à bas niveau
(ouvrir une fenêtre, dessiner des primitives simples).

La classe *Label* a **une réalisation** en Java
qui **utilise** cette abstraction du **système graphique**.

BàO : Approche minimaliste

avantages :
+ apparence et comportement
uniformes d'un hôte à l'autre ;
+ disponibilité sur tous les hôtes.

inconvénients :
- comportements non-natifs ;
- lenteur.

BàO : Approche minimaliste

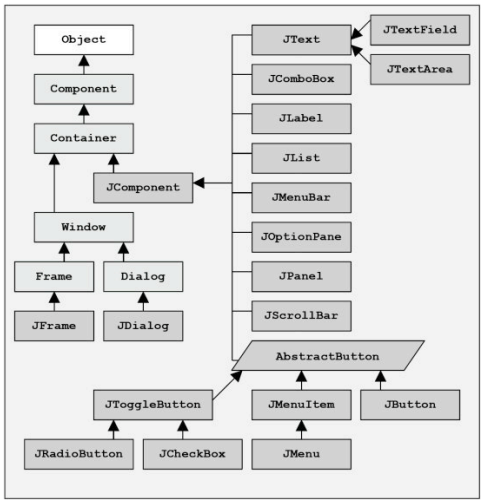
C'est l'approche suivie par
la BàO actuelle de Java ***Swing***,
fournie par les paquets `javax.swing.*`,
très utilisée de nos jours.

BàO : Approche minimaliste

C'est l'approche suivie par
la BàO actuelle de Java ***Swing***,
fournie par les paquets `javax.swing.*`,
très utilisée de nos jours.

La BàO *AWT* est toujours présente, en particulier
car les composants *Swing* sont construits
en utilisants les composants *AWT* de bas niveau.

Swing : classes de *widgets*



2.1 Gestion des événements

Mécanisme général des événements Java

La base de tous les événements :
`java.util.EventObject`
contient une information :
sa source (quel objet est à l'origine de l'événement).

```
class EventObject {
   public EventObject(Object source) { ... }
   public Object getSource() { ... }
   ...
};
```

Les événements *AWT*

AWT spécialise `EventObject` en
`java.awt.AWTEvent`
en encodant le **type** de l'événement dans un entier.

```
class AWTEvent extends EventObject {
   public AWTEvent(Object source, int id) { ... }
   public int getID() { ... }
   ...
};
```

Les événements *AWT*

Les types d'événements *AWT* :

```
COMPONENT_EVENT_MASK
CONTAINER_EVENT_MASK
FOCUS_EVENT_MASK
KEY_EVENT_MASK
MOUSE_EVENT_MASK
MOUSE_MOTION_EVENT_MASK
WINDOW_EVENT_MASK
ACTION_EVENT_MASK
ADJUSTMENT_EVENT_MASK
ITEM_EVENT_MASK
TEXT_EVENT_MASK
INPUT_METHOD_EVENT_MASK
PAINT_EVENT_MASK
INVOCATION_EVENT_MASK
HIERARCHY_EVENT_MASK
HIERARCHY_BOUNDS_EVENT_MASK
MOUSE_WHEEL_EVENT_MASK
WINDOW_STATE_EVENT_MASK
WINDOW_FOCUS_EVENT_MASK
```

Les événements *AWT*

Le paquet **`java.awt.event`** spécialise `AWTEvent` en plusieurs classes d'événements d'assez hauts niveaux :

`ActionEvent`, `AdjustmentEvent`, `AncestorEvent`, **`ComponentEvent`**, `HierarchyEvent`, `InputMethodEvent`, `InternalFrameEvent`, `InvocationEvent`, **`ItemEvent`**, **`TextEvent`**.

Les événements *AWT*

Enfin, la classe `ComponentEvent` est elle-même spécialisée en événements de plus bas niveaux :

`ContainerEvent`, `FocusEvent`, **`InputEvent`** (d'où dérivent **`KeyEvent`** et **`MouseEvent`**), `PaintEvent`, `WindowEvent`.

Les événements *Swing*

Le paquet **`javax.swing.event`** définit quelques événements supplémentaires (`CaretEvent`, `MenuEvent`, ...) à partir de `EventObject`.

Cependant, *Swing* réutilise principalement les événements de *AWT*.

Les écouteurs *(listeners)* d'événements

Les événements contiennent un champs qui précise leur **source**.

Un objet qui désire recevoir les **événements** d'une **classe particulière** doit
- **réaliser** une **interface associée à ce type** d'événement ; et
- **s'enregistrer** auprès de la source d'événement.

Les écouteurs d'événements

L'interface déclare les méthodes qu'il faudra réaliser pour recevoir les événements.

Les écouteurs d'événements

L'interface déclare les méthodes qu'il faudra réaliser pour recevoir les événements.

Par exemple, pour recevoir des `ActionEvents`, il faut réaliser l'interface qui leur est liée :

```
package java.awt.event;
import java.util.EventListener;

interface ActionListener extends EventListener {
   public void actionPerformed(ActionEvent event);
};
```

Les écouteurs d'événements

Plus généralement, on a :

événement	*listener*
ActionEvent	**Action**Listener
KeyEvent	**Key**Listener
TextEvent	**Text**Listener
MouseEvent	**MouseInput**Listener (**Mouse**Listener + **MouseMotion**Listener) **(!)**
MouseWheelEvent	**MouseWheel**Listener
...	...

Les écouteurs d'événements

Chaque interface définit un **ensemble** de méthodes correspondantes aux **sous-types d'événements** représentés par la classe.

Les écouteurs d'événements

La classe `MouseEvent` définit par exemple **7 sous-types** identifiés par les constantes :

```
MOUSE_CLICKED
MOUSE_PRESSED
MOUSE_RELEASED
MOUSE_ENTERED
MOUSE_EXITED

MOUSE_MOVED
MOUSE_DRAGGED
```

Les écouteurs d'événements

La classe `MouseEvent` définit par exemple **7 sous-types** correspondants à **2 interfaces** :

```
interface MouseListener implements EventListener {
   public void mouseClicked(MouseEvent event);
   public void mousePressed(MouseEvent event);
   public void mouseReleased(MouseEvent event);
   public void mouseEntered(MouseEvent event);
   public void mouseExited(MouseEvent event);
};

interface MouseMotionListener implements EventListener {
   public void mouseMoved(MouseEvent event);
   public void mouseDragged(MouseEvent event);
};
```

Les écouteurs d'événements

Pour résumer, en général on a :

• une **classe d'événement** avec ses sous-types :

```
class SomethingEvent extends EventObject {
   public static final int SOMETHING_HAPPENED = ...
   static static final int SOMETHING_OCCURED  = ...
   ...
};
```

• un ***listener*** (parfois plusieurs) associé avec une méthode par sous-type d'événement :

```
interface SomethingListener implements EventListener {
   public void somethingHappened(SomethingEvent e);
   public void somethingOccured(SomethingEvent e);
};
```

Les sources d'événements

Les événements contiennent un champs qui précise leur **source**.

Un objet qui désire recevoir les **événements** d'une **classe particulière** doit

• **réaliser** une **interface associée à ce type** d'événement ; et

• **s'enregistrer** auprès de la source d'événement.

Les sources d'événements

Les objets susceptibles d'émettre des événements *(event source)* doivent permettre aux *listeners* adaptés à leur type d'événements de **s'inscrire** / **se désinscrire** pour être notifiés.

Les sources d'événements

Les objets susceptibles d'émettre des événements *(event source)* doivent permettre aux *listeners* adaptés à leur type d'événements de **s'inscrire** / **se désinscrire** pour être notifiés.

Ils gèrent ainsi une liste de leurs *listeners* et appellent leurs méthodes correspondantes lorsqu'un événement se produit.

Les sources d'événements

```
class SomethingEmitter {
   private Vector<SomethingListener> listeners =
                                    new Vector<SomethingListener>();
   public void addSomethingListener(SomethingListener listener) {
      listeners.append(listener);
   }
   public void removeSomethingListener(SomethingListener listener) {
      listeners.remove(listener);
   }
   private void onSomethingHappened() {
      for(SomethingListener listener : listeners) {
         SomethingEvent event =
                        new SomethingEvent(this, SOMETHING_HAPPENED);
         listener.somethingHappened(event);
      }
   }
   ...
};
```

Les sources d'événements

Avec ce mécanisme d'abonnement, **une source** peut avoir **plusieurs écouteurs** :

```
class Receiver implements SomethingListener { ... };

SomethingEmitter emitter = new SomethingEmitter();
Receiver rec1 = new Receiver();
Receiver rec2 = new Receiver();

emitter.addSomethingListener(rec1);
emitter.addSomethingListener(rec2);
... // somethingHappened -> rec1 & rec2

emitter.removeSomethingListener(rec2);
... // somethingHappened -> rec1
```

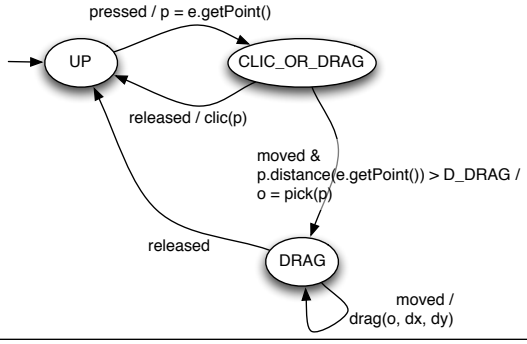
Les sources d'événements

Avec ce mécanisme d'abonnement, **un écouteur** peut avoir **plusieurs sources** :

```
class Receiver implements SomethingListener {
   public void somethingHappened(SomethingEvent event) {
      SomethingEmitter emitter = (SomethingEmitter)event.getSource();
      if(emitter == ...) {
         ...
      }
   }
};

SomethingEmitter emi1 = new SomethingEmitter();
SomethingEmitter emi2 = new SomethingEmitter();
Receiver receiver = new Receiver();

emi1.addSomethingListener(receiver);
emi2.addSomethingListener(receiver);
... // somethingHappened -> receiver
```

Techniques objets pour la gestion des événements

Les techniques suivantes sont illustrées pour le **multiplexage du *clic* et du *drag***.



Exemple

Le rendu et les actions de l'interface sont assurés par une classe fournie, **`ClicAndDrag`**, il reste à réaliser l'écouteur **`Listener`** :

```
public class ClicAndDrag extends JPanel {
   public ClicAndDrag() {
      Listener listener = new Listener(this);
      addMouseListener(listener);
      addMouseMotionListener(listener);
      ...
   }

   public void clic(Point p) { ... }
   public Object pick(Point p) { ...}
   public void drag(Object o, int dx, int dy) { ... }
}
```

Exemple

```
public class Listener implements MouseInputListener {
   ClicAndDrag cnd; // related object
   public Listener(ClicAndDrag cnd) {
      this.cnd = cnd;
   }

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

   public void mouseDragged(MouseEvent e) { ... }
   public void mouseMoved(MouseEvent e) {}
}
```

De l'automate aux *listeners*

Les états de l'automates peuvent être codés par une **énumération**.

De l'automate aux *listeners*

```
public class Listener implements ... {
   enum State { UP, CLIC_OR_DRAG, DRAG } // possible states
   State state = State.UP;               // current state

   Point p; // mouse pressed position

   public void mousePressed(MouseEvent e) {
      switch(state) {
      case UP:
         p = e.getPoint();
         state = State.CLIC_OR_DRAG;
         break;
      default:
         throw new RuntimeException();
      }
   }

   ...
```

De l'automate aux *listeners*

```
   ...
   static final int D_DRAG = 5; // dragging max distance
   Object o;                    // object to drag

   public void mouseDragged(MouseEvent e) {
      switch(state) {
      case CLIC_OR_DRAG:
         if(p.distance(e.getPoint()) > D_DRAG) {
            o = cnd.pick(p);
            state = State.DRAG;
         }
         break;
      case DRAG:
         Point n = e.getPoint(); // new position
         cnd.drag(o, n.getX()-p.getX(), n.getY()-p.getY());
         p = n;
         break;
      default:
         throw new RuntimeException();
      }
   }
```

De l'automate aux *listeners*

```
   ...
   public void mouseReleased(MouseEvent e) {
      switch(state) {
      case CLIC_OR_DRAG:
         cnd.clic(p);
         state = State.UP;
         break;
      case DRAG:
         state = State.UP;
         break;
      default:
         throw new RuntimeException();
      }
   }
}
```

Les classes internes *(inner class)*

Souvent les *listeners* doivent accéder **à l'intérieur de la classe** pour laquelle ils travaillent.

Les classes internes *(inner class)*

Souvent les *listeners* doivent accéder **à l'intérieur de la classe** pour laquelle ils travaillent.

Or on ne veut pas forcément que les méthodes puissent être utilisées par ailleurs.

Les classes internes *(inner class)*

Souvent les *listeners* doivent accéder **à l'intérieur de la classe** pour laquelle ils travaillent.

Or on ne veut pas forcément que les méthodes puissent être utilisées par ailleurs.

La solution : une **classe interne**.

Avant ...

```
public class ClicAndDrag extends JPanel {
   public ClicAndDrag() {
      Listener listener = new Listener(this);
      addMouseListener(listener);
      addMouseMotionListener(listener);
      ...
}
public class Listener implements MouseInputListener {
   ClicAndDrag cnd;
   public Listener(ClicAndDrag cnd) { this.cnd = cnd; }
   ...
   public void mouseReleased(MouseEvent e) {
      switch(state) {
      case CLIC_OR_DRAG:
         cnd.clic(p);
         state = State.UP;
         break;
      ...
```

... après (solution brutale)

```
public class ClicAndDrag extends JPanel
                        implements MouseInputListener {
   public ClicAndDrag() {
      addMouseListener(this);
      addMouseMotionListener(this);
      ...

   public void mouseReleased(MouseEvent e) {
      switch(state) {
      case CLIC_OR_DRAG:
         clic(p);
         state = State.UP;
         break;
      ...
```

Les *inner classes*

Une classe définie **à l'intérieur** *(inner)* **d'une autre classe**.

Elle a **accès** aux membres (méthodes, attributs), y compris **privés,** de sa **classe englobante** *(outer class)*.

Elle suit les **règles de visibilité des attributs** (`public`, `protected`, `private`, *package private*)

... après (avec une *inner class*)

```
public class ClicAndDrag extends JPanel {
   public ClicAndDrag() {
      Listener listener = new Listener();
      ...
   }

   enum State { UP, CLIC_OR_DRAG, DRAG }

   class Listener implements ... {
      State state = State.UP;
      ...
      public void mouseReleased(MouseEvent e) {
         switch(state) {
         case CLIC_OR_DRAG:
            clic(p);
            state = State.UP;
            break;
         ...
```

Les adaptateurs d'écouteurs *(adapters)*

Souvent les *listeners* définissent
beaucoup de méthodes
dont seulement **un sous-ensemble**
est **pertinent** pour l'application envisagée.

Exemple

```
public class Listener implements MouseInputListener {
   ...
   // MouseListener interface
   public void mousePressed(MouseEvent e) { ... }
   public void mouseReleased(MouseEvent e) { ... }
   public void mouseClicked(MouseEvent e) {}
   public void mouseEntered(MouseEvent e) {}
   public void mouseExited(MouseEvent e) {}

   // MouseMotionListener interface
   public void mouseDragged(MouseEvent e) { ... }
   public void mouseMoved(MouseEvent e) {}
}
```

Les *adapters*

La solution offerte par Java :
à chaque **interface d'écouteur**
est adjointe une **réalisation adaptatrice**
dont **les méthodes ne font rien**.

Réalisation

Pour le *listener* générique suivant :

```
interface SomethingListener implements EventListener {
   public void somethingHappened(SomethingEvent e);
   public void somethingOccured(SomethingEvent e);
}
```

on dispose de l'*adapter* :

```
class SomethingAdapter implements SomethingListener {
   public void somethingHappened(SomethingEvent e) {}
   public void somethingOccured(SomethingEvent e) {}
}
```

Réalisation

et si seuls les événements `SOMETHING_OCCURED`
nous intéressent, au lieu de :

```
class Listener implements SomethingListener {
   public void somethingHappened(SomethingEvent e) {}
   public void somethingOccured(SomethingEvent e) {
      // do something
   }
}
```

on écrira :

```
class Listener extends SomethingAdapter {
   public void somethingOccured(SomethingEvent e) {
      // do something
   }
}
```

Exemple

```
public class Listener implements MouseInputListener {
   ...
   // MouseListener interface
   public void mousePressed(MouseEvent e) { ... }
   public void mouseReleased(MouseEvent e) { ... }
   public void mouseClicked(MouseEvent e) {}
   public void mouseEntered(MouseEvent e) {}
   public void mouseExited(MouseEvent e) {}

   // MouseMotionListener interface
   public void mouseDragged(MouseEvent e) { ... }
   public void mouseMoved(MouseEvent e) {}
}
```

Exemple

```
public class Listener extends MouseInputAdapter {
   ...
   public void mousePressed(MouseEvent e) { ... }
   public void mouseReleased(MouseEvent e) { ... }

   public void mouseDragged(MouseEvent e) { ... }
}
```

Problème non résolu

Si **plusieurs types d'événements** doivent être traités (e.g., `MouseEvent` et `MouseWheelEvent`), à défaut d'**héritage multiple**, on ne peut utiliser qu'**un seul adaptateur** à la fois.

```
class Listener extends MouseInputAdapter
               implements MouseWheelListener {
   public void mousePressed(SomethingEvent e) {
      ...
   }

   public void mouseWheelMoved(SomethingEvent e) {
      ...
   }
}
```

Adapters et classe anonymes *(anonymous class)*

Les adaptateurs permettent parfois de réaliser
des écouteurs **très concis**,
définis dans **une classe interne**, à **usage unique**.

```
class OuterClass {
   public OuterClass() {
      MouseListener listener = new Listener();
   }
   class Listener extends MouseAdapter {
      public void mousePressed(MouseEvent e) { ... }
   }
}
```

Adapters et classe anonymes *(anonymous class)*

Pour rapprocher la définition,
on peut utiliser une **classe locale** :

```
class OuterClass {
   public OuterClass() {
      class Listener extends MouseAdapter {
         public void mousePressed(MouseEvent e) {
            ...
         }
      }
      MouseListener listener = new Listener();
   }
}
```

(elle a en plus accès aux variables de la méthode)

Adapters et classe anonymes *(anonymous class)*

Une **syntaxe particulière** permet de rendre le code
encore plus concis en utilisant une **classe anonyme** :

```
class OuterClass {
   public OuterClass() {
      MouseListener listener = new MouseAdapter() {
         public void mousePressed(MouseEvent e) {
            ...
         }
      };
   }
}
```

(définit et instancie une classe dérivée, sans nom)

Adapters et classe anonymes *(anonymous class)*

Une **syntaxe particulière** permet de rendre le code encore plus concis en utilisant une **classe anonyme**.

À **utiliser avec parcimonie** car :
• usage unique ;
• pas de constructeur ;
• pas de nom ...

Énumérations : utilisation avancée

En Java, les **éléments d'une énumération** peuvent avoir des **méthodes** :

```
enum Test {
   ZERO,
   UN {
      void test() { System.out.println(UN); }
   };
   void test() { System.out.println("default"); }
}

Test.ZERO.test(); // -> "default"
Test.UN.test();   // -> "UN"
```

Énumérations : utilisation avancée

En Java, les **éléments d'une énumération** peuvent avoir des **méthodes**, partager des **données** :

```
enum Test {
   ZERO,
   UN;
   static int i;
}

Test.ZERO.i = 3; // Test.ZERO.i == TEST.UN.i == 3
Test.UN.i = 12;  // Test.ZERO.i == TEST.UN.i == 12
```

Énumérations : utilisation avancée

En Java, les **éléments d'une énumération** peuvent avoir des **méthodes**, partager des **données**.

Elles permettent donc de réaliser des automates.

Exemple

```
enum State {
   UP { ... }, CLIC_OR_DRAG { ... }, DRAG { ... };
   State press(MouseEvent e)   { throw new RuntimeError(); }
   State move(MouseEvent e)    { throw new RuntimeError(); }
   State release(MouseEvent e) { throw new RuntimeError(); }
}

MouseInputListener listener = new MouseInputAdapter()  {
   State state = State.UP;
   public void mousePressed(MouseEvent e) {
      state = state.press(e);
   }
   public void mouseDragged(MouseEvent e) {
      state = state.move(e);
   }
   public void mouseReleased(MouseEvent e) {
      state = state.release(e);
   }
};
```

Exemple

```
enum State {
   UP {
      State press(MouseEvent e) {
         p = e.getPoint();
         return CLIC_OR_DRAG;
      }
   },
   CLIC_OR_DRAG { ... },
   DRAG { ... };

   static Point p;
   ...
}
```

Exemple

```
enum State {
   UP { ... },
   CLIC_OR_DRAG {
      State move(MouseEvent e) {
         if(p.distance(e.getPoint()) > D_DRAG) {
            o = pick(p); // ideally forwarded to outer class
            return DRAG;
         }
         return CLIC_OR_DRAG;
      }
      State release(MouseEvent e) {
         clic(p); // ideally forwarded to outer class
         return UP;
      }
   },
   DRAG { ... };

   static Object o;
   static final int D_DRAG = 5;
   ...
```

Exemple

```
enum State {
   UP { ... },
   CLIC_OR_DRAG { ... },
   DRAG {
      State move(MouseEvent e) {
         Point n = e.getPoint();
         drag(o, n.getX()-p.getX(), n.getY()-p.getY());
         p = n;
         return DRAG;
      }
      State release(MouseEvent e) {
         return UP;
      }
   };

   ...
}
```

Énumérations : utilisation avancée

Une **limitation** arbitraire **du langage** (contrairement aux *inner class*, les énumérations n'ont pas accès à la classe qui les englobe) oblige malheureusement à quelques **aménagements**.

Exemple

```
enum State {
   UP { ... }, CLIC_OR_DRAG { ... }, DRAG { ... };
   ...
   static OuterClass outer;
   static State init(OuterClass o) {
      outer = o;
      return UP;
   }
}

MouseInputListener listener = new MouseInputAdapter()  {
   State state = State.init(OuterClass.this);
   ...
};
```

Exemple

```
enum State {
   UP { ... },
   CLIC_OR_DRAG {
      State move(MouseEvent e) {
         if(p.distance(e.getPoint()) > D_DRAG) {
            o = outer.pick(p);
            return DRAG;
         }
         return CLIC_OR_DRAG;
      }
      State release(MouseEvent e) {
         outer.clic(p);
         return UP;
      }
   },
   DRAG { ... };
   ...
}
```

Exemple

```
enum State {
   UP { ... },
   CLIC_OR_DRAG { ... },
   DRAG {
      State move(MouseEvent e) {
         Point n = e.getPoint();
         outer.drag(o, n.getX()-p.getX(),
                       n.getY()-p.getY());
         p = n;
         return DRAG;
      }
      State release(MouseEvent e) {
         return UP;
      }
   };

   ...
}
```

Énumérations : utilisation avancée

Avantage :
alors que dans les ***listeners*** les **transitions** sont regroupées **par type d'événement**,
avec les **énumérations**,
elles sont regroupées **par état**.

2.2 Les composants interactifs et le modèle MVC

Les composants interactifs

Des objets graphiques interactifs :
les ***widgets*** *(window objects / window gadgets),*

embarquant :
• des **données** (état du widget) ;
• des **méthodes** (comportement) ; et
• un **protocole** d'interaction.

Les composants interactifs

Les *widgets* sont le cas d'école
de la programmation par objets :
1 *widget* = 1 objet (parfois plusieurs)

- **encapsulation** (portabilité) ;
- **réutilisation** (enrichissement par héritage) ;
- **modularité**
  (en particulier grâce au protocole embarqué).

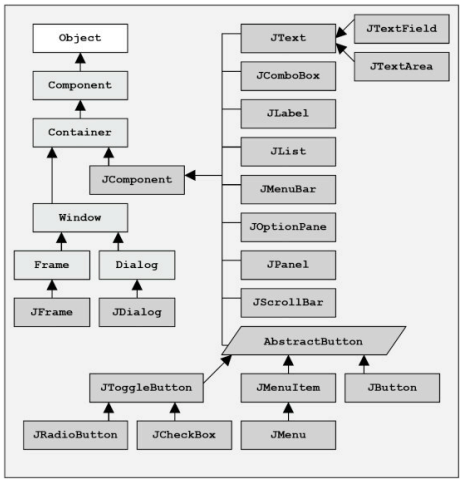
Les composants interactifs

Attention !

On va parler de **deux arbres** :
- l'arbre des classe, dû à **l'héritage**
  pour réutiliser le code ; et
- l'arbre des instances, dû à **l'imbrication**
  des *widgets* pour construire l'interface.

Les composants interactifs

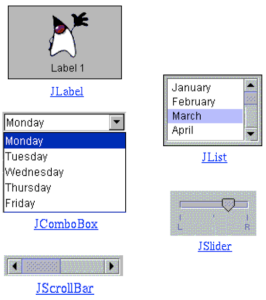
L'arbre (partiel)
d'héritage
des *widgets*
de Java/Swing



Les composants interactifs

Les ***widgets*** simples (fils directs de JComponent) :

- **JLabel**
- **JComboBox**
- **JScrollBar**
- **JSlider**, JSpinner
- **JList**, JTable, JTree
- JToolTip
- JSeparator
- JProgressBar



- *AbstractButton* (factorise les différents boutons)
- *JTextComponent* (factorise les *widgets* texte*)*

Les composants interactifs

Les **boutons** (fils d'AbstractButton) :

- **JButton**
- JToggleButton (bouton inverseur) qui factorise :
  - **JCheckBox**
  - **JRadioButton**
- **JMenuItem** qui regroupe :
  - JRadioButtonMenuItem
  - JCheckBoxMenuItem
  - JMenu

Les composants interactifs

Le **texte** (fils de JTextComponent) :

- **JTextField** (entrée **mono-ligne**) qui factorise :
  - JFormattedTextField
  - JPasswordTextField
- **JTextArea** (texte **multi-ligne** simple)
- **JEditorPane** (texte multi-ligne **formaté** txt/rtf/html)
  - JTextPane

Le modèle MVC [smalltalk, 1981]

Rappel des principes fondamentaux :
- **séparer** le noyau fonctionnel de l'interface ; et
- concevoir le noyau fonctionnel **indépendamment** d'une interface particulière.

Le modèle MVC [smalltalk, 1981]

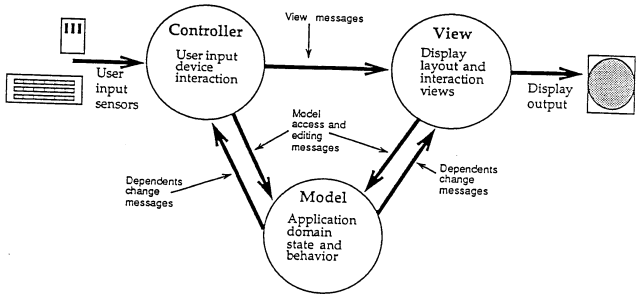


Figure 1: Model-View-Controller State and Message Sending

Le modèle MVC

Un agent MVC est composé de trois facettes réalisées par des objets :
- le **modèle** (*Model)*,
- la **vue** *(View)* et
- le **contrôleur** *(Controller).*

Les communications entre la **vue** et le **contrôleur** ne peuvent normalement se faire sans passer par le **modèle** qui est **garant de la cohérence** de l'état de l'agent.

Le modèle MVC

Le **modèle**
maintient son état et **notifie** de ses modifications.

Le modèle MVC

Le **modèle**
maintient son état et **notifie** de ses modifications.

Le **contrôleur**
écoute l'utilisateur et **demande** des modifications au modèle.

Le modèle MVC

Le **modèle**
maintient son état et **notifie** de ses modifications.

Le **contrôleur**
écoute l'utilisateur et **demande** des modifications au modèle.

La **vue**
écoute les notifications et **interroge** le modèle pour le représenter correctement.

Le modèle MVC

Attention :
MVC a été réutilisé dans le domaine des applications Web, mais il ne s'agit pas exactement du même modèle.

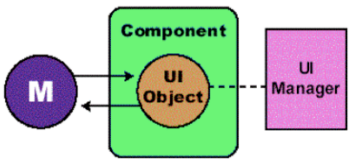
MVC à la Swing

Dans **Java/Swing**,
comme dans PAC [Coutaz, 1987],
la vue et le contrôleur sont **réunis** au sein d'une même facette (UIComponent),
ce qui résout le problème du couplage vue/contrôleur.

"Separable model architecture"

MVC à la Swing

Chaque JComponent encapsule un **UIComponent** qui peut être changé dynamiquement par un **UIManager**.



"Pluggable look-and-feel"

MVC à la Swing

"Pluggable look-and-feel" (plaf)

```
String lf = UIManager.getSystemLookAndFeelClassName();
// ou UIManager.getCrossPlatformLookAndFeelClassName(),
// "com.sun.java.swing.plaf.windows.WindowsLookAndFeel",
// "com.sun.java.swing.plaf.motif.WindowsLookAndFeel",
// "javax.swing.plaf.metal.MetalLookAndFeel",

UIManager.setLookAndFeel(lf);
SwingUtilities.updateComponentTreeUI(
                            SwingUtilities.getRoot(this));
```

MVC à la Swing

Chaque JComponent crée un **modèle**
qui peut être partagé avec un autre JComponent.

exemple :
JSlider et **JScrollBar** ont pour modèle
un **BoundedRangeModel**.

MVC à la Swing

```
// dans JSlider (et JScrollBar) on a :
public BoundedRangeModel getModel();
public void setModel(BoundedRangeModel model);


// synchroniser un slider et une scrollbar :
BoundedRangeModel model = new DefaultBoundedRangeModel() {
   public void setValue(int n) {
      System.out.println("setValue(" + n + ")");
      super.setValue(n);
   }
};

JSlider slider = new Slider();
slider.setModel(model);
JScrollBar scrollbar = new JScrollBar();
scrollbar.setModel(model);
```

MVC à la Swing

```
// ça marche aussi sans modèle explicite :
int v = slider.getValue();
slider.setValue(15);

// car dans JSlider on a :
public int getValue() {
   return getModel().getValue();
}

public void setValue(int v) {
   getModel().setValue(v);
}
```

2.3 Les conteneurs et les gestionnaires de géométrie

Les conteneurs

Les conteneurs sont des composants qui permettent de **réunir** d'autres composants grâce à une **imbrication** géométrique.

Gestion de l'imbrication

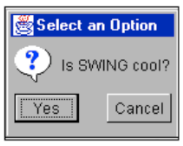
Les composants sont ajoutés à leur parent grâce à la méthode **add**.

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

class Main extends JFrame {
    Main() {
        add(new ContentPanel());
        pack();
    }

    public static void main(String[] argv) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() { new Main().setVisible(true); }
        });
    }
}
```

Les conteneurs externes

Les conteneurs **externes** 1/2
(fenêtres du système hôte) :

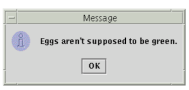
- JWindow (fenêtre sans décorations)
- **JFrame** (fenêtre avec décorations)
- **JDialog**

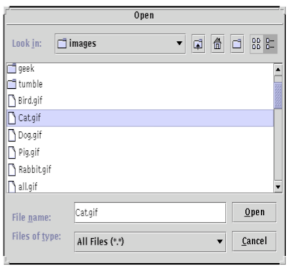


Les conteneurs externes

Les conteneurs **externes** 2/2
(boîte de dialogues préparées) :

- **JFileChooser**
- **JOptionPane**





Les conteneurs internes

Les conteneurs **internes** 1/2
permettent de **structurer** l'affichage
dans les fenêtres.

- JPanel
- **JViewport**
- **JScrollPane**
- **JSplitPane**
- **JTabbedPane**

- JTable, JTree

Les conteneurs internes

Les conteneurs **internes** 2/2
permettent de **structurer** l'affichage
dans les fenêtres.

- **MenuBar**
- AbstractButton
  - JMenuItem
    - **JMenu**
- **JPopupMenu**

- **JApplet**

Les gestionnaires de géométrie

Lorsqu'on ajoute des composants à un conteneur,
ils sont placés par son **gestionnaire de géométrie**.

Chaque conteneur a un gestionnaire de géométrie
par défaut qui peut être changé :

```
import java.awt.BorderLayout;
import javax.swing.JPanel;

public class Example extends JPanel {
   Example() {
      setLayout(new BorderLayout());
      ...
   }
}
```

Les gestionnaires de géométrie

Le gestionnaire de géométrie tient compte de la place **réclamée** par chaque fils pour accorder à chacun un rectangle.

Cette dimension préférée peut se choisir et se récupérer grâce à :

```
component.setPreferredSize(new Dimension(width,
                                         height));
component.getPreferredSize();
```

Le *BorderLayout*

Le **BorderLayout** permet au plus 5 fils, un par bord plus un central.

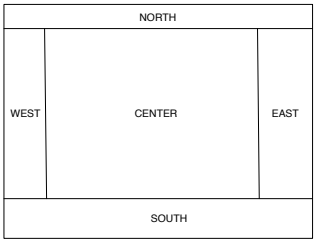
Il est approprié pour placer des **barres d'outils** ou d'état en périphérie d'une zone de travail centrale.

C'est le gestionnaire par défaut des **JFrame**.

Le *BorderLayout*

```
import java.awt.BorderLayout;
import javax.swing.JPanel;

public class Example extends JPanel {
   Example() {
      setLayout(new BorderLayout());
      add(new Toolbar(), BorderLayout.NORTH);
      add(new WorkArea()); // BorderLayout.CENTER
      pack();
   }
}
```

La **hauteur** préférée est utilisée pour NORTH & SOUTH, la **largeur** pour WEST & EAST. Les autres dimensions sont fixées par la taille du conteneur.



Le *FlowLayout*

Le **FlowLayout** place les fils horizontalement de gauche à droite en passant à la ligne lorsqu'il n'y a plus de place.

C'est le gestionnaire par défaut des **JPanel**.

Le *FlowLayout*

```
import java.awt.FlowLayout;
import javax.swing.JFrame;

public class Example extends JFrame {
   Example() {
      setLayout(new FlowLayout(FlowLayout.LEFT));
      add(new Button());
      add(new Button());
      add(new Button());
   }
}
```

Le *GridLayout*

Le **GridLayout** place les fils en grille.

Les cellules de la grille ont toutes la même taille (fixée par le conteneur, sans prendre en compte la taille préférée des fils).

On choisit le nombre de ligne (ou le nombre de colonnes) et les fils sont répartis en ligne.

Autres *layouts*

Le **BoxLayout** place les fils en grille en prenant en compte leur taille préférée.

Le **GridBagLayout** place les fils sur une grille non régulière et leur permet de se répartir sur plusieurs cellules (grâce à des **GridBagConstraint**, ardues à utiliser).

Le **SpringLayout** répartit les fils selon des contraintes (bords collés, ressorts ...).
Il est complexe à utiliser sans outils graphique de construction d'interface.