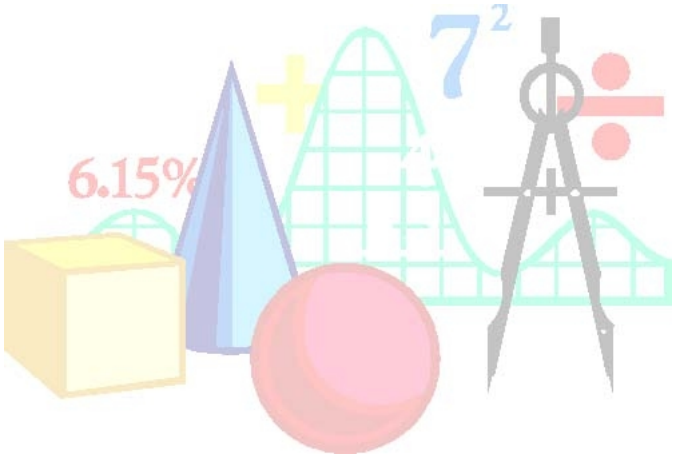
Swing

Les interfaces utilisateurs en Java



Avertissement

Ce document présente les principales entités graphiques nécessaires à la construction d'une interface utilisateur en Java. Les applications qui seront étudiées sont de type SDI (Simple Document Interface). Elles ne comporteront qu'une seule fenêtre incluant uniquement les contrôles simples, que l'on rencontre le plus couramment.

Certains passages de ce document ont pour origine

1. l'ouvrage *Swing La synthèse* de Valérie Berthié (chez Dunod)
2. le Guide le l'utilisateur Microsoft (Visual Basic)

Avant propos

Extrait du guide de l'utilisateur Microsoft

L'interface est peut-être la partie la plus importante d'une application mais elle est aussi celle que l'on remarque le plus. Pour les utilisateurs, elle représente l'application et certains ne doivent même pas savoir qu'un programme est exécuté en arrière-plan.

Lorsque le développeur.euse crée une application, il doit prendre certaines décisions concernant l'interface (le style d'interface simple document ou multi-documents, le nombre de feuilles différentes, les commandes à inclure dans les menus, les barres d'outils à prévoir pour reproduire des fonctions de menu ...etc) mais il doit avant tout tenir compte de l'utilisateur final. Une application destinée à des débutants exige une conception simple, alors que celle destinée à des utilisateurs expérimentés peut être plus complexe. Les autres applications dont se sert l'utilisateur peuvent aussi influencer ses attentes quant au comportement de l'application à développer. En outre, si l'application doit être diffusée en plusieurs langues, il ne faut pas oublier l'importance de la langue et de la culture.

La création d'une interface utilisateur doit être considérée comme un processus itératif; il est rare de parvenir à la perfection à la première tentative. Ce chapitre présente les composants nécessaires à la conception d'une application Swing, ainsi que le processus de création d'une interface.

Table des matières

Les bibliothèques graphiques de l'API

Il existe 2 bibliothèques graphiques dans l'API Java : l'Abstract Window Toolkit (l'AWT) et Swing.

L'Abstract Window Toolkit (l'AWT)

L'Abstract Window Toolkit est historiquement la première qui a été proposée, dès le JDK 1.0. La particularité de l'AWT, est que Java fait appel au système d'exploitation sous-jacent pour afficher les composants graphiques. Pour cette raison, l'affichage de l'interface utilisateur d'une application peut diverger sensiblement: chaque système d'exploitation dessine à sa manière un bouton. L''AWT garantira que la fonctionnalité recherchée sera dans tous les cas fournie mais elle sera présentée différemment.

Or Java se veut être 100% indépendant de la plate-forme utilisée. Pour cette raison, une nouvelle API a été définie

L'API Swing

Swing a donc été mis en place pour assurer 100% de portabilité (même un pixel doit avoir la même couleur). Pour assurer cette portabilité, un composant graphique est dessiné non plus par l'OS, mais par Java (ce qui en terme de temps d'exécution a un prix).

Swing est présenté comme étant écrit uniquement en Java. La machine virtuelle Java doit au moins être capable d'interagir avec l'OS pour pouvoir tracer un point. Or c'est ce que font les classes de base de l'AWT (telles que *Component*, *Container*, ...). En conséquence, tous les composants de Swing dérivent d'une classe de l'AWT.

Les mécanismes de gestion d'évènements et de positionnent des composants de l'AWT restent utilisable avec Swing. Certaines classes déjà utilisées avec l'AWT sont donc toujours d'actualité. Une application Swing doit importer des packages spécifiques mais aussi certains packages de l'AWT. Tout source utilisant Swing commence par:

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

Swing est bien plus riche que l'AWT. Même si un composant n'existe pas en natif sur l'OS, rien n'empêche de complètement le redessiner. Les classes qui font appel aux fonctions système de la machine sous-jacente sont appelées communément des composants lourds. Elles utilisent du code natif. C'est le cas d'un nombre réduit de classes comme celles qui conceptualisent les fenêtres. Les composants légers sont eux écrits en java pur. C'est le cas par exemple des contrôles qui ne sont pas dessinés par l'OS mais par du code Java. Ces contrôles auront toujours le même aspect et seront toujours disponibles quelle que soit la machine.

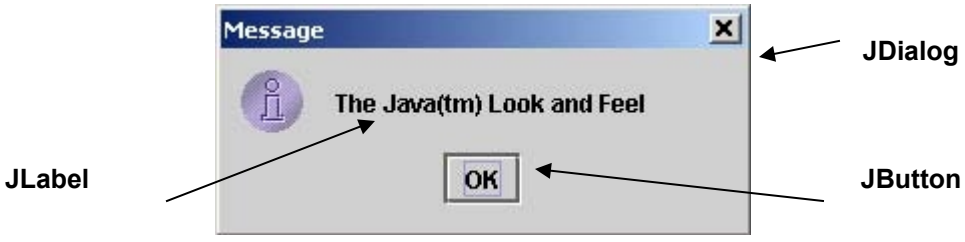
Hiérarchie des classes

La bibliothèque Swing est constituée d'une hiérarchie de classes. Chaque entité de l'interface graphique est associée à une classe inscrite dans une hiérarchie de classes liées par des liens d'héritage.

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
              |     |
              |     +--java.awt.Dialog
              |     |     |
              |     |     +--javax.swing.JDialog
              |     +--java.awt.Frame
              |           |
              |           +--javax.swing.JFrame
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JLabel
                    |
                    +--javax.swing.JList
```

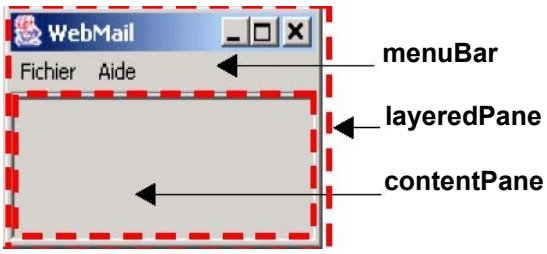
La classe **JFrame** est associée à la fenêtre principale de l'application. Elle comprend une barre de titre, une bordure permettant le re-dimensionnement les boutons standard de fermeture, d'agrandissement et de mise en icône. Elle peut contenir une barre de menu, une barre d'outils et/ou une barre d'état

La classe **JDialog** est associée à une boite de dialogue en général ouverte en mode modal ou non modal par la fenêtre application. Elle contient des contrôles tous dérivés directement ou indirectement de **JComponent** (liste, bouton, label, …).



L'une comme l'autre fenêtre est composée d'objets panneaux (au sens de la composition UML) instances d'une sous-classe de *Container*

Un premier panneau appelé *layeredPane* (de classe *JLayeredPane*) représente la zone utile de la fenêtre. Elle est elle-même composée d'une barre de menu (de classe *JMenuBar*) et une zone client appelée *contentPane* (de classe *JPanel*) conteneur pour les contrôles éventuels.



Architecture d'une application Swing

Proposition d'architecture

Une application graphique Java sera constituée d'au moins deux classes dont l'une sera dérivée des classes Swing.

Toute application sera constituée d'une:

Classe Application Contenant la méthode statique main, point d'entrée de l'application qui définit le "look and feel" et crée un seul objet instance de la classe application.

```
public class Application {

  public static void main(String args[])
  {
     try {
       UIManager.setLookAndFeel(
            UIManager.getCrossPlatformLookAndFeelClassName());
     }
     catch(Exception e) { }
     new Application ();
   }
```

Le constructeur de la classe crée un objet "Fenêtre principale", définit son état et éventuellement sa position à l'écran.

```
  public Application ()
  {
     FenetrePrincipale frame = new FenetrePrincipale();
     frame.setVisible(true);
  }
}
```

Classe Fenêtre Principale Classe dérivée de *JFrame*. C'est la fenêtre principale de l'application. *JFrame* est une classe représentant une fenêtre évoluée qui peut s'afficher sur le bureau. Elle possède une barre de titre et peut être réduite, mise en icône ou fermée par des boutons. Elle recevra des contrôles et possèdera éventuellement une barre

de menu ou une barre d'état. Cette classe aura toujours la structure suivante:

```
public class FenetrePrincipale extends JFrame {

   // Champs privés: Réferences d'objets contrôles, menus, …

   public FenetrePrincipale()
   {
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      // Définition de l'état initial de la fenêtre,
      initControles ();
   }

   private void initControls ()
   {
      // Définition de l'état initial des contrôles
      // Positionnement des controles
   }

   // Gestion des événements
}
```

Elle comportera des champs privés, un pour chaque contrôle (bouton, label, zone de texte, …) ou composant (barre de menu, barre d'état, …) qu'elle contiendra. Le constructeur définit l'état initial de la fenêtre principale (barre de titre, style de bordure ..) puis appèle une méthode privée qui initialise les objets éventuellement contenus.

Réflexions techniques

Programmation objet et interface graphique

Il est important de préciser qu'a toute entité graphique de l'interface utilisateur (fenêtre, menu, boutons, contrôles, …) sera associée un objet, instance d'une classe Swing. Tout appel de méthode sur un objet Swing agira sur l'état de l'objet et aura un effet visuel sur l'entité graphique associée.

Le look and feel

Le dessin d'un composant Swing est pris en charge par des classes java. Une application Swing peut donc avoir le même rendu graphique (*look and feel*) quel que soit le système d'exploitation de la plate-forme sur laquelle s'exécute l'application. C'est pour cette raison qu'une application java peut utiliser un contrôle même si celui-ci n'est pas disponible dans l'API graphique du système d'exploitation. Java gère un *Pluggable look and feel* (Le rendu graphique est pris en charge par des classes à part). Certains *L&F* ont été pré-définis comme celui spécifique aux applications java (*metal*) ou celui de Windows, de Motif et il est relativement aisé de re-définir son propre *L&F*

La méthode statique *main* de la classe application utilise la méthode statique *setLookAndFeel* de la classe *UIManager* pour imposer le *L&F* désiré. Ci-dessous les méthodes imposent le *L&F metal*

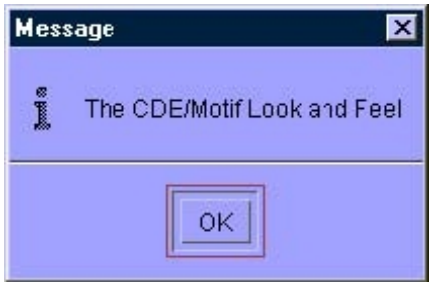
```
UIManager.setLookAndFeel(
                UIManager.getCrossPlatformLookAndFeelClassName());
JFrame.setDefaultLookAndFeelDecorated(true) ;
```

Il est possible aussi de faire en sorte qu'une application prenne automatiquement le *L&F* de la plate-forme sur laquelle elle s'exécute.

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

Sous réserve qu'il soit disponible, rien ne s'oppose à choisir un *L&F* quelconque (Motif …)

```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```



Un changement de *L&F* n'impacte ni la nature, ni le positionnement des composants, mais seulement le rendu visuel des composants

Ouverture et fermeture de la fenêtre

La création d'une instance de la classe FenetrePrincipale a pour effet de créé un objet fenêtre en mémoire. Cette fenêtre n'est pas visible. C'est le rôle de la méthode `setVisible(true)` de la rendre visible. Par défaut, une demande de fermeture de la fenêtre rend la fenêtre invisible. La méthode `setDefaultCloseOperation()` modifie le mode de fermeture. Avec l'argument `EXIT_ON_CLOSE` la fermeture de la fenêtre provoque la terminaison de l'application.

Quelques méthodes utilisables

Le constructeur de la classe FenetrePrincipale définit l'état de la fenêtre application. Quelques méthodes sont souvent utilisées. Par exemple:

`setSize()`	Fixe la taille de la fenêtre
`setTitle()`	Définit le libellé de la barre de titre
`setCursor()`	Impose un curseur particulier
`setLocation()`	Positionne la fenêtre par rapport au bureau
`setResizable()`	Autorise ou non à retailler la fenêtre

Première application

La première application contient un *JPanel* (zone rectangulaire dans laquelle il est possible de dessiner) dont la couleur de fond sera blanche. Elle occupe toute la partie utile (zone client) de la fenêtre principale. Dans cette zone graphique l'application devra dessiner un carré jaune et un cercle rouge.

Soit une classe zone de dessin. Une zone de dessin est un *JPanel* avec un rectangle jaune et un cercle rouge.

```
public class ZoneDessin extends JPanel
{
   public ZoneDessin ()
   {
      this.setBackground(Color.white);
   }

}
```

La fenêtre application doit créer un objet ZoneDessin et l'ajouter `add()` dans la zone client. La méthode `getContentPane()` permet d'obtenir une référence sur la zone client.

```
public FenetrePrincipale()
{
   this.setSize(300,200);
   this.setTitle("Application de dessin");
   this.setDefaultCloseOperation(EXIT_ON_CLOSE);

   JPanel zoneClient = (JPanel) this.getContentPane();
   zoneClient.add (new ZoneDessin ());
}
```

Chaque fois que la fenêtre application change de taille, ou qu'une fenêtre se superpose sur la fenêtre application, la fenêtre et toutes les entités graphiques qu'elle contient doivent être re-dessinée. Lorsqu'une entité est re-dessinée, Swing appèle la méthode `paintComponent()` sur l'objet associé. Cette méthode va être re-définie dans la classe ZoneDessin pour que soient effectuées les opérations de dessin spécifiques

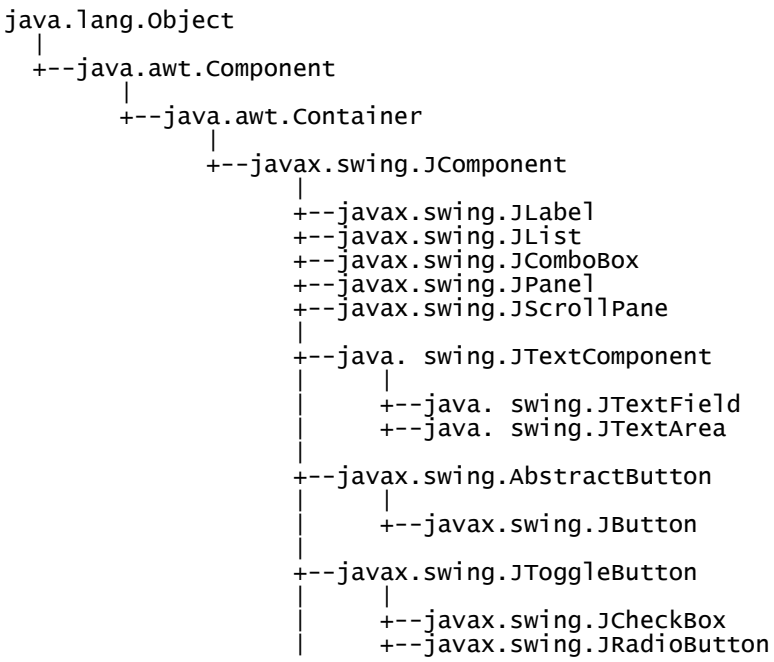
```
public class ZoneDessin extends JPanel
{
   . . .

   public void paintComponent(Graphics g)
   {
      super. paintComponent (g);
      g.setColor(Color.yellow);
      g.fillRect(5,5,30,30);
      g.setColor(Color.red);
      g.fillOval(15,15,30,30);
   }
}
```

Les opérations de dessins sont des actions de bas niveau. Il n'existe donc aucune méthode permettant de dessiner directement dans un composant graphique. Ces opérations ne seront possibles que via un "contexte graphique" associé au composant et géré par Swing. La méthode `paintComponent()` le fourni en argument. L'argument g de classe *Graphics* va permettre tous les

`setColor()`	Fixe la couleur du tracé
`drawRect()`	Trace un rectangle (ou un carré) vide
`drawLine()`	Trace une ligne
`drawOval()`	Trace un ovale (ou un cercle)
`drawImage()`	Affiche une image
`drawString()`	Affiche du texte
`fillRect()`	Trace un rectangle plein
`fillOval()`	Trace un ovale plein
`setFont()`	Définit la fonte

Les composants Swing

Le but de ce chapitre est de présenter rapidement les principaux **composants** Swing désignant les **contrôles** qui permettent une interaction classique entre l'opérateur et le logiciel et les **panneaux** qui regroupent ou contiennent des contrôles. Chaque composant est associé à une classe spécifique placée dans la hiérarchie des classes Swing. Chaque composant de l'Interface Utilisateur de l'application sera un objet, instance d'une de ces classes sur lequel il sera possible d'appliquer un grand nombre de méthodes (définies en direct dans la classe ou héritées des classes parents).



Dans une application, ces composants vont être posés sur la zone client de la fenêtre principale, ils seront en très grand nombre. Ils correspondront parfois à des objets locaux dans des méthodes de la classe `FenetrePrincipale` et parfois à des champs de cette même classe. Il est conseillé de respecter une convention de nommage qui permette, dans le source java, de différencier du premier coup d'œil, les composants des objets "applicatifs". Le nom de chaque composants est préfixé par 3 lettres qui rappèlent sa classe (donc sa nature).

JLabel	Label ou image	**lbl**xxx, **pic**xxx	lblAuteur, picLogo
JTextField, JTextArea	Zone de saisie de texte	**txt**xxx	txtNom, txtPrenom
JButton	Bouton simple	**cmd**xxx	cmdAnnuler, cmdOK
JCheckBox	Case à cocher	**chk**xxx	chkValide
JRadioButton	Bouton radio	**opt**xxx	optArial, optTMS
JList	Liste déroulante	**lst**xxx	lstAgence
JComboBox	Boîte combo	**cbo**xxx	cboVille

Tour d'horizon des contrôles simples

JLabel - Le label

Un label permet d'afficher un texte fixe et/ou une image. Il ne réagit pas aux sollicitations de l'opérateur. Il est purement décoratif et sert à décrire les autres contrôles de l'interface utilisateur. Il correspond à un objet instance de la classe *JLabel* dont le texte est fixé à l'appel du constructeur

```
JLabel lblDescr = new JLabel ("Label simple");
```

Le texte peut-être modifié à l'exécution


```
lblDescr.setText ("Label simple avec image");
```

Une image peut être ajoutée


```
lblDescr.setIcon(new ImageIcon("boulerouge.gif"));
```

Nota: L'image et le libellé peuvent être définis directement lors de l'appel du constructeur

Ce même contrôle peut servir à afficher une image seule

```
JLabel logo = new JLabel (new ImageIcon("javalogo.gif"));
```

Le texte peut-être au format HTML


```
JLabel lblDescr = new JLabel ();
lblDescr.setText ("<html>Mon label <p>a <I>moi</I></html>");
```

JTextField – La zone de texte simple

Ce contrôle permet la saisie d'informations de type texte sur une seule ligne.

Il est possible depuis l'application de lire l'information tapée par l'opérateur


```
TextField txtNom = new TextField();
String valeur = txtNom.getText();
```

ou de forcer une valeur à afficher

```
txtNom.setText("Jean DUBOIS");
```

ou de changer les attributs de la zone de texte

```
txtNom.setColumns(15);     // Fixe la longeur de la zone de saisie
txtNom.setEditable(false); // Empeche l'écriture dans la zone
```

Notons que la classe *JPasswordField* dérivée de *JTextField* est plus appropriée pour assurer la saisie des mots de passe.

JButton – Le bouton

Le bouton permet de déclencher un traitement. Il possède un libellé et éventuellement une icône et un accélérateur.

Bouton simple

```
JButton cmdDepart1 = new JButton ("Départ");
```

Bouton initialement grisé, avec image

```
JButton cmdDepart2 = new JButton ("Départ");
cmdDepart2.setIcon(new ImageIcon("boulerouge.gif"));
cmdDepart2.setEnabled(false);
```

Bouton avec icône et accélérateur (Alt+D simule un clic sur le bouton)

```
JButton cmdDepart3 = new JButton("Départ");
. . . ;
cmdDepart3.setMnemonic('D');
```

JCheckBox – La case à cocher

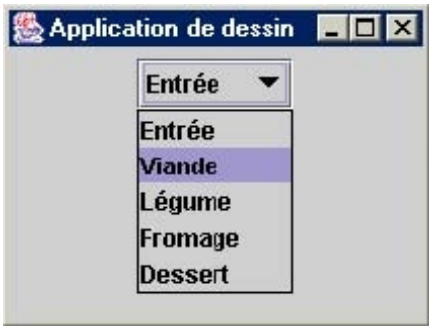
La case à cocher permet à l'opérateur de sélectionner une option vraie/fausse ou active/inactive. Elle est constituée d'une case et d'un libellé. Le libellé et l'état initial peuvent être définis lors de l'appel du constructeur à la création de l'instance.


```
JCheckBox chkOpt1 = new JCheckBox("Option #1");
JCheckBox chkOpt2 = new JCheckBox("Option #2");
chkOpt2.setSelected(true);
```

Des méthodes peuvent être respectivement utilisées pour cocher/décocher la case ou connaître son état. Il s'agit de `setSelected()` et `isSelected()`

JComboBox – La boîte combo

La boîte combo permet de proposer une liste de valeurs possibles au sein desquelles l'opérateur effectue son choix. Elle est composée d'un bouton ou d'une zone de texte associée à une liste déroulante qui apparaît sur demande de l'opérateur. Si la boîte est de type "éditable" la zone de texte permet à l'opérateur d'effectuer un choix autre que ceux proposés



Boîte combo simple

```
      String  []  choix  =  {"Entrée",  "Viande",
"Légume",
                      "Fromage", "Dessert"};


   JComboBox cboChoix = new JComboBox ();
   for (int i = 0; i<choix.length; i++)
                         cboChoix.addItem(choix[i]);
```

Boite combo éditable (3 éléments maximum visibles)

```
   cboChoix = new JComboBox ();
   . . . ;
   cboChoix.setEditable(true);
   cboChoix.setMaximumRowCount(3);
```

L'application peut faire évoluer le nombre d'éléments de la liste par les méthodes `removeItem()`, `removeItemAt()` et `addItem()`

L'application peut connaître le nombre d'éléments de la liste et savoir quel élément a été sélectionné par les méthodes `getItemCount()` et `getSelectedItem()`

Ajout des contrôles à la fenêtre application

Ce chapitre propose une démarche rigoureuse pour coder correctement la classe FenetrePrincipale. Celle qui est proposée ici est librement inspirée de l'approche utilisée par les outils RAD du marché actuel (JBuilder, NetBean, SunOne ou Forte).

Règle 1 Tous les contrôles de l'interface seront une instance d'une classe Swing

Règle 2 Tous les composants dont l'aspect varie ou qui seront sollicités par l'opérateur seront définis comme des champs privés de la classe FenetrePrincipale

Règle 3 Les composants doivent être crées et initialisés en même temps que la fenêtre principale, c'est la raison pour laquelle les initialisations sont regroupées dans une méthode privée `initControles()` appelée par le constructeur de la classe.

La classe aura la structure suivante

```
public class FenetrePrincipale extends JFrame
{
   // Champs privés: Référence vers les contrôles et instanciation

   public FenetrePrincipale ()
   {
      // Initialisation de la fenêtre
      initControles();
   }


   private void initControles ()
   {
      // Définition de l'état initial des contrôles
   }
}
```

Exemple d'application

Cette application dispose de 3 contrôles: un label purement décoratif qui ne variera jamais, une boîte combo (on souhaite connaître la valeur choisie) et un bouton simple (qui déclenchera par la suite une action).

Déclaration d'une référence et création de 2 objets pour les contrôles "actifs"

```
private JButton cmdOK = new JButton("OK");
private JComboBox cboChoix = new JComboBox();
```

Les 3 contrôles vont être ajoutés dans la zone client de la fenêtre principale. Il faut donc obtenir une référence vers cette zone client et lui ajouter (*add*) 3 instances, l'une anonyme (le label) et les 2 autres explicites (combo et bouton). Les 3 contrôles seront "organisés" l'un à côté de l'autre par un **layout manager** qui définit la politique d'organisation des contrôles (ceci sera étudié dans le chapitre suivant acceptons-le pour le moment).

```
private void initControles()
{
   JPanel zoneClient = (JPanel) this.getContentPane();
   zoneClient.setLayout(new FlowLayout());
```

Le label est un objet anonyme ajouté à la zone client

```
zoneClient.add( new JLabel("Votre nom"));
```

Initialisation et ajout à la zone client d'une boîte combo éditable

```
String [] choix = {"DUPOND Jean","DOE John","MARTIN Marie","DURAND Jules"};
for (int i = 0; i<choix.length; i++) cboChoix.addItem(choix[i]);
cboChoix.setEditable(true);
cboChoix.setMaximumRowCount(3);
zoneClient.add (cboChoix);
```

Initialisation et ajout à la zone client d'un bouton graphique

```
cmdOK.setIcon(new ImageIcon("boulerouge.gif"));
cmdOK.setMnemonic('O');
zoneClient.add (cmdOK);
```

```
}
```

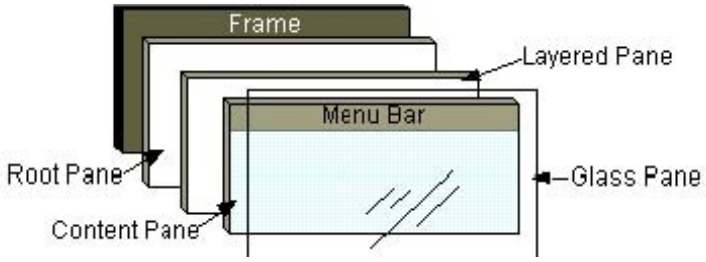
Les panneaux

Un panneau est un conteneur (ou containers), composant particulier, en général transparent (il est possible de changer sa couleur de fond), chargé de contenir des contrôles. Ils servent en général à

1• regrouper des contrôles pour mieux les positionner dans une fenêtre
2• regrouper visuellement des contrôles
3• ajouter des barres de défilement aux composants "scrollables"

Les panneaux déjà rencontrés – *JFrame et contentPane*

La fenêtre principale pouvant regrouper des contrôles est elle-même un conteneur. En fait, à regarder de plus près elle est composée de plusieurs panneaux superposés. Tout d'abord un panneau de classe *JLayeredPane* qui recouvre toute la surface de la fenêtre. Sur ce dernier sont posés 2 autres panneaux: la barre de menu (*JMenuBar*) et la zone client (Content Pane) sur laquelle sont posés les contrôles.



Pour information, le tout est recouvert d'un panneau transparent (Glass Pane) que l'on n'utilisera pas. L'ensemble de ces 4 panneaux forme un *JRootPane*.

Regroupement des contrôles - *JPanel*

La méthode `getContentPane()` fournit une référence sur la zone client. C'est un objet instance de la classe *JPanel*.

Il est possible de regrouper les contrôles dans des panneaux, eux aussi de classe *JPanel* soit pour encadrer graphiquement un groupe de contrôles, soit pour positionner globalement un groupe de contrôles comme nous le verrons dans le prochain chapitre. Le principe est simple: il faut ajouter les contrôles aux panneaux puis ajouter les panneaux à la zone client.

Le bouton et la boîte combo sont toujours des contrôles sollicités par l'opérateur donc:

```
private JButton cmdOK = new JButton("OK");
private JComboBox cboChoix = new JComboBox();
```

Le label et la boîte combo sont réunis dans un panneau avec bordure et titre. La méthode privée initControles() contiendra:

```
private void initControles()
{
   . . . ;
```

1) l'initialisation éventuelle de l'aspect des contrôles

```
   String [] choix = {"DUPOND Jean","DOE John","MARTIN Marie","DURAND Jules"};
   for (int i = 0; i<choix.length; i++) cboChoix.addItem(choix[i]);
   cmdOK.setIcon(new ImageIcon("boulerouge.gif"));
```

2) la création d'un panneau avec bordure (objet local)

```
   JPanel pan = new JPanel();
   pan.setBorder(BorderFactory.createTitledBorder("Choisissez un nom"));
```

3) l'ajout de la boîte combo et du label au panneau

```
   pan.add( new JLabel("Nom"));
   pan.add (cboChoix, null);
```

4) puis enfin l'ajout du panneau, du logo et du bouton à la zone client

```
   zoneClient.add ( new JLabel(new ImageIcon("javalogo.gif")));
   zoneClient.add (pan);
   zoneClient.add (cmdOK);
}
```

Les autres panneaux

JScrollPane est un conteneur permettant d'ajouter des barres de défilement à un seul contrôle "scrollable" (implémentant l'interface *Scrollable*) comme une liste (*JList*) et une zone de texte multi-lignes (*JTextArea*). Voir paragraphe suivant.

JTabbedPane est un groupe de panneaux qui se superposent et que l'on peut mettre individuellement au premier plan avec des onglets.

JSplitPane est un panneau séparable en 2 zones horizontales ou verticales dont la surface peut varier dynamiquement, la surface totale restant constante.

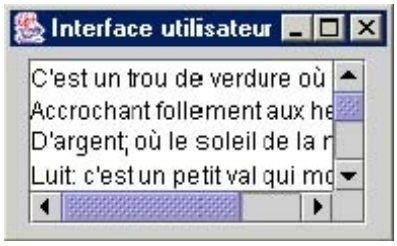
JDesktopPane est un conteneur de haut niveau se comportant comme un bureau de système d'exploitation. Il contient des fenêtres avec barre de titre et bouton d'agrandissement qui évolue en son sein. Ce conteneur permet de réaliser des applications MDI (Multiple Documents Interface).

Les contrôles toujours associés à un panneau

JTextArea – La zone de texte multi-lignes

Ce contrôle ressemble à la zone de texte simple, il permet en plus à l'opérateur de saisir un texte long sur plusieurs lignes. Le texte peut même être plus important que ce qui est visible. Dans ce cas le défilement est pris en charge par un *JScrollPane*..

Le contrôle de classe *JTextArea* est ajouté à un panneau de classe *JScrollPane* qui est lui-même ajouté à la zone client. La zone de texte sera certainement sollicitée par l'opérateur donc:




```
private JTextArea txtMsg = new JTextArea ();
```

La méthode privée initControles() contiendra:

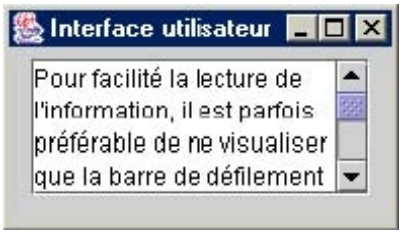
11) l'initialisation de la zone de texte (nombre de lignes et de colonnes visibles)

```
txtMsg.setColumns(10);
txtMsg.setRows(3);
```

12) La création d'un panneau permettant le défilement et l'association de la zone de texte. Puis l'ajout de ce panneau à la zone client.

```
JScrollPane panMsg = new JScrollPane(txtMsg);
zoneClient.add(panMsg);
```

Pour facilité la lecture de l'information, il est parfois préférable de ne visualiser que la barre de défilement verticale, le texte, horizontalement, étant automatique-ment renvoyé à la ligne (`setLineWrap`) avec une coupure entre 2 mots (`setWrapStyleWord`)




```
txtMsg.setLineWrap(true);
txtMsg.setWrapStyleWord(true);
```

JRadioButton – Les boutons radio

Un bouton radio n'est jamais seul. Une application dispose toujours d'un groupe de N boutons radio permettant de choisir 1 option parmi N (un seul peut être sélectionner à la fois). Il y a 2 concepts important à retenir:

Les objets boutons radio (*JRadioButton*) sont toujours ajoutés à un groupe de boutons (*ButtonGroup*) pour assurer la dynamique de sélection

Les boutons sont visuellement regroupés, c'est à dire qu'ils sont souvent placés dans un panneau avec titre et bordure

Les boutons radio seront sollicités par l'opérateur, donc

```
private JRadioButton optPomme = new JRadioButton("Pomme");
private JRadioButton optOrange = new JRadioButton("Orange");
private JRadioButton optCitron = new JRadioButton("Citron");
```

La méthode privée initControles() contiendra:

1) l'initialisation des boutons radio

```
optPomme.setSelected(true); // Sélctionné par défaut
optPomme.setMnemonic('P');  // Raccourcis clavier
optOrange.setMnemonic('O');
optCitron.setMnemonic('C');
```

2) L'ajout des boutons radio au groupe pour assurer la dynamique (1 parmi N)

```
ButtonGroup grpOption = new ButtonGroup();
grpOption.add (optPomme);
grpOption.add (optOrange);
grpOption.add (optCitron);
```

3) L'ajout des boutons radio à un panneau avec bordure et titre, puis ajout de celui-ci à la zone client

```
JPanel panOption = new JPanel ();
panOption.setBorder(BorderFactory.createTitledBorder("Fruit possible"));
panOption.add (optPomme);
panOption.add (optOrange);
panOption.add (optCitron);

zoneClient.add (panOption);
```

La méthode `isSelected()` permet de savoir si un bouton radio est ou non sélectionné.

JList – La liste simple

JList est un contrôle qui permet d'afficher une liste de N éléments, l'opérateur pouvant choisir un ou plusieurs éléments parmi N suivant le mode de sélection. Souvent le nombre d'éléments de la liste est plus grand que le nombre d'éléments montrés, c'est la raison pour laquelle une liste simple est en général associée à un panneau de classe *JScollPane*.

La liste simple sera toujours sollicitée par l'opérateur donc:

```
private JList lstNom = new JList();
```

La méthode privée `initControles()` contiendra:

11) l'initialisation des éléments de la liste, le nombre d'éléments visibles et la sélection par défaut du premier élément

```
String [] noms = {"DUPOND Jean","DOE John","MARTIN Marie","DURAND Jules"};
lstNom.setListData(noms);     // Association de la liste au tableau
```

```
lstNom.setVisibleRowCount(3); // Nb d'éléments visibles
lstNom.setSelectedIndex(1);   // Selection par défaut
```

En fait le tableau de chaînes est le "modèle" de la liste. La liste est une vue (ou vision) du modèle. La méthode `setListData()` associe le modèle à la vue. Le "modèle" de la liste peut aussi être un objet instance de la classe *Vector.* Ce mode d'utilisation de la liste ne peut convenir que dans le cas où les éléments de la liste ne changent pas.

12) La création d'un panneau permettant le défilement et l'association de la liste. Puis l'ajout du panneau à la zone client.

```
JScrollPane panNom = new JScrollPane(lstNom);
zoneClient.add(panNom);
```

Ajout ou retrait d'éléments

Lorsque le contenu de la liste varie dynamiquement il est préférable d'associer à la liste un "modèle par défaut" chargé du stockage des données. Ce modèle est une instance de la classe *DefaultListModel* qui implémente un *Vector*. Enlever/supprimer un élément du modèle aura un effet sur le contrôle *JList* associé (la vue).

Création de la liste et de son modèle. Le modèle est connecté à la vue: l'ajout d'un élément au modèle sera visualisé dans la liste:

```
DefaultListModel listData = new DefaultListModel(); // Création du modèle
JList lstMess = new JList(listData);  // Association du modèle à la liste
```

Association d'un panneau permettant le défilement (à la vue):

```
JScrollPane panTable = new JScrollPane (lstMess);
zoneClient.add(panTable, BorderLayout.CENTER);
```

Ajout dynamiquement d'un élément (au modèle):

```
listData.addElement("www.afpa.fr");
```

Suppression (au modèle) de l'élément sélectionné (dans la vue):

```
int idx = lstMess.getSelectedIndex();
if (idx!=-1) listData.remove(idx);
```

Mode de sélection

La méthode `setSelectionMode()` permet de définir le mode de sélection. Elle reçoit en argument une des 3 constantes (champs publics statiques) de l'interface *ListSelectionModel.*

SINGLE_SELECTION — Un seul élément peut être sélectionné à la fois. La sélection d'un élément entraîne la désélection du précédent.

SINGLE_INTERVAL_SELECTION	Sélection multiple d'éléments contigus. La sélection s'effectue à l'aide de la touche Shift.	
MULTIPLE_INTERVAL_SELECTION	Sélection multiple de n'importe quelle combinaison d'éléments. La sélection s'effectue à l'aide des touches Shift et Ctrl. C'est le mode par défaut.	

Par exemple, l'instruction ci-dessous force le mode de sélection simple:

```
lstNom.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

`getSelectedIndex()` et `getSelectedValue()` peuvent être utilisées sur une liste simple pour connaître l'élément sélectionné par l'opérateur. Pour une liste à sélection multiple il faut utiliser `getSelectedIndices()` et `getSelectedValues()`.

JTable – Utilisation simple d'une table

Une table ressemble à une liste avec plusieurs colonnes. C'est un objet instance de la classe *JTable* qui est associé à un modèle, instance de classe *DefaultTableModel*.

De	Email	Sujet	Taille
JC Rigal	jcrigal@fre...	hello world	12
tsaiil	tsaii@free.fr	Eupression	1
Pierre CUR...	pierre.curie...	info	2
afpa	dr.neuilly@...	Retour cou...	5

Création d'une table associée à un modèle de 4 colonnes et 0 ligne:

```
String [] cols = {"De", "Email", "Sujet", "Taille"};
DefaultTableModel listData = new DefaultTableModel(cols, 0);
JTable tblMess = new JTable(listData);
```

Association d'un panneau permettant le défilement:

```
JScrollPane panTable = new JScrollPane (tblMess);
zoneClient.add(panTable, BorderLayout.CENTER);
```

Ajout dynamiquement d'une ligne au modèle (la table enregistre la modification automatiquement)

```
Object [] mess = {"JC Rigal", "jcrigal@free.fr",
                  "hello world", new Integer(12)};
listData.addRow(mess);
```

Suppression de la ligne sélectionnée

```
int idx = tblMess.getSelectedRow();
if (idx!=-1) listData.removeRow(idx);
```

JTabbedPane – Le contrôle à onglets

Ce contrôle sert à superposer des panneaux, qu'il sera possible de mettre au premier plan avec des onglets, souvent placés en haut. La mise en œuvre consiste à :

11. créer autant de panneaux, instances de la classe *JPanel,* qu'il est nécessaire
22. ajouter à chacun d'eux, leurs contrôles
33. "accrocher" les panneaux au contrôle à onglets
44. ajouter le contrôle à onglets à la fenêtre principale

```
private JTabbedPane tabPrincipal = new JTabbedPane ();

private void initControles()
{
   JPanel zoneClient = (JPanel) this.getContentPane();

   JPanel panInfo = new JPanel();     <1>
   panInfo.add(new JLabel("Votre nom"));   <2>
   panInfo.add(new JTextField(10));
   tabPrincipal.addTab("Informations", panInfo);  <3>

   JPanel panErreur = new JPanel();     <1>
   panErreur.add(new JLabel("5 Erreurs"));   <2>
   tabPrincipal.addTab("Erreurs", panErreur);  <3>

   zoneClient.add(tabPrincipal);     <4>
}
```

Propriétés communes des composants

La classe *JComponent* regroupe les propriétés communes à tous les composants Swing. Tous ces composants ont donc un comportement commun

1• Il est possible de les rendre actifs ou inactifs (on dit aussi grisés), autorisant, ou non, les actions de l'opérateur. Cet état est fixé par la méthode `setEnabled()`
2• Les contrôles peuvent être visibles ou invisibles. Cet état est fixé par la méthode `setVisible()`
3• La couleur du texte et la couleur du fond d'un composant peuvent être imposées par, respectivement `setForeground()`et `setBackground()`. L'une et l'autre méthode reçoivent comme argument une constante définie dans la classe *Color* (`Color.red`, `Color.green`, …). Ces 2 méthodes ne doivent être utilisées que dans des cas exceptionnels. Tous les composants ont des couleurs de fond (gris, blanc ou transparent) et de texte (noir) standard que les utilisateurs connaissent bien. Evitons de changer leurs habitudes.
4• Il est souvent utile d'associer aux contrôles une bulle d'aide (*tool tip*) qui apparaît lorsque l'opérateur laisse le pointeur de la souris sur le composant. La méthode `setToolTipText()` permet de définir la bulle d'aide. Elle reçoit en argument le libellé de la bulle.
5• 3 méthodes permettent d'assurer les dimensions et/ou la position d'un composant. Il s'agit de `setLocation()`, `setSize()` et `setBounds()`
6• Le focus peut être donné à un contrôle par la méthode `requestFocus()`. Lorsqu'un contrôle à la focus toutes les actions clavier sont interceptées par le contrôle.

Agencement des contrôles

Visuellement, une interface graphique est une fenêtre (*JFrame*) conteneur, sur laquelle est placée une combinaison de contrôles (*JTextArea*, *JLabel*, …) et de panneaux eux aussi conteneurs pour d'autres contrôles et panneaux. Un problème important que doit résoudre le développeur.euse est le positionnement de ces composants les uns par rapport aux autres. Il doit aussi définir l'agencement visuel des composants par les conteneurs.

Cette particularité, en Java, consiste à associer aux conteneurs un **layout manager** particulier, objet chargé d'assurer une stratégie de positionnement. Notons bien qu'un composant est contenu dans un conteneur et que ce dernier s'attache les services d'un **layout manager** pour le positionnement dont il a la charge. Les différentes stratégies de positionnement prennent en compte la taille et la position des composants qu'il lui sont confiées. Le layout manager va essayer de fixer la taille réelle (*size*) du composant à une valeur la plus proche possible de sa taille idéale (*preferedSize*)

JLabel	La taille du texte du label et/ou éventuellement de son image
JTextField	Le contenu du texte ou le nombre de colonnes imposé par `setColumns()`
JTextArea	Le contenu du texte ou le nombre de lignes imposé par `setRows()` et de colonnes imposé par `setColumns()`
JButton	La longueur du libellé et/ou éventuellement de son image
JCheckBox	La taille du texte + la taille de la coche
JRadioButton	La taille du texte + la taille du bouton radio
JList	La taille du plus grand élément et le nombre d'éléments. Si elle est scrollable ce sera le nombre d'éléments visibles imposé par `setVisibleRowCount()`
JComboBox	La taille du plus grand élément et le nombre d'éléments visibles dans la fenêtre pop-up imposé par `setMaximumRowCount()`
JPanel	La taille idéale des composants qu'il contient

Ce mode de positionnement des contrôles est volontairement différent de celui proposé par les outils RAD sous Windows (comme Visual C++, Visual Basic, ..) où les contrôles ont une taille et un positionnement fixe et absolu définit par l'application. Ce dernier mode, bien qu'il existe aussi avec Swing, génère parfois des problèmes d'alignement ou de répartition des espaces. Ceci est dû aux caractéristiques de l'écran de la machine, à la taille de la fenêtre application, à la configuration de la carte graphique …Les layout managers garantissent la "portabilité visuelle" de l'interface utilisateur de l'application quelles que soient les caractéristiques de la machine et du système d'exploitation.

Les différentes stratégies de positionnement

L'impact des layout managers sur le code est très faible. Associer un layout manager particulier à

un conteneur se code toujours de la façon suivante:

```
objetContainer.setLayout (new xxxLayout( . . . ));
```

Ce type d'instruction sera visible par exemple dans le constructeur de la classe fenêtre sous la forme

```
this.getContentPane().setLayout(new FlowLayout());
```

ou dans la méthode initControles() de la classe fenêtre, associée à la zone client

```
zoneClient.setLayout(new FlowLayout());
```

ou à un panneau de classe *JPanel*

```
panBas.setLayout(new BorderLayout());
```

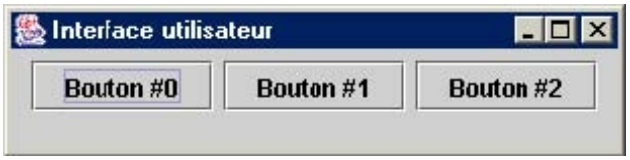
Les 3 layout managers de base qui vont être étudiés ont été définis dans le package *java.awt.* Ils peuvent être utilisés sans problème avec des composants Swing.

FlowLayout

Ce type de layout manager est principalement utilisé pour organiser le positionnement des boutons. C'est le layout manager par défaut des objets panneaux (*JPanel*). Il range les composants sur une ligne de la gauche vers la droite au fur et à mesure de l'ajout au conteneur, en utilisant leur taille idéale (*preferedSize*). Si un composant ne peut être placé, il est mis à la ligne. Les composants d'une même ligne sont centrés (par défaut) par rapport au conteneur. Si les dimensions du conteneur sont modifiées, les composants sont réarrangés.

Soit le code suivant dans la méthode initControles():

```
JPanel zoneClient = (JPanel) this.getContentPane();
zoneClient.setLayout(new FlowLayout());
for (int i = 0; i<3; i++)zoneClient.add (new JButton("Bouton #" + i));
```



Avec un tel layout, les composants auront toujours leur taille idéale. Si le conteneur est trop petit le layout "tronquera" les composants, si le conteneur est trop grand il agrandira les espaces au début et à la fin de chaque ligne.

La seule possibilité de paramétrage de ce type de layout est l'alignement à gauche, à droite ou au centre (par défaut) des composants qu'il lui sont confiés.

Le constructeur reçoit en argument une constante définissant le type d'aligne-ment.


```
zoneClient.setLayout(
  new FlowLayout(FlowLayout.LEFT));
```

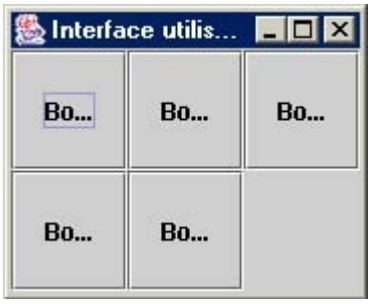
La constante `FlowLayout.RIGHT` peut utilisée p
aligner à droite

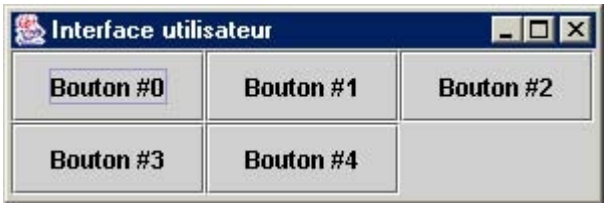
La distance par défaut inter-composant peut aussi être modifiée (*Hgap* et *Vgap*) grâce au 3ème constructeur

```
zoneClient.setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));
```

GridLayout

Ce layout manager organise le conteneur comme une grille de L lignes et C colonnes suivant les arguments du constructeur. La surface du conteneur est découpée en LxC rectangles égaux. Les composants utilisent tous cette surface élémentaire sans tenir compte de leur dimension idéale. L'ordre d'ajout est important; les composants sont ajoutés de gauche à droite et de haut en bas.






```
JPanel zoneClient = (JPanel) this.getContentPane();
zoneClient.setLayout(new GridLayout(2,3));
for (int i = 0; i<5; i++)zoneClient.add (new JButton ("Bouton #" + i));
```

Si le nombre de composants ajoutés excède LxC le nombre de lignes est conservé et le nombre de colonnes est adapté. Par exemple, si 15 composants sont ajoutés à la zone client ci-dessus, le layout manager adapte le nombre de colonne à 8.

La distance par défaut inter-composant peut aussi être modifiée (*Hgap* et *Vgap*) grâce au 3ème constructeur

```
zoneClient.setLayout(new GridLayout(2, 3, 20, 20));
```

BorderLayout

Ce type de layout manager décompose le conteneur en 5 régions baptisées

NORTH, SOUTH, EAST, WEST, et CENTER.
Il dimensionne et positionne les composants dans ces 5 régions, un et un seul composant par région. C'est le layout manager par défaut de la zone client (*Content Pane*) de la fenêtre principale.

L'ordre d'ajout des composants n'a pas d'importance. Par contre, il est nécessaire d'utiliser une autre version de la méthode `add()` qui spécifie le composant à ajouter et la contrainte de placement (chaîne de caractères)

Chaque région est donc identifiée par une constante définie dans la classe *BorderLayout*. Par exemple:

```
zoneClient.setLayout(new BorderLayout());
zoneClient.add (new JButton ("Nord"), BorderLayout.NORTH);
```

Les régions NORTH et SOUTH ont comme hauteur la hauteur idéale du composant ajouté. Leur largeur est identique à celle du conteneur.

Les régions WEST est EAST ont comme largeur la largeur idéale du composant ajouté. Leur hauteur occupe la place maximum.

La région CENTER occupe toute la place restante.

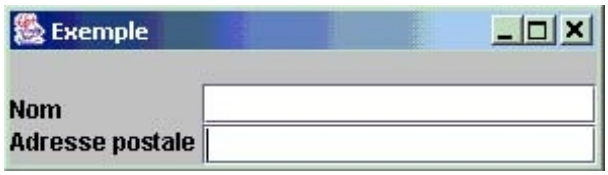
La distance par défaut inter-composant peut aussi être modifiée (*Hgap* et *Vgap*) grâce au 2ème constructeur

```
zoneClient.setLayout(new BorderLayout(5, 5));
```

Le BoxLayout

Ce type de layout manager est spécifique à Swing. Il permet d'aligner les composants horizontalement ou verticalement. Pour toutes les directions, les composants sont arrangés dans l'ordre d'ajout au conteneur et resteront alignés quelle que soit la taille de celui-ci. Si les composants sont arrangés horizontalement et qu'ils n'ont pas la même hauteur préférée le *BoxLayout* adapte la hauteur à celle du plus haut. De même si les composants sont arrangés verticalement et qu'ils n'ont pas la même largeur préférée le *BoxLayout* adapte la largeur à celle du plus large.

Soit par exemple l'application ci-dessous où les contrôles sont placés dans 2 managers verticaux, eux-mêmes placés dans un manager horizontal placé au sud de la zone client



Le panneau gauche est associé à un *BoxLayout* vertical. On remarquera que le layout manager

doit obtenir une référence vers le panneau dont il doit gérer les contrôles

```
JPanel panGauche = new JPanel();
panGauche.setLayout(new BoxLayout(panGauche, BoxLayout.Y_AXIS ));
panGauche.add( new JLabel("Nom "));
panGauche.add( new JLabel("Adresse postale "));
```

Idem pour le panneau de droite

```
JPanel panDroite = new JPanel();
panDroite.setLayout(new BoxLayout(panDroite, BoxLayout.Y_AXIS ));
panDroite.add( new JTextField());
panDroite.add( new JTextField());
```

Les 2 panneaux sont ajoutés à un *BoxLayout* vertical

```
JPanel panComplet = new JPanel();
panComplet.setLayout(new BoxLayout(panComplet, BoxLayout.X_AXIS ));
panComplet.add( panGauche);
panComplet.add( panDroite);
```

Le tout est ajouté au sud de la zone client.

```
zoneClient.add( panComplet, BorderLayout.SOUTH);
```

X_AXIS met les composants les uns derrière les autres.
Y_AXIS met les composants les uns au dessus des autres.

Absence de layout manager

Dans le cas ou la fenêtre principale n'est pas re-taillable et que le nombre de composant n'est pas important, il est possible de ne pas utiliser les services d'un layout manager.

```
zoneClient.setLayout(null);
```

Dans ce cas les composants seront placés en spécifiant leur position `setLocation()`, leur taille `setSize()` ou les deux à la fois `setBounds()` exprimées en pixels dans un repère dont l'origine (0,0) est placée en haut et à gauche (relativement au conteneur).

Les zones de texte sont consultables, donc:

```
private JTextField txtNom = new JTextField();
private JTextField txtPrenom = new JTextField();
```

La fenêtre principale n'est pas re-taillée

```
public FenetrePrincipale()
{
   this.setSize(300,100);
   this.setResizable(false);
   initControles();
}
```

Dans la méthode initControles(): Définition de la stratégie, ajout et positionnement des contrôles

```
   zoneClient.setLayout(null);
   JLabel lblNom = new JLabel ("Votre nom");
   lblNom.setBounds (5,5,100,20);
   zoneClient.add (lblNom);

   JLabel lblPrenom = new JLabel ("Votre prénom");
   lblPrenom.setBounds (5,30,100,20);
   zoneClient.add (lblPrenom);

   txtNom.setBounds (100,5,100,20);
   zoneClient.add (txtNom);


   txtPrenom.setBounds (100,30,100,20);
   zoneClient.add (txtPrenom);
```

Critères de choix

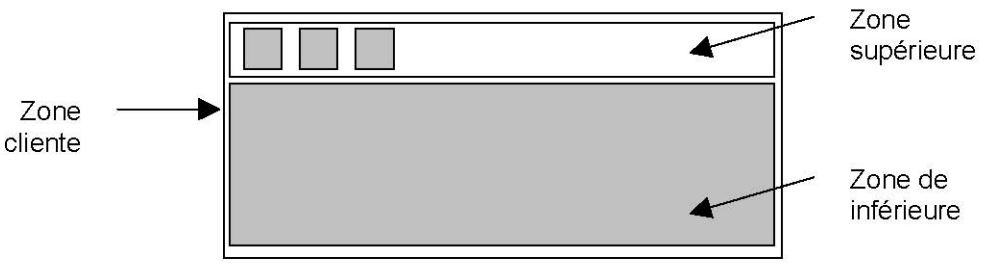
La quasi-totalité des interfaces utilisateur peuvent être obtenues en posant les contrôles dans des panneaux, chaque panneau ayant sa propre stratégie d'organisation (une des quatre précédemment décrites); ces panneaux pouvant à leur tour être posés dans d'autres panneaux … etc.

La seule façon rationnelle de définir rapidement une interface utilisateur avec Swing est de:

11. dessiner sur une feuille de papier l'interface souhaitée
22. identifier les zones géographiques de contrôles
33. déduire les panneaux et leur stratégie d'organisation

Un exemple

Réalisation de l'interface utilisateur d'un éditeur de texte composée de 3 boutons graphiques (Ouvrir, Fermer et Aide) de taille fixe et d'une zone de texte avec ascenseur dont les dimensions évolueront lorsque la fenêtre application sera retaillée. On identifie rapidement 3 zones: la zone client, la zone supérieur dont la hauteur ne doit pas varier et la zone inférieure contenant une zone de texte scrollable dont la taille est asservie à celle de la fenêtre.



Les 3 boutons de classe *JButton* doivent conserver leur taille idéale fixée par la taille de leur l'image. La zone supérieure doit donc être un panneau de classe *JPanel* associé à un *FlowLayout* alignant à gauche puisque c'est le seul qui conserve la taille idéale des contrôles qu'il contient. La hauteur idéale du panneau haut sera celle des boutons.

La zone inférieure sera un panneau de classe *JScrollPane* associé à une *JTextArea* puisqu'elle doit avoir une barre de défilement.

Les 2 panneaux seront ajoutés à la zone cliente. Une région de cette zone devant évoluer seuls un *GridLayout* ou un *BorderLayout* peuvent être utilisés. Le *GridLayout* découperait la zone client en 2 parties égales; ce n'est pas l'effet recherché. Nous choisirons donc un *BorderLayout* avec la zone supérieure au Nord et la zone inférieure au centre (c'est d'ailleurs le layout par défaut de la zone client..

Les 3 boutons et la zone de texte seront manipulés par l'opérateur donc:

```
private JButton cmdOpen = new JButton(new ImageIcon ("openFile.png"));
private JButton cmdClose = new JButton(new ImageIcon ("closeFile.png"));
private JButton cmdHelp = new JButton(new ImageIcon ("help.png"));
private JTextArea txtTexte = new JTextArea();
```

La méthode initControles() contiendra la création des panneaux et l'ajout de ceux-ci à la zone client

```
JPanel zoneClient = (JPanel) this.getContentPane();
zoneClient.setLayout(new BorderLayout());          // Pour le fun

JPanel panHaut = new JPanel (new FlowLayout(FlowLayout.LEFT));
panHaut.add (cmdOpen);
```

```
panHaut.add (cmdClose);
panHaut.add (cmdHelp);
zoneClient.add (panHaut, BorderLayout.NORTH);

JScrollPane panEdit = new JScrollPane(txtTexte);
zoneClient.add (panEdit, BorderLayout.CENTER);
```

Un petit exercice Agencement des Controles

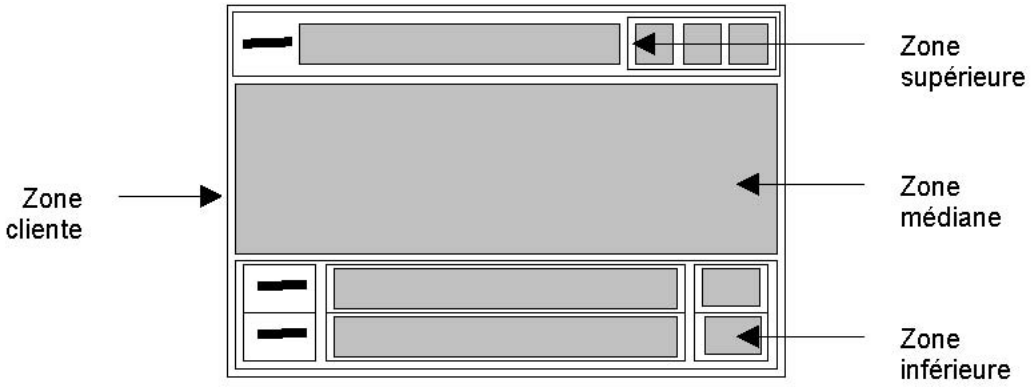
Ecrire une application ayant l'interface utilisateur décrit ci-dessous. La fenêtre application comporte 3 zones :

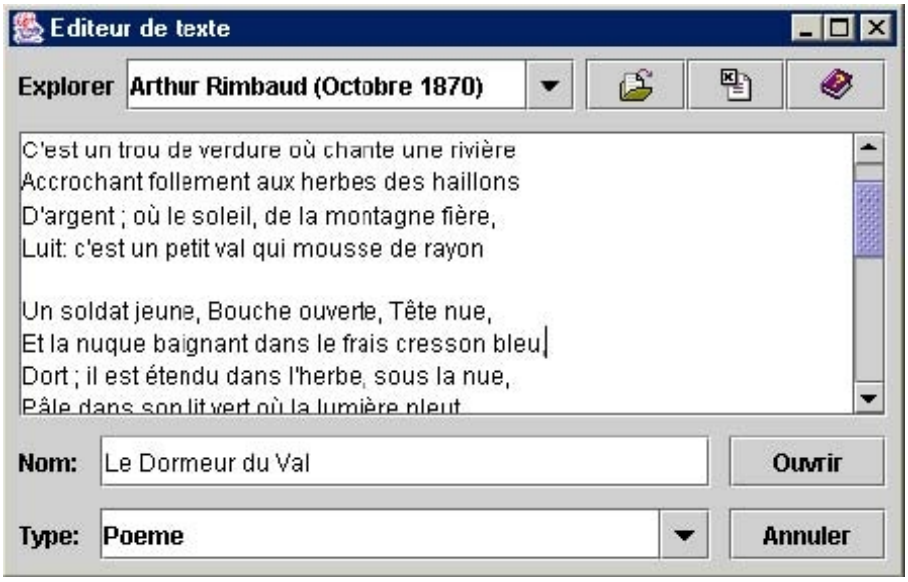
La partie supérieure de hauteur constante comprend :

- un JLabel (une étiquette) qui affiche le mot : Explorer
- une boîte JCombo qui occupe le maximum de largeur,
- une barre d'outils de 3 boutons graphiques (ouvrirFichier, fermerFichier, aidezMoi).
  Chaque bouton graphique est associé à une image :
  - ouvrirFichier.png pour le bouton ouvrirFichier
  - fermerFichier.png pour le bouton fermerFichier
  - aidezMoi.png pour le bouton aidezMoi

1- La partie médiane est une zone de texte multi-lignes (JTextArea) qui occupe le maximum d'espace en largeur avec un ascenseur (JScrollPane vertical toujours apparent)

2- La partie inférieure est constituée elle-même de 3 zones sur 2 lignes :
première ligne :
une JLabel (étiquette) qui affiche « Nom : »,
une zone de saisie du type JTtextField qui prend le maximum de largeur
un JButton (bouton) ouvrir
deuxième ligne :
une étiquette qui affiche « type : »,
une JComboBox qui prend toute la largeur
un bouton Quitter




Editeur de texte
Explorer
Arthur Rimbaud (Octobre 1870)
C'est un trou de verdure où chante une rivière
Accrochant follement aux herbes des haillons
D'argent ; où le soleil, de la montagne fière,
Luit: c'est un petit val qui mousse de rayon
Un soldat jeune, Bouche ouverte, Tête nue,
Et la nuque baignant dans le frais cresson bleu
Dort ; il est étendu dans l'herbe, sous la nue,
Pâle dans son lit vert où la lumière pleut
Nom:
Le Dormeur du Val
Ouvrir
Type:
Poeme
Annuler

Gestion des évènements

La programmation d'interface utilisateur graphique (Swing et les autres) repose sur une approche événementielle. Chaque action manuelle de l'opérateur (clic, déplacement de la souris, …) déclenche un événement. Si l'application doit avoir un comportement particulier en réponse à cette action il faut associer à cet événement un traitement. Deux difficultés seront à traiter par le développeur.euse:

méthodologique Il doit avoir identifié les traitements que l'application doit effectuer pour pouvoir définir **quelle action déclenche quel traitement**. Cette difficulté est commune à toutes les interfaces utilisateur graphique (VB et C++ sous Windows, Motif, ….)

technique Il doit connaître les mécanismes de codage permettant l'association d'un événement avec une portion de code. Chaque technologie est différente, il faut ré-apprendre des concepts nouveaux pour chaque langage (Java, C++, VB, …) et chaque API (Swing, Windows, Motif, ..)

L'objectif de ce chapitre est d'étudier les aspects techniques de la programmation Java/ Swing c'est à dire:

1• Qu'est-ce qu'un événement Java
2• Comment récupérer des événements spécifiques
3• Comment déclencher une méthode particulière en réponse à ces événements

L'application qui va être décrite dans les prochains paragraphes consiste à afficher le message "Hello World" dans la zone de texte en réponse à un clic sur le bouton OK. Lorsque la zone de texte prend le focus, elle s'efface automatiquement.

Les événements Java

En réponse à une action de l'opérateur un objet Java va être créé. **C'est le composant qui a détecté l'action qui va créer cet objet**. L'objet est une instance d'une classe d'événement spécifique (une classe d'événement pour chaque type d'interaction). La classe de l'objet sera différente suivant qu'il s'agit d'un clic sur un bouton, d'un clic sur un menu ou d'un déplacement de souris au-dessus d'une image. Par exemple:

Les événements les plus fréquents sont

Evénement émis	Causes possible
FocusEvent	Changement de focus sur un composant
MouseEvent	Clic sur un panneau
KeyEvent	Frappe d'une touche dans une zone de texte
WindowEvent	Agrandissement d'une fenêtre
ActionEvent	Clic sur un bouton
ListSelectionEvent	Sélection d'un élément d'une liste

L'objet événement pourra être consulté pour connaître le composant qui l'a émis et avoir un certain nombre de renseignements concernant le contexte d'apparition (position de la souris, valeur de la touche frappée, …)

Mise en place d'un écouteur

Le mécanisme qui va intercepter l'événement pour déclencher le traitement (Affichage du message dans la zone de texte) s'appelle un **listener** (à l'écoute de …). **Un *listener* ou écouteur est une classe java qui doit avoir une fonction spécifique** (adapté à l'écoute d'un clic, d'une sélection, du focus, ….).

Par exemple l'interface *ActionListener* définit les caractéristiques d'un écouteur capable d'intercepter un clic sur un bouton. L'écouteur à mettre en place dans l'application doit être un *ActionListener.* **Notre écouteur spécifique à notre l'application hérite (implémente) donc de l'interface *ActionListener*.**

```
class AppActionListener implements ActionListener {

}
```

Cette classe est spécifique à la fenêtre principale, elle ne pourra pas être réutilisée dans une autre application. Elle sera donc incluse dans la classe FenetrePrincipale. Les classes incluses (*inner class*) sont définies au même niveau d'imbrication que les méthodes. Elles auront toutes le modifieur par défaut (rien).

Implémenter une interface oblige à re-définir toutes ses méthodes. Il faudra ici re-définir la méthode `actionPerformed()`

```
class AppActionListener implements ActionListener {
   public void actionPerformed(ActionEvent e)
   {
      // Traitement à exécuter
   }
}
```

Pour que la méthode `actionPerformed()` soit appelée lors d'un clic sur le bouton, il faut enfin associer le *listener* au bouton cmdOK. Le *listener* doit s'**abonner** à l'événement auprès du bouton.

L'abonnement s'effectue de la façon suivante:

```
cmdOK.addActionListener(new AppActionListener ());
```

Les boutons appèlent la méthode `actionPerformed()` de tous les *listeners* abonnés, lorsqu'ils reçoivent un clic. On utilise ici le terme de **notification**.

Le traitement à exécuter (Affichage du message dans la zone de texte) pourrait très bien être écrit directement dans la méthode `actionPerformed()`.

```
class AppActionListener implements ActionListener {

   public void actionPerformed(ActionEvent e)
   {
      txtTexte.setText("Hello World");
   }
}
```

Cette forme ci-dessus, bien que correcte, est déconseillée car il est préférable de bien séparer dans le code, les méthodes "applicatives" (ce que doit faire l'application) des méthodes "techniques" (mécanique de programmation Swing). Il est préférable de présenter le code ainsi:

```
class AppActionListener implements ActionListener {

   public void actionPerformed(ActionEvent e)
   {
      if (e.getSource() == cmdOK) cmdOK_click();

   }
}

private void cmdOK_click()
{
   txtTexte.setText("Hello World");
}
```

`actionPerformed()` est une méthode technique. Ce même écouteur peut s'abonner à un autre composant. On vérifie quel composant a envoyé la notification et on déclenche le traitement spécifique.

`cmdOk_Click()` est une méthode applicative. Elle est privée car spécifique à la classe FenetrePrincipale. Par convention elle porte un nom significatif où apparaît clairement le nom du composant d'origine et le nom de l'événement propagé.

Autre exemple

Le second traitement consiste à effacer la zone de texte lorsqu'elle prend le focus. Il faut lui abonner un écouteur capable de détecter le changement de focus. Il s'agit de *FocusListener*.

```
txtTexte.addFocusListener(new AppFocusListener());
```

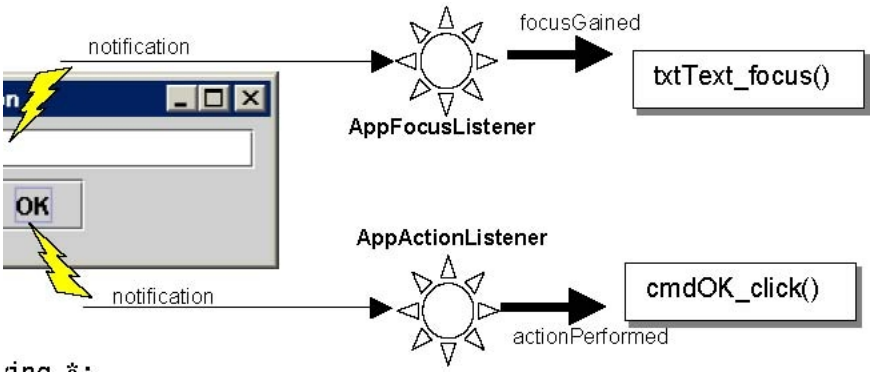
L'écouteur à mettre en place dérive donc de *FocusListener*. Il faudra redéfinir ses 2 méthodes, l'une qui est appelée en réponse à la prise du focus, l'autre à la perte du focus

```
class AppFocusListener implements FocusListener {

   public void focusGained(FocusEvent e) { . . . }
   public void focusLost(FocusEvent e)   { . . . }
}
```

Dans l'exemple, seule la prise de focus `focusGained()` déclenche un traitement. L'autre méthode `focusLost()`doit avoir un bloc d'instruction vide

```
class AppFocusListener implements FocusListener {

   public void focusGained(FocusEvent e)
   {
      if (e.getSource() == txtTexte) txtTexte_focus();
   }

   public void focusLost(FocusEvent e)   {}
}

private void txtTexte_focus()
{
   txtTexte.setText("");
}
```

Code complet




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

public class FenetrePrincipale extends JFrame
{
   private JTextField txtTexte = new JTextField(20);
   private JButton cmdOK = new JButton("OK");

   public FenetrePrincipale()
   {
      . . . ;
      initControles();
   }

   private void initControles()
   {
      JPanel zoneClient = (JPanel) this.getContentPane();
      zoneClient.setLayout (new FlowLayout());
      zoneClient.add (txtTexte);
      zoneClient.add (cmdOK);
      cmdOK.addActionListener(new AppActionListener());  // Abonnements
      txtTexte.addFocusListener(new AppFocusListener());
   }

   // -------- Mise en place des listeners

   class AppActionListener implements ActionListener {

      public void actionPerformed(ActionEvent e)
      {
         if (e.getSource() == cmdOK) cmdOK_click();
      }
   }

   class AppFocusListener implements FocusListener {

      public void focusGained(FocusEvent e)
      {
         if (e.getSource() == txtTexte) txtTexte_focus();
      }

      public void focusLost(FocusEvent e)   {}
   }


   private void cmdOK_click()
   {
      txtTexte.setText("Hello World");
```

```
    }

    private void txtTexte_focus()
    {
       txtTexte.setText("");
    }
}
```

Démarche générale

Il est important de raisonner en terme de TRAITEMENT et de se poser 3 questions auxquelles on apporte une réponse immédiate.

Question 1

Pour un traitement donné à exécuter, quel composant déclenche ce traitement ?

En réponse à cette question, la classe du composant est identifiée et les différentes versions des méthodes addXXXListener() renseignent sur les types de *listener* qu'il est possible d'abonner.

Le tableau ci-dessous reprend les principaux *listeners* (en gras ceux qui sont spécifiques aux composants et en maigre ceux qui sont définis dans la classe *Component* donc disponibles pour tous les composants)

JLabel	FocusListener, KeyListener, MouseListener, MouseMotionListener
JTextField	**ActionListener, CaretListener,** FocusListener, KeyListener, MouseListener, MouseMotionListener
JTextArea	**CaretListener,** FocusListener, KeyListener, MouseListener, MouseMotionListener
JButton, JCheckBox, JRadioButton	**ActionListener, ChangeListener,** FocusListener, KeyListener, MouseListener, MouseMotionListener
JList	**ListSelectionListener,** FocusListener, KeyListener, MouseListener, MouseMotionListener
JComboBox	**ActionListener, ItemListener, PopupMenuListener,** FocusListener, KeyListener, MouseListener, MouseMotionListener
JFrame, JDialog	WindowListener

Question 2

Quelle type d'action l'opérateur effectue-t-il sur le composant ?

Sur le composant, l'opérateur déplace-t-il la souris, modifie-t-il le focus, sélectionne-t-il un élément, … Parmi les classes *listener* utilisables une seule est adaptée. Le tableau ci-dessous reprend les principales fonctionnalités des *listeners* les plus utilisés

FocusListener	La perte ou prise du focus par le composant
KeyListener	Une action clavier (appuie, relâchement) lorsque le composant a le focus
MouseListener	L'apparition et disparition de la souris sur le composant ou clic sur les boutons
MouseMotionListener	Le déplacement de la souris sur le composant
ActionListener	L'action sur un contrôle
ChangeListener	Au changement d'état (enfoncé et relâché) d'un bouton, bouton radio, une case à cocher ou à la sélection d'un élément de menu
CaretListener	Au déplacement du curseur dans une zone de texte
ListSelectionListener	Au changement de sélection dans une liste

ItemListener	A la sélection ou la dé-sélection d'un élément de combo.
PopupMenuListener	Au changement d'état de la fenêtre pop-up d'une combo
WindowListener	La perte, la prise du focus par la fenêtre ou a son changement d'état (agrandissement, mise en icône, …)

Lorsque l'interface est déterminée, il est possible d'écrire la classe incluse dérivant l'interface et re-définissant ses méthodes.

```
class AppXxxListener extends XxxListener {
   public void methode1 ( … ) { }
   public void methode2( … ) { }
}
```

L'abonnement peut ainsi être effectué

```
leComposant.addXxxListener (new AppXxxListener());
```

Question 3

Quelle action précise l'opérateur effectue-t-il sur le composant ?

En réponse à cette question, une méthode de l'interface est identifiée (chaque méthode correspond à une action précise: prise de focus, enfoncement d'un bouton, …). C'est elle qui contiendra le code d'appel de la méthode applicative:

```
public void methodeN (XxxEvent e)
{
   if (e.getSource() == leComposant) methode_applicative();
}
```

Nota:
La méthode applicative n'est pas forcément située dans la classe FenetrePrincipale. Ce peut-être une méthode particulière de la hiérarchie des classes de l'application.

Autre type d'écouteur

Un écouteur est logiquement une interface Java. Implémenter une interface oblige à re-définir systématiquement toutes les méthodes qui la composent. Certaines interfaces (*FocusListener*, *ActionListener*) n'ont qu'une ou 2 méthodes. D'autres par contre en ont un très grand nombre (*WindowListener*, *MouseListener*). Si seules une ou 2 méthodes déclenchent une méthode applicative il faudra re-définir malgré tout les autres méthodes en leur associant un bloc vide.

C'est la raison pour laquelle Swing offre des **Adapters**, classes concrètes qui implémentent certaines interfaces *listener*. Les écouteurs de l'application pourront ainsi dériver les classes concrètes *XxxAdapter* et ne re-définir que les méthodes nécessaires.

Le tableau ci-dessous donne la correspondance entre le *XxxListener* (Interface) et son *XxxAdapter* (classe concrète) disponible avec Swing

FocusListener	FocusAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter

MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

Par exemple: Effacer la zone de texte lorsque l'application est mise en icône.

Abonnement
```
this.addWindowListener(new      AppWindowAdapter
());
```

Ecouteur
```
class AppWindowAdapter extends WindowAdapter {

   public void windowIconified(WindowEvent e)
   {
      fenetrePrincipale_iconified();
   }
}
```

Applicatif
```
private void fenetrePrincipale_iconified()
{
   txtTexte.setText("");
}
```

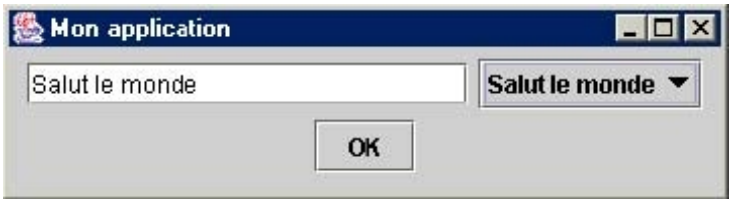
Nota:
On remarquera ici que la méthode `windowIconified()` ne teste pas l'origine de l'événement. En effet, les événements récupérés par *WindowListener* ne peuvent provenir que de la fenêtre principale.

Optimisation du code source "technique"

La démarche qui vient d'être étudiée semble lourde surtout pour coder une application simple. Cette démarche trouve son intérêt lors du développement d'applications réelles disposant d'un nombre important de contrôles, chacun étant susceptible de propager des événements qui déclenchent des traitements applicatifs. Pour ne pas alourdir le code il est préférable de ne pas multiplier les écouteurs.

Plusieurs composants génèrent le même type d'événement

Faisons évoluer l'application précédente en rajoutant une combo contenant des messages prédéfinis à afficher dans la zone de texte suite à une sélection



La combo génère aussi `actionPerformed()` lorsqu'une sélection est effectuée. Il sera donc possible ici d'abonner la combo au même écouteur que pour le bouton.

```
cboChoix.addActionListener(new AppActionListener());
```

La même méthode sera appelée en réponse à un clic sur le bouton OK ou en réponse à une sélection de la combo. Le test du composant émetteur permettra de lancer l'une ou l'autre des

méthodes applicatives

```
class AppActionListener implements ActionListener {

   public void actionPerformed(ActionEvent e)
   {
      if (e.getSource() == cmdOK) cmdOK_click();
      if (e.getSource() == cboChoix) cboChoix_click();
   }
}

private void cboChoix_click()
{
   txtTexte.setText(cboChoix.getSelectedItem().toString());
}
```

Des événements différents déclenchent le même traitement

Cette particularité est très fréquente avec les interfaces utilisateurs graphiques. Par exemple la sauvegarde d'un document peut être effectuée par un menu ou un bouton de la barre d'outil ou par la fermeture de l'application.

Avec la démarche proposée la mise en œuvre est évidente et ne nécessite pas d'exemple. Les écouteurs nécessaires sont créés et les méthodes re-définies. Chacune d'elles appellera la même méthode applicative. C'est tout.

La classe Timer

La classe Timer fait partie du package swing mais n'est pas un composant graphique. Les objets instance de cette classe sont capables d'envoyer un événement *ActionEvent* à intervalle régulier. Il sera possible ainsi de déclencher régulièrement un traitement sans action physique de l'opérateur (interrogation régulière d'une base de donnée, clignotement d'une information …).

Exemple

Toutes les secondes (1000ms) un message pourra être envoyé et récupéré par un listener instance de la classe AppActionListener

```
private Timer tmWarning = new Timer (1000, new AppActionListener);
```

Mise en route du timer

```
tmWarning.start();
```

Récupération de l'événement généré

```
class AppActionListener implements ActionListener {

   public void actionPerformed(ActionEvent e)
   {
      Object source = e.getSource();
      if (source == tmWarning ) tmWarning_time();
   }
}
```

Traitement exécuté toutes les secondes (ici un simple bip sonore)

```
private void tmWarning_time()
{
   Toolkit.getDefaultToolkit ().beep();
```

```
}
```

Interface utilisateur complet

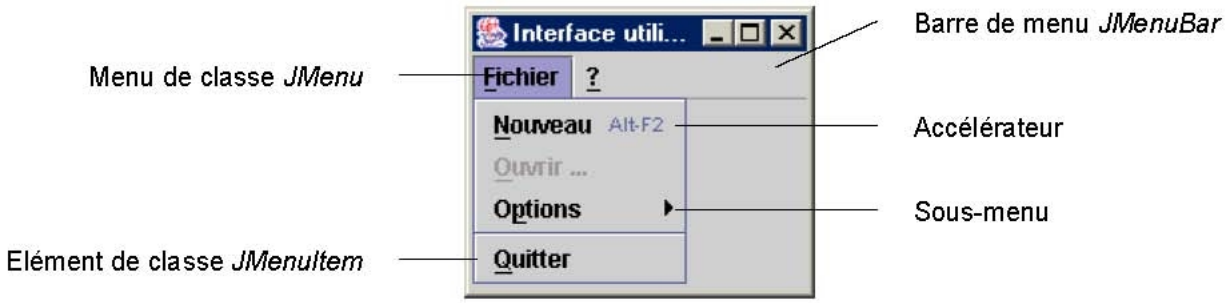
L'objectif de ce chapitre est d'étudier les composants supplémentaires rencontrés couramment dans les applications: la barre de menu, la barre d'état, la barre d'outils et les boîtes de dialogue simples dont la boîte "A propos" qui donne des informations sur l'application.

```
+--javax.swing.JComponent
      |
      +--javax.swing.JToolBar
      +--javax.swing.JMenuBar
      +--javax.swing.AbstractButton
      |     |
      |     +--javax.swing.JButton
      |     +--javax.swing.JMenuItem
      |           |
      |           +--javax.swing.JMenu
```

La barre de menus

Une barre de menus est une liste horizontale de chaînes de caractères, appelées menus, ouvrant une fenêtre déroulante (pop-up). Cette dernière contient des éléments permettant de déclencher des actions lorsqu'ils sont sélectionnés. Les éléments peuvent être grisés et/ou cochés. Si un élément d'un menu ouvre une boîte de dialogue il se termine par 3 points. Une cascade de menu (sous-menu) est signalée par un triangle. Le dernier élément du menu le plus à gauche permet de quitter l'application. Le menu le plus à droite est le menu d'aide qui contient obligatoirement l'élément "A propos ...".

En standard, les éléments d'un menu peuvent être sélectionnés par la souris mais aussi par l'action combinée de la touche *Alt* et d'une lettre; c'est un raccourcis clavier (*Mnemonic*). Dans le libellé de l'élément, la lettre soulignée indique à l'opérateur la touche concernée. Il est aussi possible de définir des accélérateurs (*Accelerator*) ou combinaison de touches, qui permettent de sélectionner un élément sans dérouler le menu. Si un accélérateur est associé à un élément il est spécifié dans le libellé de l'élément.



La fenêtre principale a la possibilité d'ajouter, en plus de la zone client, une seule barre de menu à la *layeredPan*. Celle-ci est toujours placée comme panneau supérieur. Une barre de menu est une instance de *JMenuBar*. Elle est composée de menus, instances de *JMenu* eux-mêmes composés d'éléments instances de *JItemMenu*.

Elément de classe *JMenuItem*

Un menu (*JMenu*) et un élément (*JMenuItem*) sont en fait des boutons, le premier qui déclenche l'ouverture d'une fenêtre pop-up, le second qui est associé à une méthode applicative

Les éléments de menu déclenchent des actions et vont parfois changer d'état. Il faut donc les définir comme champs privés. Les autres objets pourront être déclarés localement à la méthode `initControls()`. Par convention l'élément de libellé "Ouvrir" dans le menu "Fichier" sera nommé `mnuFichierOuvrir`.

```
private JMenuItem mnuFichierNouveau = new JMenuItem("Nouveau", 'N');
private JMenuItem mnuFichierOuvrir  = new JMenuItem("Ouvrir ...", 'O');
private JMenuItem mnuFichierQuitter = new JMenuItem("Quitter", 'Q');
```

Ces 3 éléments de menu vont déclencher des traitements en réponse à un clic souris. Un *JMenuItem* est un *AbstractButton*, il sera donc possible de lui associer un écouteur dérivé de *ActionListener*.

```
class MenuActionListener implements ActionListener {

    public void actionPerformed(ActionEvent e)
    {
       Object source = e.getSource();
       if (source == mnuFichierNouveau) mnuFichierNouveau_click();
       if (source == mnuFichierOuvrir ) mnuFichierOuvrir_click();
       if (source == mnuFichierQuitter) mnuFichierQuitter_click();
    }
}
```

Le code de création d'une barre de menu est généralement linéaire et très long, Il est conseillé d'une part d'avoir une approche de programmation rigoureuse et d'autre part de regrouper ce code dans une méthode privée spécifique `initMenu()` appelée depuis le constructeur

```
public FenetrePrincipale()
{
   // initialisation de la fenêtre
   initMenu();
   initControles();
}
```

11) Création de la barre de menu (la barre de menu n'est pas ajoutée à la *contentPane* mais à la fenêtre)

```
private void initMenu()
{
   JMenuBar mbPrincipale = new JMenuBar ();
   this.setJMenuBar(mbPrincipale);
```

12) Création des menus et ajout à la barre de menu

```
    JMenu mnuFichier = new JMenu ("Fichier");
    mnuFichier.setMnemonic('F');         // Ajout du raccourcis clavier
    mbPrincipale.add(mnuFichier);
    JMenu mnuEdition = new JMenu ("?");
    mnuAide.setMnemonic ('?');
    mbPrincipale.add(mnuAide);
```

13) Initialisation des éléments, affectation de l'état initial (accélérateur, grisé, …) et ajout au menu

```
    mnuFichierNouveau.setAccelerator(KeyStroke.getKeyStroke("alt F2"));
    mnuFichierNouveau.addActionListener(new MenuActionListener()); // ABONNEMENT
    mnuFichier.add (mnuFichierNouveau);
    mnuFichierOuvrir.setEnabled(false);  // Le menu est grisé
    mnuFichierOuvrir.addActionListener(new MenuActionListener());  // ABONNEMENT
    mnuFichier.add (mnuFichierOuvrir);
    mnuFichier.addSeparator();
    //. . . etc pour tous les éléments
}
```

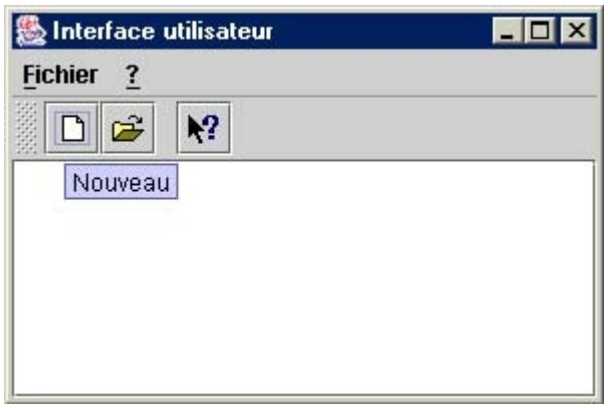
14) Reste ensuite à implémenter toutes les méthodes applicatives

```
private void mnuFichierQuitter_click()
{
   System.exit(0);
}
//. . . etc pour toutes les méthodes
```

La barre d'outils

Une barre d'outils (*JToolBar*) est un composant (*JComponent*) contenant des boutons graphiques (*JButton*) associés à certains éléments du menu (les plus utilisés). C'est la raison pour laquelle la barre d'outils est souvent placée sous la barre de menu (dans la partie Nord de la zone client qui a un *layout* par défaut de classe *BorderLayout)*.

Les images des boutons peuvent être au format GIF, JPEG et PNG et doivent être de taille identique (en général 16x15 pixels). Il est conseillé d'associer une bulle d'aide à chaque bouton



Conseil pratique:

Les boutons de la barre d'outils déclenchent les mêmes actions que certains éléments de menu. Il est fortement conseillé de coder d'abord le menu (aspect et dynamique) puis, seulement ensuite, de coder la barre d'outils. Ceci permettra que le clic sur un bouton notifie la même méthode applicative que l'élément correspondant.

Mise en œuvre

Les boutons génèrent des événements donc:

```
private JButton cmdToolNouveau = new JButton (new ImageIcon("new.gif"));
private JButton cmdToolOuvrir = new JButton (new ImageIcon("open.gif"));
private JButton cmdToolAide = new JButton (new ImageIcon("help.gif"));
```

Après l'initialisation du menu, création d'une barre et ajout au nord de la zone client

```
private void initMenu()
{
   JMenuBar mbPrincipale = new JMenuBar ();
   this.setJMenuBar(mbPrincipale);
   // initialisation du menu

   JToolBar tbPrincipale = new JToolBar ();
   zoneClient.add (tbPrincipale, BorderLayout.NORTH);
```

Initialisation des boutons et abonnement pour chacun du même écouteur que pour les éléments du menu

```
   cmdToolNouveau.setToolTipText("Nouveau");
   cmdToolNouveau.addActionListener(new MenuActionListener());
   tbPrincipale.add(cmdToolNouveau);
   // . . . etc pour tous les boutons
}
```

Mise à jour de la re-définition de la méthode `actionPerformed()` pour déclencher les mêmes méthodes applicatives que les éléments correspondant

```
class MenuActionListener implements ActionListener {


    public void actionPerformed(ActionEvent e)
    {
       Object source = e.getSource();
       if (source == cmdToolNouveau) mnuFichierNouveau_click();
       if (source == cmdToolOuvrir) mnuFichierOuvrir_click();

       if (source == mnuFichierNouveau ) mnuFichierNouveau_click();
       if (source == mnuFichierOuvrir) mnuFichierOuvrir_click();
       if (source == mnuFichierQuitter) mnuFichierQuitter_click();
    }
 }
```

Précisions supplémentaires

Le séparateur est obtenu par la méthode `addSeparator()` appelée entre l'ajout du second et du troisième bouton

```
tbPrincipale.addSeparator();
```

Par défaut la barre d'outil est flottante, c'est à dire que l'opérateur peut l'accrocher sur l'un des 4 cotés de son conteneur (en général la zone client) .

Si cette particularité n'est pas souhaitée elle peut être invalidée par:

```
tbPrincipale.setFloatable(false);
```

La barre d'état

La barre d'état est un panneau contenant de simples labels (*JLabel*) avec bordure creuse ou en relief, placé dans la partie Sud de la zone client. Elle est utilisée pour afficher des informations d'état pour l'opérateur (renseignement, message d'erreur, …).

Souvent elle est utilisée pour afficher un message d'aide sur l'utilisation des éléments de menu. Ce message s'affiche dès que l'élément est sélectionné.

La barre d'état est un objet dont le contenu est modifié. (utiliser ce constructeur pour obtenir une hauteur idéale correcte)

```
private JLabel statusBar= new JLabel(" ");
```

La barre d'état est ajoutée à la partie Sud de la zone client

```
private void initControles()
{
   JPanel zoneClient = (JPanel) this.getContentPane();
   statusBar.setBorder(BorderFactory.createLoweredBevelBorder());
   zoneClient.add(statusBar, BorderLayout.SOUTH);
```

Lorsque la souris passe sur l'élément de menu, un événement de classe *MouseEvent* est propagé. Il faut donc créer un écouteur, dérivé de *MouseAdapter,* pour intercepter cet événement (il est préférable de prendre ici l'*adapter* que le *listener* car on ne traite que 2 méthodes sur 5). Abonnement des éléments à un tel écouteur.

```
   mnuFichierNouveau.addChangeListener(new MenuMouseAdapter ());
   mnuFichierOuvrir.addChangeListener (new MenuMouseAdapter ());
   mnuFichierQuitter.addChangeListener(new MenuMouseAdapter ());
}
```

Création de la classe écouteur et re-définition de 2 méthodes

```
class MenuMouseAdapter extends MouseAdapter {
   public void mouseEntered(MouseEvent e)
   {
      Object source = e.getSource();
      if (source == mnuFichierNouveau )
                              statusBar.setText("Nouveau fichier");
      if (source == mnuFichierOuvrir)
                              statusBar.setText("Ouvrir un fichier");
      if (source == mnuFichierQuitter)
                              statusBar.setText("Quitter l'application");
   }
   public void mouseExited(MouseEvent e)
   {
      statusBar.setText(" ");
   }
}
```

Les fenêtres prédéfinies

Il existe des fenêtres particulières qui permettent à l'utilisateur de saisir des informations. Celles-ci doivent avoir le même L&F que le reste des fenêtres l'application. La méthode *main()* doit comporter l'instruction suivante dans le cas du L&F Metal

```
JDialog.setDefaultLookAndFeelDecorated(true);
```

Il en existe 2 types de fenêtre prédéfinies:

Les boîtes de message

Ce sont des fenêtres qui affichent un message et attendent une réponse simple de l'opérateur (OK, OUI, NON, ANNULER, ...). Elles comprennent un titre, une ou plusieurs lignes de texte, un ou plusieurs boutons et une icône. La classe *JOptionPane*, contient des méthodes statiques permettant l'ouverture de telle boîte

Boîte d'information

La méthode `showMessageDialog()` permet d'ouvrir une simple boîte d'information

```
JOptionPane.showMessageDialog(this, "Erreur de saisie", "Erreur",
                              JOptionPane.ERROR_MESSAGE);
```

Les icônes possibles sont:


QUESTION_MESSAGE


INFORMATION_MESSAGE


WARNING_ MESSAGE


ERROR_MESSAGE

Boîte de confirmation

La méthode `showConfirmDialog()` attend un choix de l'opérateur (OUI, NON et éventuel-lement ANNULER)

```
int rep = JOptionPane.showConfirmDialog(this, "Fin de l'application",
                              "Voulez-vous quitter l'application",
                              JOptionPane.YES_NO_OPTION);
```

Le type de boutons peut être aussi YES_NO_CANCEL_OPTION
La méthode retourne une des constantes suivantes: YES_OPTION, NO_OPTION ou CANCEL_ OPTION (champs statiques de la classe *JOptionPane*)

Les boîtes de dialogue prédéfinies

Il existe 2 boîtes de dialogue prédéfinies *JColorChooser* et *JFileChooser* dérivées de *JComponent* qui permettent respectivement de choisir une couleur et de choisir un fichier.

JColorChooser

Cette classe dispose d'une méthode statique pour ouvrir la boîte en mode modal (L'instruction est bloquante tant que l'opérateur n'a pas cliqué sur OK ou ANNULER). La méthode reçoit une référence sur la fenêtre parent, le titre de la boîte de dialogue et la couleur par défaut.

```
Color col = JColorChooser.showDialog(this,
                                     "Choisissez une couleur de fond",
                                     Color.gray);
```

La méthode retourne un objet couleur ou *null* si l'opérateur a cliqué sur ANNULER

JFileChooser

Permet d'ouvrir ou sauver un fichier suivant la méthode utilisée. Il faut créer un objet instance de cette classe, appliquer une méthode pour l'ouverture en mode modal puis enfin appliquer un *getter* pour récupérer l'information sélectionnée.

Création de l'objet. Le constructeur reçoit le répertoire choisit à l'ouverture

```
JFileChooser dlg = new JFileChooser("C:/usr");
```

Ouverture en mode modal de la boîte (mode ouverture de fichier). La méthode reçoit la référence vers la fenêtre parent et retourne une constante différente suivant que la boîte a été fermée par OK ou ANNULER.

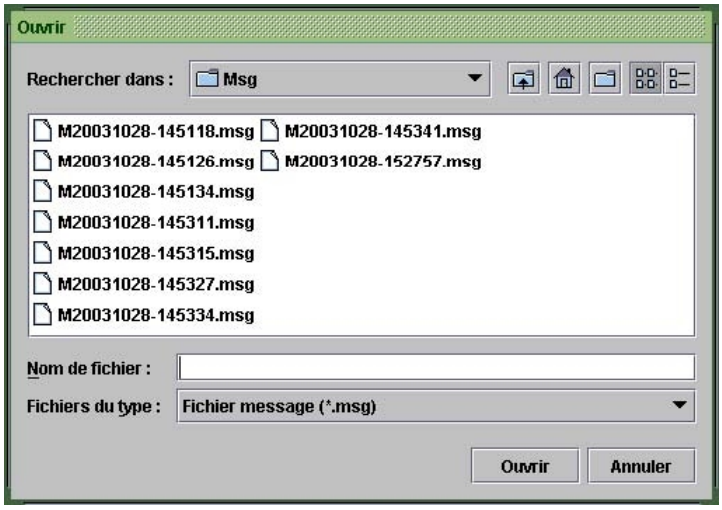
```
int ret = dlg.showOpenDialog(this);
```

Pour une ouverture en mode "sauvegarde de fichier" il faudra utiliser

```
int ret = dlg.showSaveDialog(this);
```

Si la boîte a été fermée par OK il faut récupérer le fichier sélectionné (de classe *File*)

```
if (ret == JFileChooser.APPROVE_OPTION) {
      File f = dlg.getSelectedFile();
}
```



Il est possible d'associer un filtre permettant de sélectionner seulement les fichiers répondant à un certain critère. Dans l'exemple ci-dessous la boîte *JFileChooser* ne visualise que les répertoires et les fichiers d'extension `msg`.

Un filtre est une classe dérivée de la classe abstraite *FileFilter* dans laquelle 2 méthodes sont re-définies. La première méthode définit les critères de visualisation, la seconde le libellé dans la boîte combo "Fichiers du type". Cette classe peut être implémentée comme une *inner class* dans la fenêtre principale.

```
class MessageFilter extends FileFilter {
   public boolean accept(File f) {
      return f.isDirectory() || f.getName().endsWith(".msg");
   }
   public String getDescription() {
      return "Fichier message (*.msg)";
   }
}
```

Le filtre doit être ajouté à la boite (il est possible d'ajouter ainsi plusieurs filtres)

```
JFileChooser dlg = new JFileChooser ("C:/usr");
dlg.addChoosableFileFilter(new MessageFilter());
```

La boîte à propos

Outre les fenêtre prédéfinies il est possible de définir des boîtes de dialogue spécifiques à l'application qui permettent une saisie particulière d'informations. Elles sont dérivées de la classe *JDialog* et peuvent être:

modale Lorsqu'une boîte modale est ouverte l'opérateur ne peut plus activer une fenêtre de l'application tant que la boîte n'est pas fermée. C'est le cas par exemple de la boîte *ouvrir* du menu *Fichier* des éditeurs de texte ou de la boîte *A propos.* Par contre l'opérateur peut toujours activer les fenêtres d'une autre application.

non modale Une boîte non modale permet à l'opérateur de se déplacer entre la boîte et les autres fenêtres de l'application. C'est en général le cas de la boîte *Rechercher* du menu *Edition* des éditeurs de texte.

Cette étude sort du cadre de ce document. Etudions simplement une boîte de

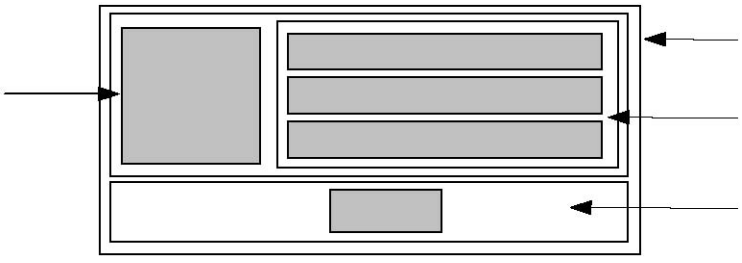
dialogue que toute application doit posséder. Il s'agit de la boîte "A Propos", ouverte en réponse à un clic sur l'élément "A Propos …" du menu de droite "?"

Cette boîte est de type modal et de taille fixe. L'opérateur ne dispose que d'un bouton OK. Elle ne comporte que des contrôles statiques de renseignement pour l'opérateur: Une description, un copyright, un numéro de version et l'icône de l'application

Le nombre de contrôles et leur disposition sont les mêmes quelle que soit l'application. Seul change leur contenu. C'est l'occasion d'en faire un composant logiciel réutilisable. Ce sera donc une classe indépendante, dérivée de *JDialog* dont la structure sera proche de celle de la fenêtre principale.

```
public class APropos extends JDialog
{
   // reférence vers les Controles
   public APropos(JFrame parent, String titre)
   {
      super(parent, "A Propos de " + titre, true);
      // Initialisation de la boîte de dialogue
      initControles();
   }
   // Autres méthodes
}
```

Dans notre cas de la boîte "A Propos" est modale. Les contrôles sont ordonnés de la façon suivante:



La boîte "A Propos" sera ouverte en mode modal dans une procédure d'événement de la fenêtre principale de la façon suivante

```
private void mnuAideAPropos_click()
{
   APropos dlg = new APropos(this, "WeibMail");
   // Initialisations
```

```
    dlg.show();
}
```

La méthode `show()` rend la boîte visible et la met au premier plan. Cette instruction est bloquante tant que la boîte de dialogue n'est pas fermée.

Code complet de la boîte "A Propos"

La boîte A Propos dérive de *JDialog* et implémente *ActionListener* pour gérer le bouton OK (cette façon de gérer les événements évite la classe interne)

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

public class APropos extends JDialog implements ActionListener
{
```

Plusieurs méthodes vont accéder aux contrôles donc:

```
    private JLabel lblDescription = new JLabel ();
    private JLabel lblCopyright = new JLabel ();
    private JLabel lblVersion = new JLabel () ;
    private JLabel lblIcone = new JLabel ();
    private JButton cmdOK = new JButton ("OK");
```

Le constructeur initialise la boîte (titre, modale, position, non re-taillable) et fait en sorte que la mémoire utilisée soit "déposée" (récupérée) à la fermeture.

```
    public APropos(JFrame parent, String titre)
    {
        super(parent, "A Propos de " + titre, true);

        Point loc = parent.getLocation();
        this.setLocation((int) loc.getX()+50,(int)loc.getY()+100);
        this.setResizable(false);
        this.setDefaultCloseOperation(DISPOSE_ON_CLOSE );
        initControles();
    }
```

Les contrôles sont initialisés et ajoutés aux panneaux. Les panneaux sont eux-mêmes ajoutés à la zone client.

```
private void initControles()
{
    JPanel zoneClient = (JPanel) this.getContentPane();
    zoneClient.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

    JPanel panDroite = new JPanel (new GridLayout(3,1,10,10));
    panDroite.add (lblDescription);
    panDroite.add (lblCopyright);
    panDroite.add (lblVersion);

    JPanel panHaut = new JPanel(new FlowLayout(FlowLayout.CENTER, 20, 0));
    panHaut.add (lblIcone);
    panHaut.add (panDroite);

    JPanel panBas = new JPanel ();  // FlowLayout par défaut
    cmdOK.addActionListener(this);  // Voir plus bas
    panBas.add(cmdOK);

    zoneClient.add (panHaut, BorderLayout.NORTH);
    zoneClient.add (panBas, BorderLayout.SOUTH);
}
```

Pour éviter une classe incluse, il est possible de faire en sorte que la classe APropos implémente, elle-même, l'interface *ActionListener*. Elle est donc ainsi capable de récupérer l'événement

ActionEvent, et devient elle-même écouteur. C'est la raison pour laquelle le bouton est associé à l'écouteur *this*. En réponse à un clic sur le bouton OK la boîte est "déposée" (fermée et vidée de la mémoire) comme défini dans le constructeur

```
   public void actionPerformed(ActionEvent e)
   {
      if (e.getSource() == cmdOK) this.dispose();
   }
```

Pour rendre la classe réutilisable il faut prévoir des *setters* publics permettant de définir la description, la version, le copyright et l'icône qui seront affichés dans les contrôles. Remarquons l'instruction `this.pack()` qui permet d'ajuster la taille de la boîte de dialogue en fonction de la taille idéale des contrôles (dépendant du contenu).

```
   public void setDescription (String texte)
   {
      lblDescription.setText(texte);
      this.pack();
   }


   public void setVersion (String texte)
   {
      lblVersion.setText(texte);
      this.pack();
   }

   public void setCopyright (String texte)
   {
      lblCopyright.setText(texte);
      this.pack();
   }

   public void setIcone (String url)
   {
      lblIcone.setIcon(new ImageIcon(url));
      this.pack();
   }
}
```

Code complet de l'ouverture de la boîte

Création d'un objet instance de la classe APropos, appel des *setters* pour fixer la valeur des contrôles et ouverture de la boîte

```
   private void mnuAideAPropos_click()
   {
      APropos dlg = new APropos(this, "WebMail");
      dlg.setDescription("Envoi de mails à un destinataire");
      dlg.setCopyright("JC Rigal (c) 2003");
      dlg.setVersion("Version 1.0");
      dlg.setIcone("MailLogo.gif");
      dlg.show();
   }
```

Sous-classement des contrôles

Ce chapitre correspond à une approche ultra simplifiée du concept de JavaBean sur lequel repose tous les composants Swing. Un composant JavaBean est un composant réutilisable qui, s'il est graphique, peut être intégré à la barre d'outils d'environnement de développement de type RAD comme JBuilder, NetBeans … Un composant JavaBean à ses champs propres, ses méthodes et propage des événements spécifiques à des listener spécialisés. Seule la notion de composant logiciel réutilisable sera étudiée ici.

Il existe de nombreux exemples, dans l'API Swing, de classes conceptualisant des contrôles

spéciaux, dérivées d'autres classes plus générales. Ainsi, un *JPasswordField* est un *JTextField* spécial qui affiche des * rendant la saisie non visible. Le *JTextField* standard à été sous-classé pour répondre à un besoin spécifique. Il est possible d'étendre cette particularité aux contrôles particuliers de l'application. Par exemple une application de supervision pourrait avoir besoin d'un *JVoyant* considéré comme un *JTextField* non éditable, avec une bordure et un fond qui passe en couleur ou en blanc suivant son état, allumé ou éteint.


```
        public   class   JVoyant   extends
JTextField
   {
      private Color m_colOn;
      private Color m_colOff;

      public Voyant(Color col)
      {


         m_colOn = col;
         m_colOff = this.getBackground();
         this.setPreferredSize(new Dimension (50,20));
         this.setEditable(false);
         this.setBorder(BorderFactory.createLoweredBevelBorder());
      }

      public void setOn()
      {
         this.setBackground(m_colOn);
      }

      public void setOff()
      {
         this.setBackground(m_colOff);
      }
   }
```

Dans l'application les *JVoyant* seront utilisés comme n'importe quel contrôle Swing

```
   JVoyant v1 = new JVoyant(Color.RED);
   v1.setToolTipText("Alarme");     // méthode héritée
   if ( . . . ) v1.setOn();
```

Ces contrôles sous-classés peuvent disposer de leur propre politique de gestion interne des événements. Ainsi le *JVoyant* peut disposer d'un objet de classe *Timer* assurant le clignotement du voyant.

```
   public class Voyant extends JTextField
   {
      . . .

      private Timer tempo = new Timer(300, new AppActionListener());


      public void setBlink()
      {
         tempo.start();
      }

      class AppActionListener implements ActionListener {

         public void actionPerformed(ActionEvent e)
         {
            if (e.getSource() == tempo) tempo_time();
         }
      }

      private void tempo_time()
      {
         this.setBackground(this.getBackground()==m_colOff ? m_colOn
                                                           : m_colOff );
      }
```

```
}
```

Séparation classes UI et classes métier

Le but de ce chapitre va au-delà du contexte initial d'étude car il aborde des recommandations d'architecture pour une application quelconque. Celles-ci s'appliquent bien sur aux applications SDI utilisant une interface utilisateur basée sur Swing.

Modèle MVC

Le Model-View-Controller (MVC) est un modèle de conception logicielle largement répandu; qui a été créé dans les années 1980 par Xerox PARC pour Smalltalk-80. Plus récemment, il a été recommandé comme modèle pour la plate-forme J2EE de Sun et il gagne fortement en popularité auprès des développeurs. Le modèle MVC représente un complément fort utile aux outils du développeur, quel que soit le langage utilisé.

MVC est un modèle de conception qui impose la séparation entre les données, les traitements et la présentation. C'est pour cette raison que l'application est divisée en trois composants fondamentaux: le modèle, la vue et le contrôleur.

Chacun de ces composants tient un rôle bien défini.

La **vue** correspond à l'interface avec laquelle l'utilisateur interagit. Dans notre cas il s'agit d'une ou plusieurs classes UI utilisant des composants Swing. Aucun traitement n'est effectué dans la vue (hormis ceux correspondant à la "mécanique" de l'interface) elle sert uniquement à afficher les données et permettre à l'utilisateur d'agir sur ces données.

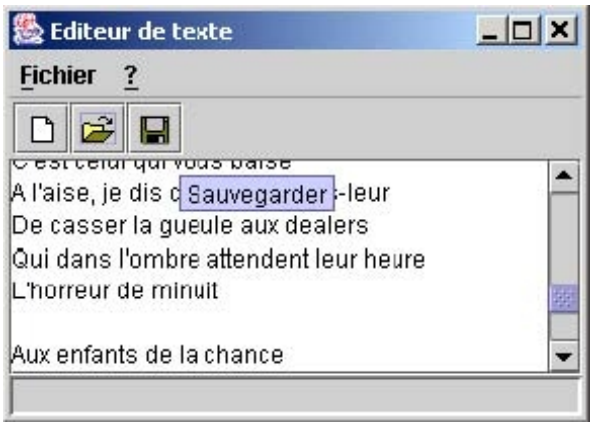
Le deuxième composant, le **modèle**, représente les données et les règles métier. C'est là que s'effectuent les traitements. Les bases de données en font partie, de même que des objets métier. Les données renvoyées par le modèle sont indépendantes de la présentation, c'est-à-dire que le modèle ne réalise aucune mise en forme. Les données d'un seul modèle peuvent ainsi être affichées dans plusieurs vue.

Enfin, le **contrôleur** interprète les requêtes de l'utilisateur et appelle les méthodes du modèle et de la vue, nécessaires pour répondre à la requête. Ainsi, lorsque l'utilisateur clique sur un contrôle, le contrôleur intercepte la requête et détermine quelle portion du modèle et quelle portion

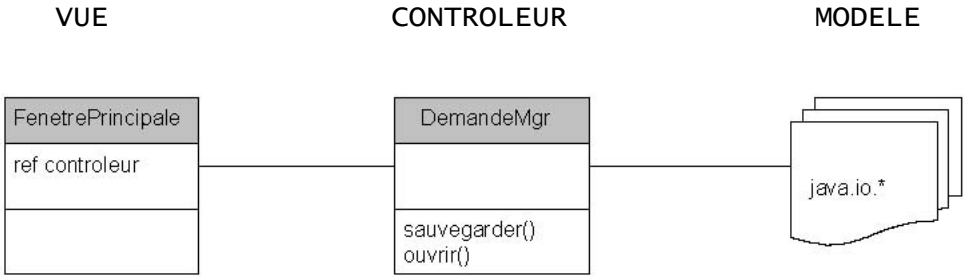
de la ou des vues doivent êtres associés.

Application Editeur de texte

L'application qui va être présentée est un éditeur de texte simplifié ou seules les fonctionnalités "Nouveau" "Ouvrir" et "Sauvegarder" sont implémentées



L'application est composée de 2 classes et utilise le package *io* pour accéder à des fichiers texte



La classe Application crée la vue et le contrôleur. L'objet instance de la classe FenetrePrincipale doit avoir une référence sur l'objet instance de la classe DemandeMgr pour pouvoir appeler les méthodes.

```
public class Application
{

   public Application ()
   {
      DemandeMgr ctrl = new DemandeMgr ();
      FenetrePrincipale frame = new FenetrePrincipale(ctrl);
   }

   public static void main(String args[]) . . .
}
```

Le constructeur de la classe FenetrePrincipale est légèrement différent des exemples précédents. Il doit mémoriser une référence sur le contrôleur.

```
public class FenetrePrincipale extends JFrame
{
   // Création des objets composants

   private DemandeMgr m_ctrl;

   public FenetrePrincipale(DemandeMgr ctrl)
   {
      m_ctrl = ctrl;
      // Intitialisation de la fenêtre
      initMenu();

      initControles();
   }
```

Les méthodes applicatives ne contiennent que du "code Swing" et appèlent les méthodes de la classe contrôleur en leur passant en argument le contrôle sur lequel elles doivent agir.

```
   private void mnuFichierOuvrir_click()
   {
      JFileChooser dlg = new JFileChooser();
      if (dlg.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
      m_ctrl.ouvrir(txtTexte, dlg.getSelectedFile());
   }

   private void mnuFichierSauver_click()
   {
      JFileChooser dlg = new JFileChooser();
      if (dlg.showSaveDialog(this) == JFileChooser.APPROVE_OPTION)
      m_ctrl.sauvegarder(txtTexte, dlg.getSelectedFile());
   }
}
```

La classe DemandeMgr contient des méthodes publiques assurant la prise en compte des actions de l'opérateur (appelées par les méthodes applicatives de FenetrePrincipale), l'extraction des informations du modèle (objets de classe *PrintWriter* et *BufferedReader*) et la mise à jour de la vue (*JTextArea*)

```
public final class DemandeMgr
{
   public DemandeMgr()
   {
   }

   public void sauvegarder (JTextArea txt, File f)
   {
      try {
         PrintWriter WModele = new PrintWriter (new FileWriter (f));
         WModele.print(txt.getText());
         WModele.close();
      } catch (IOException e) {}
   }

   public void ouvrir (JTextArea txt, File f)
   {
      try {
         BufferedReader RModele = new BufferedReader (new FileReader (f));
         txt.setText("");
         while (RModele.ready()) txt.append(RModele.readLine() + "\n");
         RModele.close();
      } catch (IOException e) {}
   }
}
```

Règle de conception de l'interface utilisateur

(*extrait du guide de l'interface Utilisateur Microsoft*)

Quelques outils RAD facilitent la création d'une interface utilisateur Swing en offrant la possibilité de déplacer les contrôles sur une feuille. Un minimum d'organisation préalable fera toute la différence quant à la fonctionnalité de l'application.

La composition ou la présentation d'une *JFrame* ne doit pas nuire à son caractère esthétique qui a aussi un impact considérable sur la fonctionnalité d'une application. La composition comprend des facteurs déterminants tels que le positionnement des contrôles, la cohérence entre les éléments, l'accessibilité, l'utilisation d'espaces vierges et une certaine simplicité dans la conception.

La simplicité est sans doute un des principes les plus importants à respecter lors de la création d'une interface. Si celle-ci semble complexe au niveau des applications, c'est qu'elle l'est réellement. Pour créer une interface conviviale, il faut prendre le temps de réfléchir au préalable à sa conception. De même, d'un point de vue esthétique, une conception nette et simple est toujours préférable.

Positionnement des contrôles

Tous les éléments ne sont pas d'une importance égale dans la plupart des interfaces. Une conception soigneuse est indispensable pour que l'utilisateur puisse immédiatement repérer les éléments les plus importants. Les éléments importants ou souvent utilisés doivent être placés bien en vue, alors que les éléments moins importants peuvent être relégués à des emplacements moins visibles.

La plupart des langues se lisent de gauche à droite et de haut en bas par rapport à une page. Il en est de même sur un écran d'ordinateur. Comme les yeux sont d'abord attirés par la partie supérieure gauche de l'écran, l'élément le plus important doit être placé à cet endroit. Par exemple, si les informations d'une feuille concernent un client, le champ du nom doit apparaître à l'endroit le plus visible. Les boutons OK ou Suivant doivent être placés dans la partie inférieure droite de l'écran car l'utilisateur n'y accédera que lorsqu'il aura terminé de traiter la feuille.

Il est aussi important de regrouper les éléments et les contrôles. Il faut les regrouper d'une façon logique, selon leurs fonctions ou leurs relations. Comme leurs fonctions sont liées, les boutons permettant de consulter une base de données doivent être regroupés visuellement, et non dispersés sur la feuille. La même règle s'applique aux informations: les champs de noms et d'adresses sont généralement regroupés, car ils sont étroitement liés. Dans la plupart des cas, il est possible d'utiliser des panneaux avec bordures de classe *JTitleBorder* pour

souligner les relations entre les contrôles.

Cohérence entre les éléments de l'interface

La cohérence est une vertu dans la conception d'une interface utilisateur. Une présentation cohérente contribue à l'harmonie d'une application qui doit constituer un ensemble coordonné. Le manque de cohérence peut rendre une application confuse, chaotique et désorganisée. L'utilisateur peut alors en conclure que l'application est d'une qualité médiocre et il doutera de sa fiabilité.

Pour obtenir une cohérence visuelle, il est conseillé de définir une stratégie de conception et des conventions de styles avant de passer à la création. Certains éléments tels que les types de contrôles, les normes de dimensionnement et de regroupement de contrôles et les choix de polices doivent être définis à l'avance.

Swing offre une vaste gamme de contrôles. Ne pas essayer de tous les utiliser; sélectionner plutôt un sous-ensemble de contrôles qui s'adapte le mieux à l'application. Les contrôles de zone de liste, de zone de liste éditable, de grille et d'arborescence peuvent tous servir à présenter des listes d'informations, mais il est préférable de se limiter à un seul style dans la mesure du possible.

Essayer également d'utiliser les contrôles de façon appropriée. Alors qu'un contrôle *JTextField* peut être défini en lecture seule et servir à afficher du texte, un contrôle *JLabel* est généralement plus approprié à cette fin. La cohérence entre les différentes feuilles de l'application est également primordiale. Il est préférable de sélectionner un style et le conserver pour toutes les feuilles de l'application.

Utilisation d'espaces vierges

L'utilisation d'espaces vierges peut contribuer à mettre en valeur des éléments et à améliorer les fonctionnalités de l'interface. Un espace vierge n'est pas nécessairement un espace blanc; il peut aussi s'agir de l'intervalle entre les contrôles et autour d'eux. Si une feuille contient trop de contrôles, l'interface risque de devenir confuse et la recherche d'un champ ou d'un contrôle peut s'avérer fastidieuse. N'hésiter pas à utiliser des espaces vierges pour mettre en valeur les éléments.

Un intervalle constant entre les contrôles et l'alignement des éléments verticaux et horizontaux rend l'application plus lisible, exactement comme un magazine où le texte est disposé en colonnes égales, avec un espacement homogène entre les lignes.

Couleurs et images

L'utilisation de couleurs dans l'interface peut améliorer son aspect visuel, mais ne pas en abuser. Le choix des couleurs est très subjectif; le goût de l'utilisateur ne correspond pas forcément à celui du développeur.euse. Ne pas oublier que la couleur peut évoquer de vives émotions et, pour les applications internationales, certaines couleurs peuvent avoir une signification culturelle. Il est donc conseillé d'utiliser des couleurs traditionnelles, douces et neutres.

Le choix des couleurs peut aussi être influencé par les utilisateurs auxquels s'adresse l'application, ainsi que par le ton ou l'esprit à transmettre. Des couleurs vives rouge, vert et jaune peuvent être utilisées pour attirer l'attention sur une zone importante ou la mettre en valeur. En règle générale, il faut limiter le nombre de couleurs dans l'application et se tenir à la palette des 16 couleurs standard car le dégradé peut entraîner la disparition de certaines couleurs si l'application est exécutée sur un écran 16 couleurs. Ne pas oublier les personnes daltoniennes qui ne peuvent distinguer les différentes combinaisons des couleurs de base telles que le rouge et le vert. Par exemple, il leur est impossible de lire un texte en caractères rouges sur un fond vert.

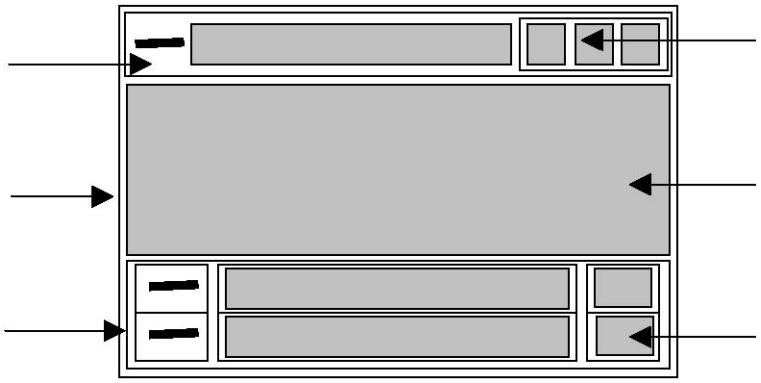
Images et icônes

Les images et les icônes peuvent aussi ajouter un attrait visuel à l'application. Mais, une fois encore, elles doivent être utilisées avec parcimonie et prudence. Les images peuvent transmettre des informations concises sans qu'il soit nécessaire d'ajouter un texte, mais elles sont souvent perçues différemment selon la culture des utilisateurs.

Les barres d'outils avec des icônes représentant diverses fonctions s'avèrent très utiles dans une interface, mais si l'utilisateur ne peut identifier la fonction représentée par l'icône, elles peuvent ralentir sa productivité. Par exemple, beaucoup d'applications utilisent une feuille de papier avec un angle replié pour représenter l'icône *Nouveau fichier.* Il peut exister une meilleure métaphore pour cette fonction, mais l'utilisateur peut être troublé si elle est représentée différemment.

Il est aussi important de tenir compte de la signification culturelle des images. Beaucoup de programmes utilisent l'image d'une boîte aux lettres de style rural avec un drapeau pour représenter les fonctions de courrier. Ce type de boîte aux lettres étant utilisé essentiellement aux Etats-Unis, il est possible que les utilisateurs d'autres pays ou d'une autre culture ne comprennent pas la signification de cette icône.

Correction de l'exercice sur l'agencement des contrôles



```
/**
 * @author DISERVER
 * Version de Philippe Bouget

Liste des composants / type de gestionnaire de positionnement :

      panCentre / JScrollPane
      panBasDroit, panBasGauche, panBasMilieu / GridLayout

      panTool / FlowLayout

      zoneClient / BorderLayout

      panHaut / BorderLayout

      panBas / BorderLayout
 */

// gestionnaires de positionnement :
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;

import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;


public class FenetrePrincipale extends JFrame
      {
   private JButton cmdOuvrirIcon = new JButton(new ImageIcon("ouvrirFichier.png"));
   private JButton cmdFermerIcon = new JButton(new ImageIcon("fermerFichier.png"));
   private JButton cmdAidezMoi = new JButton(new ImageIcon ("aide.png"));
   private JButton cmdOuvrir = new JButton("Ouvrir");
   private JButton cmdAnnuler = new JButton("Annuler");
   private JTextArea txtTexte = new JTextArea();
   private JTextField txtNom = new JTextField();
   private JComboBox cboExplorer = new JComboBox();
   private JComboBox cboType = new JComboBox();

   public FenetrePrincipale()
   {
            this.setSize(500,300);
            this.setTitle("Exemple Editeur de texte");
            this.setDefaultCloseOperation(EXIT_ON_CLOSE);
            // appelle du programme d'initialisation des composants :
            initControles();
         }

         private void initControles()
         {
        cmdOuvrirIcon.setToolTipText("Permet d'ouvrir un fichier");
```

```
        cmdFermerIcon.setToolTipText("Permet de fermer un fichier");
        cmdAidezMoi.setToolTipText("affichez l'aide ! (mais y en a pas !)");
        cmdOuvrir.setToolTipText("Ouvrir !");
        cmdAnnuler.setToolTipText("Annuler tout !");
        cboExplorer.setEditable(true);
        cboType.setEditable(true);

        JPanel zoneClient = (JPanel) this.getContentPane();
        zoneClient.setLayout(new BorderLayout(10,10));
        zoneClient.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

        // panneau des boutons outils :
        JPanel panTool = new JPanel (new FlowLayout(FlowLayout.CENTER, 0, 0));
        panTool.add (cmdOuvrirIcon);
        panTool.add (cmdFermerIcon);
        panTool.add (cmdAidezMoi);

        // panneau du haut comprenant la combo et le panneau boutons outils
        JPanel panHaut = new JPanel (new BorderLayout(5,5));
        panHaut.add(new JLabel("Explorer"), BorderLayout.WEST);
        panHaut.add(cboExplorer, BorderLayout.CENTER);
        panHaut.add(panTool, BorderLayout.EAST);

        // panneau du centre avec la zone de texte Area :
        JScrollPane panCentre = new JScrollPane (txtTexte);
        // barre verticale toujours présente

panCentre.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

        JPanel panBas = new JPanel (new BorderLayout(10,10));
        JPanel panBasGauche = new JPanel (new GridLayout(2,1,10,10));
        panBasGauche.add (new JLabel ("Nom :"));
        panBasGauche.add (new JLabel ("Type :"));
        panBas.add (panBasGauche, BorderLayout.WEST);

        JPanel panBasDroit = new JPanel (new GridLayout(2,1,10,10));
        panBasDroit.add (cmdOuvrir);
        panBasDroit.add (cmdAnnuler);
        panBas.add (panBasDroit, BorderLayout.EAST);

        JPanel panBasMilieu = new JPanel (new GridLayout(2,1,10,10));
        panBasMilieu.add (txtNom);
        panBasMilieu.add (cboType);
        panBas.add (panBasMilieu, BorderLayout.CENTER);

        zoneClient.add (panHaut, BorderLayout.NORTH);
        zoneClient.add (panCentre, BorderLayout.CENTER);
        zoneClient.add (panBas, BorderLayout.SOUTH);
        }
         public static void main(String args[])
         {
               (new FenetrePrincipale()).setVisible(true);
         }
      }
```

Astuce pour que les braillistes

Se déplacer dans une JTextArea :

Pour pouvoir utiliser les touches de Tab, Shift-Tab, etc avec un composant JTextArea, il faut créer une classe qui hérite de JTextArea et redéfinir une seule méthode, **IsManagingFocus()** pour l'obliger à renvoyer false afin que ce composant ne gère pas automatiquement les touches habituelles pour se déplacer dans une fenêtre :

```
public class JTextAreaMoi extends JTextArea
{
…
        public Boolean isManagingFocus()
        {
                Return false;
        }

}
```