Java

Programmation des interfaces graphiques

Jean-Baptiste Vioix

Table des matières

Avant-propos Ce document n'a pas pour vocation d'être exhaustif, il est conçu pour être nécessaire et suffisant pour pouvoir suivre le cours. Le lecteur trouvera de nombreux compléments sur Internet et notamment sur le site de Sun qui documente l'API : `http://java.sun.com/j2se/1.5.0/docs/api/` et `http://java.sun.com/docs/books/tutorial/uiswing/` Par convention, les mots clefs sont écrits avec une fonte particulière pour être facilement distingués. De même, seul le nom des méthodes est spécifié, les paramètres ne sont pas systématiquement donnés pour éviter d'alourdir le texte. On retrouvera toutes les méthodes détaillées sur le site précédent. Enfin, les packages contenant les classes ne sont pas spécifiés. Les outils de développement actuels (comme Eclipse) sont capables de compléter les importations automatiquement.

Chapitre 1

Présentation de Swing

1.1 Historique de Swing

Dès ses premières versions Java comprenait une bibliothèque de création d'interfaces utilisateurs graphiques (Graphic User Interface). Cette bibliothèque nommée AWT (Abstract Window Toolkit) devait permettre d'obtenir des interfaces graphiques identiques sur tous les systèmes. AWT utilisait une approche par peers. Les composants utilisés par AWT étaient associés à leurs équivalents dans le système d'exploitation (on parle de composants lourds). Par exemple, pour construire un bouton, AWT appelait le composant bouton du système d'exploitation (donc un bouton Windows sous Windows, un bouton Mac sur un Mac,...). Pour chaque composant, il y avait donc une couche d'adaptation entre Java et le système d'exploitation. Rapidement, des problèmes de compatibilité sont apparus, certains composants se comportaient différemment d'un système à l'autre. De plus, des éléments importants manquaient. Durant l'existence de Java 1.1, Sun a travaillé sur une nouvelle API (Application Programming Interface) de création de GUI. Reprenant les éléments efficace de AWT, Sun a complètement refondu la bibliothèque et l'a nommée Swing. Parmi les modifications importantes, les composants sont devenus des éléments de la JVM (ce sont des composants légers). Dans une application Swing, seule la fenêtre principale est construite par le système d'exploitation, tous les autres composants sont dessinés par la JVM. La relation entre Java et Swing est très intime. Contrairement à la plupart des interfaces graphiques, Swing est directement liée au langage Java. A l'inverse de l'interface Windows qui peut être programmée avec C++, Pascal Object (Delphi), Basic(,...) ou des bibliothèques GTK/GTK+ qui peuvent être programmées en C/C++, ADA, Perl(,...) Swing ne peut être utilisée qu'avec Java. Cette relation permet d'utiliser toute la puissance de la programmation objet de Java pour construire des interfaces graphiques.

1.2 Concepts de bases

Swing partage de nombreux concepts avec les bibliothèques de création d'interfaces graphiques tout en les adaptant aux spécificités de Java. La construction d'interfaces graphiques est basée sur quatre éléments principaux.

1.2.1 Les composants

Les éléments constituant les interfaces graphiques sont appelés composants (ce sont des boutons, des textes, des images, des fenêtres,...). Dans Swing, tous les composants descendent de la même

classe. Cette hiérarchisation très forte permet de mettre en oeuvre facilement les composants et d'en créer d'autre à l'aide de l'héritage.

1.2.2 Les conteneurs

Certains composants sont capables d'en accueillir d'autres, ce sont les conteneurs. Swing propose une grande variété de conteneurs pour pouvoir répondre à tous les besoins. D'un point de vue informatique, les conteneurs sont des descendants de la classe `Container`.

1.2.3 Les gestionnaires de placement

Le placement des composants est souvent confié à un gestionnaire de placement. Contrairement aux applications écrites pour un seul système d'exploitation, les applications Java s'exécutent sur des environnements très différents (polices de caractères, taille et résolution de l'écran,...). Un positionnement absolu des composants est donc déconseillé, on préfère utiliser un positionnement relatif. Swing propose de nombreux gestionnaires de placement pour construire des applications dont la probabilité est assurée.

1.2.4 Les gestionnaires d'événements

Les actions de l'utilisateur sont représentées par des événements. Le programme peut modifier son comportement en fonction de certains événements. Swing propose une méthode souple de gestion des événements. Les événements sont des objets qui sont transmis par un composants vers un gestionnaire d'événements. Ce gestionnaire modifie ensuite le programme pour prendre en compte les besoin de l'utilisateur. D'une manière informatique, la gestion des événements est séparée de la création de l'interface ce qui facilite les modifications éventuelles du programme.

Chapitre 2

Les principaux composants de Swing

L'API Swing permet de construire des interfaces graphiques très riches à l'aide d'une collection de composants complète (boutons, menus, fenêtres...). Nous allons présenter une grande partie des composants de Swing dans les prochaines sections. Ce chapitre est rédigé en fonction d'une hiérarchie logique, il est tout à fait normal que certains exemples soit difficilement compréhensible lors de la première lecture. Ces exemples font appel à des notions abordées ultérieurement. Une seconde lecture attentive doit permettre la compréhension de ces exemples...

2.1 Les classes mères : `Component` et `JComponent`

Tous les composants de Swing descendent de la classe JComponent, qui descend elle même de la classe Component de AWT. Cette hiérarchie permet de proposer de nombreuses méthodes communes à tous les éléments.

Relation entre les composants

Tous les composants sont hébergés par un conteneur (sauf les conteneurs primaires). Il est possible de connaître le conteneur d'un composant en utilisant la méthode `getParent`. On ajoute un composant dans un conteneur en utilisant la méthode `add`.

Les propriétés géométriques

La position d'un composant peut être obtenue avec la méthode `getLocation`. Le coin supérieur gauche du composant parent est utilisé comme position de référence. La position peut être modifiée avec la méthode `setLocation`. Toutefois, il n'est pas recommandé de positionner les objets de manière manuelle. Swing propose les gestionnaires de placement pour disposer les composants automatiquement tout en assurant une compatibilité graphique entre les différents systèmes d'exploitation. La taille d'un composant peut être fixée avec la méthode `setSize` (et lue avec la méthode `getSize`). En pratique, on utilise peut ces méthodes car les composants sont capable de calculer la taille qui leur est préférable pour assurer un affichage optimal. La fonction `getPreferedSize` permet d'obtenir cette information. Par contre, il est déconseillé d'utiliser la méthode `setPreferedSize` pour modifier cette valeur.

```
//...
   monComposant.setSize(monComposant.getPreferedSize());
```

```
//...
```

Si le composant est déjà affiché quand sa taille (ou sa position) est modifiée, sa méthode `revalidate` doit être appelée pour qu'il soit redessiné. On peut « cacher » un composant en utilisant la méthode `setVisible`. Le composant prend son emplacement mais il n'est pas dessiné, donc l'utilisateur ne le voit pas. Souvent, on préfère utiliser la méthode `setEnabled`. Dans ce cas, le composant est quand même affiché mais il est inactif. Sur la plupart des systèmes d'exploitation les composants inactifs ont un aspect particulier (souvent ils sont grisés).

Les infobulles : `JToolTip`

Tous les composants dérivant de `JComponent` peuvent être agrémentés d'info-bulles. Ce sont des petites « boites » contenant un texte qui présente le rôle du contrôle. C'est la classe `JToolTip` qui contrôle la bulle d'aide. En pratique, il n'est pas nécessaire d'instancier un objet. La classe `JComponent` propose une méthode qui permet de créer directement le composant `JToolTip` et de l'associer au composant. C'est la méthode `setToolTipText` qui instancie un composant `JToolTip` et l'associe au composant :

```
//...
   monComposant.setToolTipText("Coller depuis le presse-papier")
      ;
//...
```

Lorsque la souris reste quelques secondes sur le composant la bulle d'aide s'affiche (figure 2.1) :

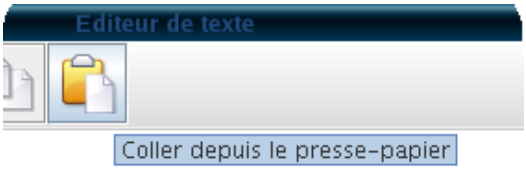


FIG. 2.1 – Utilisation d'une bulle d'aide pour un bouton

Les bordures

Les composants peuvent être encadrés par une bordure. Plusieurs bordures sont disponibles en creux, en relief, avec un texte,... La classe `BorderFactory` fournit de nombreuses méthodes statiques pour créer des bordures prédéfinies. Par exemple pour créer une bordure avec un titre :

```
//...
   monComposant.setBorder(BorderFactory.createTitleBorder("Une
      bordure"));
//...
```

Cette bordure est souvent utilisée pour matérialiser des blocs d'informations.

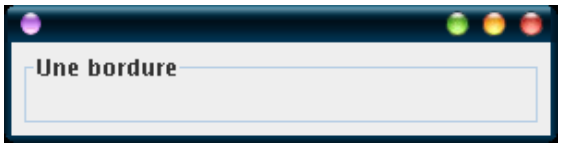


FIG. 2.2 – Un exemple de bordure

2.2 Les conteneurs primaires

Les conteneurs primaires sont les seuls composants qui sont dessinés par le système d'exploitation, ce sont donc des composants lourds. Toute application graphique doit utiliser un conteneur primaire. Trois types de conteneurs primaires existent : les applets (`JApplet`), les fenêtres (`JFrame` et `JWindow`) et les boites de dialogues (`JDialog`).

2.2.1 Propriétés communes

Les quatre conteneurs primaires sont construit sur le même modèle. La couche supérieure est un le `GlassPane`, un panneau transparent qui est utilisé pour la gestion des événements. Sous ce panneau se situe le `contentPane` qui accueille les composants graphiques, par défaut c'est un `JPanel`. Il est contenu par le `LayeredPane` qui peut être utilisé pour empiler des composants à des "profondeurs" différentes. Pour les `JApplet`, `JFrame` et les `JDialog`, il contient aussi une barre de menu, accessible via la méthode `setJMenuBar`, par défaut cet élément est `null`. Tous ces éléments sont contenu dans un élément principal de type `JRootPane`.

Accès au `contentPane` pour ajouter des composants

Les composants (boutons, labels,...) sont placés sur le `contentPane`. Par défaut ce conteneur est un `JPanel`. Il est possible d'accéder à ce conteneur avec la méthode `getContentPane`, puis d'appeller la méthode `add` pour ajouter des composants. Cette méthode accepte un paramètre, un objet de type `JComponent` (un de ces nombreux descendants car la classe est abstraite).

```
JFrame fen = new JFrame();
Container cont = fen.getContentPane();
cont.add(myComponent);
```

On peut simplifier l'écriture en omettant la variable intermédiaire `cont`.

```
fen.getContentPane().add(myComponent);
```

Une autre approche consiste à utiliser un nouveau conteneur que l'on place ensuite dans le conteneur primaire. On utilise la méthode `setContentPane` pour spécifier un nouveau conteneur.

```
fen.setContentPane(new monConteneur());
```

La seconde méthode conduit à une création d'objet (et une destruction) supplémentaire. Elle est toutefois obligatoire si l'on veut utiliser un conteneur qui n'est pas un `JPanel` (le conteneur par défaut) comme par exemple un `JTabbedPane` ou un `JDesktopPane`.

2.2.2 Les applets : `JApplet`

Une des idées fortes de Java lors de sa création était la possibilité de pouvoir exécuter des applications à l'intérieur du navigateur Internet.

Cette idée ingénieuse n'a pas eu le retentissement voulu dans les premières années de Java, la lenteur des premières connexions Internet étant pour beaucoup dans cette échec. De plus, diverses querelles judiciaires entre Sun et Microsoft ont bloquées pendant longtemps les applets à la version 1.1 de Java[1]. Ces conflits sont maintenant résolus, les applets disposent de toute la puissance de l'interface Swing.

Une applet est une application téléchargée par le navigateur et exécutée par le poste client via la machine virtuelle. L'applet est capable de réagir aux événements du clavier et de la souris comme n'importe quelle application Java.

Les applets sont exécutées dans un environnement sécurisé à l'intérieur du navigateur. Les restrictions suivantes sont généralement appliquées :

- Aucun accès aux fichiers sur la machine locale
- Pas de connexion réseau vers l'extérieur sauf vers le serveur où elles ont été téléchargées
- Interdiction de créer de nouveaux processus

Une certaine souplesse vis-à-vis de ces règles est toutefois possible en signant l'applet. Une applet signée comporte une identification numérique de son concepteur. L'utilisateur peut lui donner certains droits la considérant comme fiable.

Pour construire une applet, on doit utiliser la classe `JApplet` comme conteneur principal de l'application. L'applet est ensuite insérée dans une page web grâce à la balise HTML `<applet>`, dans le cas d'un site en XHTML, on utilise la balise `<object>`

Les applets ne seront pas plus détaillées dans ce document. De nombreuses références sont disponibles sur le sujet, que ce soit sous la forme de pages internet ou d'ouvrage.

2.2.3 Les fenêtres : `JWindow`, `JFrame`

Swing propose deux types de fenêtres : les `JWindow` et les `JFrame`. Ces deux composants sont proches, ils descendent tous les deux de `Window` (composant de AWT) et ont donc de nombreux points communs.

Méthode communes à `JWindow` et `JFrame`

Pour créer une fenêtre, le constructeur est appelé. Par défaut, un fenêtre qui est crée n'est pas affichée. Pour l'afficher, il faut faire appel à la méthode `setVisible`.

```
JFrame fen = new JFrame(); // identique pour JWindow
fen.setVisible(true);
```

La taille d'une fenêtre dépend des éléments qu'elle contient. Afin d'éviter de l'estimer ou de la calculer, Swing propose une méthode (`pack`) qui calcule la taille de la fenêtre en fonction de la taille préférée de ses les composants internes.

```
fen.pack();
```

Lors de l'appel de la méthode `pack`, la méthode `getPreferredSize` est appelée sur tous les composants pour connaître leurs dimensions. Ces informations sont utilisées pour calculer la dimension de la fenêtre.

[1] et donc à une interface graphique AWT limitée alors que Swing fonctionne beaucoup mieux...

Pour créer un programme exécutable en mode graphique, il suffit de créer une (ou plusieurs) fenêtre dans la classe qui contient la méthode statique `main`.

Lorsqu'une fenêtre n'est plus utile, elle peut être caché (avec `setVisible(false)`) ou détruite. Si cette fenêtre doit être réaffichée rapidement le mieux est de la cacher. Par contre, si elle n'est plus utile il convient de libérer les ressources associée avec la méthode dispose. Cette méthode libère les ressources alloués par le système d'exploitation, puis le *garbage collector* détruit la fenêtre. Quand il n'y a plus aucune fenêtre dans une application elle est quittée.

Différences entre `JFrame` et `JWindow`

La classe `JWindow` permet de créer une fenêtre graphique dans le système de fenêtrage utilisée. Cette fenêtre n'a aucune bordure et aucun bouton. Elle ne peut être fermée que par le programme qui l'a construite (ou en mettant fin à l'application via le système d'exploitation en tuant le processus associé). D'une manière générale, `JWindow` n'est utilisée que pour créer des fenêtres particulières comme les splash screens ou des écrans d'attente.

```
import javax.swing.JWindow;
public class TestJWindow {
   public static void main(String[] args) {
      JWindow fenetre = new JWindow();
      fenetre.setSize(300,300);
      fenetre.setLocation(500,500);
      fenetre.setVisible(true);
   }
}
```

Comme pour tous les composants la méthode `setLocation` permet de positionner le composant à l'intérieur du conteneur. Dans le cas des `JFrame` et des `JWindow`, le conteneur est l'écran. Ce programme construit une fenêtre sans bordure et sans bouton, elle ne peut pas être fermée (sauf en détruisant la tâche associée).

FIG. 2.3 – Une fenêtre `JWindow`

La plupart des applications sont construites à partir d'une (ou plusieurs) `JFrame`. En effet, `JFrame` construit des fenêtres qui comportent une bordure, un titre, des icônes et éventuellement un menu.

```
import javax.swing.JFrame;
public class TestJFrame {
   public static void main(String[] args) {
      JFrame fenetre = new JFrame();
      fenetre.setSize(300,300);
      fenetre.setVisible(true);
      fenetre.setLocation(500,500);
   }
}
```

La fenêtre JFrame est une fenêtre standard du système d'exploitation.

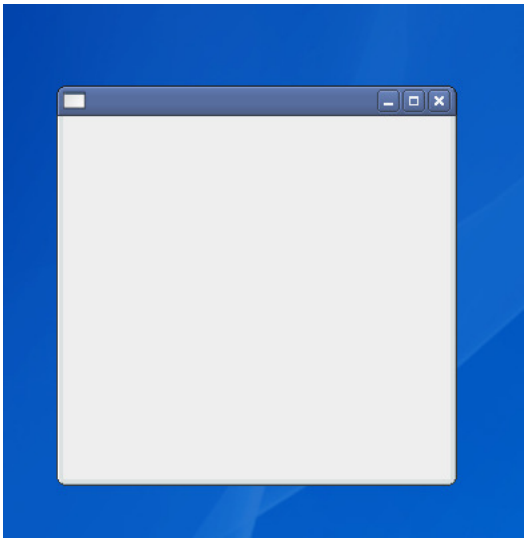


FIG. 2.4 – Une fenêtre `JFrame`

La position des icônes et la police du titre dépend donc du système. Par contre, l'icône représentant la fenêtre lorsqu'elle est réduite peut être modifié en utilisant la méthode `setIconImage`.

Par défaut, une `JFrame` n'est pas fermée lorsque l'on clique sur l'icône de fermeture mais cachée (comme avec `setVisible(false)`). Ce comportement peut être modifié avec la méthode `setDefaultCloseOperation`. Plusieurs comportement sont définis dans la classe `JFrame`. Pour fermer la fenêtre, on utilise le comportement `EXIT_ON_CLOSE` :

```
fra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Pour modifier le titre d'une `JFrame`, on utilise la méthode `setTitle()`. On peut aussi utiliser la version surchargée du constructeur `JFrame` :

```
JFrame fen = new JFrame("Ma premiere fenetre");
```

Un menu peut être ajouté au `JFrame` à l'aide de la méthode `setJMenuBar` (voir section 2.4.8).

2.2.4 Les boites de dialogues

Swing propose des boites de dialogues afin de communiquer rapidement avec l'utilisateur. Ces boites de dialogues sont crées par le système d'exploitation. La classe `JOptionPane` permet de créer des boites de dialogues génériques mais aussi de les modifier en fonction des besoins de l'application (en ajoutant des composants à l'intérieur). Les méthodes d'affichage sont statiques, il n'est donc pas nécessaire d'instancier une classe pour les utiliser.

Les dialogues de message

Ce sont les boites de dialogue les plus simple, elles informent l'utilisateur en affichant un texte simple et un bouton de confirmation. On peut aussi afficher un icône correspondant au message (point d'exclamation, symbole d'erreur,...). Elles restent affichée tant que l'utilisateur n'a pas cliqué sur le bouton et bloquent l'application en cours : on parle de boites de dialogues modales. Pour afficher un message on utilise la méthode `showMessageDialog` de la classe `JOptionPane`. Cette méthode propose de nombreuses options accessibles via la surcharge. `JOptionPane.showMessageDialog( fen,"Un message")`.

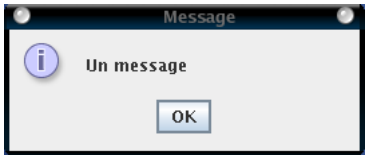


FIG. 2.5 – Une boite de dialogue simple

La variable `fen` est le composant parent. C'est souvent la fenêtre principale de l'application. La boite de dialogue sera positionnée au milieu de cette fenêtre. La variable `fen` peut être `null`. On peut modifier l'aspect de la fenêtre en fonction du type de message, par exemple pour afficher un dialogue d'avertissement :

```
JOptionPane.showMessageDialog(fen,"èProblme de érseau","Connexion au
érseau impossible.",JOptionPane.WARNING_MESSAGE);
```

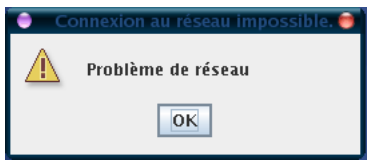


FIG. 2.6 – Une boite de dialogue d'alerte

Tous les éléments de la boite de dialogue peuvent être modifiés en utilisant la méthode `showMessageDialog`, de plus de nombreuses constantes (icônes, type de message,...) sont prévues.

Les dialogues de confirmation/question

Les boites de dialogues peuvent aussi être utilisées pour demander un renseignement à l'utilisateur. Le cas le plus courant est une question fermée[2]. La méthode `showConfirmDialog` affiche ce type de boite. Sans paramètre particulier, elle ajoute un bouton pour annuler la question :

```
JOptionPane.showConfirmDialog(fen,"Choisir une érponse");
```

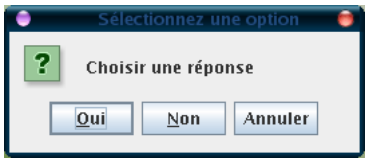


FIG. 2.7 – Une boite de dialogue de confirmation

La syntaxe est la même que pour `showMessageDialog`. Comme précédemment, on peut personnaliser la boite de dialogue avec des chaînes de caractères, par exemple : `JOptionPane .showConfirmDialog(fen,"Aimez vous Java ?","Examen de Java",JOptionPane. YES_NO_OPTION);`

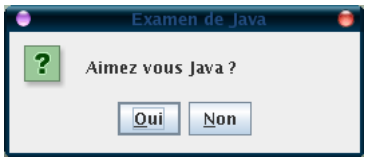


FIG. 2.8 – Une boite de dialogue de confirmation modifiée

La boite de dialogue renvoie un entier en fonction du choix de l'utilisateur. Cette valeur peut être utilisée pour modifier le comportement du programme. Cet entier est définit comme une constante de la classe `JOptionPane`. Par exemple pour le cas précédent :

```
int rep = JOptionPane.showConfirmDialog(fen,"Aimez vous Java ?",
   "Examen de Java",JOptionPane.YES_NO_OPTION);
  if (rep==JOptionPane.YES_OPTION){
  //...
  }
  if (rep==JOptionPane.NO_OPTION){
  //...
  }
```

Les dialogues de saisie

La méthode `showInputDialog` affiche une boite de dialogue comprenant une entrée de saisie (`JTextField`) en plus des boutons de validation. Après validation elle retourne une chaîne de caractères (`String`) si l'utilisateur a cliqué sur OK, sinon elle renvoie `null`. La ligne suivante :

[2] dont la réponse est oui ou non.

```
String rep = JOptionPane.showInputDialog(fen,"Entrez votre nom d
   'utilisateur");
```

conduit à l'affichage de la boite de dialogue de la figure 2.9.

FIG. 2.9 – Une boite de dialogue de saisie

La méthode `showInputDialog` est surchargée afin de pouvoir modifier l'aspect de la boite de dialogue. Pour changer le titre de la boite on va utiliser le code suivant :

```
String rep = JOptionPane.showInputDialog(fen,"Entrez votre nom
   d'utilisateur","Renseignement",JOptionPane.
      INFORMATION_MESSAGE);
```

afin d'obtenir le résultat présenté dans la figure 2.10.

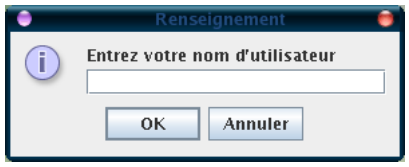


FIG. 2.10 – Une boite de dialogue de saisie modifiée

Il est aussi possible de proposer une valeur par défaut pour le champ de saisie lors de l'appel.

Construction de dialogues personnalisés

Pour construire des boites de dialogue plus complexes on utilise la méthode `showOptionDialog` . Cette méthode admet 8 paramètres (certains pouvant être à `null`). Elle renvoie un entier qui correspond au bouton qui a été cliqué. Le prototype est le suivant : `showOptionDialog( Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object initialValue)` La première variable `parentComponent` est la fenêtre parent de la boite de dialogue. La variable `message` est un (ou plusieurs si c'est un tableau) objet que l'on souhaite afficher. La troisième variable est une chaîne de caractères qui définit le titre de la fenêtre (`title`). Les boutons affichés sont configurés à l'aide de la variable `optionType` comme pour les autres boites de dialogue. La variable `messageType` permet d'afficher l'un des icônes prédéfinis dans la classe `JOptionPane`. Cet icône peut aussi être modifié à l'aide de la variable `icon`. Diverses options peuvent être configurées à l'aide de l'avant dernière variable. La variable `initialValue` permet de fournir des valeurs par défaut pour les différents champs.

Mise en oeuvre On peut par exemple construire une boite de dialogue qui demande à l'utilisateur de saisir ses nom et prénom. La boite contient 3 labels et 2 champs de saisie. Le code qui suit présente la construction de cette boite de dialogue. Les éléments constituant la boite sont instanciés et placés dans un tableau. La méthode `showOptionDialog` est appelée avec les différents paramètres de configuration.

```
JLabel labelNom = new JLabel("Nom :");
JLabel labelPrenom = new JLabel("Prenom :");
JTextField nom = new JTextField();
JTextField prenom = new JTextField();
JLabel lab = new JLabel("Entrez vos nom et éprnom");

Object[] tab = new Object[]{labelNom, nom, labelPrenom, prenom,
   lab};

int rep = JoptionPane.showOptionDialog(
   fen,
   tab,
   "Saisie des nom et prenom",
   JOptionPane.OK_CANCEL_OPTION,
   JoptionPane.INFORMATION_MESSAGE,
   null,
   null,
   null);
```

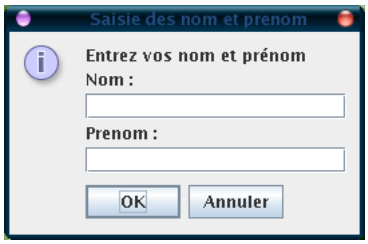


FIG. 2.11 – Une boite de dialogue personalisée

Après l'appel il ne reste plus qu'à lire les valeurs saisie par l'utilisateur s'il a validé son action.

```
if (rep==JOptionPane.OK_OPTION){
   System.out.println("Nom : "+nom.getText()+" prenom :
   "+prenom.getText());
}
```

2.3 Les conteneurs secondaires

Swing propose de nombreux conteneurs secondaires pour créer des interfaces ergonomiques. Ce sont des composants légers. Il est possible d'en créer d'autre à l'aide des méthodes de programmation orientée objets (par exemple en utilisant un descendant d'une classe existante). La plupart des

interfaces possibles dans les systèmes d'exploitation usuels sont présentes comme par exemple les panneaux à onglets, les panneaux défilant,...

2.3.1 Le panneau : `JPanel`

Le conteneur léger le plus simple de Swing et le panneau (`JPanel`), c'est le conteneur par défaut de `JFrame` et de `JWindow`. Il permet de grouper des composants selon une politique de placement (le chapitre 3 est consacré aux gestionnaires de placement). Pour ajouter un composant à un panneau, on utilise la méthode `add` (ou l'une de ses surcharges). La méthode réciproque `remove` permet d'enlever un composant.

L'exemple suivant présente la construction d'un panneau comportant plusieurs composants : trois `JLabel`, deux `JTextField` et un `JTextArea`. Il s'agit d'un formulaire de renseignements simple. On crée une classe `FicheIdentite` qui est une fille de `JPanel`, ainsi elle hérite de toutes les méthode de `JPanel`. Les composants graphiques (`JXxx`) sont des attributs de la classe. Dans le constructeur de la classe, le constructeur de la classe mère est appelé (`super()`) puis les variables sont instanciées par un appel à leurs constructeurs respectifs. Enfin, les composants sont ajoutés en utilisant la méthode `add`.

```
public class FicheIdentite extends JPanel {
      private JTextField nom;
      private JLabel labelNom;
      private JTextField prenom;
      private JLabel labelPrenom;
      private JTextArea adresse;
      private JLabel labelAdresse;

      public FicheIdentite(){
         super();
         labelNom = new JLabel("Nom : ");
         labelPrenom = new JLabel("éPrnom : ");
         labelAdresse = new JLabel("Adresse : ");

         nom = new JTextField(5);
         prenom = new JTextField(5);
         adresse = new JTextArea("",3,10);

         this.add(labelNom);
         this.add(nom);
         this.add(labelPrenom);
         this.add(prenom);
         this.add(labelAdresse);
         this.add(adresse);
      };
}
```

Pour pouvoir utiliser ce panneau, il faut l'inclure dans une fenêtre (par exemple une `JFrame`). Le code suivant peut être utilisé :

```
// import
```

```
public class TestPanel {
   public static void main(String[] args) {
      JFrame fen = new JFrame();
      FicheIdentite laFiche = new FicheIdentite();
      fen.setContentPane(laFiche);
      fen.pack();
      fen.setVisible(true);
      fen.setTitle("Renseignements");
   }
}
```

Cette classe est exécutable (présence de la méthode statique `main`), on obtient la fenêtre de la figure 2.12 :

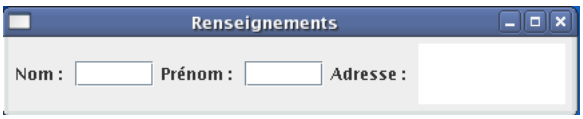


FIG. 2.12 – Un `JPanel` utilisé pour construire une fiche de renseignements

Le composant `JPanel` est le plus utilisé pour construire des interfaces. Comme tous les conteneurs, il peut être imbriqué dans d'autre conteneurs (y compris `JPanel`).

2.3.2 Le panneau à défilement : `JScrollPane`

Les applications traitant du texte ou des images n'affichent souvent qu'une partie du document pour éviter de monopoliser toute la surface d'affichage. Des glissières (appelés « ascenseurs ») sont affichées sur les cotés afin de pouvoir se déplacer dans le document. Le composant `JScrollPane` permet d'implémenter cette fonction. Il peut accueillir un autre composant (et un seul) mais permet d'ajouter des ascenseurs pour déplacer la partie affichée si elle est trop grande. Ce conteneur ne peut héberger qu'un seul composant, et en utilisant son propre gestionnaire de placement. Si l'on le conteneur doit accueillir plusieurs composants on les place dans un `JPanel` que l'on place dans le `JScrollPane`. L'exemple suivant présente l'utilisation d'un `JScrollPane` pour afficher une image de grande taille. L'image est chargée en utilisant un label. Ce label est passé au constructeur surchargé du `JScrollPane`. Une fenêtre est crée (`fra`), on remplace son conteneur originel par le `JScrollPane` . Ensuite, une taille est imposée à la fenêtre (volontairement trop petite pour contenir l'image).

```
JFrame fra = new JFrame();
JLabel monImage = new JLabel(new ImageIcon("Image.jpg"));
JScrollPane lePanneau = new JScrollPane(monImage);

fra.setContentPane(lePanneau);
fra.setTitle("La Terre vue du ciel");
fra.setSize(400,400);
fra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fra.setResizable(false);
fra.setVisible(true);
```

L'exécution de ce programme affiche la fenêtre de la figure 2.13.

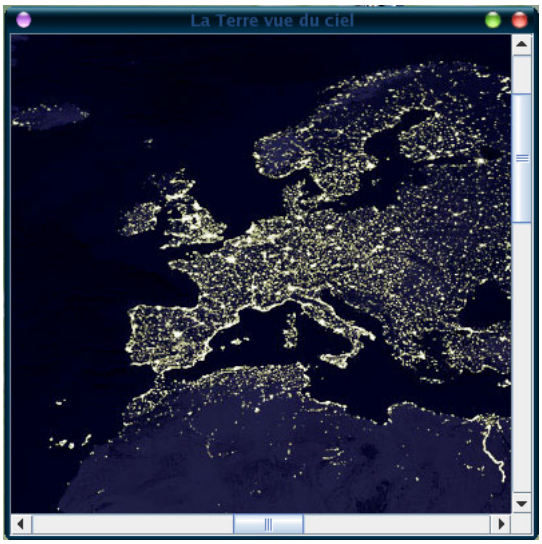


FIG. 2.13 – Utilisation d'un panneau à défilement

Par défaut, les ascenseurs ne sont affichés que s'ils sont nécessaires. La politique d'affichage peut être modifiée à l'aide des méthodes `setVerticalScrollBarPolicy` et `setHorizontalScrollBarPolicy`.

2.3.3 Le panneau divisé : `JSplitPane`

Séparer une interface en deux volets est utilisé pour mettre en vis-à-vis deux documents, ou un document et une barre d'outils. La séparation peut être horizontale ou verticale selon les interfaces. Ce type d'interface fait appel au `JSplitPane`. Ce conteneur ne peut accueillir que deux composants qui seront séparés par une barre de division (verticale ou horizontale). L'exemple suivant propose l'affichage de deux textes (à l'aide de composants `JTextArea`) séparés par une barre verticale. Une version surchargée du constructeur permet d'instancier le conteneur tout en spécifiant les deux composants à afficher et le sens de la division :

```
JFrame fra = new JFrame();
JTextArea source = new JTextArea();
JTextArea traduction = new JTextArea();

JSplitPane lePanneau = new JSplitPane(JSplitPane.
   HORIZONTAL_SPLIT, source, traduction);

fra.setContentPane(lePanneau);
```

Après construction, le conteneur est associé à la fenêtre principale à l'aide de la méthode `setContentPane`. La disposition suivante de la figure 2.14 est alors obtenue.

Si le positionnement et/ou les composants ne sont pas définis lors de la construction du conteneur, il est possible de les définir à l'exécution. La méthode `setOrientation` permet de

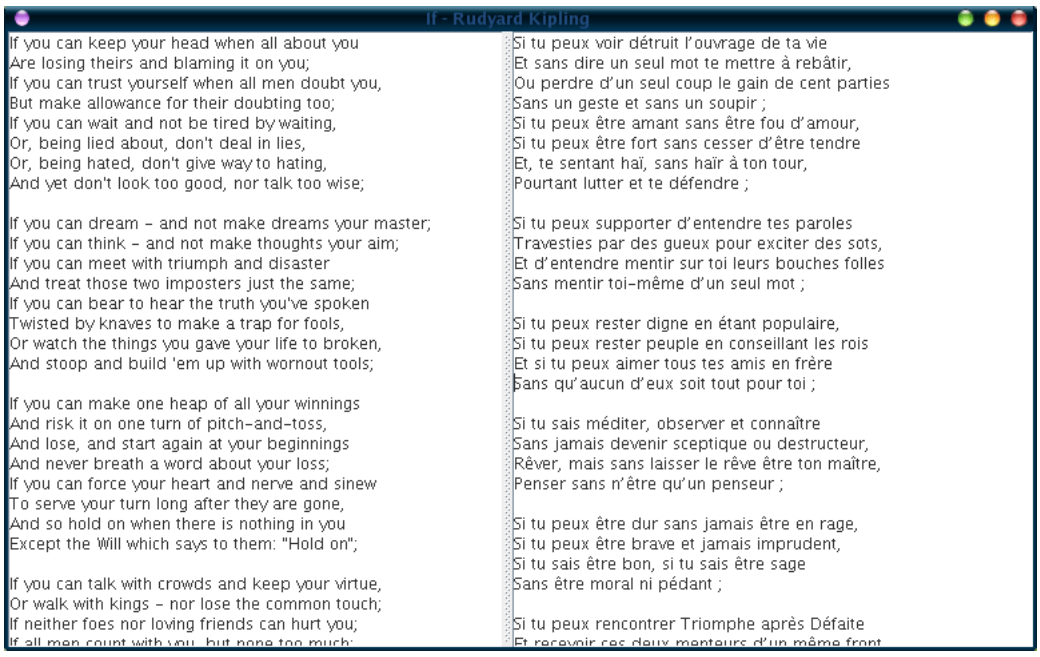


FIG. 2.14 – Un `SplitPane` utilisé pour présenter un texte et sa traduction

choisir une barre de séparation verticale ou horizontale. Pour placer les composants, on utilise les méthodes `setBottomComponent` et `setTopComponent` pour une disposition horizontale et `setLeftComponent` et `setRightComponent` pour une disposition verticale. Deux icônes peuvent rajoutés à la barre de séparation. Ils ajoutent la possibilité de ramener la barre de séparation aux extrêmes afin d'utiliser toute la surface d'affichage pour un seul des deux composants. Pour autoriser cette fonctionnalité il faut utiliser la méthode `setOneTouchExpandable`.

2.3.4 Le panneau à onglets : `JTabbedPane`

Le composant `JTabbedPane` permet de construire des interfaces en utilisant des onglets. Les composants peuvent ainsi être regroupés de manière thématique pour obtenir des interfaces allégées. La méthode `addTab` permet d'ajouter un composant dans une nouvelle feuille. Généralement on utilise un conteneur, le plus souvent `JPanel`. Une version surchargée de la méthode `addTab` permet de donner un titre à la feuille crée. Dans le code suivant, on utilise deux classes dérivées de `JPanel` (voir 2.3.1) pour construire des formulaires de saisie. La classe principale dérive de `JTabbedPane`; les éléments sont ajoutés avec la méthode `addTab()`.

```
public class MenuOnglets extends JTabbedPane {

   private FicheIdentite identite;
   private FicheEtude etudes;

   public MenuOnglets() {
      super();
```

```
        identite = new FicheIdentite();
        etudes = new FicheEtude();
        this.addTab("éIdentit",identite);
        this.addTab("Etudes",etudes);
    }
}
```

Deux onglets sont ainsi crées, l'un est nommé "Identité" l'autre "Etudes". Par défaut le premier onglet est présenté à l'exécution comme le montre la figure 2.15

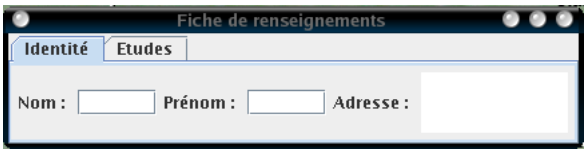


FIG. 2.15 – Utilisation d'un panneau à onglets (première vue)

Lorsque l'utilisateur clique sur un onglet, il est affiché comme présenté en figure 2.16

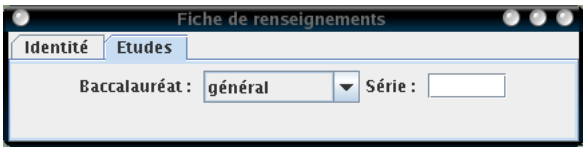


FIG. 2.16 – Utilisation d'un panneau à onglets (deuxième vue)

Les onglets sont numérotés à partir de 0. L'onglet affiché peut être modifié à l'aide de la méthode `setSelectedIndex`. De même, la méthode réciproque `getSelectedIndex` retourne l'onglet visible[3]. La position des onglets peut être modifiée en utilisant la méthode `setTabPlacement`. Les onglets peuvent ainsi être positionnés en haut, en bas, à droite ou à gauche de la fenêtre. Cette approche permet de classer facilement des informations variées en ne faisant apparaître que celles qui sont pertinentes. Il convient de remarquer que tous les composants présents dans les onglets sont crées même s'ils ne sont pas visibles. Cela signifie qu'il faut éviter de trop surcharger ce conteneur pour ne pas allouer trop de ressources (mémoire et processeur). Pour éviter ces problèmes, on peut modifier les onglets présents à l'aide de la méthode `addTab` pour ajouter seulement les onglets nécessaires (en fonction d'un choix, par exemple). La méthode `insertTab` permet d'insérer un onglet dans le conteneur. Enfin, la méthode `removeTabAt` permet de supprimer un onglet.

2.3.5 Les barres d'outils : `JToolBar`

Les applications complexes font souvent appel à des barres d'outils pour pouvoir accéder facilement aux fonctions les plus utilisées. Swing propose un conteneur (`JToolBar`) pour construire des barres d'outils regroupant n'importe quel composant. Comme pour tous les conteneurs, les composants sont ajoutés avec la méthode `add`. Il est possible de grouper les composants entre eux en matérialisant un espace entre deux groupes avec la méthode `addSeparator`. L'exemple suivant

[3]Cette méthode peut être utilisée dans les gestionnaire d'événements

présente la construction d'une barre d'outil pour un éditeur de texte simple, le résultat est présenté en figure 2.17.

```
JButton bOuvrir = new JButton(new ImageIcon("icons/fileopen.
   png "));
JButton bEnregistrer = new JButton(new ImageIcon("icons/
   filesave.png " ));
JButton bCouper = new JButton(new  ImageIcon("icons/editcut.
   png "));
JButton bCopier = new JButton(new ImageIcon("icons/editcopy.
   png "));
JButton bColler = new JButton(new ImageIcon("icons/editpaste.
   png "));

bOuvrir.setToolTipText("Ouvrir un fichier");
bEnregistrer.setToolTipText("Enregistrer le fichier");
bCouper.setToolTipText("Couper vers le presse-papier");
bCopier.setToolTipText("Copier vers les presse-papier");
bColler.setToolTipText("Coller depuis le presse-papier");

JToolBar tb = new JToolBar();
tb.add(bOuvrir);
tb.add(bEnregistrer);
tb.addSeparator();
tb.add(bCouper);
tb.add(bCopier);
tb.add(bColler);
```

Dans cette application, la barre d'outil est placée dans un `JPanel` qui a pour gestionnaire de placement un `BorderLayout`. La barre d'outil est placé dans l'emplacement `NORTH`, ce qui permet de s'assurer un placement cohérent. Généralement les barres d'outils peuvent être déplacées par l'utilisateur afin d'organiser son environnement de travail. Si il est nécessaire, cette fonctionnalité peut être modifiée à l'aide de la méthode `setFloatable`. A la création, cette propriété est activé, les barres d'outils peuvent dont être déplacées.

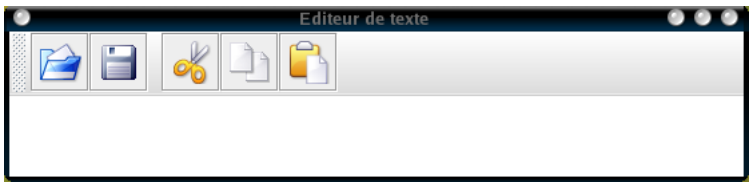


FIG. 2.17 – Une barre d'outils pour un éditeur de texte

2.3.6 Les bureaux `JDesktopPane`

De nombreuses applications autorisent l'ouverture de plusieurs documents simultanément. Ces interfaces sont nommées MDI (Multiple Document Interface). Le composant `JDesktopPane` propose une implémentation de ce comportement. Il permet d'afficher des fenêtres (`JInternalFrame`) à l'intérieur de l'application.

Les fenêtres `JInternalFrame` Elles proposent des méthodes proches de celles de `JFrame` (bien qu'il n'existe pas de relation d'héritage[4]). Pour des application MDI, une classe fille est souvent crée à partir de `JInternalFrame`. Par exemple, pour une application d'affichage d'image, la classe fille sera une fenêtre capable d'afficher une image en fonction du nom du fichier. Le code suivant propose une implémentation possible :

```
public class FrameMDI extends JInternalFrame {

   public FrameMDI() {
      super("",true,true,true,true);
   }

   public void setImage(String nomImage){
      JPanel unPanneau = new JPanel();
      JLabel uneImage = new JLabel(new ImageIcon(nomImage));
      unPanneau.add(uneImage);
      this.setContentPane(unPanneau);
      this.setTitle(nomImage);
   }
}
```

Les méthodes `setContentPane` et `setTitle` ont le même comportement que leurs homonymes de `JFrame`. Le constructeur de `JInternalFrame` permet de préciser le comportement de la fenêtre face aux opérations de redimensionnement, fermeture, agrandissement et réduction. Pour chaque capacité, un booléen permet d'autoriser/interdire cette possibilité. Dans l'exemple précédent, toutes les possibilités sont autorisées.

Utilisation d'un `JDesktopPane` Les fenêtres `JInternalFrame` doivent être placées dans un `JDesktopPane` à l'aide de la méthode `add`. Dans l'exemple suivant, on construit une classe (`TestDesktop`) à partir de `JDesktopPane`. Elle est utilisé comme conteneur de la fenêtre principale de l'application. Lors de la création, les fenêtres (`FrameMDI` voir ci-dessus) sont construites et ajoutées dans la classe `TestDesktop`. Il est tout à fait possible d'ajouter ou de retirer (méthode `remove`) des fenêtres à tout moment. Comme pour des fenêtre `JFrame`, on utilise la méthode `setSize` pour imposer la taille. Il aurait aussi été possible d'utiliser la méthode pack pour adapter la fenêtre à son contenu. La méthode `setVisible` permet de rendre la fenêtre visible.

```
public class TestDesktop extends JDesktopPane{
```

[4]Les `JFrame` sont des composants lourds, les `JInternalFrame` sont des composants légers

```
    FrameMDI[] planetes;
    String[] nomsFichier={"saturn.jpg","venus.jpg","jupiter.jpg"
       };

    public TestDesktop(){
       super();
       planetes = new FrameMDI[3];
       for (int i = 0; i < 3; i++) {
          planetes[i] = new FrameMDI();
          planetes[i].setSize(300,300);
          planetes[i].setLocation(100*i,50*i);
          planetes[i].setImage(nomsFichier[i]);
          planetes[i].setVisible(true);
          this.add(planetes[i]);
       }
    }

    public static void main(String[] args) {
       JFrame fra = new JFrame();
       fra.setContentPane(new TestDesktop());
       fra.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       fra.setVisible(true);
       fra.setSize(600,600);
       fra.setTitle("Afficheur d'images");
    }

}
```

La figure 2.18 présente l'affichage obtenu.

2.4 Les composants atomiques

Les composants atomiques sont tous les composants élémentaire de Swing. Ce sont les boutons, les labels, les menus,...

2.4.1 Les labels : `JLabel`

Un label est une simple chaîne de caractères informative (il peut aussi contenir une image). Pour créer un nouveau label il suffit d'appeler le constructeur `JLabel`. Ce constructeur est surchargé, généralement, on utilise ceux qui permettent l'initialisation en même temps, par exemple avec une chaîne :

```
JLabel monLabel = new JLabel("Une îchane de ècaractres");
```

Ce label est ajouté à un conteneur avec la méthode `add`.

```
JFrame fen = new JFrame();
JPanel pan = new JPanel();
JLabel unLabel = new JLabel("Une chaine de ècaractres");
```

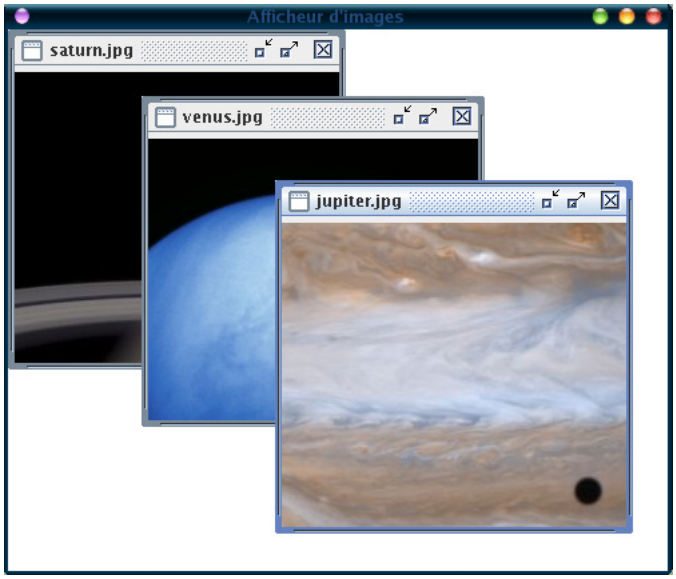


FIG. 2.18 – Une application MDI à base de `JDesktopPane`

```
pan.add(unLabel);
fen.setContentPane(pan);
fen.pack();
fen.setVisible(true);
```

Ce code placé dans une classe exécutable produit l'affichage de la figure 2.19.

FIG. 2.19 – Un label affichant une chaîne de caractères

Contrairement à d'autres interfaces graphiques, Swing ne propose pas de composant image dédié, toutefois, il est possible d'utiliser `JLabel` pour afficher des image[5]. Comme de nombreux composants de Swing, `JLabel` peut afficher un objet `ImageIcon`. Il suffit alors de créer un objet `ImageIcon` à partir de l'image à afficher et d'associer cet icône à un `JLabel`. Le constructeur de `ImageIcon` permet de charger directement un fichier de type `JPEG`, `GIF` ou `PNG` en passant le nom du fichier en paramètre lors de l'appel du constructeur. L'exemple précédent modifié pour afficher une image devient :

```
ImageIcon img = new ImageIcon("venus.jpg");
JLabel unLabel = new JLabel(img);
pan.add(unLabel);
```

Le résultat suivant obtenu est présenté en figure 2.20.

[5]Il est aussi possible de surcharger la méthode `paint` du composant. Cette approche est beaucoup plus complexe

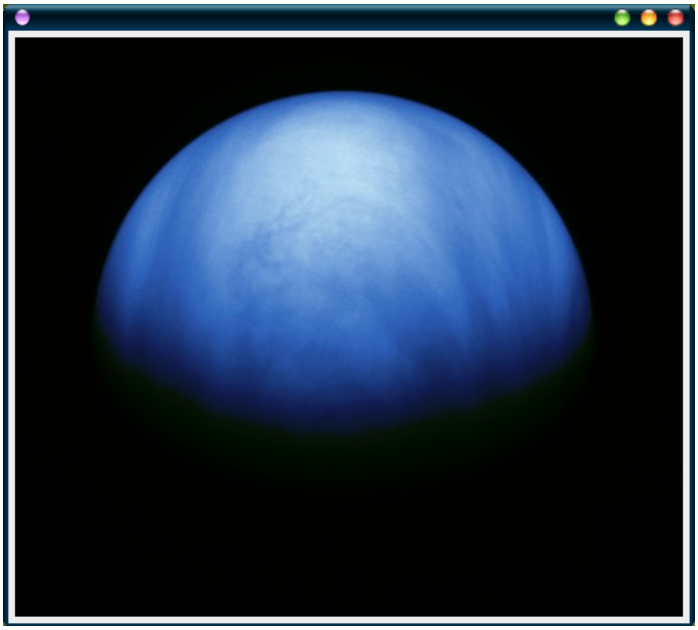


FIG. 2.20 – Un `JLabel` utilisé pour afficher une image

2.4.2 Les boutons : `JButton` et `JToggleButton`

Les boutons sont les composants les plus utilisés pour interagir avec l'utilisateur. Swing propose plusieurs types de boutons. Tous les boutons[6] héritent de la classe abstraite `AbstractButton`.

La classe `AbstractButton`

Elle fournit de nombreuses méthodes pour paramétrer les boutons. Généralement, les méthodes sont surchargées dans les classes filles pour adapter leur comportement. Tous les boutons peuvent être accompagnés d'un texte, la méthode `setText` est utilisée pour le modifier. Les descendants de `AbstractButton` peuvent être décorés à l'aide d'icônes (comme pour `JLabel`). La méthode `setIcon` permet de spécifier l'icône par défaut du composant. Si l'élément est désactivé, l'icône affiché sera grisé (en général), dans certains cas, on préfère spécifier un icône particulier pour signifier l'état désactivé, la méthode `setDisabledIcon` sera utilisée. La méthode `setMnemonic` permet de définir une touche de raccourcis pour le clavier. Les touches de raccourci pour le clavier sont des combinaisons de `ALT` et d'une autre touche. Cette méthode utilise les constantes définies dans la classe `KeyEvent`. Il est d'usage de définir une lettre contenue dans le label du bouton.

```
//...
monBouton.setText("Un bouton");
//...
monBouton.setMnemonic(KeyEvent.VK_B);
//...
```

[6] De nombreuses classes héritent de `AbstractButton` : les boutons `JButton`, les boutons à bascule `JToggleButton`, les boutons radios `JRadioButton`, les cases à cocher `JCheckBox`, les éléments de menus : `JMenuItem`, `JCheckBoxMenuItem` et `JRadioButtonMenuItem`.

Dans cet exemple, la lettre `'b'` sera souligné et l'appui sur `ALT+B` (ou `ALT+b`) aura le même effet qu'un clic sur le bouton.

Le composant `JButton`

Le bouton le plus utilisé est le `JButton`. Il crée un bouton qui peut être cliqué par l'utilisateur à l'aide de la souris. Généralement le texte affiché dans le bouton est passé comme paramètre au constructeur. Toutefois, il est possible de le modifier à l'aide de la méthode `setText`. Pour ajouter un bouton à l'exemple précédent, les modifications sont les suivantes :

```
//...
JPanel pan = new JPanel();
JLabel unLabel = new JLabel("Un label");
JButton unBouton = new JButton("Un Bouton");
pan.add(unLabel);
pan.add(unBouton);
//...
```

Le bouton est ajouté au panneau en fonction de la politique de placement du gestionnaire (voir chapitre 3). L'affichage par défaut est présenté en figure 2.21

FIG. 2.21 – Un `JButton` avec du texte

Pour ajouter une icône, on peut utiliser la méthode `setIcon` de la classe mère, mais le plus simple et d'utiliser le constructeur surchargé. Par exemple, pour afficher une icône à coté du texte d'un bouton (le résultat est présenté en figure 2.22) :

```
//...
ImageIcon img = new ImageIcon("sun-java.png");
JButton unBouton = new JButton("Un Bouton", img); //...
```

FIG. 2.22 – Un `JButton` avec du texte et une image

L'icône est placé à coté du texte du bouton. Le placement peut être modifié avec les méthodes `setVerticalTextPosition` et `setHorizontalTextPosition`. Par exemple, pour placer le texte centré sous le bouton, les deux lignes suivantes sont ajoutées :

```
//...
unBouton.setHorizontalTextPosition(SwingConstants.CENTER);
unBouton.setVerticalTextPosition(SwingConstants.BOTTOM);
//...
```

Le texte est alors positionné sous le bouton, comme pour une barre d'outils (figure 2.23).

FIG. 2.23 – Modification de la position du texte dans un `JButton`

Le composant `JToggleButton`

Swing propose un type de bouton particulier, les boutons à bascule : `JToggleButton`. Ces boutons conservent leur état après le relâchement de la souris. Ils sont utilisés pour représenter un état booléen (comme par exemple l'effet souligné dans un traitement de texte). On peut comparer les `JButton` aux boutons poussoirs et les `JToggleButton` aux interrupteurs. La mise en oeuvre est identique à celle des `JButton`. L'état du composant peut être lu avec `isSelected` et modifié avec `setSelected`[7].

```
//...
JToggleButton unBouton = new JToggleButton("Un Bouton", img);
//...
```

Après un clic de souris le bouton a l'aspect de la figure 2.24.

FIG. 2.24 – Un `JToggleButton` enfoncé

2.4.3 Les cases à cocher : `JCheckBox`

Les cases à cocher permettent de matérialiser des choix binaires d'une manière plus usuelle que les boutons à bascules (`JToggleButton`.) Comme les boutons, elles héritent de la classe abstraite `AbstractButton`. Plusieurs constructeurs sont implémentés pour attribuer le label et l'état lors de l'initialisation. L'exemple suivant présente deux cases à cocher, la seconde est cochée lors de la création (cette méthode est souvent utilisée pour les options par défaut). A l'exécution, on obtient l'affichage suivant (figure 2.25) :

```
JCheckBox casePasCochee = new JCheckBox("Une case à cocher");
JCheckBox caseCochee = new JCheckBox("Une case écoche", true);
```

[7]Considérons une application de traitement de texte, on peut mettre le texte en gras en cliquant sur le bouton de la barre d'outil, en allant dans le menu format ou en utilisant un raccourci clavier. Si l'utilisateur utilise le raccourci clavier, il faut mettre à jour l'état du bouton de la barre d'outil et l'élément de menu correspondant, on utilise alors `setSelected`.

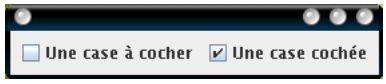


FIG. 2.25 – Les deux états des `JCheckBox`

Comme pour les boutons à bascule, on lit l'état d'une case à cocher à l'aide de la méthode `isSelected()` qui renvoie une valeur booléenne et on la modifie avec `setSelected`.

```
if (maCaseACocher.isSelected()==true){
   // la case est écoche
}else{
   // la case n'est pas écoche
}
```

2.4.4 Les boutons radio : `JRadioButton`

Les boutons radio `JRadioButton` sont des boutons à choix exclusif[8], il permettent de choisir un (et un seul) élément parmi un ensemble. Comme les `AbstractButton` dont ils héritent, ils ont deux états. Pour créer un bouton radio, on peut utiliser différentes surcharges du constructeur. L'une des plus utilisée permet d'initialiser la variable de texte associée au bouton : `JRadioButton(String )`. Il est possible de choisir l'état du bouton de la création avec le constructeur `JRadioButton(String , boolean)`. On comprend facilement que si cette valeur n'est pas renseignée, le bouton n'est pas coché[9]. Pour créer deux boutons radios présentant les différents états, le code suivant peut être utilisé :

```
JRadioButton bouton1 = new JRadioButton("Un bouton radio");
JRadioButton bouton2 = new JRadioButton("Un bouton radio écoch",
    true);
```

Les boutons crées sont présentés dans la figure 2.26.

FIG. 2.26 – Les deux états des `JRadioButton`

Les boutons radio doivent être regroupées dans un `BoutonGroup` pour avoir un comportement exclusif[10]. Il s'agit d'un groupe logique où un et seul bouton peut être actif (`true`). Le `BoutonGroup` n'a aucune incidence sur l'affichage. Il est donc utile lors de la conception de matérialiser les groupes de boutons radio. Généralement, cette tache est confiée à différents panneaux matérialisés par des bordures. Pour obtenir la présentation suivante (figure 2.27) :

le code suivant peut être utilisé :

```
public class TestRadio extends JPanel {
```

[8] Ce nom vient des ancien poste de radio sur lesquels on choisissait la bande (GO, PO,...) à l'aide de boutons mécaniques. Un seul bouton pouvait être enfoncé à la fois.

[9] Tout simplement pour éviter d'avoir plusieurs boutons cochés lors de la création...

[10] On peut aussi grouper des `JToggleButton`.

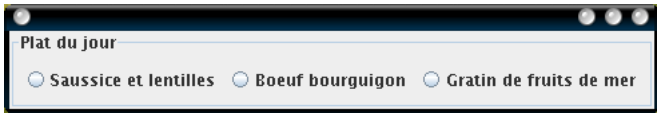


FIG. 2.27 – Mise en place d'une bordure pour matérialiser un groupe de boutons radio

```
    JRadioButton plat1, plat2, plat3;
    ButtonGroup plat;

    public TestRadio(){
        plat1 = new JRadioButton("Saussice et lentilles");
        plat2 = new JRadioButton("Boeuf bourguigon");
        plat3 = new JRadioButton("Gratin de fruits de mer");
        plat = new ButtonGroup();
        plat.add(plat1);
        plat.add(plat2);
        plat.add(plat3);
        this.add(plat1);
        this.add(plat2);
        this.add(plat3);
        this.setBorder(BorderFactory.createTitledBorder("Plat du
            jour"));
    };
}
```

Il reste à créer une fenêtre et à placer un objet `TestRadio` dans cette fenêtre :

```
import ...
public class PlatDuJour{
    public static void main(String[] args) {
        JFrame fen = new JFrame();
        TestRadio panneau = new TestRadio();
        fen.setContentPane(panneau);
        fen.pack();
        fen.setVisible(true);
    }
}
```

2.4.5 Les listes de choix : `JList`

Ce composant permettent de choisir un (ou plusieurs) élément(s) parmi un ensemble prédéfini. Cet ensemble peut être un tableau ou vecteur d'objets quelconques[11]. Les éléments sont présentés sous la forme d'une liste dans laquelle les éléments choisis sont surlignés. La liste d'éléments peut être

[11] La méthode `toString` est utilisée pour l'affichage

définie lors de la construction mais aussi modifiée ultérieurement. L'extrait de code suivant présente une utilisation d'une `JList` pour obtenir le résultat de la figure 2.28.

```
String[] lesElements={"Guitare" , "Basse" , "Clavier" , "
   Batterie" , "Percussions" , "Flute" , "Violon"};
JList instruments = new JList(lesElements);
JPanel pan = new JPanel();
JLabel text = new JLabel("Choississez un(des) instrument(s) :");
JFrame fen = new JFrame("Musique");

pan.add(text);
pan.add(instruments);
```

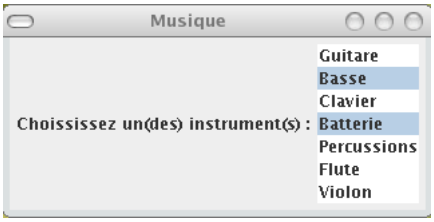


FIG. 2.28 – Une sélection d'éléments à partir d'une `JList`

Pour pouvoir exécuter ce programme, il faut placer `pan` dans une fenêtre. Un ou plusieurs éléments peuvent être sélectionnés en cliquant dessus. Les éléments sont numérotés à partir de `0` (comme pour un tableau). Les méthodes `getSelectedIndex` et `getSelectedIndices` permettent de connaître le premier indice ou tous les indices des éléments sélectionnés. De même `getSelectedValue` et `getSelectedValues` fournissent le premier élément ou tous les éléments sélectionnés. Il est possible de choisir le mode de sélection parmi trois modes possibles : sélection unique, sélection multiple, sélection d'un intervalle. Pour configurer le mode de sélection, on utilise la méthode `setSelectionMode`.

2.4.6 Les boites combo : `JComboBox`

Les boites combo permettent de choisir un seul élément parmi une liste proposée. Elles ont un comportement proche des boutons radio. On les utilise quand l'ensemble des éléments à afficher n'est pas connu lors de la conception. En effet, il est difficile de concevoir une interface avec un nombre de boutons radio variable. Comme les listes de choix, on peut les construire en passant un tableau d'objet en paramètre de construction.

```
JComboBox instruments = new JComboBox(lesElements);
```

L'utilisation d'une boite combo permet de sélectionner un seul objet dans la liste (figure 2.29).

Les méthodes `getSelectedIndex` et `getSelectedItem` permettent de connaître l'indice et l'objet sélectionné. Les boites combo proposent un avantage supplémentaire sur les bouton radio, elles peuvent être éditées par l'utilisateur. La méthode `setEditable` permet d'autoriser l'édition de la boite combo.

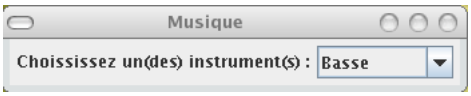


FIG. 2.29 – La liste précédente affichée avec une `JComboBox`

2.4.7 Les glissières : `JSlider`

Les glissières permettent de proposer à l'utilisateur une interface de saisie plus intuitive qu'un champ de texte pour régler certains paramètres. Swing propose le composant `JSlider` pour représenter un réglage variable. L'exemple suivant présente l'utilisation de glissières pour régler les trois composantes principales d'une couleur (figrue 2.30) :

```
JSlider sBlue = new JSlider();
JSlider sRed = new JSlider();
JSlider sGreen = new JSlider();
//...
JPanel pan = new JPanel();
//...
pan.add(sRed);
pan.add(sGreen);
pan.add(sBlue);
```

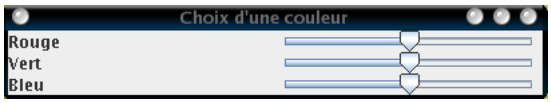


FIG. 2.30 – Des `JSlider` utilisés pour choisir une couleur

La position du curseur est fixée avec `setValue` et lue avec `getValue`. L'orientation peut être changée en utilisant la méthode `setOrientation`. Les `JSlider` peuvent être configurés pour couvrir une étendue à l'aide de `setMaximum` et `setMinimum`. Il est possible de tracer des repères, deux types sont proposés, les repères majeurs et les repères mineurs ; les intervalles de chaque type sont définis avec `setMinorTickSpacing` et `setMajorTickSpacing`. On demande le tracé des repères avec la méthode `setPaintTicks`.

```
sBlue.setMinimum(0);
sBlue.setMaximum(255);
sBlue.setValue(127);
sBlue.setMajorTickSpacing(127);
sBlue.setMinorTickSpacing(32);
sBlue.setPaintTicks(true);
```

Certains paramètres peuvent être initialisés lors de la construction du composant. Par exemple, on peut configurer les bornes et la position du curseur (figure 2.31) :

```
JSlider sBlue = new JSlider(JSlider.HORIZONTAL,0,255,127);
sBlue.setMajorTickSpacing(127);
```

```
sBlue.setMinorTickSpacing(32);
sBlue.setPaintTicks(true);
```

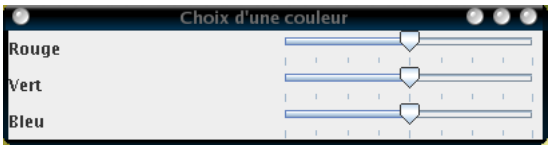


FIG. 2.31 – Ajout de repères dans un `JSlider`

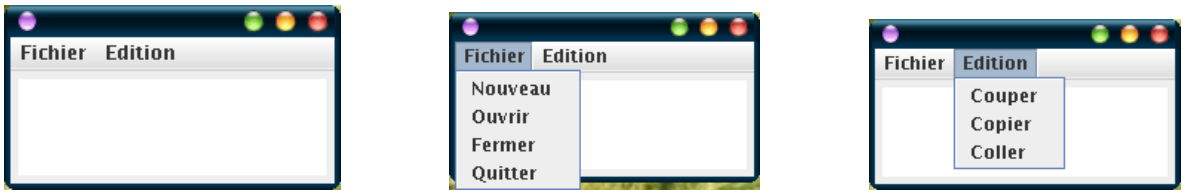
2.4.8 Les menus : `JMenu`, `JMenuBar`, `JMenuItem`, `JRadioButtonMenuItem`, `JCheckBoxMenuItem`

Les barres de menus permettent de regrouper de nombreuses fonctions d'une manière ergonomique. La construction de menus est très hiérarchisé. On utilise un composant `JMenuBar` pour construire une barre de menus. Les menus sont construits à partir de la classe `JMenu`. Ils sont placés dans la `JMenuBar` avec la méthode `add`. Ces menus sont constitués d'éléments appartenant à la classe `JMenuItem` ou à l'une de ses classes filles (`JRadioButtonMenuItem`, `JCheckBoxMenuItem`). La classe `JMenuItem` est une classe héritant de `JToggleButton`, elle profite donc de toutes ses méthodes (gestion des icônes,...). Les éléments de menus sont ajoutés aux menus à l'aide de la méthode `add`. Cette hiérarchisation complexe est en fait très facile de mise en oeuvre. L'exemple suivant présente la construction d'une barre de menus qui comporte deux menus : Fichier et Edition. Dans le menu Fichier, on trouve les éléments Nouveau, Ouvrir, Fermer et Quitter. Le menu Edition est composé des éléments Couper, Copier et Coller (figure 2.32).

```
JFrame fen = new JFrame();

JMenu menuFichier = new JMenu("Fichier");

JMenuItem menuFichierNouveau = new JMenuItem("Nouveau");
JMenuItem menuFichierOuvrir = new JMenuItem("Ouvrir");
JMenuItem menuFichierFermer = new JMenuItem("Fermer");
JMenuItem menuFichierQuitter = new JMenuItem("Quitter");

menuFichier.add(menuFichierNouveau);
menuFichier.add(menuFichierOuvrir);
menuFichier.add(menuFichierFermer);
menuFichier.add(menuFichierQuitter);

JMenu menuEdition = new JMenu("Edition");

JMenuItem menuEditionCouper = new JMenuItem("Couper");
JMenuItem menuEditionCopier = new JMenuItem("Copier");
JMenuItem menuEditionColler = new JMenuItem("Coller");

menuEdition.add(menuEditionCouper);
```

```
menuEdition.add(menuEditionCopier);
menuEdition.add(menuEditionColler);

JMenuBar barreMenu = new JMenuBar();

barreMenu.add(menuFichier);
barreMenu.add(menuEdition);

fen.setJMenuBar(barreMenu);
```

La barre de menus (`barreMenu`) pourrait être ajoutée à un conteneur. Ici, comme nous utilisons une fenêtre `JFrame`, nous pouvons ajouter directement la barre de menus à la fenêtre à l'aide de la méthode set`JMenuBar`.



(a) Création d'une barre de menus

(b) Le premier menu (`MenuFichier`) déployé

(c) Le premier menu (`MenuEdition`) déployé

FIG. 2.32 – Les éléments de base des menus

Tous les éléments composant un menu (`JMenuItem` ou ses descendants) peuvent avoir une mnémonique attribuée par la méthode `setMnemonic` afin de pouvoir contrôler les menus avec la touche `ALT` (ou une touche équivalente). Cette approche est complétée par la possibilité de créer des touches de raccourcis pour les opérations les plus courantes (par exemple, l'élément de menu copier est équivalent à la combinaison `CTRL+C`). Ces combinaisons sont définies à l'aide de la méthode `setAccelerator`. Cette méthode a pour paramètre d'appel un objet `KeyStroke` qui code une touche ou une combinaison de touches. On utilise la méthode statique `KeyStroke.getKeyStroke` pour obtenir directement l'objet nécessaire. Cette méthode permet de définir une combinaison de touche de la forme `touche + modificateur`. La touche est une constante définie dans `KeyEvent` (par exemple `KeyEvent.VK_C`) et le modificateur est une constante définie dans `Event`. Les modificateurs de touches sont présentés dans le tableau 2.1.

Si on veut associer la combinaison CTRL+C à l'élément de menu menuCopier, l'appel de la fonction sera le suivant :

```
menuEditionCopier.setAccelerator(KeyStroke.getKeyStroke(KeyEvent
    .VK_C, Event.CTRL_MASK);
```

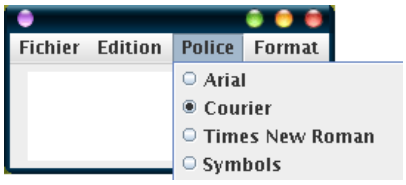
La classe `JMenuItem` a deux classes filles qui implémentent des boutons radio et des cases à cocher à l'intérieur des menus. La classe `JRadioButtonItemMenu` permet de créer des boutons radio adaptés aux menus, comme pour les `JRadioButton`, les éléments sont regroupés dans des ButtonGroup. De même, la classe `JCheckBoxMenuItem` fournit des cases à cocher adaptées aux menus. L'exemple suivant présente l'ajout de deux menus, l'un pour choisir la police de caractères basé sur des boutons radio , l'autre pour choisir le format des caractères basé sur des cases à cocher (figure 2.33) :

TAB. 2.1 – Les principaux modificateurs de touche

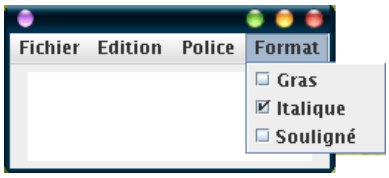
Modificateur	Action
Event.SHIFT_MASK	Touche SHIFT enfoncée
Event.CTRL_MASK	Touche CONTROL enfoncée
Event.META_MASK	Touche ALT GR enfoncée
Event.ALT_MASK	Touche ALT enfoncée
Event.BUTTON1_MASK	Bouton 1 de la souris
Event.BUTTON2_MASK	Bouton 2 de la souris
Event.BUTTON3_MASK	Bouton 3 de la souris

```
JMenu menuFonte = new JMenu("Police");
JRadioButtonMenuItem policeArial = new JRadioButtonMenuItem("
   Arial");
JRadioButtonMenuItem policeCourier = new JRadioButtonMenuItem("
   Courier");
JRadioButtonMenuItem policeTimes = new JRadioButtonMenuItem("
   Times New Roman");
JRadioButtonMenuItem policeSymbol = new JRadioButtonMenuItem("
   Symbols");

ButtonGroup fontes = new ButtonGroup();
fontes.add(policeArial);
fontes.add(policeCourier);
fontes.add(policeTimes);
fontes.add(policeSymbol);

menuFonte.add(policeArial);
menuFonte.add(policeCourier);
menuFonte.add(policeTimes);
menuFonte.add(policeSymbol);

JMenu menuFormat = new JMenu("Format");
JCheckBoxMenuItem formatGras = new JCheckBoxMenuItem("Gras");
JCheckBoxMenuItem formatItalique = new JCheckBoxMenuItem("
   Italique");
JCheckBoxMenuItem formatSouligne = new JCheckBoxMenuItem("
   éSoulign");

menuFormat.add(formatGras);
menuFormat.add(formatItalique);
menuFormat.add(formatSouligne);
//...
barreMenu.add(menuFonte);
```

```
barreMenu.add(menuFormat);
//...
```



(a) Utilisation de bouton radio dans un menu



(b) Utilisation de cases à cocher dans un menu

FIG. 2.33 – Les éléments modaux des menus

Pour organiser les menus de manière conviviale, on peut insérer des séparateurs entre les différents éléments d'un même menu avec la méthode `addSeparator`. Pour séparer l'élément `Quitter` des éléments relatif à la gestion des fichiers :

```
menuFichier.add(menuFichierNouveau);
menuFichier.add(menuFichierOuvrir);
menuFichier.add(menuFichierFermer);
menuFichier.addSeparator();
menuFichier.add(menuFichierQuitter);
```

2.4.9 Les dialogues de sélection de fichiers : `JFileChooser`

Comme toutes les interfaces graphiques, Swing propose une boite de sélection de fichier(s) avec la classe `JFileChooser`. Son comportement est proche d'une boite de dialogue modale (bien qu'elle n'hérite pas de `JOptionPane`). La classe propose trois méthodes pour afficher un dialogue d'ouverture de fichier. La première méthode (`showOpenDialog`) présente une boite de dialogue pour l'ouverture d'un fichier, la seconde (`showSaveDialog`) pour la sauvegarde d'un fichier, enfin, la troisième méthode (`showDialog`) permet de spécifier des chaînes de caractères pour le bouton de validation et le titre de la fenêtre afin de créer des boîtes personnalisées.

Contrairement aux boites de dialogues, un objet doit être instancié :

```
JFileChooser fc = new JFileChooser();
fc.showOpenDialog(fen);
```

On obtient la boite de sélection présentée dans la figure 2.34[12].

Les trois méthodes renvoient un entier dont la valeur correspond au bouton qui a été cliqué. La méthode `getSelectedFile` renseigne sur le nom du fichier sélectionné.

```
if (fc.showOpenDialog(fen)==JFileChooser.APPROVE_OPTION){
   System.out.println("Le fichier est : " + fc.getSelectedFile()
      );
}
```

[12]Si on utilise le code proposé, la boite va s'ouvrir sur le répertoire de l'utilisateur (`home` sous Unix/Linux et `Mes Documents` sous Windows)

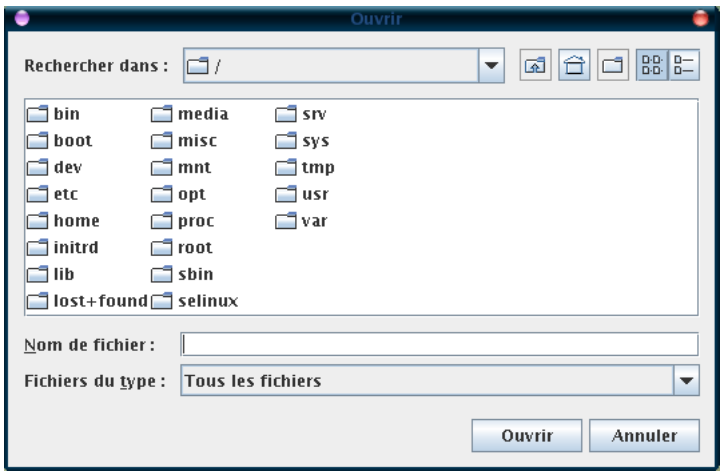


FIG. 2.34 – Une boite de dialogue d'ouverture de fichier

Pour certaines applications, il est intéressant de pouvoir sélectionner plusieurs fichiers en même temps ; la méthode `setMultiSelectionEnabled` autorise (ou bloque) les choix multiples. Si on autorise les choix multiples, la méthode `getSelectedFiles` renvoie un tableau contenant les noms des fichiers sélectionnés. Si on souhaite accéder à un répertoire particulier lors de l'ouverture de la boite, il faut faire appel à la procédure `setCurrentDirectory`. L'exemple suivant permet d'afficher l'arborescence à partir de la racine (sur un système Unix/Linux) :

```
JFileChooser fc = new JFileChooser();
fc.setCurrentDirectory(new File("/"));
fc.showOpenDialog(fen);
```

Filtrage des types de fichiers Les applications ne gèrent souvent qu'un seul (ou quelques) type(s) de fichiers, comme par exemple des images, des textes,... Il intéressant de ne pouvoir afficher que les fichiers d'un type donné dans la boite de dialogue. La classe `FileFilter` permet de construire des filtres de sélection. Cette classe abstraite définit deux méthodes : `accept` et `getDescription`. La méthode `accept` reçoit un fichier en paramètre et renvoie `true` s'il doit être affiché, `false` sinon. La seconde méthode renvoie une chaîne de caractères décrivant le filtre. Pour construire un filtre, on surcharge ces deux méthodes afin d'obtenir le comportement voulu. La classe `ExampleFileFilter` présente dans les exemples livrés avec l'installation de Java[13] illustre bien la construction d'un tel filtre. La méthode `setFileFilter` permet de spécifier un filtre pour une boite de dialogue `JFileChooser`.

2.4.10 Les composants orientés texte : `JTextField`,`JTextArea`, `JEditorPane`

Swing propose cinq composants pour travailler avec du texte : `JTextField`, `JFormatedTextField`, `JPasswordField`, `JTextArea`, `JEditorPane` et `JTextPane`. Tous ces composants descendent (directement ou non) de `JTextComponent`.

[13] Présente dans `/demo/jfc/FileChooserDemo/src` dans le répertoire d'installation de Java.

Présentation de la classe `JTextComponent`

Cette classe implémente une architecture MVC (Model-View-Controler). Sans rentrer dans les détails, elle sépare la partie affichage des données des données elles-mêmes. Pour la classe `JTextComponent` cela se manifeste sous la forme d'une classe membre interne de type `Document` qui contient les données texte. Cette classe membre interne est accessible via les méthodes `setDocument` et `getDocument`. Elle est automatiquement mise à jour quand on ajoute des éléments au composant. De plus, deux éléments partageant le même `Document` afficheront les mêmes données. Un accès aux données est aussi possible avec les fonctions `getText` et `setText` qui manipulent des `String`. La méthode `setEditable` permet d'autoriser (ou d'interdire) la modification des données. Il est possible de sélectionner du texte (souvent à l'aide de la souris). Le texte sélectionné est accessible via la méthode `getSelectedText`. Un accès complet au bloc notes du système d'exploitation est assuré par les méthodes `cut`, `copy` et `paste`. Elles peuvent être utilisées pour transférer des informations à l'intérieur de l'application Java mais aussi avec le système d'exploitation.

Le champ de saisie `JTextField`

Pour saisir une seule ligne de texte, on utilise le composant `JTextField`. Il construit un champ de saisie dont la largeur peut être fixée avec `setColumns`. Il est préférable de fixer la largeur du champ de saisie pour éviter des déformations des interfaces graphiques lors du remplissage. L'exemple qui suit présente un champ de saisie simple (figure 2.35) :

```
JPanel pan = new JPanel();
JLabel lNom = new JLabel("Entrez votre nom :");
JTextField tfNom = new JTextField();
tfNom.setColumns(15);
pan.add(lNom);
pan.add(tfNom);
```

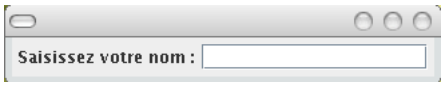


FIG. 2.35 – Utilisation d'un champ de saisie

Les principales méthodes proviennent de la classe mère (`JTextComponent`), `JTextField` ne propose que peu de méthodes supplémentaires. La position du texte dans la boite est fixée par `setHorizontalAlignment` et la police utilisée par `setFont`. Certaines valeurs peuvent aussi être initialisées par les versions surchargées du constructeur.

Plusieurs classes descendent de `JTextField`, parmi celle-ci deux offrent des possibilités intéressantes. `JFormatedTextField` permet de spécifier un format de données (comme par exemple une date, un numéro de téléphone,...) pour le champ d'entrée. `JPasswordField` se comporte comme un `JTextField` sauf que le texte saisi est remplacé par des astérisques pour l'affichage. Ce type de dialogue est utilisé pour saisir les mots de passe pour des applications sécurisées.

La zone de texte `JTextArea`

Il est souvent nécessaire d'afficher un texte sur plusieurs lignes. Le composant `JTextArea` est le composant le plus simple pour effectuer cette tache. Le texte est affiché en bloc, avec une police

unique (mais modifiable). Pour remplir le composant on utilise la méthode `setText` (de la classe mère `JTextComponent`) :

```
JTextArea taNotes = new JTextArea();
taNotes.setText("Trois anneaux pour ...
...
les ombres.\n");
taNotes.setEditable(false);
pan.add(taNotes);
```

Le composant `JTextArea` se présente comme un bloc note (figure 2.36) :

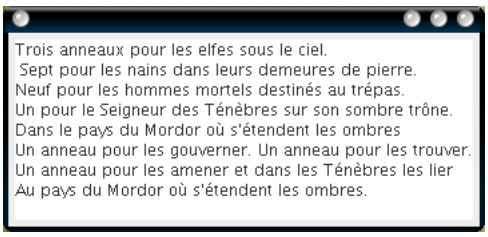


FIG. 2.36 – Un bloc note à partir d'un `JTextArea`

Le texte peut être modifié dynamiquement, via les méthodes de la classe mère (accès au `Document` , accès à une `String`), ou en ajoutant des lignes avec la méthode `append`. Pour obtenir un affichage plus confortable, le composant gère le retour à la ligne automatique. Cette fonctionnalité peut être activée (ou désactivée) avec la méthode `setLineWrap`.

Visualisateur de documents formatés `JEditorPane`

Pour afficher des mises en page plus complexes ou/et des documents plus riches les formats `RTF` et `HTML` sont souvent reconnus comme des standards. Le composant `JEditorPane` permet de gérer des affichages complexes notamment l'affichage de page web via la méthode `setPage` :

```
JEditorPane epNotes = new JEditorPane();
epNotes.setEditable(false);
epNotes.setPage("http://www.google.fr");
```

Ce composant permet de réaliser un navigateur simple en l'intégrant à une fenêtre (figure 2.37) :

Le gestionnaire de style peut être choisi en utilisant la méthode `setEditorKit`. Cette méthode est utile pour assigner un style à un document vide. Elle nécessite un composant `StyledEditorKit`. Deux styles sont prédéfinis, l'un pour le `HTML` : `HTMLEditorKit`, l'autre pour le `RTF` : `RTFEditorKit` . Il est aussi possible de définir un style pour d'autres types de document. La méthode `read` utilise un flux entrant pour charger un document. Le composant `JTextPane` est un descendant de `JEditorPane`. Il propose des méthodes d'insertion de style (gras, italique,...) dans le document à l'aide des `StyleConstants` et des `SimpleAttributeSet`.

FIG. 2.37 – Un navigateur web construit à partir d'un `JEditorPane`

2.5 Pour aller plus loin...

2.5.1 Les graphismes

Tous les composants se redessinent automatiquement sans intervention de l'utilisateur. Ce processus est basé sur un appel de la méthode `paintComponent` du composant qui se charge de dessiner le composant à l'écran. Il est possible de forcer la mise à jour de l'affichage du composant en appelant la méthode `repaint`. Pour modifier le dessin d'un composant il faut surcharger sa méthode `paintComponent`. Avant d'ajouter ses propres instructions graphiques, il est souvent nécessaire de faire appel à la méthode de dessin de la classe mère :

```
public class PanelDessin extends JPanel {
//...
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D)g;
      g2.drawRect(10,10,50,50);
   //...
   }
//...
```

La méthode `paintComponent` n'a qu'un argument g, qui représente le contexte graphique du composant. Cette argument est de type abstrait `Graphics`. Pour pouvoir l'utiliser directement il faut transtyper une classe descendante de `Graphics`. Généralement on utilise la classe `Graphics2D` qui propose de nombreuses méthodes de dessin (lignes, polygones, images,...). Il est alors possible de profiter de toute l'API 2D pour dessiner dans le composant.

2.5.2 D'autres composants de Swing

Les composants Swing sont bien plus nombreux que ceux présentés ici. Les arbres (`JTree`), permettent de représenter des données hiérarchisées (par exemple un explorateur de fichiers/-

dossiers). Les tables (`JTable`) qui peuvent être utilisées pour présenter des données en tableau, avec des champs de type texte, numérique, booléen...Il est possible d'y insérer des boites combo pour l'édition, de personnaliser les bulles d'aides par cellule, de réaliser toutes les opérations de tri (avec la classe `TableSorter`)...Parmi les différentes sources de composants supplémentaires, on peut citer `http://www.jfree.org` qui propose plusieurs composants libres dont `JFreeChart` qui permet de construire toutes sorte de graphiques (barres, camemberts, Gantt, ...). Sur `http://www.lowagie.com/iText/` on trouvera le composant `iText` pour créer des documents `PDF` à la volée. Un ensemble de calendriers est disponible sur `http://www.toedter.com/en/jcalendar/index.html`. Cette liste n'est pas exhaustive, avant de penser à construire un composant, une recherche approfondie sur Internet peut faire gagner beaucoup de temps.

2.5.3 Gestion du HTML

La plupart des composants (`JButton, JLabel,...`) qui affichent du texte sont capables d'interpréter du HTML[14]. Il est alors facile de personnaliser un affichage, il suffit de précéder le contenu de la balise `<html>` puis de faire suivre les balise `HTML` conventionnelles :

```
JLabel monLabelHTML = new JLabel« (<html><b>En gras</b »>);
```

La gestion du `HTML` est assez poussée, notamment la possibilité d'ajouter des images avec la balise `<img>`. C'est une approche complémentaire des icônes pour placer une image dans un label.

2.5.4 Apparence modifiable

Comme nous l'avons vu précédemment les composants de Swing sont dessinés par la `JVM`. De fait, il est possible de choisir un thème graphique pour une application Swing. C'est la classe `UIManager` qui permet de modifier l'apparence graphique d'une application. Un thème graphique est un objet de la classe `LookAndFeel`. Swing propose plusieurs thèmes graphiques par défaut lors de l'installation de Java. Le style Metal est le thème par défaut de Swing, une application graphique écrite en Java aura donc l'aspect de la figure 2.38 si le thème n'a pas été spécifié.

FIG. 2.38 – Le look-and-feel Metal

Sur toutes les implémentation de Java, on retrouve un deuxième thème, CDE/Motif qui reprend l'aspect graphique de certaines stations de travail (figure 2.39).

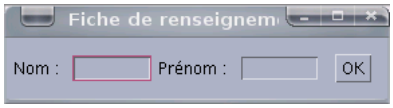


FIG. 2.39 – Le look-and-feel CDE/Motif

[14] Attention à ne pas transformer un label en une page web, l'utilisation des balises `HTML` est utile pour mettre des éléments en gras, les souligner,...

Un troisième thème est disponible par défaut, mais il dépend du système d'exploitation (Windows, Unix/Linux et MacOS). Ce thème permet une interface très proche de celle du système d'exploitation (figure 2.40).

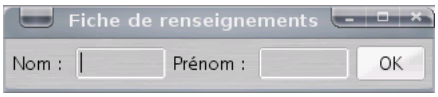


FIG. 2.40 – Le look-and-feel du système d'exploitation (ici Linux avec un interface Gnome)

Pour choisir ce thème il suffit d'ajouter cette ligne avant la construction des objets graphiques :

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

Pour lister tous les thème installés on peut s'inspirer du code suivant :

```
LookAndFeelInfo[] listLF;
listLF = UIManager.getInstalledLookAndFeels();
for (int i = 0; i < listLF.length; i++) {
   System.out.println(listLF[i].getName()+" - " + listLF[i].
      getClassName());
}
```

Pour utiliser un des thèmes listés :

```
UIManager.setLookAndFeel(listLF[2].getClassName());
```

Si on change le thème graphique durant l'exécution du programme, il faut forcer la mise à jour de l'affichage avec la méthode statique `updateComponentTreeUI` de la classe `SwingUtilities`. Le seul paramètre est la fenêtre principale de l'application.

Chapitre 3

Le positionnement des composants

Une application portable doit pouvoir être exécutée sur différents systèmes ayant des résolutions graphiques disparates. Un placement absolu des éléments graphiques conduit souvent à des problèmes d'affichage comme des textes qui débordent des boutons,...Pour éviter ce problème, Java propose de disposer les composants graphiques en fonction de règles simples qui permettront un aspect visuel quasiment identique d'un système à l'autre. Ces règles de placement sont définies à l'aide d'objets : les gestionnaires de placement. Nous n'allons présenter ici que quelques uns des gestionnaires, comme précédemment nous renvoyons le lecteur à la documentation officielle ainsi qu'aux nombreux sites internet consacrés à ce sujet.

3.1 Principe des gestionnaires de placement

Le placement des composants dans un conteneur est défini par un gestionnaire de placement. Lors de la création d'un conteneur, un gestionnaire est crée en même et lui est associé. Le tableau (tab. 3.1) ci-dessous présente les gestionnaires par défaut pour les 4 principaux conteneurs (certains composants (comme `JTabbedPane`, `JScrollPane`,...) ont leurs propres gestionnaires de placements) .

TAB. 3.1 – Gestionnaire par défaut pour les principaux conteneurs

Conteneur	Gestionnaire par défaut
`JPanel`, `JApplet`	`FlowLayout`
`JFrame`, `JWindow`	`BorderLayout`

Les gestionnaires de placement implémentent l'interface `LayoutManager`. Ils utilisent les méthodes `getPreferedSize`/`getMinimumSize`, pour connaître la taille préférée/minimale des composants. Ensuite, ils calculent les tailles et les positions des composants. Ces informations sont ensuite appliquées en utilisant les méthodes `setSize` et `setLocation`. Pour accéder au gestionnaire de placement d'un conteneur on utilise la méthode `getLayout`. La méthode `setLayout` permet de spécifier un nouveau gestionnaire.

3.2 Le positionnement absolu

Bien que cette approche soit peu recommandée, il est possible de ne pas utiliser de gestionnaire de placement. Dans ce cas, on utilise `setLayout(null)` pour désactiver le gestionnaire par défaut du conteneur. Les composants sont ensuite placés en utilisant les coordonnées absolues à l'aide de la méthode `setLocation`. La taille du composant peut être imposée en utilisant la méthode `setSize` et des valeurs :

```
JButton unBouton, unAutreBouton;
JLabel unLabel;
//...
unBouton.setSize(100,20);
unAutreBouton.setSize(150,20);
unLabel.setSize(200,50);
//...
unBouton.setLocation(20,20);
unAutreBouton.setLocation(50,50);
unLabel.setLocation(100,50);
```

D'une plate-forme à l'autre les polices d'affichage sont très variables. La taille des composants contenant du texte risque donc de varier. Il est préférable d'utiliser la méthode `getPreferedSize` pour être certain d'afficher les textes en entier.

```
unBouton.setSize(unBouton,getPreferedSize());
unAutreBouton.setSize(unBouton,getPreferedSize());
unLabel.setSize(unBouton,getPreferedSize());
```

D'une manière générale, le positionnement absolu doit être réservé à des applications particulières pour éviter des problèmes d'affichage.

3.3 Le gestionnaire `FlowLayout`

Le gestionnaire le plus élémentaire est le `FlowLayout`. Il dispose les différents composants de gauche à droite et de haut en bas (sauf configuration contraire du conteneur). Pour cela il remplit une ligne de composants puis passe à la suivante comme le ferrait un éditeur de texte. Le placement peut suivre plusieurs justifications, notamment à gauche, au centre et à droite. Une version surchargée du constructeur permet de choisir cette justification (bien qu'elle puisse aussi être modifiée en utilisant la méthode `setAlignement`). Les justifications sont définies à l'aide de variables statiques `FlowLayout.LEFT`, `FlowLayout.CENTER` et `FlowLayout.RIGHT`. L'extrait suivant présente l'utilisation d'un gestionnaire de placement `FlowLayout` dans un panneau ainsi que l'ajout de six boutons :

```
public class TestFlowLayout extends JPanel {
   public TestFlowLayout() {
      FlowLayout fl = new FlowLayout();
      this.setLayout(fl);
      this.add(new JButton("Un"));
      this.add(new JButton("Deux"));
      this.add(new JButton("Trois"));
      this.add(new JButton("Quatre"));
```

```
        this.add(new JButton("Cinq"));
        this.add(new JButton("Six"));
    }
//...
```

Par défaut, le gestionnaire tente de mettre tous les composants sur une seule ligne (figure 3.1).

FIG. 3.1 – Comportement par défaut du gestionnaire `FlowLayout`

Si la fenêtre est réduite, une nouvelle ligne de composants est crée comme le montre la figure 3.2.

FIG. 3.2 – Le gestionnaire `FlowLayout` redispose les composants après une réduction de la fenêtre

Les composants qui ne peuvent plus être placés sur la première ligne sont placés sur la seconde ligne. L'espacement entre deux composants de la même ligne est de 5 pixels par défaut. Il peut être modifié avec la méthode `setHgap`. De même l'espacement entre deux lignes est de 5 pixels par défaut, modifiable avec `setVgap`. Une version surchargée du constructeur permet de configurer la justification et les espacement lors de la création du gestionnaire : `FlowLayout(int align, int hgap, int vgap)`.

Remarque sur l'instanciation Généralement, l'objet gestionnaire de placement n'est pas utilisé dans le programme. On peut donc le créer sans l'instancier :

```
public class TestFlowLayout extends JPanel {
   public TestFlowLayout() {
      this.setLayout(new FlowLayout());
...
```

Lorsque l'on utilise cette méthode, l'objet crée ne peut pas être accédé facilement. Il faut utiliser un transtypage sur la valeur renvoyée par `getLayout` par exemple, pour utiliser la méthode `setHgap` :

```
((FlowLayout) this.getLayout()).setHgap(10);
```

3.4 Le gestionnaire `GridLayout`

Le gestionnaire `GridLayout` propose de placer les composants sur une grille régulièrement espacée. Généralement les composants sont disposés de gauche à droite puis de haut en bas. Cette disposition peut être modifiée en utilisant la méthode `ComponentOrientation` du conteneur. Le

TAB. 3.2 – Règles régissant le comportement d'un `GridLayout`

Nb. de lignes	Nb. de colonnes	Comportement
> 0	> 0	Seul le nombre de lignes et pris en compte, le nombre de colonnes est calculé en fonction du nombre de composants.
> 0	$= 0$	Seul le nombre de lignes et pris en compte, le nombre de colonnes est calculé en fonction du nombre de composants.
$= 0$	> 0	Seul le nombre de colonnes et pris en compte, le nombre de lignes est calculé en fonction du nombre de composants.
$= 0$	$= 0$	Une exception est levée.

nombre de lignes et de colonnes sont fixés à l'aide des méthodes `setRows` et `setColumns`. Une version surchargée du constructeur permet d'initialiser ces valeurs. Dans tous les cas, le comportement du gestionnaire est régi par le tableau 3.2.

Les composants sont placés dans l'ordre où ils apparaissent, de gauche à droite puis de haut en bas lorsqu'une ligne est complète. L'espacement entre les différents composants peut être modifié en utilisant les méthodes `setHgap` et `setVgap`. Ces deux méthodes imposent, respectivement, un espacement horizontal et un espacement vertical en pixels. Cet espacement est constant, quelque soit la taille du conteneur (certains composants peuvent donc être altérés dans ce but). On peut utiliser un constructeur surchargé pour définir les espacements et verticaux en plus du nombre de lignes et/ou de colonnes. Par défaut, l'espacement entre les composants est nul dans les deux directions. L'exemple qui suit présente un cas où le nombre de composants n'est pas un multiple du nombre de lignes.

```
public class TestGridLayout extend JPanel {
   public TestGridLayout() {
      this.setLayout(new GridLayout(3,0));
      this.add(new JButton("Un"));
      this.add(new JButton("Deux"));
      this.add(new JButton("Trois"));
      this.add(new JButton("Quatre"));
      this.add(new JButton("Cinq"));
      this.add(new JButton("Six"));
      this.add(new JButton("Sept"));
   }
```

Le placement est présenté dans la figure 3.3.

3.5 Le gestionnaire `BorderLayout`

Le gestionnaire `BorderLayout` définit cinq zones dans le conteneur : un zone centrale (`CENTER`) et quatre zones périphériques (`EAST`, `WEST`, `NORTH`, `SOUTH`). Les composants placés dans les zones `NORTH` et `SOUTH` sont dimensionnés à la hauteur qu'ils souhaitent puis étendus en largeur. A l'inverse

FIG. 3.3 – Comportement par défaut du gestionnaire `GridLayout`

les composant des zones `EAST` et `WEST` sont placés à leurs largeurs voulues et étendus en hauteur. Le composant placé dans la zone `CENTER` est utilisé pour combler le vide (il croit si la place est suffisante, il est réduit si elle est trop faible). Pour ajouter un composant, on une utilise une version surchargée de la méthode `add`, `add(component, constraints)` où `component` est un composant et `constraints` un objet représentant la position voulue. Des constantes (définies sous la forme `static`) permettent de définir les cinq positions (tab 3.3).

TAB. 3.3 – Constantes définies dans la classe `BorderLayout`

Constante	Position
`BorderLayout.NORTH`	Haut
`BorderLayout.SOUTH`	Bas
`BorderLayout.EAST`	Droite
`BorderLayout.WEST`	Gauche
`BorderLayout.CENTER`	Centre

Comme pour les gestionnaires vus ci-dessus, l'espacement entre les différents composants peut être modifié en utilisant les méthodes `setHgap` et `setVgap`. Ces deux méthodes imposent, respectivement, un espacement horizontal et un espacement vertical en pixels. On peut utiliser une version surchargée du constructeur pour définir les espacements horizontaux et verticaux. Cet espacement est nul par défaut. L'exemple qui suit présente la création d'un `BorderLayout` dans un `JPanel` et l'ajout de 5 boutons :

```
public class TestBorderLayout extends JPanel {
   public TestBorderLayout() {
      this.setLayout(new BorderLayout());
      this.add(new JButton("North"),BorderLayout.NORTH);
      this.add(new JButton("South"),BorderLayout.SOUTH);
      this.add(new JButton("East"),BorderLayout.EAST);
      this.add(new JButton("West"),BorderLayout.WEST);
      this.add(new JButton("Center"),BorderLayout.CENTER);
   }
...
```

La figure 3.4 présente le résultat obtenu.

Il n'est pas obligatoire de placer cinq composants dans le conteneur, les emplacements non pourvu seront alors ignoré. Par exemple, si on utilise que les emplacements `NORTH`, `WEST` et `CENTER` on obtient le résultat de la figure 3.5.

FIG. 3.4 – Comportement par défaut du gestionnaire `BorderLayout`

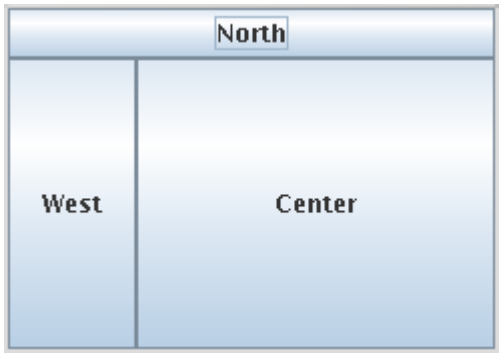


FIG. 3.5 – Comportement du gestionnaire `BorderLayout` avec 3 composants

Cette interface peut être utilisée pour placer, par exemple, une barre d'outils en haut, un affichage en arbres (fichiers, répertoires,...) à gauche et un document au centre.

3.6 Le gestionnaire `GridBagLayout`

Le `GridBagLayout` est le gestionnaire le plus complet et le plus souple. Comme le `GridLayout`, il place les composants sur une grille tout en autorisant les composants à prendre des libertés. Ainsi, les composants peuvent occuper plusieurs cellules, se répartir l'espace restant,...Le constructeur de `GridBagLayout` n'admet aucun paramètre, ils sont passés par la méthode `add` via un objet de type `GridBagConstraints`.

3.6.1 La classe `GridBagConstraints`

Le gestionnaire `GridBagLayout` utilise plus d'une dizaine de paramètres, pour éviter une surcharge trop lourde de la méthode `add`, on passe un objet de la classe `GridBagConstraints`qui représente tout ou partie de ces paramètres.

```
GridBagConstraints contraintes;
...
monConteneur.setLayout(new GridBagLayout());
...
contraintes = new GridBagConstraints();
...
monConteneur.add(monComposant, contraintes);
...
```

Dans la classe `GridBagConstraints` les variables membres sont publiques pour en faciliter l'accès.

3.6.2 Positionnement sur la grille

Les deux premières variables sont la position sur la grille, représentée par `gridx` et `gridy` dans la classe `GridBagConstraints`. La taille de la grille est calculée en fonction des maxima des variables `gridx` et `gridy` mais aussi de leurs valeurs relatives. L'exemple suivant propose une disposition « originale » de cinq boutons à partir d'un `GridBagLayout` :

```
public class TestGridBagLayout extends JPanel {
...
   this.setLayout(new GridBagLayout());
   GridBagConstraints contraintes = new GridBagConstraints();

   contraintes.gridx=0;
   contraintes.gridy=0;
   this.add(new JButton("Un"), contraintes);

   contraintes.gridx=1;
   contraintes.gridy=0;
   this.add(new JButton("Deux"), contraintes);

   contraintes.gridx=0;
   contraintes.gridy=1;
   this.add(new JButton("Trois"),contraintes);

   contraintes.gridx=1;
   contraintes.gridy=1;
   this.add(new JButton("Quatre"),contraintes);

   contraintes.gridx=2;
   contraintes.gridy=2;
   this.add(new JButton("Cinq"),contraintes);
...
```

Par commodité, on réutilise à chaque fois l'objet `contraintes`, cette approche ne pose pas de problème particulier si on veille bien à initialiser toutes les valeurs nécessaires. On obtient la disposition présentée dans la figure 3.6.

FIG. 3.6 – Une disposition à base de `GridBagLayout`

3.6.3 Remplissage des cellules

Dans l'exemple précédent, les boutons sont disposés selon le plan voulu mais ils n'occupent pas complètement leur cellule (espace entre les boutons "Un" et "Deux" par exemple). Il est possible de demander le remplissage des cellules à l'aide de la variable `fill`. Quatre constantes sont utilisées pour régler cette valeur :

- `GridBagConstraints.NONE` : aucun remplissage des cellules (résultat ci-dessus),
- `GridBagConstraints.HORIZONTAL` : remplissage dans le sens horizontal,
- `GridBagConstraints.VERTICAL` : remplissage dans le sens vertical,
- `GridBagConstraints.BOTH` : remplissage dans les deux sens.

Pour que cette valeur soit prise en compte il faut que les valeurs `weightx` et `weighty` de l'objet `GridBagConstraints` soient non nulles. Ce comportement est aussi utilisé si la fenêtre est redimensionnée. Dans l'exemple précédent on ajoute les lignes suivantes :

```
...
    GridBagConstraints contraintes = new GridBagConstraints();
    contraintes.fill = GridBagConstraints.BOTH;
    contraintes.weightx=1.0;
    contraintes.weighty=1.0;
...
```

Les valeurs `fill`, `weightx` et `weighty` étant les mêmes pour tous les composants on ne les définit qu'une seule fois. Les composants occupent alors tout l'espace possible comme le montre la figure 3.7.

FIG. 3.7 – Utilisation des contraintes de remplissage avec un `GridBagLayout`

3.6.4 Nombre de cases occupées

L'un des points forts de `GridBagLayout` est la possibilité de placer des composants sur plusieurs lignes et/ou plusieurs colonnes. Ce sont les variables `gridwidth` et `gridheight` qui définissent la largeur et la hauteur d'un composant (en nombre de cellules). L'exemple suivant présente les différent cas possibles :

```
...
    contraintes.fill = GridBagConstraints.BOTH;
    contraintes.weightx=1.0;
    contraintes.weighty=1.0;

    contraintes.gridx=0;
    contraintes.gridy=0;
    this.add(new JButton("Un"), contraintes);
```

```
    contraintes.gridx=1;
    contraintes.gridy=0;
    contraintes.gridwidth=2;
    this.add(new JButton("Deux"), contraintes);

    contraintes.gridx=0;
    contraintes.gridy=1;
    contraintes.gridwidth=1;
    contraintes.gridheight=2;
    this.add(new JButton("Trois"),contraintes);

    contraintes.gridx=1;
    contraintes.gridy=1;
    contraintes.gridwidth=1;
    contraintes.gridheight=1;
    this.add(new JButton("Quatre"),contraintes);

    contraintes.gridx=2;
    contraintes.gridy=1;
    contraintes.gridwidth=1;
    contraintes.gridheight=1;
    this.add(new JButton("Cinq"),contraintes);

    contraintes.gridx=2;
    contraintes.gridy=2;
    contraintes.gridwidth=1;
    contraintes.gridheight=1;
    this.add(new JButton("Six"),contraintes);
...
```

Les composants se placent alors en fonction des contraintes pour obtenir l'affichage de la figure 3.8.

FIG. 3.8 – Composants répartis sur plusieurs lignes/colonnes avec un `GridBagLayout`

Le comportement de `GridBagLayout` peut toutefois surprendre dans un premier temps. En effet, si on supprime le composant "Six", les éléments "Trois", "Quatre" et "Cinq" seront alignés car la notion de hauteur relative n'aura plus de sens et la contrainte imposée par `fill` conduira au remplissage de l'espace libre.

3.6.5 Poids des composants

Lorsque la variable `fill` est utilisée, les composants se redimensionnent pour occuper tout l'espace possible. Il est possible de paramétrer la manière dont les composants vont se partager cet espace supplémentaire à l'aide des poids : `weightx` et `weighty`. Les poids sont évalués et les composants occupent l'espace libre de manière proportionnelle à leurs poids (ceux ayant un poids plus important prenant plus d'espace). L'exemple suivant propose trois composants ayant des poids proportionnels :

```
//...
this.setLayout(new GridBagLayout());
GridBagConstraints contraintes = new GridBagConstraints();

contraintes.fill = GridBagConstraints.BOTH;
contraintes.weightx=1;
contraintes.weighty=1;

contraintes.gridx=0;
contraintes.gridy=0;
this.add(new JButton("Un"), contraintes);

contraintes.gridx=1;
contraintes.gridy=0;
contraintes.weightx=2;
this.add(new JButton("Deux"), contraintes);

contraintes.gridx=2;
contraintes.gridy=0;
contraintes.weightx=4;
this.add(new JButton("Trois"),contraintes);
//...
fen.setSize(300,50);
//...
```

La taille de la fenêtre accueillant les composants a volontairement été imposée pour illustrer le propos (figure 3.9).

FIG. 3.9 – Trois composants ayant des poids différents dans un `GridBagLayout`

Chapitre 4

Construction d'une interface graphique

4.1 Une interface simple

4.1.1 Présentation de l'interface voulue

On souhaite réaliser un convertisseur monétaire. L'application est basé sur une seule fenêtre ayant l'aspect suivant présentée dans la figure 4.1.

4.1.2 Identification des différents composants

Dans un premier temps, les différents éléments graphiques (boutons, menus,...) doivent être identifiés et nommés. La figure 4.2 présente le résultat de cette démarche.

4.1.3 Choix du/des gestionnaires de placement

Grille de placement

Pour déterminer le (ou les) gestionnaire(s) de placement à utiliser, le nombre de colonnes et de lignes de composants sont comptés. La figure 4.3 présente le découpage retenu.

Identification des gestionnaires de placement

Dans notre cas, il y a 2 colonnes et 3 lignes ; un des composant (`bConversion`) occupe les 2 colonnes. Le choix des gestionnaire est liée au choix des conteneurs (l'un va avec l'autre). Plusieurs solutions sont possibles :

– Un seul conteneur, et un gestionnaire `GridBagLayout`

FIG. 4.1 – Le résultat voulu

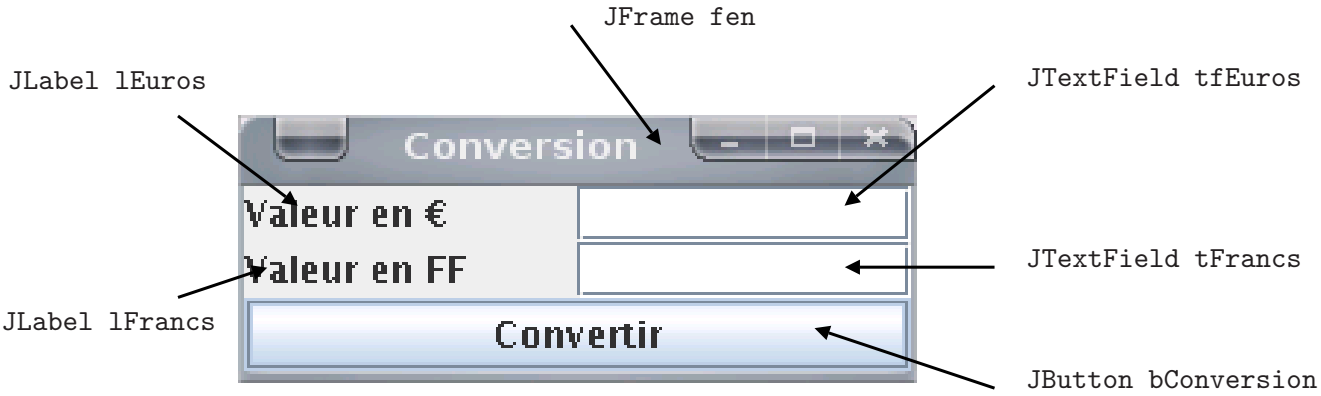


FIG. 4.2 – Les différents éléments visibles

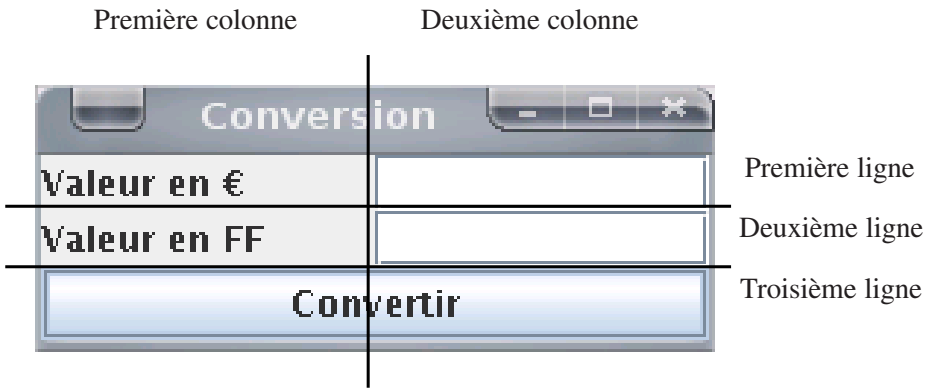


FIG. 4.3 – La grille retenue pour le placement des composants

FIG. 4.4 – Les deux `JPanel` utilisés

– Deux conteneurs l'un dans l'autre, l'un géré par un `GridLayout`, l'autre géré par un `BorderLayout`.

Si on considère la deuxième solution, deux conteneurs sont nécessaires. Ils n'ont aucune fonction particulière, des `JPanel` sont donc suffisant. Le découpage se fait selon la figure 4.4.

Le panneau le plus interne (`pane`) utilise un gestionnaire de placement `GridLayout`. Le panneau externe utilise un `BorderLayout`.

4.1.4 Programme obtenu

Le programme peut être écrit à partir de l'étude précédente. Nous avons choisi de créer une classe à partir du panneau externe et de construire la fenêtre dans la méthode `main`.

```
1  public class ConversionEuro extends JPanel {
2     JLabel lEuros, lFrancs;
3     JTextField tfEuros, tfFrancs;
4     JButton bConversion;
5     JPanel pane;
6
7     public ConversionEuro() {
8        super();
9        lEuros = new JLabel("Valeur en E");
10       lFrancs = new JLabel("Valeur en FF");
11       tfEuros = new JTextField(10);
12       tfFrancs = new JTextField(10);
13       bConversion = new JButton("Convertir");
14       pane = new JPanel();
15
16       pane.setLayout(new GridLayout(2,0));
17       pane.add(lEuros);
18       pane.add(tfEuros);
19       pane.add(lFrancs);
20       pane.add(tfFrancs);
21
22       this.setLayout(new BorderLayout());
23       this.add(pane, BorderLayout.CENTER);
24       this.add(bConversion, BorderLayout.SOUTH);
25    }
```

```
26
27      public static void main(String[] args) {
28          JFrame fen = new JFrame("Conversion");
29          fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30          fen.setContentPane(new ConversionEuro());
31          fen.pack();
32          fen.setVisible(true);
33      }
34  }
```

Les différents composants graphiques sont déclarés dans les lignes 2 à 5 ; ils sont ensuite instanciés (lignes 9 à 14). Le gestionnaire de placement du panneau le plus intérieur (`pane`) est crée dans la ligne 16 puis les différents éléments sont ajoutés (lignes 17 à 20). Le gestionnaire de placement de la classe est construit ligne 22. Les deux éléments `pane` et `bConversion` sont placé lignes 23 et 24.

La fenêtre principale de l'application est crée en ligne 28. Une instance du panneau est directement désignée comme contenu en ligne 30. Le titre de la fenêtre est spécifié lors de la construction. Sa taille est définie à l'optimum (ligne 31), elle est rendu visible à la fin du programme après toutes les modifications graphiques (ligne 32).

4.2 Une interface plus complète

4.2.1 Présentation de l'interface voulue

L'interface à réaliser est un visualisateur de fichiers images. La figure 4.5 présente la vue générale de l'application. La barre de menu est composée des menus : Fichier (Ouvrir, Répertoire utilisateur, Quitter) et Image (Précédente, Suivante, Informations). Les icônes utilisés sont présents dans le répertoire de l'application. L'application est basé sur une seule fenêtre.

4.2.2 Identification des différents composants

Dans un premier temps, les différents éléments graphiques (boutons, menus,...) doivent être identifiés et nommés. La figure 4.6 présente le résultat de cette démarche.

4.2.3 Choix du/des gestionnaires de placement

Dans cette application, le choix des gestionnaires de placement est assez aisé. Plusieurs fonctionnalités ne peuvent être crées qu'en utilisant le bon gestionnaire de placement. Ainsi, la barre d'outils fait appel à un `JToolBar`, les panneaux à défilement à des `JScrollPane` et le panneau divisé ne peut être qu'un `JSplitPane`. La figure 4.7 illustre ces choix.

Seul le gestionnaire du `JPanel` doit être choisi. Il contient une barre d'outils (devant être en haut) et un panneau divisé (devant prendre l'espace restant). Le gestionnaire `BorderLayout` semble être le plus approprié.

4.2.4 Programme obtenu

Le programme peut être écrit à partir de l'étude précédente. Nous avons choisi de créer une classe à partir du panneau externe et de construire la fenêtre dans la méthode `main` comme dans l'exemple précédent. Le code comporte quelques commentaires suffisants pour comprendre le programme.

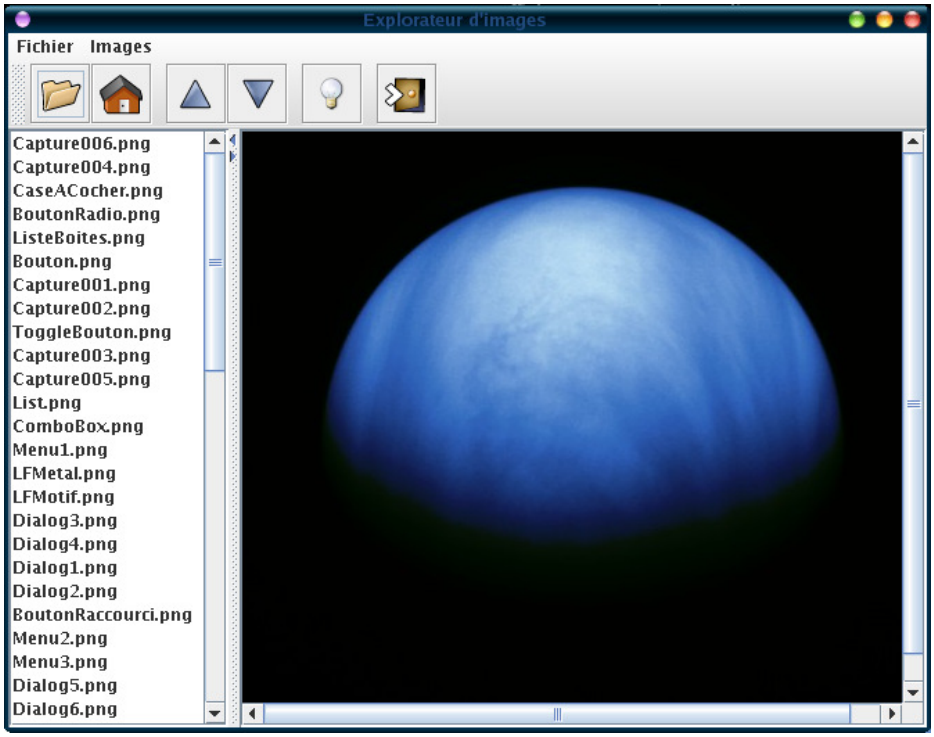


FIG. 4.5 – Le résultat voulu

```
1  public class PanneauExplorateurImage extends JPanel {
2
3      /*
4       * Les éélments de la barre de menu
5       */
6      private JMenuBar laBarreDeMenu;
7      private JMenu mFichier, mImages;
8      private JMenuItem mOuvrir, mHome, mQuitter;
9      private JMenuItem mPrecedente, mSuivante, mInfo;
10     /*
11      * La barre de menu et ses boutons
12      */
13     private JToolBar laBarreDOutils;
14     private JButton bOuvrir, bHome, bQuitter, bHaut, bBas, bInfo;
15     /*
16      * Le panneau de droite
17      */
18     private JScrollPane pDefilantDroit;
19     private JLabel leLabelImage;
20     private ImageIcon lImage;
21     /*
22      * Le panneau de gauche
```

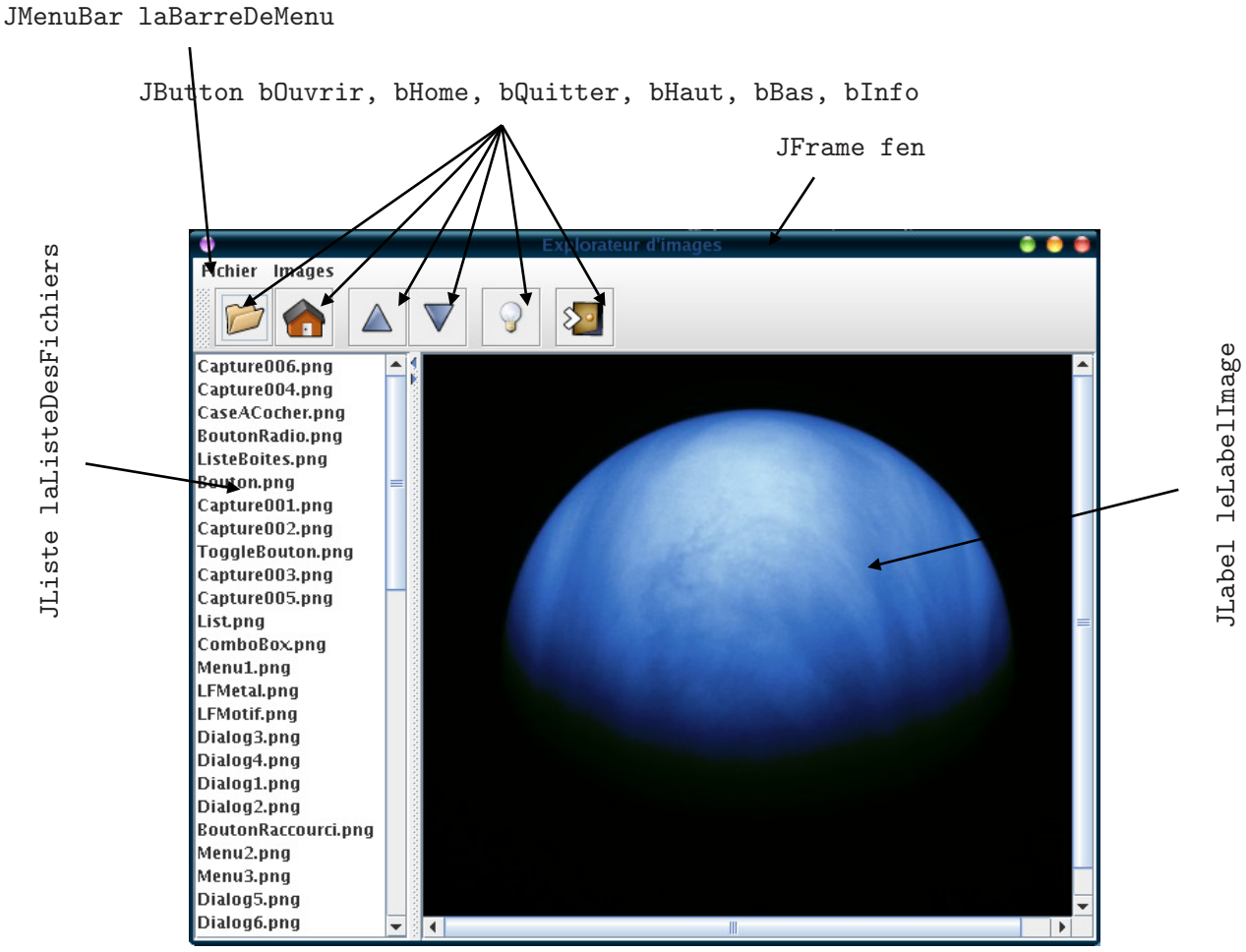


FIG. 4.6 – Les différents éléments visibles

```
23      */
24     private ListeurFichier leListeur;
25     private JList laListeDesFichiers;
26     private JScrollPane pDefilantGauche;
27     /*
28      * Le panneau ééspar
29      */
30     private JSplitPane pSepare;
31
32     public PanneauExplorateurImage() {
33        super();
34        this.setLayout(new BorderLayout());
35        /*
36         * La barre de menus
37         */
38        laBarreDeMenu = new JMenuBar();
```

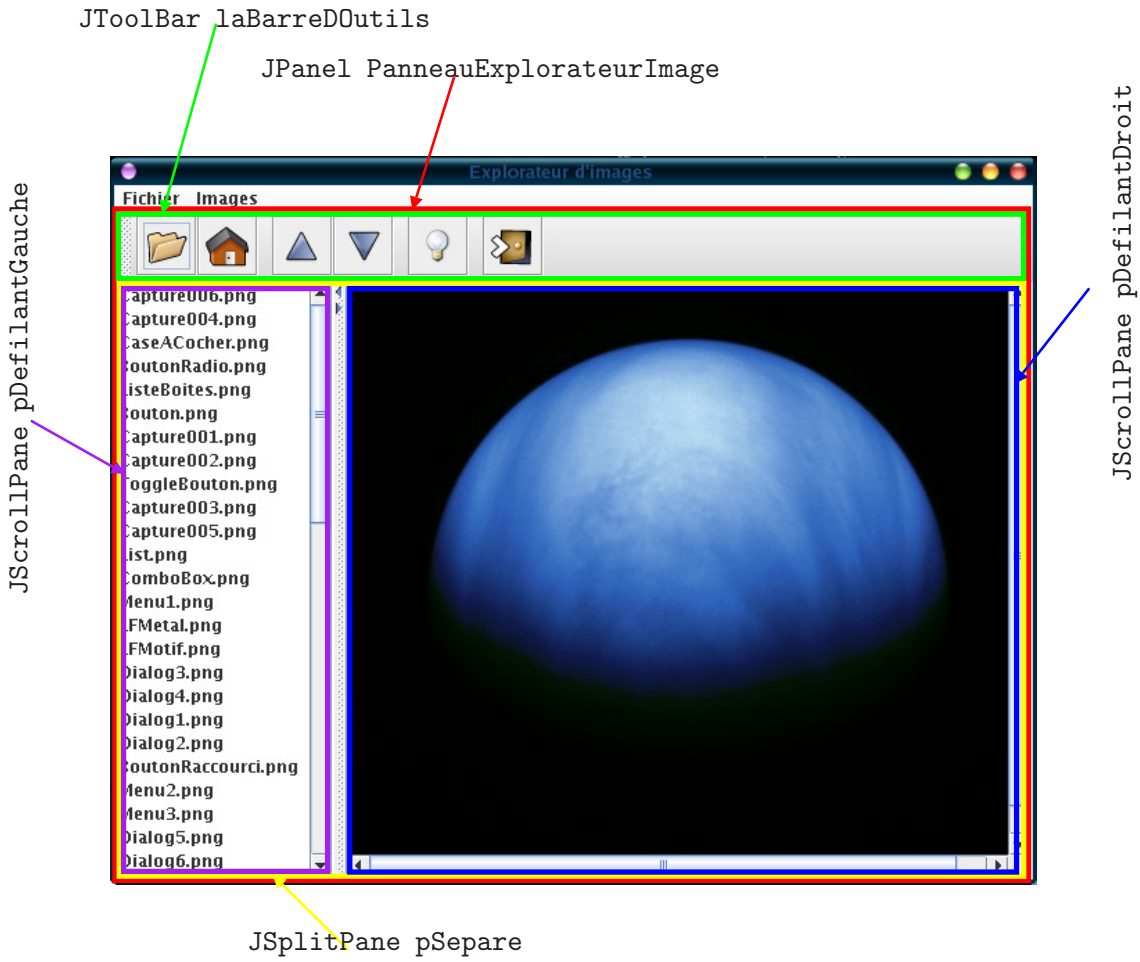


FIG. 4.7 – Les différents conteneurs présents

```
39        mFichier = new JMenu("Fichier");
40        mImages = new JMenu("Images");
41        mOuvrir = new JMenuItem("Ouvrir");
42        mHome = new JMenuItem("éRpertoire utilisateur");
43        mQuitter = new JMenuItem("Quitter");
44        mPrecedente = new JMenuItem("ééPrcdente");
45        mSuivante = new JMenuItem("Suivante");
46        mInfo = new JMenuItem("Information sur l'image");
47
48        mFichier.add(mOuvrir);
49        mFichier.add(mHome);
50        mFichier.addSeparator();
51        mFichier.add(mQuitter);
52        laBarreDeMenu.add(mFichier);
53
54        mImages.add(mPrecedente);
```

```
55        mImages.add(mSuivante);
56        mImages.add(mInfos);
57        laBarreDeMenu.add(mImages);
58
59        /*
60         * Construction de la barre d'outils et de ses boutons
61         */
62        laBarreDOutils = new JToolBar();
63        bOuvrir = new JButton(new ImageIcon("open.png"));
64        bHome = new JButton(new ImageIcon("home.png"));
65        bHaut = new JButton(new ImageIcon("up.png"));
66        bBas = new JButton(new ImageIcon("down.png"));
67        bInfo = new JButton(new ImageIcon("info.png"));
68        bQuitter = new JButton(new ImageIcon("quit.png"));
69        laBarreDOutils.add(bOuvrir);
70        laBarreDOutils.add(bHome);
71        laBarreDOutils.addSeparator();
72        laBarreDOutils.add(bHaut);
73        laBarreDOutils.add(bBas);
74        laBarreDOutils.addSeparator();
75        laBarreDOutils.add(bInfo);
76        laBarreDOutils.addSeparator();
77        laBarreDOutils.add(bQuitter);
78        this.add(laBarreDOutils,BorderLayout.NORTH);
79
80        /*
81         * Le panneau de droite
82         */
83        lImage = new ImageIcon("venus.jpg");
84        leLabelImage = new JLabel(lImage);
85        pDefilantDroit = new JScrollPane(leLabelImage);
86        pDefilantDroit.setHorizontalScrollBarPolicy(JScrollPane.
             HORIZONTAL_SCROLLBAR_AS_NEEDED);
87        pDefilantDroit.setVerticalScrollBarPolicy(JScrollPane.
             VERTICAL_SCROLLBAR_AS_NEEDED);
88
89
90        /*
91         * Le panneau de gauche
92         */
93        leListeur = new ListeurFichier();
94        leListeur.ConstruireList(".");
95        laListeDesFichiers = new JList(leListeur.
             getListeDesFichiers());
96
97        pDefilantGauche = new JScrollPane(laListeDesFichiers);
```

```
98          pDefilantGauche.setHorizontalScrollBarPolicy(JScrollPane.
               HORIZONTAL_SCROLLBAR_NEVER);
99          pDefilantGauche.setVerticalScrollBarPolicy(JScrollPane.
               VERTICAL_SCROLLBAR_AS_NEEDED);
100
101         /*
102          * Le panneau ééspar
103          */
104         pSepare = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
               pDefilantGauche, pDefilantDroit);
105         pSepare.setOneTouchExpandable(true);
106         this.add(pSepare, BorderLayout.CENTER);
107     }
108
109
110     public static void main(String[] args) {
111         JFrame fen = new JFrame("Explorateur d'images");
112         PanneauExplorateurImage pan = new PanneauExplorateurImage
               ();
113         fen.setContentPane(pan);
114         fen.setVisible(true);
115         fen.setJMenuBar(pan.laBarreDeMenu);
116         fen.pack();
117
118     }
119 }
```

Chapitre 5

Les événements et leur gestion

5.1 Principe des événements

Java propose des mécanismes de communication entre les objets basés sur des événements. L'événement est émis par un objet (la source) et reçu par un ou plusieurs objets (les auditeurs). On dit que la source notifie un événement aux auditeurs. Pour pouvoir recevoir un (ou des) événement(s) les auditeurs doivent s'enregistrer au près de la (des) source(s).

5.1.1 Les événements

Les événements sont des héritiers de la classe `EventObject`. Cette classe ne propose qu'une méthode : `getSource` qui renvoie l'objet qui a crée cet événement. Le constructeur de `EventObject` admet l'objet qui l'a crée comme seul paramètre. Il est évidement possible de surcharger ce constructeur lors de l'héritage en fonction des besoins. Par convention, les noms des objets sont de la forme `XXXEvent`. Le constructeur de la classe `EventObject` nécessite la source comme paramètre.

5.1.2 La source

La source doit être capable de créer des événements et de les transmettre aux auditeurs. Les auditeurs doivent pouvoir s'enregistrer auprès de la source mais peuvent aussi se libérer de cette relation.

5.1.3 Les auditeurs

Les auditeurs doivent implémenter une interface qui dérive de `EventListener`. En pratique, cette interface a un nom de la forme `XXXListener`. Elle doit comprendre une méthode qui admet comme argument un élément de type `XXXEvent`.

5.1.4 Mise en oeuvre

A des fins pédagogiques[1] les exemples suivants vont présenter l'utilisation des événements. Tout d'abord un événement simple peut être construit ainsi :

[1]Les visibilités des méthodes ainsi que les synchronisations des objets ont été ignorés pour éclaircir les exemples. De plus la notation proposée n'a pas été respectée pour alléger la lecture.

```
public class MonEvenement extends EventObject {
   public MonEvenement(Object source) {
      super(source);
      System.out.println("Lé'évnement est écre ...");
   }
}
```

Les objets voulant recevoir des événements `MonEvenement` doivent implémenter l'interface suivante :

```
public interface MonEvenementListener extends EventListener {
   void MonEvenementOccurs(MonEvenement e);
}
```

Une source d'événements doit tenir à jour une liste des auditeurs d'événements. Cette liste est souvent basée sur un objet de type `Vector`. Pour la mettre à jour deux méthodes sont crées : `addXXXListener` et `removeXXXListener`. Dans cet exemple les événements sont crées à la demande par la méthode `notifyMonEvenement`. Un événement (`evt`) est crée puis il est transmit à tous les auditeurs (boucle `for...`).

```
public class MaSource {
   private Vector monEvenementListeners = new Vector();

   public synchronized void addMonEvenementListener(
      MonEvenementListener mel ){
      if( !monEvenementListeners.contains( mel ) )
         monEvenementListeners.addElement( mel ) ;
   }

   public synchronized void removeMonEvenementListener(
      MonEvenementListener mel ){
      if( monEvenementListeners.contains( mel ) )
         monEvenementListeners.removeElement( mel ) ;
   }

   public void notifyMonEvenement(){
      MonEvenement evt = new MonEvenement(this) ;

      for( int i = 0; i < monEvenementListeners.size(); i++ ){
         MonEvenementListener listener = (MonEvenementListener)
            monEvenementListeners.elementAt( i ) ;
         listener.MonEvenementOccurs( evt ) ;
      }
   }

}
```

Pour tester le mécanisme des événements, on construit des classes qui mettent en oeuvre ces différents éléments. Une classe auditeur est construite :

```
public class MaClasseAuditeur implements MonEvenementListener {

   public void MonEvenementOccurs(MonEvenement e) {
      System.out.println("J'ai çreu un evenement de "+e.
         getSource());

   }

}
```

Elle affichera un simple message quand elle recevra un événement[2]. La classe exécutable contient deux objets : une source (classe `MaSource`) et un auditeur (classe `MaClasseAuditeur`) :

```
public class TestEvenement {

   public static void main(String[] args) {
      MaSource uneSource = new MaSource();
      MaClasseAuditeur unAuditeur= new MaClasseAuditeur();
      uneSource.ajouteMonEvenementListener(unAuditeur);
      uneSource.envoieMonEvenement();
   }

}
```

A l'exécution on obtient le résultat suivant :

```
L'evenement est cree ...
J'ai recu un evenement de MaSource@1f6a7b9
```

5.2 Gestion des événements dans Swing

5.2.1 Principes de la gestion d'événements dans Swing

Tous les composants de Swing (et de AWT) créent des évènements en fonction des actions de l'utilisateur. Les évènements sont des descendants de `EventObject`, notés `XXXEvent`. Les composants proposent des méthodes du type `AddXXXListener` pour enregistrer un auditeur. L'auditeur doit implémenter l'interface correspondante qui est de la forme `XXXListener`.

Par exemple, le composant `JAbstractButton` (et donc tous ses descendants) possède une méthode `addActionListener` utilisée pour enregistrer une classe implémentant l'interface `ActionListener`. L'interface `ActionListener` n'a qu'une méthode `actionPerformed( ActionEvent e)` qui doit être implémentée pour gérer l'évènement (l'évènement est disponible via le paramètre `e`). La classe `ActionEvent` propose plusieurs méthodes utiles comme `ActionEvent.getSource` pour identifier la source de l'évènement[3].

[2] Utiliser `e.getSource` pour afficher la source de l'évènement dans un `println` n'est pas très élégant, mais ça marche !

[3] En fait, `getSource` est une méthode de la classe mère `EventObject`.

5.2.2 Les différents événements

Dans le cadre de Swing deux types d'événements existent : les événements "de base" et les événements sémantiques. Les premiers, qui sont communs à tous les descendants de `Component` (et donc de `JComponent`), couvrent les événements bas-niveau comme les mouvements de la souris, la modification des composants ou l'utilisation du clavier. Les événements sémantiques représentent des actions de haut-niveau de l'utilisateur comme la sélection d'un menu, la modification du curseur dans les composants textuels,...Souvent, il s'agit de combinaisons d'événements bas-niveau (sélectionner un menu correspond à des mouvements de souris, l'appui sur le bouton gauche puis le relâchement du bouton gauche). Les événements "de base" sont présentés dans le tableau 5.1.

TAB. 5.1 – Les événements "de base"

Événements	Description
`ComponentEvent`	Un composant a bougé, changé de taille, changé de visibilité.
`ContainerEvent`	Un composant a été ajouté/enlevé du conteneur.
`FocusEvent`	Le composant a gagné/perdu le focus.
`KeyEvent`	Une touche a été appuyée/relâchée.
`MouseEvent`	La souris a bougée ou l'un des boutons a changé d'état.
`MouseWheelEvent`	La molette de souris a été tournée.
`WindowEvent`	La fenêtre a été réduite/redimensionnée/agrandie.

Le tableau ci-dessous (tableau 5.2)présente quelques événements sémantiques, la liste est trop longue pour tous les présenter ici. Le lecteur trouvera facilement ceux qui manquent dans la documentation officielle de Java[4].

5.2.3 Gestion des événements par la classe graphique

L'approche la plus simple de mise en oeuvre est l'utilisation de la classe graphique comme récepteur d'événements. La (ou les) procédure(s) liée(s) à l'interface est (sont) simplement implémentée(s) dans la classe graphique. Par exemple, pour gérer l'action sur un bouton dans un `JPanel` l'interface `ActionListener` est ajouté à la déclaration de la classe, ainsi que la méthode `actionPerformed`. L'implémentation peut être la suivante :

```
public class Test extends JPanel implements ActionListener{
   ...
   JButton leBouton;
   ...
   public Test()  {
      ...
      leBouton.addActionListener(this);
      ...
   }
   public void actionPerformed(ActionEvent e) {
      // Partie utile
```

[4]On retrouve les événements en recherchant les méthodes `addXXXListener`

TAB. 5.2 – Quelques événements sémantiques

Événements	Description
`ActionEvent`	De nombreux éléments génèrent cet événement : un `JButton` lors d'un clic de souris, un élément de menu (`JXXXMenuItem`) est sélectionné, la touche `Entrée` a pressée dans `JTextField`,...
`ChangeEvent`	Un changement d'état a eu lieu dans un composant : un élément radio (bouton ou menu), une case à cocher (bouton ou menu), un bouton ou un élément de menu a changé d'état, une glissière a bougée, changement d'onglet dans un `JTabbedPane`,...
`CaretEvent`	Modification de la position du curseur dans un élément `JTextComponent` ou un de ses descendants.
`ItemEvent`	Un nouvel élément a été sélectionné dans une `JComboBox`.
`ListSelectionEvent`	Un nouvel élément a été sélectionné dans une `JList`.
...	...

```
    }
    ...
```

De même, pour implémenter plusieurs interfaces :

```
public class Tests extends JPanel implements ActionListener,
    KeyListener{
    ...
    JButton leBouton;
    JTextField leTextField;
    ...
    public TestActionListener()  {
        ...
        leBouton.addActionListener(this);
        leTextField.addKeyListener(this);
        ...
    }

    public void actionPerformed(ActionEvent e) {
        // Partie utile
    }

    public void keyTyped(KeyEvent e) {
        // Partie utile
    }
```

```
    public void keyPressed(KeyEvent e) {
        // Partie utile
    }

    public void keyReleased(KeyEvent e) {
        // Partie utile
    }
    ...
```

Si deux (ou plus) composants émettent le même événement, il faut identifier la source de l'événement dans la procédure concernée à l'aide de `getSource`.

```
public class TestDeuxBouton extends JPanel implements
    ActionListener{
    ...
    JButton leBouton1, leBouton2;
    ...
    public TestDeuxBouton()  {
        ...
        leBouton1.addActionListener(this);
        leBouton2.addActionListener(this);
        ...
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==leBoutton1){
        // La source est leBouton1
        }
        if (e.getSource()==leBoutton2){
        // La source est leBouton2
        }
    }
...
```

Cette méthode est très simple lorsque le nombre d'événements à gérer est faible. L'imbrication entre l'interface graphique et l'auditeur est dans ce cas extrêmement forte. Ce peut être une source d'erreurs et cela rend le code difficile peut évolutif (si on ajoute un bouton, il faut retoucher la méthode `actionPerformed` et la modifier).

5.2.4 Gestion des événements par des classes dédiées

L'approche opposée consiste à construire une classe par auditeur d'évènement. Deux évènements sont présents dans notre exemple, un `ActionEvent` généré par le `Bouton1` et un `ActionEvent` généré par le `Bouton2`. Le premier auditeur écoute les évènements du `Bouton1` :

```
public class ListenerBouton1 implements ActionListener {
...
    public void actionPerformed(ActionEvent e) {
```

```
            // Partie utile
    }
...
}
```

De même, le second auditeur écoute les évènements du `Bouton2` :

```
public class ListenerBouton2 implements ActionListener {
...
    public void actionPerformed(ActionEvent e) {
            // Partie utile
    }
...
}
```

Les deux auditeurs sont instanciés dans les méthodes d'enregistrement (lignes 10 et 11) car aucune référence n'est nécessaire :

```
1  public class TestClassesExternes extends JPanel{
2      ...
3      JButton leBouton1, leBouton2;
4      ...
5      public TestClassesExternes()  {
6          ...
7          leBouton1 =new JButton("Bouton 1");
8          leBouton2 = new JButton("Bouton 2");
9          ...
10         leBouton1.addActionListener(new ListenerBouton1());
11         leBouton2.addActionListener(new ListenerBouton2());
12     ...
13     }
```

Les classes auditeurs n'ont pas directement accès aux variables membres de la classe visuelle. Ils est donc nécessaire de créer des références entre ces deux éléments. Dans le chapitre 6, ces éléments seront discutés.

5.2.5 Gestion des événements par des classes membres internes

Le problème évoqué ci-dessus peut être résolu en utilisant des classes membres internes. Les classes membres peuvent accéder facilement aux éléments de la classe graphique. Cette méthode conduit à créer une classe membre interne par composant et par événement. Si on reprend l'exemple précédent :

```
public class TestClassesMembres extends JPanel{
    ...
    JButton leBouton1, leBouton2;
    ...
    public TestClassesMembres()  {
        ...
        leBouton1 =new JButton("Bouton 1");
```

```
        leBouton2 = new JButton("Bouton 2");
        ...
        leBouton1.addActionListener(new ListenerBouton1());
        leBouton2.addActionListener(new ListenerBouton2());
        ...
    }
    private class ListenerBouton1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            // Partie utile
        }
    }
    private class ListenerBouton2 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            // Partie utile
        }
    }
}
```

5.2.6 Gestion des événements par des classes membres internes anonymes

Présentation des classes anonymes

Si une classe est crée pour n'être utilisée qu'une seule fois dans le programme, on peut utiliser une construction particulière : les classes anonymes. Cette approche permet de construire un objet qui étend une classe ou construire un objet qui implémente une interface donnée sans l'associer à une variable. Si l'objet hérite d'un parent, il est possible de surcharger les méthodes de la classe mère, mais il n'est pas possible d'en ajouter d'autre. Si l'objet implémente une interface, seules les méthodes de l'interface peuvent être crées.

Dans le code suivant l'objet `unObjet` n'est utilisé que lors de l'appel de la méthode `maMethodeDAppel`.

```
class UneClasse(){
    ...
    ... laMethode(){
    ...
    }
}
...
// Instanciation de la classe
...
    UneClasse unObjet = new UneClasse
...
// Utilisation de la classe
...
    maMethodeDAppel(unObjet);
...
```

Une classe anonyme peut être utilisée de la manière suivante :

```
...
   maMethodeDAppel(new UneClasse(){
   });
```

On peut aussi surcharger la méthode la classe mère :

```
...
   maMethodeDAppel(new UneClasse(){
      ... laMethode(){
      }
   });
```

La valeur de retour de `new` est directement utilisé comme argument pour la méthode. Par contre, il n'est pas possible d'accéder à l'objet ainsi crée. Toutefois, le compilateur construit des fichiers classes qui correspondent à ces classes.

La même approche peut être utilisée pour les interfaces :

```
...
   maMethodeDAppel(new UneInterface(){
      ... uneMethodeDeLInterface(){
      }
   });
```

En pratique, les classes anonymes sont peu utilisées car elles rendent le code peu lisible sauf pour les auditeurs d'évènements.

Application aux auditeurs d'événements

Dans le cas des auditeurs, les classes sont simplement instanciées pour être ensuite utilisée comme paramètre de la fonction `addXXXListener`. Dans ce cas, on peut utiliser des classes anonymes sans nuire à la qualité du code de la manière suivante :

```
public class TestClassesAnonymes extends JPanel{
   ...
   JButton leBouton;
   JTextField leTextField;
   ...
   public TestClassesAnonymes()  {
      ...
      leBouton =new JButton("Bouton 1");
      leTextField = new JTextField();
      ...
      leBouton.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            // Partie utile
         }
      });
      leTextField.addKeyListener(new KeyListener() {
         public void keyTyped(KeyEvent e) {
            // Partie utile
```

```
            }
            public void keyPressed(KeyEvent e) {
               // Partie utile
            }
            public void keyReleased(KeyEvent e) {
               // Partie utile
            }
        });
    }
```

Les instanciations inutiles sont supprimées et chaque auditeur est rattaché à un seul objet. Cette approche permet de facilement modifier le code, si un on ajoute (ou on enlève) un composant ou/et un événement.

5.2.7 Gestion des événements par des classes d'adaptations (ou adaptateurs factices)

Les interfaces auditeurs les plus utilisées comportent de nombreuses méthodes[5]. Lors de la conception d'un auditeur on doit donc déclarer toutes ces méthodes mêmes si elles ne sont pas utilisées. Ce développement prend du temps et alourdit le code. Afin d'éviter ceci, on peut utiliser les classes d'adaptation (ou adaptateurs factices). Ce sont des classes abstraites qui contiennent toutes les méthodes de l'interface auditeur d'événement.

Par exemple pour l'interface `KeyListener` on doit définir les trois méthodes `keyTyped`, `keyPressed` et `keyReleased`. Si seul l'événement `keyTyped` est utilisé, le code suivant est nécessaire :

```
public class ListenerClavier implements KeyListener {
    public void keyTyped(KeyEvent e) {
       // Partie utile
    }
    public void keyPressed(KeyEvent e) {
    }
    public void keyReleased(KeyEvent e) {
    }
}
```

La classe abstraite `KeyAdapter` implémente les trois méthodes de la manière suivante :

```
public void keyTyped(KeyEvent e) {}
public void keyPressed(KeyEvent e) {}
public void keyReleased(KeyEvent e) {}
```

Pour construire l'auditeur on crée une classe fille de la classe `KeyAdapter` qui surcharge les méthodes utiles. Ce qui conduit à l'implémentation suivante :

```
public class ListenerClavier extends KeyAdapter {
    public void keyTyped(KeyEvent e) {
       // Partie utile
    }
}
```

[5] Par exemple 5 méthodes pour `MouseListener`.

Avec cette implémentation, seuls les événements utilisés sont présents dans le code. Celui-ci est donc plus clair et plus compact. Cette méthode de conception a toutefois une limite : Java ne supportant pas l'héritage multiple, on doit donc multiplier le nombre de classes auditeurs. Par exemple, un auditeur qui implémente les interfaces `KeyListener` et `MouseListener` peut avoir la déclaration suivante :

```
public class ListenerMultiple implements KeyListener,
   MouseListener {

   public void keyTyped(KeyEvent e) {
      // Partie utile
   }
   public void keyPressed(KeyEvent e) {}
   public void keyReleased(KeyEvent e) {}
   public void mouseClicked(MouseEvent e) {
      // Partie utile
   }
   public void mousePressed(MouseEvent e) {}
   public void mouseReleased(MouseEvent e) { }
   public void mouseEntered(MouseEvent e) {}
   public void mouseExited(MouseEvent e) {}
}
```

Cet auditeur peut être utilisé de la manière suivante :

```
...
ListenerMultiple unListener = new ListenerMultiple();
...
unComposant.addMouseListener(unListener);
unComposant.addKeyListener(unListener);
...
```

Pour utiliser les adaptateurs, on doit créer deux classes listeners, l'une pour `KeyListener` et l'autre pour `MouseListener`. Par exemple, pour l'interface `KeyListener` :

```
public class ListenerClavier extends KeyAdapter {
   public void keyTyped(KeyEvent e) {
      // Partie utile
   }
}
```

et pour l'interface `MouseListener` :

```
public class ListenerSouris extends MouseAdapter {
   public void mouseClicked(MouseEvent e) {
      // Partie utile
   }
}
```

Ces listeners pourront être utilisés ainsi :

```
...
ListenerClavier unListenerClavier = new ListenerClavier();
```

```
ListenerSouris unListenerSouris = new ListenerSouris();
...
unComposant.addMouseListener(unListenerSouris);
unComposant.addKeyListener(unListenerClavier);
...
```

Les classes adaptateurs factices peuvent être utilisées pour créer des classes auditeurs externes comme ci-dessus, mais aussi des classes membres internes ou des classes membres anonymes. L'exemple suivant présente l'utilisation d'une classe anonyme :

```
unComposants.addMouseListener(new MouseAdapter(){
   public void mouseClicked(MouseEvent e) {
      //Partie utile
      }
   });
```

Cette approche est la plus utilisée, notamment par les environnements de développement. Le tableau suivant présente les principaux adaptateurs :

TAB. 5.3 – Les principaux adaptateurs

Événements	Classe abstraite	Méthodes
ComponentEvent	ComponentAdapter	componentHidden componentMoved componentResized componentShow
FocusEvent	FocusAdapter	focusGained focusLost
KeyEvent	KeyAdapter	keyPressed keyReleased keyTyped
MouseEvent	MouseAdapter	mouseClicked mouseEntered mouseExited mousePressed mouseReleased
MouseEvent	MouseMotionAdapter	mouseDragged mouseMoved
MouseEvent	MouseInputAdapter	mouseClicked mouseEntered mouseExited mousePressed mouseReleased mouseDragged mouseMoved
ContainerEvent	ContainerAdapter	componentAdded componentRemoved
InternalFrameEvent	InternalFrameAdapter	internalFrameActivate internalFrameClosed internalFrameClosing internalFrameDeactivated internalFrameDeiconified internalFrameIconifie internalFrameOpened
WindowEvent	WindowAdapter	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowGainedFocus windowIconified windowLostFocus windowOpened windowStateChanged

Chapitre 6

La construction d'applications graphiques

Une application graphique est composée de trois types d'éléments :
- l'interface graphique qui regroupe les composants, les conteneurs et les gestionnaires de placement,
- les événements qui sont émis par les composants et reçus par les auditeurs,
- la ou les classes d'applications qui traitent des données, manipulent des fichiers, accèdent aux bases de données, ...

Réaliser une application graphique consiste donc à faire cohabiter le plus harmonieusement ces trois types d'éléments. De plus, le code doit pouvoir être facilement modifié pour les améliorations futures (et pour faciliter la chasse aux bugs !)

6.1 Principes et vocabulaire

Un paradigme de programmation issu des *desing-patterns* souvent utilisé est le modèle MVC (Modèle, Vue, Contrôleur). On retrouve sensiblement les trois éléments évoqués ci-dessus :
- le modèle est la partie qui gère le comportement de l'application : traitement des données, manipulation de fichiers, interactions avec les bases de données, connections réseaux,...
- la vue est l'interface par laquelle l'utilisateur agit. Ce peut être une interface graphique, une applet, des pages HTML,...
- le contrôleur se place entre ces deux éléments est transmet les requêtes de la vue au modèle. En théorie, il n'effectue aucun traitement.

Cette approche permet, d'une part de réutiliser facilement des éléments et d'autre part de partager le modèle entre plusieurs interfaces. Par exemple, un site de e-commerce basé sur un seul modèle peut utiliser deux contrôleurs et deux vues. Le modèle permet de passer des commandes, d'ajouter ou de supprimer des produits, de gérer les clients,...Une des vues peut être une applet utilisée par les clients pour passer des commandes, par contre les clients n'ont pas d'accès aux fonctions de gestion des stocks. A l'inverse, la gestion des stocks est assurée par une application Swing qui permet aussi de gérer les clients. Par contre, il n'est pas nécessaire de pouvoir passer des commandes via cette application. La figure 6.1 présente une synthèse de cette approche en proposant des solutions possibles pour gérer les liaisons entre les différents éléments. La liaison entre le modèle et la vue sont assuré par le couple `Observer/Observable` pour les cas complexes. Dans les cas simple, une approche basée sur des `get/set` est suffisante.

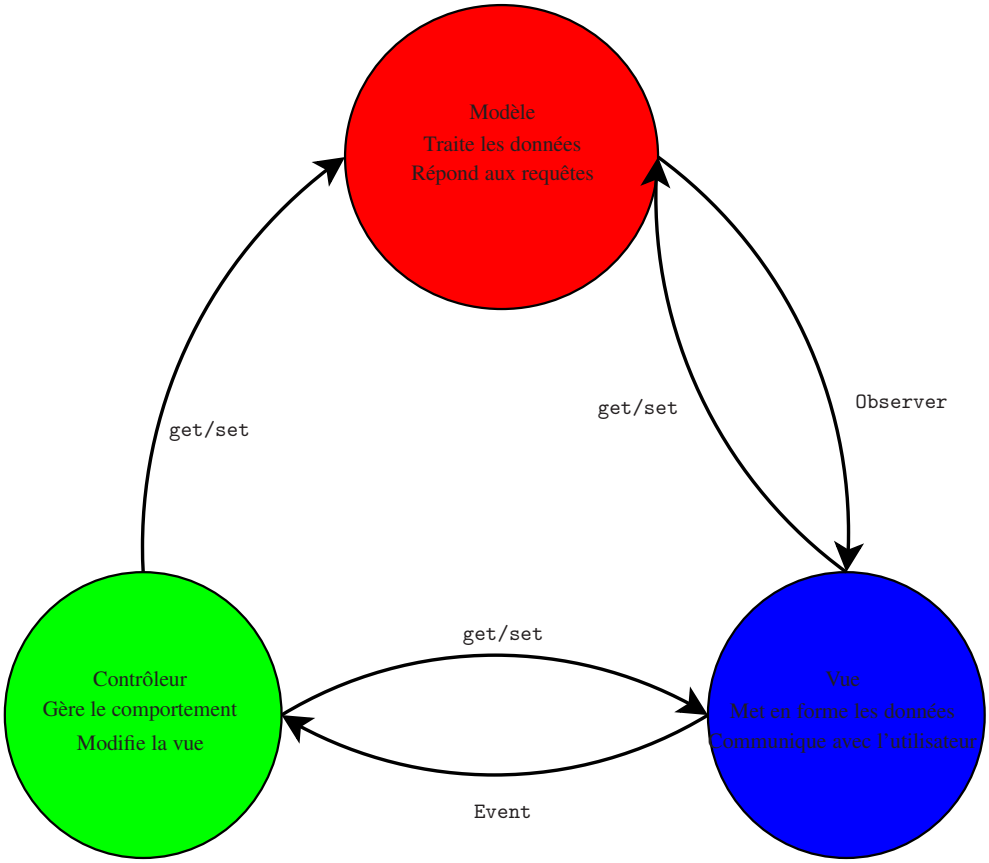


FIG. 6.1 – Relations entre les différentes entités dans une modélisation MVC

6.2 Les éléments modèle, vue et contrôleur ne sont pas distincts

L'approche la plus simple consiste à mettre tous les éléments dans la même classe. Dans l'exemple qui suit, on retrouve les éléments graphiques `pHaut`, `lEuros`, `lFrancs`, `tfEuros`, `tfFrancs` et `bConversion` qui sont construits et ajoutés au conteneurs dans les lignes 21 à 35.

La gestion des événements est faite par des classes membres internes anonymes. Trois événements sont gérés : changement de focus sur `tfEuros`, changement de focus sur `tfFrancs`, et action du bouton `bConversion`.

Lorsque `tfEuros` ou `tfFrancs` prennent le focus on efface le contenu de `tfEuros` et `tfFrancs` (lignes 39 à 57).

L'événement action du bouton est géré dans les lignes 58 à 73. On cherche quel champ de saisie est vide, et on calcul la conversion en utilisant les méthodes de conversion `calculEuros` et `calculFrancs` (lignes 75 à 81). La figure 6.4 présente le diagramme UML de l'application.

```
1  import ...
2
3  public class ConversionEuroMVC extends JPanel {
4
5     /* Le panneau du haut */
6     private JPanel pHaut;
7     private JLabel lEuros, lFrancs;
8     private JTextField tfEuros, tfFrancs;
9     /* Le bouton de conversion */
10    private JButton bConversion;
11    /* Le taux de conversion */
12    private static final float TAUX = (float) 6.55957;
13    /* Les valeurs monetaires */
14    private float valeurEuros;
15    private float valeurFrancs;
16
17
18    public ConversionEuroMVC() {
19       super();
20       /* Partie superieure */
21       lEuros = new JLabel("Valeur en E");
22       lFrancs = new JLabel("Valeur en FF");
23       tfEuros = new JTextField("",10);
24       tfFrancs = new JTextField("",10);
25       pHaut = new JPanel();
26       pHaut.setLayout(new GridLayout(2,0));
27       pHaut.add(lEuros);
28       pHaut.add(tfEuros);
29       pHaut.add(lFrancs);
30       pHaut.add(tfFrancs);
31       /* Inclusion du bouton et de la partie superieure */
32       bConversion = new JButton("Convertir");
33       this.setLayout(new BorderLayout());
```

```
34        this.add(pHaut, BorderLayout.CENTER);
35        this.add(bConversion, BorderLayout.SOUTH);
36        /*
37         * GESTION DES EVENEMENTS
38         */
39        /* Pour tfEuros */
40        tfEuros.addFocusListener(new FocusListener(){
41           public void focusGained(FocusEvent e) {
42              tfEuros.setText("");
43              tfFrancs.setText("");
44           }
45           public void focusLost(FocusEvent e) {
46           }
47        });
48        /* Pour tfFrancs */
49        tfFrancs.addFocusListener(new FocusListener(){
50           public void focusGained(FocusEvent e) {
51              tfEuros.setText("");
52              tfFrancs.setText("");
53           }
54           public void focusLost(FocusEvent e) {
55           }
56        });
57        /* Pour bConversion */
58        bConversion.addActionListener(new ActionListener(){
59           public void actionPerformed(ActionEvent e) {
60              if(tfFrancs.getText().length()==0){
61                 valeurEuros= Float.parseFloat(tfEuros.getText());
62                 calculFF();
63        tfFrancs.setText(Float.toString(valeurFrancs));
64              }
65              else{
66                 valeurFrancs = Float.parseFloat(tfFrancs.getText
                      ());
67                 calculEuros();
68        tfEuros.setText(Float.toString(valeurEuros));
69              }
70           }
71        });
72
73     }
74
75     private void calculEuros(){
76        valeurEuros = valeurFrancs/TAUX;
77     }
78
79     private void calculFF(){
```

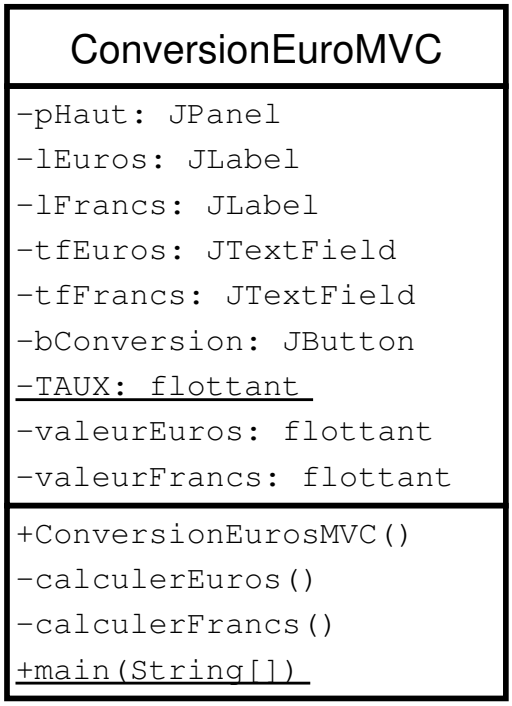


FIG. 6.2 – Le diagramme UML de la classe `ConversionEuroMVC`

```
80          valeurFrancs = valeurEuros*TAUX;
81      }
82
83
84      public static void main(String[] args) {
85          JFrame fen = new JFrame("Conversion FF/E");
86          fen.setContentPane(new ConversionEuroMVC());
87          fen.pack();
88          fen.setVisible(true);
89          fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
90      }
91  }
```

Cette implémentation est fonctionnelle mais peu évolutive. En effet, si les méthodes de conversion changent on doit reprendre la classe complètement. Toutefois, pour des applications simples cette solution est rapide de mise en oeuvre.

6.3 Séparation des éléments vue, contrôleur et modèle

L'implémentation la plus stricte du paradigme MVC consiste à séparer les éléments dans différentes classes. Pour assurer la sécurité de l'application, les données doivent être stockées sous la forme de membres `private` et accédées à l'aide de méthodes publiques (comme le couple `get/set`). En pratique, pour les cas les plus simple, on peut alléger le code sans prendre trop de risque

et placer les membres en `protected`. C'est cette approche qui va être utilisée dans la suite de ces explications pour alléger les codes sources.

6.3.1 La classe modèle

On place tous les éléments liés à la modélisation du problème (dans notre cas, la conversion FF/E) dans une classe. Les variables membres sont déclarées `private` et accédées via des méthodes de la forme `get/set`. Les calculs étant simples, ils sont effectués dès qu'une valeur change. Pour des problèmes plus lourds, il faut faire appelle à des relations observeur/observé[1] entre la partie modèle et le partie vue.

```
1  public class Convertisseur {
2     static final float TAUX = (float) 6.55957;
3     private float valeurEuro;
4     private float valeurFranc;
5
6     public void setValeurEuro(float aFloat) {
7        valeurEuro = aFloat;
8        valeurFranc = valeurEuro * TAUX;
9     }
10
11    public void setValeurFranc(float aFloat) {
12       float temp;
13       valeurFranc = aFloat;
14       valeurEuro = valeurFranc/TAUX;
15    }
16
17    public float getValeurFranc() {
18       return valeurFranc;
19    }
20
21    public float getValeurEuro() {
22       return valeurEuro;
23    }
24 }
```

6.3.2 Un seul contrôleur

La classe contrôleur

La classe contrôleur implémente les deux interfaces FocusListener (pour tfFrancs et tfEuros) et ActionListener (pour bConversion). La classe modèle est instanciée dans la classe contrôleur (ligne 7), elle aurait aussi pu être instanciée dans la classe vue et passée en paramètre dans le constructeur du contrôleur. Les deux implémentations sont très proches. A part le constructeur, la classe ne comporte que les méthodes liées aux évènements. Pour l'événement `focusGained` la source est identifiée avant

[1] Le modèle héritera de la classe `Observable` et la vue implémentera l'interface `Observer`.

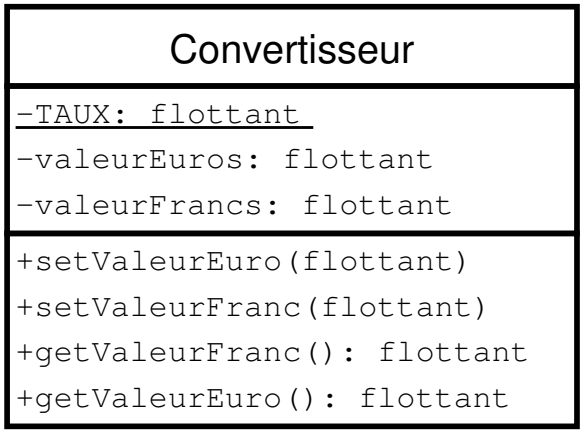


FIG. 6.3 – Le diagramme UML de la classe `Convertisseur`

de modifier les éléments de la vue[2].

```
1  public class ControleurConversionEuro implements FocusListener,
       ActionListener {
2      /* Le Modele Convertisseur */
3      private Convertisseur leModele;
4      private ConversionEuroV laVue;
5
6      public ControleurConversionEuro(ConversionEuroV V){
7          leConvertisseur = new Convertisseur();
8          laVue=V;
9      }
10
11     public void focusGained(FocusEvent e) {
12         /* tfEuros est la source de lé'évnement */
13         if (e.getComponent()==laVue.tfEuros){
14             laVue.tfEuros.setText("");
15             laVue.tfFrancs.setText("");
16         }
17         /* tfFrancs est la source de lé'évnement */
18         if (e.getComponent()==laVue.tfFrancs){
19             laVue.tfEuros.setText("");
20             laVue.tfFrancs.setText("");
21         }
22     }
23
24     public void focusLost(FocusEvent e) {
25     }
26
```

[2]L'exemple est purement pédagogique puisque les lignes 14-15 et 19-20 sont identiques, l'identification de la source n'est pas nécessaire dans ce cas.

```
27     public void actionPerformed(ActionEvent e) {
28        if(laVue.tfFrancs.getText().length()==0){
29           leModele.setValeurEuro(Float.parseFloat(laVue.tfEuros.
                getText()));
30           laVue.tfFrancs.setText(Float.toString(leModele.
                getValeurFranc()));
31        }
32        else{
33           leModele.setValeurFranc(Float.parseFloat(laVue.tfFrancs
                .getText()));
34           laVue.tfEuros.setText(Float.toString(leModele.
                getValeurEuro()));
35        }
36     }
37  }
```

La classe vue

Les deux champs de saisie ne sont plus `private` mais `protected` pour que la classe contrôleur puisse y accéder. Un contrôleur associé à la vue est crée en ligne 14. La gestion des événements est très simple puisqu'elle consiste simplement à associer les événements à surveiller au contrôleur (lignes 35 à 39).

```
1  public class ConversionEuroV extends JPanel {
2
3     /* Le panneau du haut */
4     private JPanel pHaut;
5     private JLabel lEuros, lFrancs;
6     protected JTextField tfEuros, tfFrancs;
7     /* Le bouton de conversion */
8     private JButton bConversion;
9     /* Le Controleur */
10    private ControleurConversionEuro leControleur;
11
12    public ConversionEuroV() {
13       super();
14       leControleur = new ControleurConversionEuro(this);
15       /* Partie ésuprieure */
16       lEuros = new JLabel("Valeur en E");
17       lFrancs = new JLabel("Valeur en FF");
18       tfEuros = new JTextField("",10);
19       tfFrancs = new JTextField("",10);
20       pHaut = new JPanel();
21       pHaut.setLayout(new GridLayout(2,0));
22       pHaut.add(lEuros);
23       pHaut.add(tfEuros);
24       pHaut.add(lFrancs);
```

```
25          pHaut.add(tfFrancs);
26          /* Inclusion du bouton et de la partie ésuprieure */
27          bConversion = new JButton("Convertir");
28          this.setLayout(new BorderLayout());
29          this.add(pHaut, BorderLayout.CENTER);
30          this.add(bConversion, BorderLayout.SOUTH);
31          /*
32           * GESTION DES EVENEMENTS
33           */
34          /* Pour tfEuros */
35          tfEuros.addFocusListener(leControleur);
36          /* Pour tfFrancs */
37          tfFrancs.addFocusListener(leControleur);
38          /* Pour bConversion */
39          bConversion.addActionListener(leControleur);
40      }
41
42      public static void main(String[] args) {
43          JFrame fen = new JFrame("Conversion FF/E");
44          fen.setContentPane(new ConversionEuroV());
45          fen.pack();
46          fen.setVisible(true);
47          fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
48      }
49  }
```

Cette approche sépare clairement les trois éléments du modèle MVC en trois classes (la figure 6.4 présente le diagramme UML de l'application). L'ajout d'une fonctionnalité est facile, la seule limite étant la nécessité d'identifier la source des événements dans le contrôleur ce qui conduit assez vite à des « batteries de tests ».

6.3.3 Autant de contrôleurs que d'événements possible

Les différents contrôleurs doivent se référer à la même vue et au même modèle. La solution la plus simple consiste à leur passer la vue et le modèle en paramètre lors de la construction.

Les contrôleurs

Les contrôleurs ont deux membres `laVue` et `leModele`, ils sont initialisés par les constructeurs. Les contrôleurs reprennent de manière séparée les éléments du contrôleur unique présenté ci-dessus. Les champs de texte n'ont pas d'incidence sur le modèle, celui-ci n'est pas référencé dans leurs contrôleurs. Le contrôleur du champ de texte tfEuros :

```
1  public class ControleurTfEuro implements FocusListener {
2
3      private ConversionEuroV laVue;
4
5      public ControleurTfEuro(ConversionEuroV V) {
```

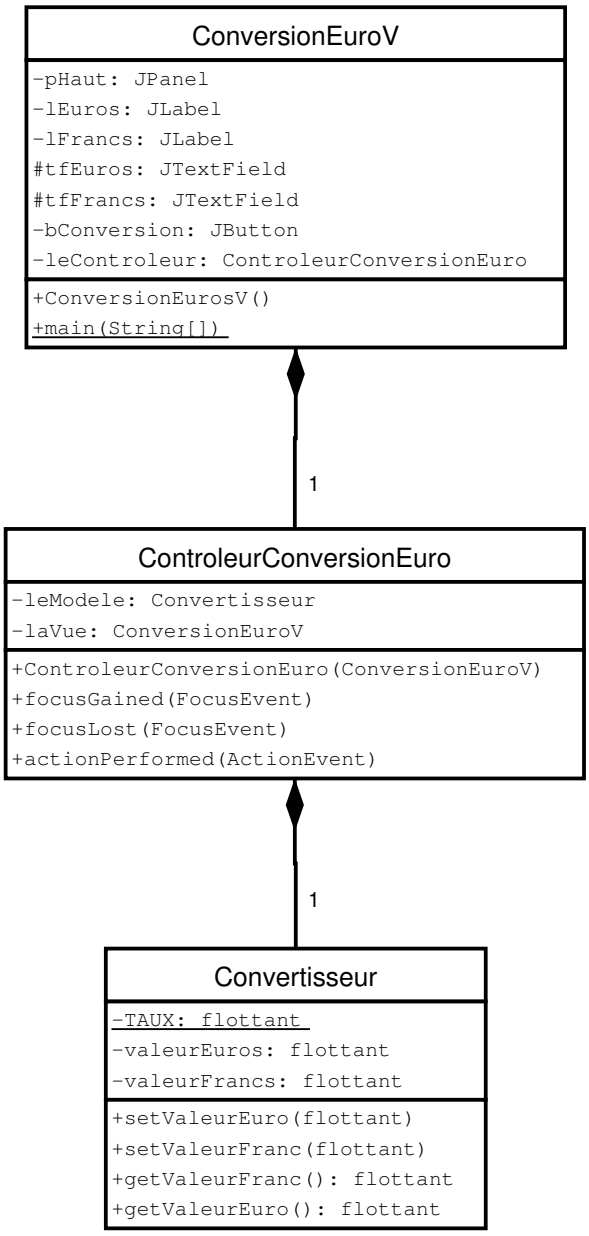


FIG. 6.4 – Le diagramme UML de l'application en décomposant les trois éléments

```
6         laVue = V;
7      }
8
9      public void focusGained(FocusEvent e) {
10        laVue.tfEuros.setText("");
11        laVue.tfFrancs.setText("");
12     }
13
14     public void focusLost(FocusEvent e) {
15     }
16
17 }
```

Le contrôleur du champ de texte tfFrancs :

```
1  public class ControleurTfFranc implements FocusListener {
2
3     private ConversionEuroV laVue;
4
5     public ControleurTfFranc(ConversionEuroV V) {
6        laVue = V;
7     }
8
9     public void focusGained(FocusEvent e) {
10       laVue.tfEuros.setText("");
11       laVue.tfFrancs.setText("");
12    }
13
14    public void focusLost(FocusEvent e) {
15    }
16
17 }
```

Le contrôleur du bouton bConversion :

```
1  public class ControleurBConversion implements ActionListener {
2     private ConversionEuroV laVue;
3     private Convertisseur leModele;
4
5     public ControleurBConversion(ConversionEuroV V, Convertisseur
          M)  {
6        laVue = V;
7        leModele = M;
8     }
9
10    public void actionPerformed(ActionEvent e) {
11       if(laVue.tfFrancs.getText().length()==0){
12             leModele.setValeurEuro(Float.parseFloat(
13 laVue.tfEuros.getText()));
```

```
14              laVue.tfFrancs.setText(Float.toString(
15 leModele.getValeurFranc()));
16       }else{
17          leModele.setValeurFranc(Float.parseFloat( laVue.
               tfFrancs.getText()));
18          laVue.tfEuros.setText(Float.toString(  leModele.
               getValeurEuro()));
19       }
20    }
21 }
```

La vue

La vue est peu modifiée par rapport à l'implémentation précédente. Le modèle est crée dans la vue (ligne 14). Les contrôleurs sont instanciés lors de l'appel à la méthode d'enregistrement. En effet, aucun lien n'est nécessaire après la construction puisque la vue et le modèle sont passés en paramètres aux constructeurs.

```
1  public class ConversionEuroV extends JPanel {
2
3     /* Le panneau du haut */
4     private JPanel pHaut;
5     private JLabel lEuros, lFrancs;
6     protected JTextField tfEuros, tfFrancs;
7     /* Le bouton de conversion */
8     private JButton bConversion;
9     /* Le Modele */
10    private Convertisseur leModele;
11
12    public ConversionEuroV() {
13       super();
14       leModele = new Convertisseur();
15
16       /* Partie ésuprieure */
17       lEuros = new JLabel("Valeur en E");
18       lFrancs = new JLabel("Valeur en FF");
19       tfEuros = new JTextField("",10);
20       tfFrancs = new JTextField("",10);
21       pHaut = new JPanel();
22       pHaut.setLayout(new GridLayout(2,0));
23       pHaut.add(lEuros);
24       pHaut.add(tfEuros);
25       pHaut.add(lFrancs);
26       pHaut.add(tfFrancs);
27       /* Inclusion du bouton et de la partie ésuprieure */
28       bConversion = new JButton("Convertir");
29       this.setLayout(new BorderLayout());
```

```
30          this.add(pHaut, BorderLayout.CENTER);
31          this.add(bConversion, BorderLayout.SOUTH);
32          /*
33           * GESTION DES EVENEMENTS
34           */
35          /* Pour tfEuros */
36          tfEuros.addFocusListener(new ControleurTfEuro(this,
               leModele));
37          /* Pour tfFrancs */
38          tfFrancs.addFocusListener(new ControleurTfFranc(this,
               leModele));
39          /* Pour bConversion */
40          bConversion.addActionListener(new ControleurBConversion(
               this, leModele));
41      }
42
43      public static void main(String[] args) {
44          JFrame fen = new JFrame("Conversion FF/E");
45          fen.setContentPane(new ConversionEuroV());
46          fen.pack();
47          fen.setVisible(true);
48          fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
49      }
50  }
```

L'utilisation d'une classe par contrôleur permet une organisation très structurée de l'application (fig 6.4. Le seul inconvénient est lié au nombre de classes (et donc de fichiers) qui croit très rapidement.

6.4 Les éléments vue et contrôleur sont groupés, l'élément modèle est autonome

Une approche souvent retenue, notamment par les outils de développement (IDE) et la constitution d'une classe qui regroupe les éléments vue et contrôleur et une classe qui comprend le modèle. On reprend la classe convertisseur de l'exemple précédent. Un objet de cette classe est instancié dans la classe graphique en ligne 15. On combine les avantages des deux méthodes vues précédemment. Les gestionnaires sont à l'intérieur de la classe graphique, ainsi on évite une multiplication des fichiers. Par contre, la partie modèle peut être modifiée sans toucher à l'interface puisque l'on utilise des méthodes d'accès (`setValeurEuro` et `setValeurFranc`).

```
1   public class ConversionEuroVC extends JPanel {
2
3       /* Le panneau du haut */
4       private JPanel pHaut;
5       private JLabel lEuros, lFrancs;
6       private JTextField tfEuros, tfFrancs;
7       /* Le bouton de conversion */
8       private JButton bConversion;
```

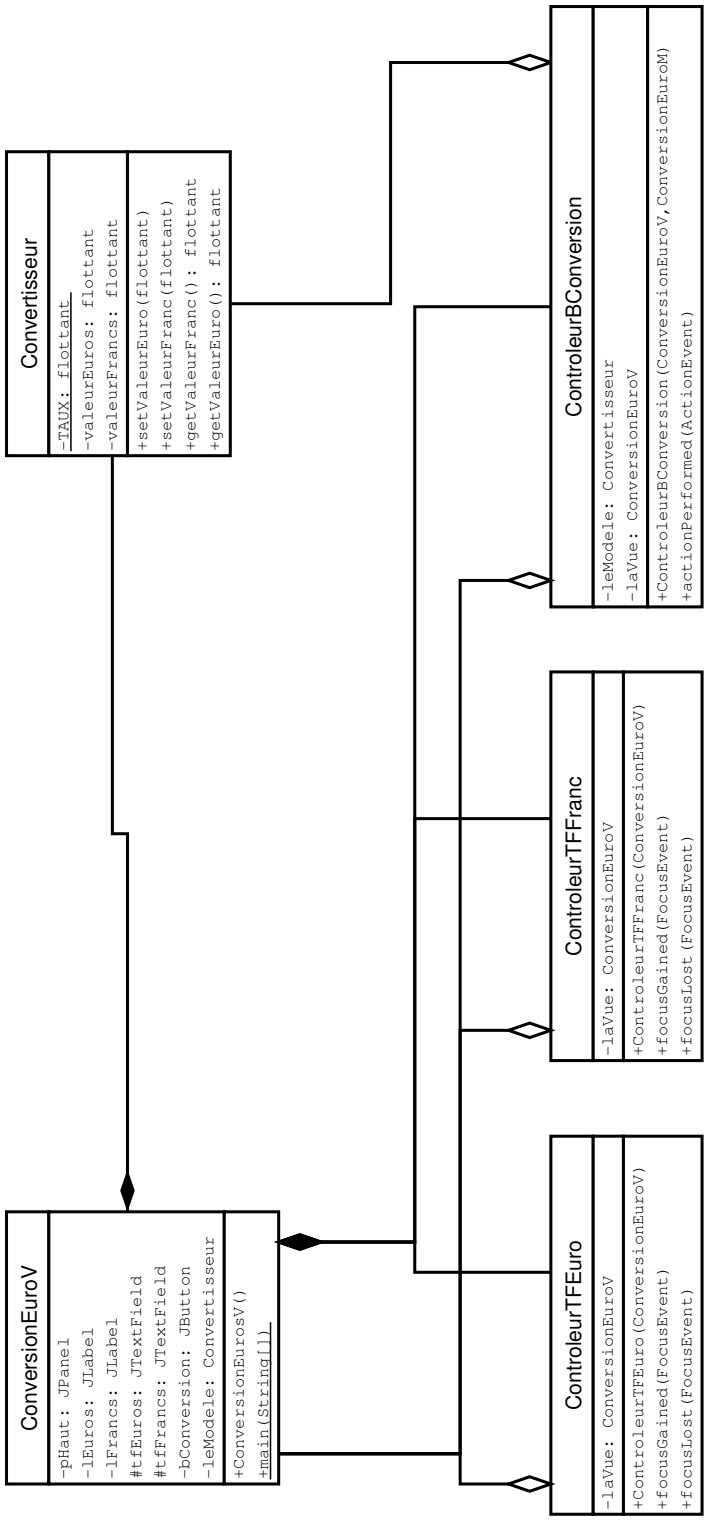


FIG. 6.5 – Le diagramme UML de l'application en décomposant les trois éléments avec un contrôleur par composant

```
9     /* Le Modele Convertisseur */
10    private Convertisseur leModele;
11
12
13    public ConversionEuroVC() {
14       super();
15       leModele = new Convertisseur();
16       /* Partie ésuprieure */
17       lEuros = new JLabel("Valeur en E");
18       lFrancs = new JLabel("Valeur en FF");
19       tfEuros = new JTextField("",10);
20       tfFrancs = new JTextField("",10);
21       pHaut = new JPanel();
22       pHaut.setLayout(new GridLayout(2,0));
23       pHaut.add(lEuros);
24       pHaut.add(tfEuros);
25       pHaut.add(lFrancs);
26       pHaut.add(tfFrancs);
27       /* Inclusion du bouton et de la partie ésuprieure */
28       bConversion = new JButton("Convertir");
29       this.setLayout(new BorderLayout());
30       this.add(pHaut, BorderLayout.CENTER);
31       this.add(bConversion, BorderLayout.SOUTH);
32       /*
33        * GESTION DES EVENEMENTS
34        */
35       /* Pour tfEuros */
36       tfEuros.addFocusListener(new FocusListener(){
37          public void focusGained(FocusEvent e) {
38             tfEuros.setText("");
39             tfFrancs.setText("");
40          }
41          public void focusLost(FocusEvent e) {
42          }
43       });
44       /* Pour tfFrancs */
45       tfFrancs.addFocusListener(new FocusListener(){
46          public void focusGained(FocusEvent e) {
47             tfEuros.setText("");
48             tfFrancs.setText("");
49          }
50          public void focusLost(FocusEvent e) {
51          }
52       });
53       /* Pour bConversion */
54       bConversion.addActionListener(new ActionListener(){
55          public void actionPerformed(ActionEvent e) {
```

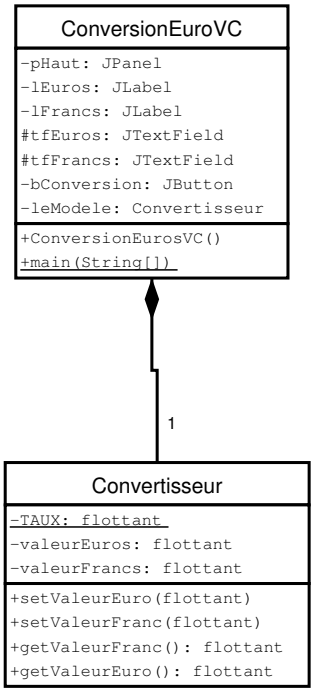


FIG. 6.6 – Le diagramme UML de l'application en regroupant la vue et le contrôleur

```
56              if(tfFrancs.getText().length()==0){
57                 leModele.setValeurEuro(Float.parseFloat(tfEuros.
                      getText()));
58              tfFrancs.setText(Float.toString(leModele .
                   getValeurFranc()));
59              }else{
60                 leModele.setValeurFranc(Float.parseFloat(tfFrancs
                      .getText()));
61                 tfEuros.setText(Float.toString(leModele .
                      getValeurEuro()));                          }
62              }
63        });
64     }
65
66     public static void main(String[] args) {
67        JFrame fen = new JFrame("Conversion FF/E");
68        fen.setContentPane(new ConversionEuroVC());
69        fen.pack();
70        fen.setVisible(true);
71        fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
72     }
73  }
```

La figure 6.6 présente le diagramme UML de cette implémentation. Il ne subsiste que deux classes.

Index