Chapitre 3 : Programmation des interfaces graphiques

Table des matières

I- Présentation de AWT et SWING

1. AWT et Swing

Les premières versions de Java utilisaient des composants graphiques définis dans le paquetage java.awt : on pouvait y trouver les classes Button, Frame, Panel, CheckBox... Ces composants étaient des composants lourds, reprenant du code natif, construisant des composants natifs des différentes plateformes. De nouveaux composants plus légers ont été définis dans le paquetage javax.swing et sont maintenant utilisés à la place des premiers ; ils se nomment JButton, JFrame, JPanel, JCheckBox...

2. Concepts de bases

La construction d'interfaces graphiques est basée sur quatre éléments principaux.

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.

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. Les conteneurs sont des descendants de la classe Container.

2.3 Les gestionnaires de placement

Le placement des composants est souvent confié à un gestionnaire de placement.

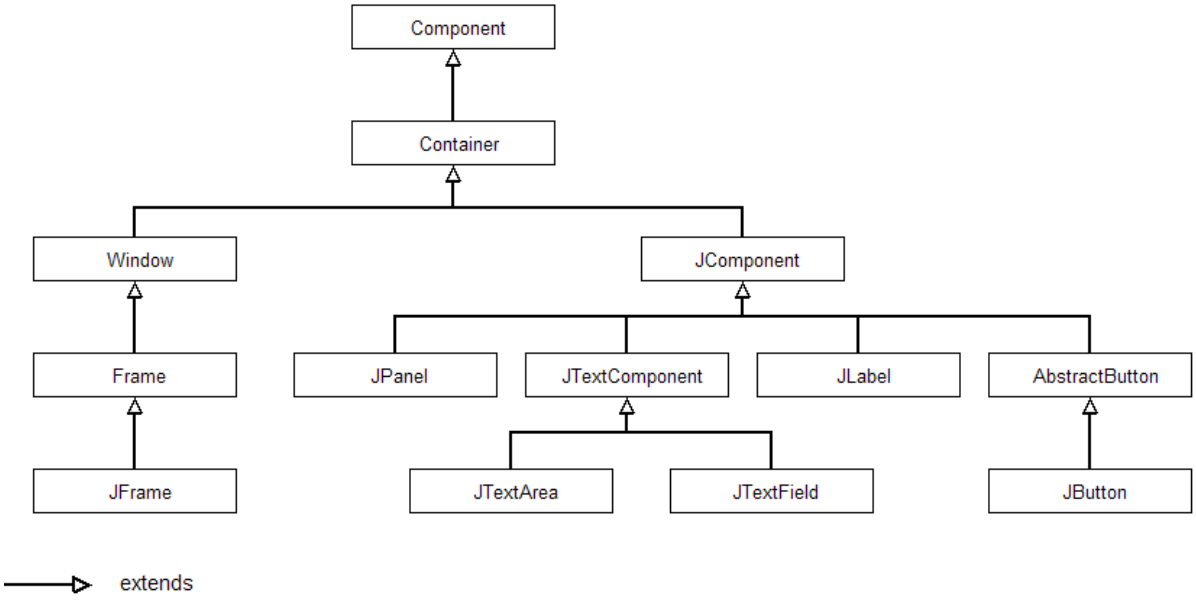
Le palcement absolu des composants est déconseillé, on préfère utiliser un positionnement relatif. Swing propose de nombreux gestionnaires de placement pour construire des applications dont la probabilité est assurée.

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.

II. Les principaux composants de Swing

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 àtous les éléments.

2. 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.

```
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);
}
}
```

Accès au contentPane **pour ajouter des composants**

Les composants (boutons, labels,. . .) sont placés sur le contentPane.. 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);
```

3. Les composants atomiques

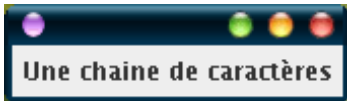
Les composants atomiques sont tous les composants élémentaire de Swing. Ce sont les boutons, les labels, les menus,. . .

3.1 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 chîne de caractères");
//Ce label est ajouté à un conteneur avec la méthode add.
JFrame fen = new JFrame ();
JLabel unLabel = new JLabel (" Une chaine de ècaractres");

fen . getContentPane().add ( unLabel );
fen . setVisible( true);
```



3.2 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.

Exemple :

Pour ajouter un bouton :

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

3.3 Les cases àcocher : JCheckBox

Les cases à cocher permettent de matérialiser des choix binaires.
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).

JCheckBox casePasCochee = new JCheckBox("Une case àcocher ");
JCheckBox caseCochee = new JCheckBox(" Une case écoche", true);

3.4 Les boutons radio : JRadioButton

Les boutons radio JRadioButton sont des boutons à choix exclusif8, il permettent de choisir un (et un seul) élément parmi un ensemble.
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);

3.5 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.

Exemple :
L'exemple qui suit présente un champ de saisie simple:

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

Utilisation d'un champ de saisie

III- 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.

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 par défaut et lui est associé.
Gestionnaire par défaut pour les principaux conteneurs :
JPanel: FlowLayout
JFrame: BorderLayout

2. Le positionnement absolu

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 àl'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);
```

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 àgauche, 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.

Exemple :
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 JFrame {
        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

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.

Le gestionnaire FlowLayout redispose les composants après une réduction de la fenêtre

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.
Le 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.

```
public class TestGridLayout extend JFrame {
        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"));
                }

.....
```

5. Le gestionnaire BorderLayout

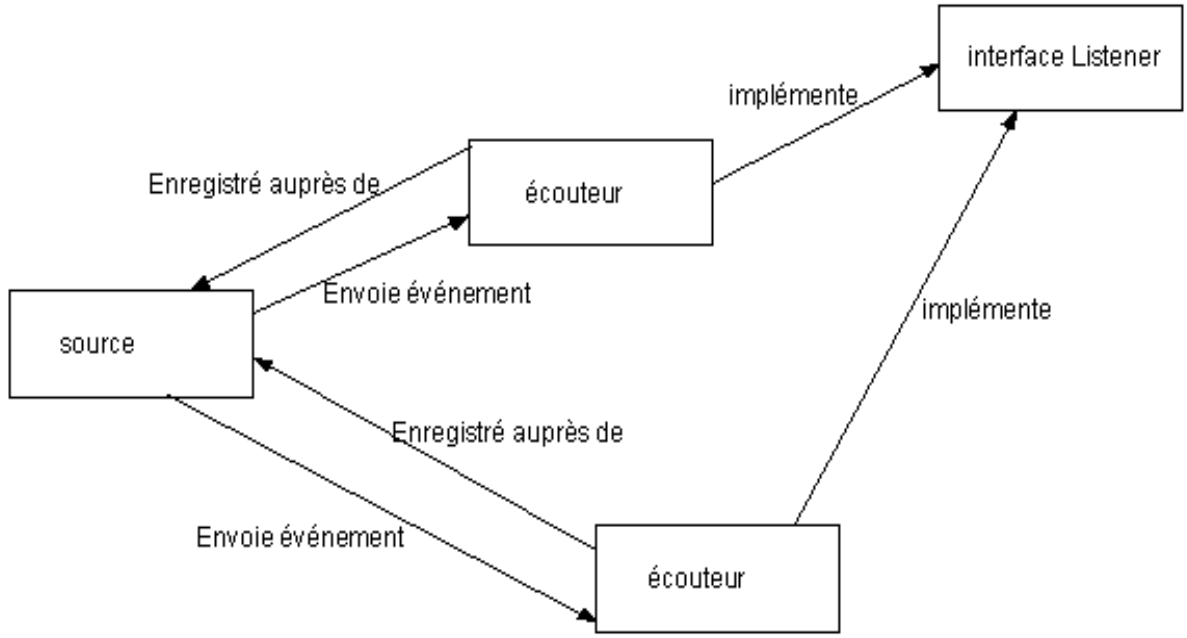
Le gestionnaire BorderLayout définit cinq zones dans le conteneur : un zone centrale (CENTER) et quatre zones périphériques (EAST, WEST, NORTH, SOUTH)

```
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 );
}
...
```

IV. Gestion des événements

1. Principes de la gestion d'événements

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(écouteur). L'auditeur(écouteur) 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) qui doit être implémentée pour gérer l'évènement



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,. .

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.

– Les événements "de base"

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.
...	...

Quelques événements sémantiques

Les interfaces suivantes permettent d'écouter ces événements :

1. ActionListener
2. AjustementListener
3. FocusListener
4. ItemListener
5. KeyListener
6. MouseListener
7. MouseMotionListener

8. MouseWheelListener
9. WindowsListener
10. WindowsFocusListener
11. WindowsStateListener

Ecouteur (adaptateur)	**Méthodes écouteur**	**Type évènement**	**Méthodes événement**	**Composants concernés**
MouseListener (MouseAdapter)	mouseClicked mousePressed mouseReleased mouseEntered mouseExited	MouseEvent	getClickCount getComponent getModifiers getSource getX getY getPoint isAltDown isAltGraphDown isControlDown isMetaDown isPopupTrigger isShiftDown	Componenent
KeyListener (KeyAdapter)	keyPressed keyReleased keyTyped	keyEvent	getComponent getSource getKeyChar getKeyCode getKeyModifiers- Text getKeyText getModifiers isAltDown isAltGraphDown isControlDown isShiftDown	Component

			isMetaDown isActionKey	
ActionListener	actionPerformed	ActionEvent	getSource getActionCommand getModifiers	*JButton*, *JCheckBox*, *JradioButton*, JMenu, JMenuItem, … JTextField

Dans la classe Component on trouve les méthodes suivantes, pour enregistrer un écouteur :

void addFocusListener (FocusListener écouteur)

void addKeyListener (KeyListener écouteur)

void addMouseListener (MouseListener écouteur)

void addMouseMotionListener (MouseMotionListener écouteur)

3. Gestion des événements par la classe graphique

L'approche la plus simple de mise en œuvre 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 JFrame 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 JFrame implements ActionListener{
...
JButton leBouton ;
...
public Test () {
...
leBouton . addActionListener( this);
...
}
public void actionPerformed( ActionEvent e)
{// Partie utile
}
```

...

De même, pour implémenter plusieurs interfaces :

```
public class Tests extends JFrame  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 JFrame 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
}
}
...
```

4. Gestion des événements par des classes dédiées

L'approche opposée consiste à construire une classe par écouteur d'évènement.
Exemple :

Deux évènements sont présents dans l'exemple, un ActionEvent généré par le Bouton1 et un ActionEvent généré par le Bouton2. Le premier auditeur(écouteur) écoute les évènements du Bouton1 :

```
public class ListenerBouton1 implements ActionListener {
...
public void actionPerformed( ActionEvent e) {
// Partie utile
}
...
}
De même, le second auditeur(écouteur) écoute les évènements du Bouton2 :
public class ListenerBouton2 implements ActionListener {
...
public void actionPerformed( ActionEvent e) {
// Partie utile
}
...
}
```

Les deux auditeur(écouteur)s sont instanciés dans les méthodes d'enregistrement :

```
public class TestClassesExternes extends JPanel {
...
JButton leBouton1 , leBouton2;
...
public TestClassesExternes() {
...
leBouton1 = new JButton (" Bouton 1");
leBouton2 = new JButton (" Bouton 2");
...
leBouton1. addActionListener(new ListenerBouton1());
leBouton2. addActionListener(new ListenerBouton2());
...
}
```

Les classes auditeur(écouteur)s n'ont pas directement accès aux variables membres de la classe visuelle. Ils est donc nécessaire de créer des références entre ces deux éléments.

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 JFrame {
...
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
}
}
}
```

6. Gestion des événements par des classes membres internes anonymes

Présentation des classes anonymes

Ce sont en fait des classes locales directement déclarées et instanciées au sein d'une même expression de code. Elles peuvent directement étendre une classe ou implémenter une interface. Elles peuvent spécifier les arguments d'un des constructeurs de la super-classe mais ne peuvent pas en définir.

Dans le cas des auditeur(écouteur)s, 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
}
});
}
```

7. Gestion des événements par des classes d'adaptations (XxxAdapter)

Les interfaces auditeur(écouteur)s les plus utilisées comportent de nombreuses méthodes[5]. Lors de la conception d'un auditeur(écouteur) on doit donc déclarer toutes ces méthodes mêmes si elles ne sont pas utilises. 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 qui contiennent toutes les méthodes de l'interface auditeur(écouteur) d'événement.

Voici les classes adaptateur les plus souvent utilisées :

FocusAdapter
KeyAdapter
MouseAdapter
MouseMotionAdapter
WindowAdapter

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 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(écouteur) 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
}
}
```

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 auditeur(écouteur)s. Par exemple, un auditeur(écouteur) 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(écouteur) 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 auditeur(écouteur)s 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.