SWING
LA SYNTHÈSE

Développement des interfaces graphiques en Java

01 INFORMATIQUE

Valérie Berthié
Jean-Baptiste Briaud

Préface de Gilles Clavel

DUNOD

Préface

Aujourd'hui, l'interaction de l'informatique avec notre vie courante se fait la plupart du temps à travers une interface graphique.

Intuitive et banalisée, cette interface nous est familière. Mais, pour le programmeur qui l'a conçue et développée, elle a représenté un travail considérable. La complexité de ce type d'interaction homme-machine est d'ailleurs à l'origine d'une profonde évolution de la programmation : celle de la technologie objet. Cependant, malgré l'apport de l'objet, l'interface graphique est très longtemps restée propriétaire et non portable : les langages qui permettaient sa programmation ne le faisaient qu'à travers des additions, sous la forme de bibliothèques de classes toujours dépendantes de la plate-forme utilisée.

Java a mis fin à ces restrictions en proposant des API (*Application Programming Interfaces*) intégrées au langage et indépendantes de la plate-forme d'exécution. Le succès du langage est à la mesure de l'apport de ces API. Parmi elles, celle qui concerne l'interface graphique est l'une des plus riches et des plus utilisées : dans sa version la plus évoluée, on l'appelle Swing.

Swing la synthèse est un ouvrage qui a l'ambition de présenter cette interface à un lecteur qui souhaite la comprendre, être capable de l'utiliser et savoir choisir la meilleure technique.

Décrire une API est toujours un exercice périlleux : il est difficile d'éviter la présentation fastidieuse d'un catalogue de recettes. Et il est encore moins aisé de distinguer, face à la richesse de l'API, entre ce qu'il est indispensable de montrer et ce qui constitue, pour le lecteur, une fastidieuse énumération de détails. Le livre de Valérie Berthié et Jean-Baptiste Briaud a su éviter ces écueils. Dès le premier chapitre, le style alerte et vivant nous explique l'indispensable, avec le recul qui convient. Les enjeux de la programmation d'une interface graphique sont présentés et illustrés à travers des exemples parlants. Au fil des chapitres, les

aspects majeurs de la programmation avec Swing (layouts, événements, composants) sont introduits avec la même approche : partir d'un exemple, faire comprendre le besoin et la solution, récapituler.

Bien entendu, ce livre suppose un prérequis, celui de la connaissance du langage Java et des principes de la programmation objet. Mais la pédagogie des exposés et la pertinence des exemples facilitent la compréhension et limitent le périmètre du prérequis. Ce qui devrait être connu mais n'est pas évident est la plupart du temps rappelé et, pour celui qui n'est pas familiarisé avec UML, une annexe précise toutes les notations utilisées.

Valérie Berthié et Jean-Baptiste Briaud ont fait le pari qu'il était possible de faire une synthèse claire, concise mais suffisamment complète d'une API aussi riche que celle de Swing. Je suis persuadé que les lecteurs de ce livre confirmeront le succès : lire *Swing la synthèse* est un excellent moyen de comprendre Swing et de savoir démarrer avec pertinence le développement d'une interface homme-machine.

Gilles Clavel
Paris, juillet 2005

Table des matières

Avant-propos

Il y a quelque temps, nous avons été amenés à réaliser un projet Java conséquent avec de fortes contraintes ergonomiques. Le choix de Swing s'imposait. Nous avons trouvé difficile de développer une vue d'ensemble, de prendre du recul, afin d'entamer la conception de l'interface graphique. Il existait de nombreux et volumineux ouvrages de référence, mais ils ne sont devenus indispensables qu'après avoir acquis ce recul par nous-mêmes.

Nous avons regretté l'absence d'ouvrages qui nous auraient exposé les principes de Swing afin de nous orienter parmi les nombreuses classes que comporte cette libraire. Notre ambition en réalisant cet ouvrage est de vous offrir cette vue d'ensemble, de vous donner les clés qui vous permettront d'appréhender tout nouveau composant.

Cet ouvrage s'adresse à des concepteurs/développeurs Java désireux d'acquérir rapidement une vision d'ensemble de Swing.

Nous étudierons en premier lieu les composants, exposant leurs capacités communes ainsi que leurs particularités. Nous ne chercherons pas à être exhaustifs, cet ouvrage n'est pas un dictionnaire technique. Nous nous efforcerons de mettre en valeur les concepts fondateurs de la librairie Swing.

Nous verrons ensuite comment les *layouts* résolvent le problème du positionnement des composants dans des contextes graphiques dynamiques, comme par exemple quand l'utilisateur modifie la taille d'une fenêtre.

Nous évoquerons la programmation événementielle qui permet de réagir à des interactions de l'utilisateur.

Nous reviendrons plus en détail sur certains composants très riches fonctionnellement, mais plus complexes à mettre en œuvre.

Le traitement du texte fait l'objet dans Swing de composants spécialisés que nous approfondirons.

Une partie plus courte est réservée au concept ergonomique du *drag and drop*, incontournable dans les interfaces graphiques actuelles.

Des problématiques habituelles lors des développements d'interfaces sont ensuite abordées : comment éviter les temps d'attente pour l'utilisateur grâce au parallélisme, les principes de base de la construction d'un *framework*, et l'inévitable casse-tête des tests.

Enfin, nous exposerons les apports des dernières versions du JDK concernant Swing.

L'annexe A présente quelques règles du formalisme UML sur lequel nous nous sommes appuyés tout au long de l'ouvrage.

Nous avons choisi de mettre en place un exemple assez conséquent qui est construit étape par étape, au fur et à mesure des connaissances acquises dans chaque chapitre. Le code complet de cette application est récapitulé en annexe B.

Nous souhaitons remercier Gilles Clavel qui nous a offert l'opportunité de réaliser ce livre. Nous remercions également toutes les personnes que nous avons mises à contribution pour relire des chapitres de cet ouvrage, en particulier Emmanuelle Mouthe, Vincent Gallot, Aude Maulny et Chanthy Lim. Nous remercions aussi Amaury Fatus, Brian Swan, l'équipe Valrisk de BNP-Paribas ainsi que Jide software. Nous remercions enfin Christelle Briaud pour son soutien constant et sa patience.

1

Concepts fondamentaux, premiers pas avec Swing

On désigne sous le nom d'interface, ou bien IHM (Interface homme-machine) la partie visible à l'écran d'une application. L'interface est le premier contact qu'a un utilisateur avec une application. Elle constitue dans un premier temps la seule partie perceptible d'un logiciel et c'est elle qui détermine la première impression des utilisateurs. Lorsque l'on connaît l'importance de cette première impression, on comprend qu'il soit nécessaire de lui accorder un intérêt tout particulier.

Une application d'une grande richesse et d'une grande souplesse fonctionnelle peut donc être « gâchée » par une interface mal perçue par l'utilisateur.

Nous allons développer dans cet ouvrage les arcanes d'une librairie Java : Swing.

Swing est une librairie Java fournie avec le JDK (Java Developpement Kit) standard. Historiquement, Swing n'est pas la première librairie fournie par Sun pour construire des interfaces graphiques. La librairie AWT est arrivée la première. Elle est toujours utilisée car Swing elle-même utilise des éléments d'AWT. Swing est donc plus une avancée qu'une remise en cause d'AWT.

1.1 AWT ET SWING, UNE PREMIÈRE APPROCHE

Dès le départ, le langage Java avec le concept de la machine virtuelle s'est voulu portable. Il ne pouvait en être autrement au niveau graphique. La librairie AWT est donc portable, c'est-à-dire que des interfaces développées à l'aide de cette

librairie s'exécutent sans recompilation sur toutes les plates-formes pour lesquelles il existe une machine virtuelle Java.

La librairie AWT utilise cependant des ressources système qu'elle encapsule par des abstractions, de sorte que le développeur ne perçoit pas directement ces ressources propres au système. Cette librairie porte donc bien son nom : Abstract Windowing Toolkit.

Si une interface graphique développée avec AWT présente à l'écran un champ et un bouton, elle s'exécute potentiellement sur différentes plates-formes, mais l'apparence du champ et du bouton est régie par les ressources graphiques du système d'exploitation. De ce fait, les éléments graphiques auront l'apparence que le système d'exploitation leur donne. De plus, il n'est pas possible avec une telle architecture d'utiliser des éléments graphiques spécifiques d'un système particulier car ils ne sauraient être dessinés à l'écran par d'autres systèmes.

C'est en partie pour corriger cet inconvénient que la librairie Swing a été développée. Les éléments graphiques développés à l'aide de Swing sont dessinés par la librairie elle-même : l'affichage n'est pas délégué au système.

Cette nouvelle architecture est beaucoup plus souple, mais est plus consommatrice en temps car les calculs sont exécutés par la machine virtuelle Java et non pas par le système d'exploitation.

Pour illustrer notre propos, un exemple simple a été réalisé avec AWT puis Swing et exécuté sous Linux (figures 1.1 et 1.2), puis sous Windows (figure 1.3).

Regardons tout d'abord le résultat de l'application AWT sous Linux.

Figure 1.1 — Un exemple en AWT sous Linux, avec Enlightment.

Figure 1.2 — L'exemple AWT sous Linux, avec WindowMaker.

Figure 1.3 — L'exemple AWT sous Windows.

Le système d'exploitation Linux délègue la gestion des fenêtres à un gestionnaire de fenêtres entièrement paramétrable, tant en terme d'apparence (thèmes…) que de comportement.

La figure 1 montre notre exemple avec un gestionnaire de fenêtre nommé *Enlightenment*. La figure 2 montre le même exemple avec *Window Maker* comme gestionnaire de fenêtre. Cet exemple « linuxien » nous permet de montrer que certaines parties du dessin des fenêtres restent à la charge du système. Ces zones sont identifiables par comparaison entre les figures 1 et 2 : tout ce qui change entre ces deux fenêtres est dû au gestionnaire de fenêtres, donc ce sont ces parties qui dépendent du système.

Nous pouvons noter des différences sur les éléments graphiques. L'apparence du champ et du bouton varie un peu alors que les couleurs et les ombrages varient beaucoup. Naturellement, les trois exécutions de ce petit programme n'ont pas nécessité de recompilation, Java est portable.

Voici maintenant le même exemple codé avec Swing (figures 1.4 et 1.5).

Figure 1.4 — L'exemple Swing sous Linux avec Enlightment.

Figure 1.5 — L'exemple Swing sous Windows.

Nous pouvons faire les constatations suivantes :

- les différences concernant les zones dessinées par le système d'exploitation sont les mêmes qu'avec AWT ;
- les différences sur les éléments graphiques sont nettement plus faibles qu'avec AWT.

Un élément graphique Swing peut donc avoir la même apparence d'une plate-forme à l'autre. Mieux, le rendu graphique d'un élément peut être paramétré puisqu'il est effectué par Swing. C'est le *Pluggable Look and Feel*. Autre conséquence, un élément graphique peut être défini par Swing sans même que ce composant ait un équivalent au niveau du système d'exploitation.

Tout au long de cet ouvrage, nous allons appliquer les notions présentées à un exemple complet. Cet exemple consiste en une application pour gérer les favoris Internet d'un utilisateur. Voici dans les figures 1.6 à 1.11 quelques variations de *look and feel* de cette application.

Le *look and feel* `metal` est un standard Java. Une application peut avoir la même apparence sur toutes les plates-formes si elle utilise ce *look and feel*. Les autres *look and feel* sont *a priori* destinés chacun à une plate-forme spécifique. Comme la librairie Swing dessine elle-même les éléments graphiques, elle doit embarquer un *look and feel* par plate-forme supportée. Tous ces *look and feel* ne sont cependant pas livrés en standard. Le JDK que nous avons utilisé pour cet exemple n'embarquait que deux *look and feel* spécifiques en plus du standard. Techniquement rien ne s'oppose à choisir un *look and feel* autre que `metal`, Swing l'affichera à l'écran à l'identique quelle que soit la plate-forme. Plus encore, rien n'interdit d'écrire son propre *look and feel*, c'est un travail assez conséquent, mais cela est tout à fait possible.

Figure 1.6 — L'application Signet en *look and feel* metal.

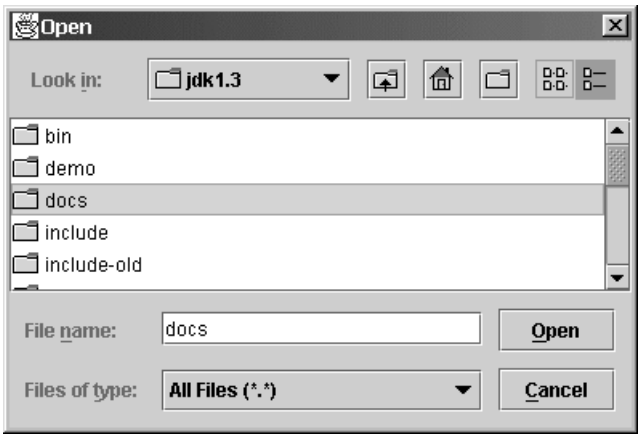


Figure 1.7 — Une boîte de dialogue standard (`JFileChooser`) en *look and feel* metal.

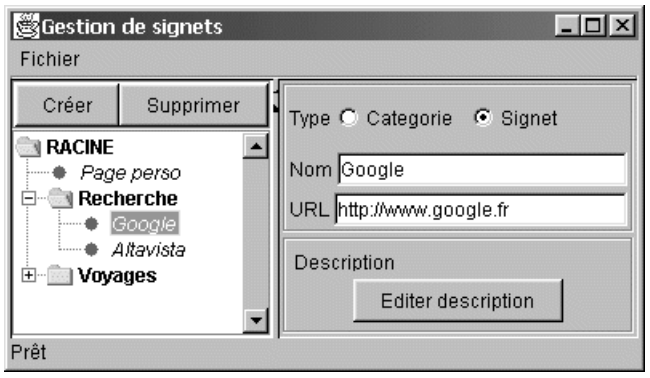


Figure 1.8 — L'application Signet en *look and feel* windows.

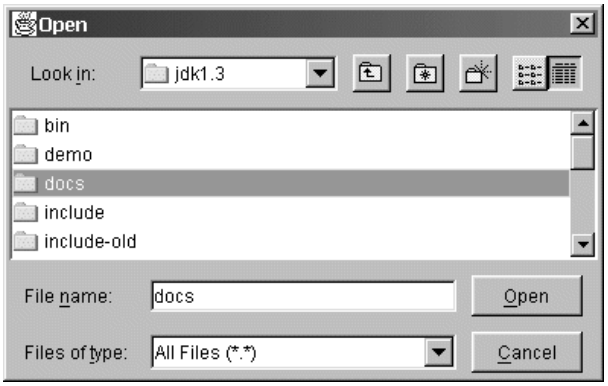


Figure 1.9 — Une boîte de dialogue standard (`JFileChooser`) en *look and feel* windows.

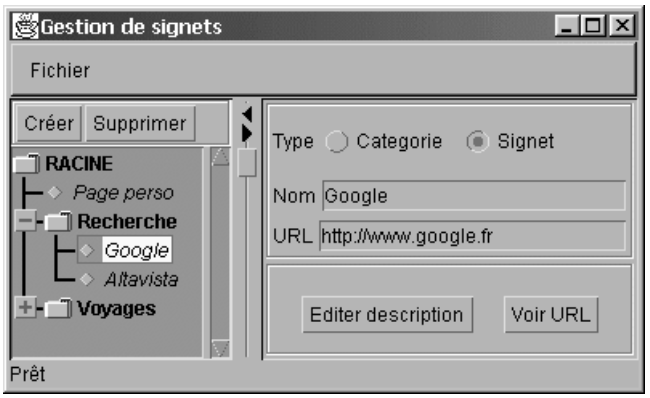


Figure 1.10 — L'application Signet en *look and feel* Motif.

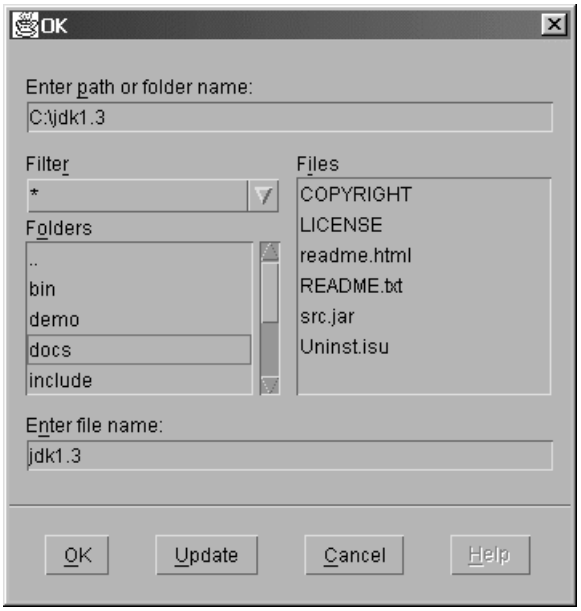


Figure 1.11 — Une boîte de dialogue standard (`JFileChooser`) en *look and feel* Motif.

1.2 SWT ENTRE DANS LA DANSE

IBM est à l'origine d'Eclipse, un environnement de développement Open Source. Cet outil est écrit en Java mais n'est pas dédié à Java en terme de cible. Le produit est construit autour de la notion de plugin, sorte de composant logiciel que l'on peut ajouter ou supprimer d'un noyau de base.

Par exemple, l'éditeur Java dans Eclipse est un plugin. Un développeur peut ajouter ses propres plugins pour permettre d'éditer d'autres types de fichiers.

Eclipse fournit des modules de base qui prennent en charge les fonctionnalités habituelles des environnements de développement (gestion des fichiers, aide en ligne, assistants…) et épargne ainsi au développeur de réécrire ce code générique.

Au cœur de l'architecture d'Eclipse se trouve la gestion des plugins eux-mêmes, ainsi qu'une « librairie » d'interface graphique : SWT pour *Standard Widget Toolkit*. Cette librairie graphique est très différente de Swing d'un point de vue technique mais se place au même niveau conceptuel : elle manipule des composants de même niveau, elle est événementielle…

La principale différence entre Swing et SWT réside dans la façon dont sont dessinés les composants graphiques à l'exécution. Swing, comme nous l'avons dit, n'est pas une librairie native. Le rendu des composants est effectué par la machine virtuelle Java elle-même. La librairie SWT est native. Le rendu des composants est délégué au système d'exploitation, tout comme avec AWT.

Cependant, à la différence d'AWT, les composants proposés par SWT sont de haut niveau et ne se restreignent pas à ceux proposés par HTML.

Du fait que SWT délègue la représentation graphique des composants au système d'exploitation, ces composants ont exactement la même allure que ceux d'une application native.

Les performances d'une application SWT constituent un sujet très controversé dans la comparaison inévitable qui s'exerce entre SWT et Swing.

La portabilité de SWT est moindre que celle de Swing car SWT nécessite une librairie graphique différente pour chaque système. Ce fichier doit être spécifiquement déployé sur le poste client.

La convergence Swing-SWT est d'ores et déjà entamée. Expérimentalement, il est maintenant possible d'exécuter du code Swing au sein d'une application SWT et réciproquement.

1.3 NOTION DE COMPOSANTS

Visuellement, une interface graphique est essentiellement une combinaison d'éléments graphiques comme des boutons, des listes ou des champs, qui permettent à l'utilisateur de « dialoguer » avec l'application : saisie d'informations, navigation dans le logiciel, visualisation de données, etc. Ces éléments graphiques, parfois appelés `widgets`, sont nommés « composants » dans la terminologie Java.

Une interface ne se résume cependant pas aux composants qui la constituent. Beaucoup d'autres éléments, parfois non visibles, sont indispensables pour agencer les composants les uns par rapport aux autres. Les fenêtres ne sont pas des composants mais sont des classes incontournables que nous apprendrons à utiliser dans cette partie.

Ce chapitre donne les éléments nécessaires pour construire des interfaces graphiques « statiques », qui ne réagissent pas aux interactions de l'utilisateur.

1.3.1 Une toute première IHM

Commençons sans attendre plus longtemps notre première interface graphique. Une tradition, dont l'origine se perd dans la nuit des temps, veut en effet que tout apprentissage débute par un exemple affichant un message de bienvenue du type *Hello world* !

Figure 1.12 — Notre premier exemple.

```
import javax.swing.*;

public class Bonjour {
     protected static final String TEXTE = "Bonjour
     ➥tout le monde !";

     public static void main(String args[])     {
        JFrame fenetre = new JFrame();
        JLabel texte = new JLabel(TEXTE);
        fenetre.getContentPane().add(texte);
        fenetre.SetVisible(true);
     }
}
```

Notre classe `Bonjour` importe tout le package `javax.swing`. La librairie, le package `swing` se trouve dans `javax` alors que le package `awt` se trouve dans le package `java`. Cela est dû au fait qu'historiquement Swing est arrivé après AWT. La lettre « x » du package `javax` correspond au mot *extended* en anglais. Le package `swing` est donc vu comme une extension présente par défaut.

Il y a un lien assez évident entre ce que l'on peut voir à l'écran et le code. Nous voyons une fenêtre avec sa barre de titre et ses icônes de manipulation

(réduction, agrandissement et fermeture). Cette fenêtre contient un composant contenant le texte « Bonjour tout le monde ! ».

Dans le code nous retrouvons facilement ces informations :

Une fenêtre est instanciée :

```
JFrame fenetre = new JFrame();
```

Un composant `JLabel` est instancié avec un texte par défaut :

```
JLabel texte = new JLabel(TEXTE);
```

Ce composant, `JLabel`, est positionné dans la fenêtre :

```
fenetre.getContentPane().add(texte);
```

On donne l'ordre à la fenêtre de s'afficher :

```
fenetre.SetVisible(true);
```

Il n'y a que deux intervenants visibles dans le code : la fenêtre et le composant. Nous verrons ultérieurement que d'autres éléments non perceptibles visuellement sont indispensables. Nous évoquerons les fenêtres plus tard, mais il est d'ores et déjà évident qu'elles sont indispensables. Il est en effet impossible d'afficher un composant sans une certaine forme de contenant, rôle que joue ici la fenêtre.

`JLabel` est un des composants les plus simples. Il permet d'afficher un texte non éditable. `JLabel` est une sous-classe lointaine de `Component` et une sous-classe directe de `JComponent`. Quelles sont ces classes dont le nom semble redondant ? S'il y a un couple de classes `JComponent` et `Component`, y aurait-il une classe `Label` puisqu'il y a une classe `JLabel` que nous avons utilisée pour l'exemple ?

1.3.2 AWT et Swing, vision technique

Quelle est la différence entre ces deux classes ? `Component` se trouve dans le package `java.awt` et `JComponent` dans le package `javax.swing`. De plus `JComponent` hérite de `Component`.

Dans les premières versions du JDK, les classes permettant de programmer des interfaces graphiques se trouvaient exclusivement dans le package `java.awt`. Ces classes permettent d'implémenter des interfaces graphiques avec des composants et un modèle événementiel, tout comme Swing.

Une des différences fondamentales entre AWT et Swing est que les composants Swing sont entièrement dessinés en Java.

L'affichage d'un composant « bouton » en Swing fait intervenir des méthodes Java de dessin d'un rectangle, par exemple `drawRect`, puis de remplissage avec une certaine couleur, l'affichage d'une chaîne de caractères au centre du rectangle, et ainsi de suite.

Le problème des classes d'AWT est qu'elles reposent sur des composants natifs et que l'apparence des composants à l'écran varie donc suivant les plates-formes. Par ailleurs, à cause de cette délégation du dessin de l'IHM au système d'exploitation, AWT ne possède que des composants existant sur les principales plates-formes. C'est pour cela que les composants d'AWT sont à peine plus riches que ceux des interfaces HTML.

Les composants Swing ont une indépendance totale vis-à-vis du système. Grâce à cette caractéristique, il est possible de leur affecter une apparence particulière, qui peut rester la même quelle que soit la plate-forme.

Swing est donc un ensemble de packages qui reprend les principes développés dans les classes du package `java.awt`, mais en étendant leur comportement. De nouvelles classes ont aussi été ajoutées.

C'est le cas de `JComponent` qui hérite de `Component`.

Le diagramme de la figure 1.13 montre que certaines classes peuvent avoir leur équivalent AWT.

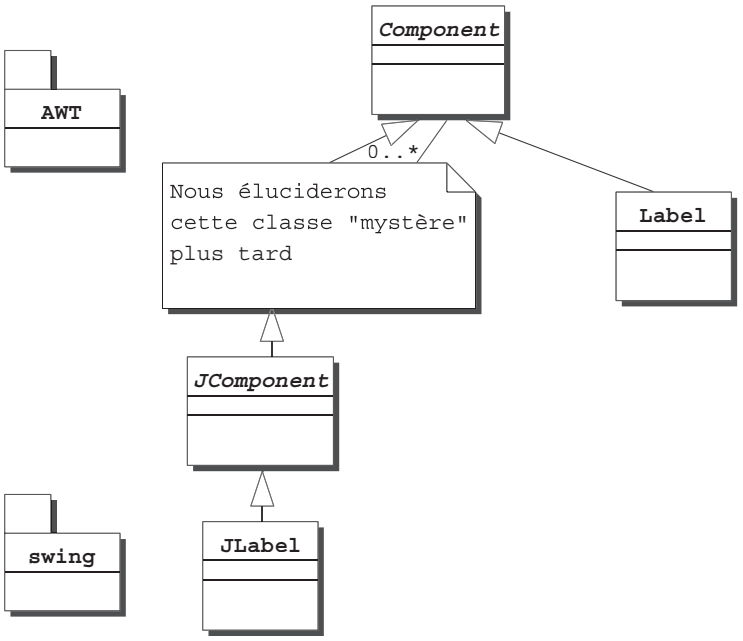


Figure 1.13 — `Component` et `Jcomponent`.

Ainsi le composant `Label` hérite de `Component`, classe abstraite « racine » du package `java.awt`. De même, `JLabel` hérite de `JComponent`, classe « racine » du package `javax.swing`.

`JComponent` est donc une classe très importante car tous les composants Swing en héritent.

> Les composants Swing s'appuient sur les composants AWT. Ainsi, le composant AWT `Exemple` aurait pour équivalent Swing `JExemple`. Cette règle n'est pas systématique. Certaines classes Swing n'ont pas d'équivalent AWT et réciproquement.

Sun préconise de ne pas mélanger des composants Swing avec des composants AWT, car cela est susceptible de créer des comportements inattendus. Mais attention, il s'agit bien de ne pas mélanger des *composants* : il est bien sûr indispensable d'utiliser les classes AWT pour programmer en Swing.

Nous allons maintenant faire un tour rapide des composants Swing les plus fréquemment utilisés, puis des capacités communes à tous les composants.

1.3.3 Tour d'horizon rapide des composants Swing

Une des clés lors de la construction d'interfaces graphiques est la connaissance des différents composants que propose Swing. Le but de cet ouvrage n'est pas d'offrir un catalogue exhaustif des composants, mais plutôt de s'attarder sur les principes. Cependant, nous n'éviterons pas une présentation sommaire des principaux composants simples sous forme de fiches signalétiques.

`JLabel`

Nom

`javax.swing.Jlabel`

Usage

Ce composant permet d'afficher du texte statique et/ou une image. Ce composant ne réagit pas aux interactions de l'utilisateur. On l'utilise souvent pour décrire les informations présentées dans une interface. Ainsi, un champ de saisie va être précédé d'un `JLabel` décrivant la nature de l'information à saisir (figure 1.14).

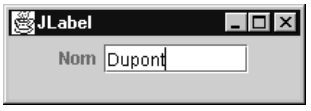


Figure 1.14 — Utilisation d'un `JLabel` « Nom ».

Méthodes

L'ajout de texte peut se faire de différentes façons. On peut passer la chaîne de caractères en paramètre du constructeur :

```
JLabel lb = new JLabel("Nom");
```

On peut également utiliser la méthode `setText` :

```
lb.setText("Nom");
```

Le texte peut naturellement être obtenu avec l'accesseur en consultation : `getText()`.

Il est possible d'ajouter une icône au `Jlabel` (figure 1.15).

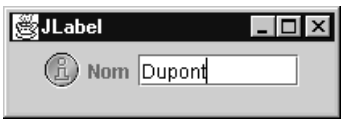


Figure 1.15 — `JLabel` avec une icône et du texte.

L'ajout de l'icône peut se faire dans le constructeur ou à l'aide de la méthode `setIcon`.

```
lb.setIcon(new ImageIcon("Information24.gif"));
```

On peut spécifier la position du texte par rapport à l'icône, par exemple pour afficher le texte sous l'icône (figure 1.16).

Figure 1.16 — Icône et texte l'un dessous l'autre.

Cela se fait de la façon suivante :

```
lb.setVerticalTextPosition(SwingConstants.BOTTOM);
lb.setHorizontalTextPosition(SwingConstants.CENTER);
```

Il est également possible de créer des `JLabel` avec du texte mis en forme (gras, italique…), car la méthode `setText` supporte le texte HTML (figure 1.17).

Figure 1.17 — `JLabel` contenant du texte formaté.

L'exemple précédent est obtenu grâce à l'instruction :

```
lb.setText("<html>Saisissez <p> votre <B><I>nom</B>
➥</I></html>");
```

Attention ! Soyez sûr de votre code HTML, sinon la compilation se passe bien, mais une exception est levée à l'exécution.

JTextField

Nom

`javax.swing.JTextField`

Usage

Ce composant présente un champ de saisie de texte. Il n'existe pas de champ de saisie typé pour des données particulières : nombres, dates… La seule variante est un champ de saisie pour les mots de passe : `JPasswordField`.

Toutefois, cela ne constitue pas une limitation car il est possible d'obtenir tous les types primitifs tels que `int`, `double`, `long`… ou encore leur version objet tel que `Integer`, `Double` ou `Long` à partir d'une chaîne de caractères.

Méthodes

Comme pour le composant `JLabel`, il est possible de spécifier un texte en paramètre du constructeur :

```
JTextField champ = new JTextField("du texte");
```

ou bien à l'aide de la méthode `setText` :

```
JTextfield champ = new JTextField();
String texte = new String("du texte");
champ.setText(texte);
```

Comme pour le composant `JLabel`, le texte peut être obtenu grâce à l'accesseur en consultation : `public String getText()`.

Si le texte est plus grand que la longueur du champ, il est possible de lire tout le texte en se déplaçant à l'aide des flèches. Cependant, on peut aussi imposer un nombre de colonnes afin que tout le texte soit visible :

```
champ.setColumns(texte.length());
```

Il est possible de transférer la valeur du champ dans le presse-papiers en utilisant la méthode `copy` ou `cut` selon que l'on souhaite copier ou couper le texte : `champ.copy()` ou `champ.cut()`.

Le presse-papiers contient donc la chaîne `"du texte"`.

Pour transférer la valeur du presse-papiers dans un autre champ, on utilise la méthode `paste` :

```
JTextField champ2 = new JTextField();
champ2.paste();
```

D'autres méthodes de paramétrage de ce composant seront vues dans le chapitre 5 consacré aux composants texte.

JButton

Nom

`javax.swing.JButton`

Usage

Ce composant présente un bouton. Il existe de nombreux types de boutons : les boutons simples, les radio-boutons, les boutons à états… `JButton` représente un bouton simple, qui peut contenir du texte et/ou une image. La position du texte par rapport à l'image est variable.

L'icône peut être indiquée au moment de l'instanciation, en paramètre du constructeur, ou bien ultérieurement avec la méthode `setIcon` (figure 1.18).

On peut également indiquer une icône spécifique qui s'affiche lorsque le curseur de la souris passe au-dessus du bouton (figure 1.19), et une autre qui s'affiche au moment du clic (figure 1.20).

Figure 1.18 — Aspect du bouton à l'état normal.

Figure 1.19 — Aspect du bouton lorsque le curseur est dessus.

Figure 1.20 — Aspect du bouton lors du clic.

Méthodes

Le paramétrage de ces différentes icônes se fait de la façon suivante :

```
JButton bt = new JButton("Info", new ImageIcon
➥("Information24.gif"));
// quand le curseur passe sur le bouton
bt.setRolloverIcon(new ImageIcon("About24.gif"));
// quand l'utilisateur clique
bt.setPressedIcon(new ImageIcon("TipOfTheDay24.gif"));
```

On peut indiquer également un *mnemonic*, c'est-à-dire un moyen de produire un clic sur le bouton à l'aide du clavier uniquement. La lettre qui sert de *mnemonic* est soulignée dans le texte du bouton. L'utilisateur peut actionner le bouton en tapant la combinaison de touches « Alt + mnemonic ».

```
// pour specifier un mnemonic
bt.setMnemonic('n');
```

Quand un bouton est désactivé, il prend un aspect grisé. Si on le souhaite il est possible de préciser explicitement une icône à afficher lorsque le bouton est dans cet état à l'aide de la méthode `setDisabledIcon`. Si rien n'est précisé, le programme se charge d'afficher l'icône de façon grisée.

Il est possible également d'indiquer si la bordure doit être affichée ou non, et si le focus doit être visible ou non. Dans les images ci-dessus, la bordure et l'indication de focus étaient affichées. Voici en figure 1.21 l'aspect du bouton sans cela.

Figure 1.21 — Bouton sans bordure et sans indication de focus.

```
// pour enlever la bordure
bt.setBorderPainted(false);
// pour enlever le cadre indiquant le focus
bt.setFocusPainted(false);
```

Il est possible de simuler un clic utilisateur à l'aide de la méthode `doClick`.

Nous verrons dans le chapitre 3 consacré aux événements comment gérer le clic de l'utilisateur, comment affecter une action au bouton.

JToggleButton

Nom

```
javax.swing.JToggleButton
```

Usage

Cette classe permet de représenter des boutons à deux états. On l'utilise donc lorsque l'aspect visuel du bouton doit refléter l'état (ON ou OFF) d'une fonctionnalité.

Un cas classique d'utilisation se trouve dans les barres d'outils des éditeurs de texte (figure 1.22). En effet, les boutons servent d'une part à effectuer des modifications sur le texte (mettre en gras, changer la police), mais aussi à refléter l'état du texte à un endroit donné.

Figure 1.22 — Boutons reflétant l'état du texte.

Méthodes

Par défaut, un `JToggleButton` aura deux aspects – enfoncé ou non – en fonction de son état. On peut spécifier une icône particulière pour chaque état : la méthode `setIcon` indique l'icône pour l'état « non sélectionné » et la méthode `setSelectedIcon` pour l'état « sélectionné ». En figure 1.23, le bouton présenté n'a pas de texte, mais seulement une icône qui varie suivant l'état.

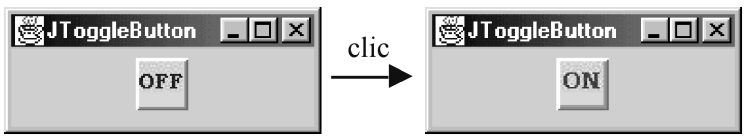


Figure 1.23 — Un `JToggleButton` et ces deux états.

Voici le code permettant d'obtenir ce résultat :

```
JToggleButton bt = new JToggleButton(new ImageIcon
➥("Off24.jpg"));
// icône pour l'état sélectionné
bt.setSelectedIcon(new ImageIcon("On24.jpg"));
// pour afficher une bordure de mise en relief
bt.setBorder(BorderFactory.createRaisedBevelBorder());
```

On peut également forcer l'état sélectionné ou non d'un bouton à l'aide de la méthode `setSelected`. On peut aussi récupérer l'état avec `isSelected`.

Remarque : ne cherchez pas trop longtemps dans la documentation Java ! Ces méthodes se trouvent dans la super-classe `AbstractButton` et non dans la classe `JToggleButton`…

`JCheckBox` et `JRadioButton`

Usage

Les classes `JCheckBox` et `JRadioButton` permettent de créer respectivement des boutons radio et des cases à cocher (figure 1.24).

La présentation d'une case à cocher est assez proche de celle d'un bouton radio, mais le mode de sélection est habituellement différent. En effet, une case à cocher peut être utilisée seule, alors qu'un bouton radio est prévu pour être utilisé au sein d'un groupe. Au sein d'un groupe de boutons radio, un seul peut être sélectionné à la fois. En revanche, il est possible de présenter plusieurs cases à cocher côte à côte, leurs états (sélectionné ou non) sont complètement indépendants.

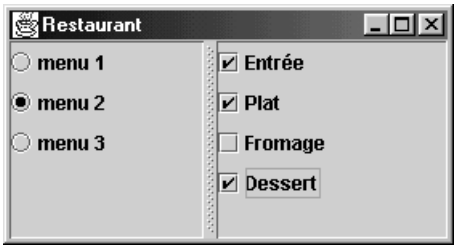


Figure 1.24 — Utilisation de cases à cocher et des boutons radio.

Méthodes

Cela se fait avec le code suivant :

```
// Création des boutons radio
ButtonGroup groupe = new ButtonGroup();
```

```
JRadioButton rbMenu1 = new JRadioButton("Menu 1");
JRadioButton rbMenu2 = new JRadioButton("Menu 2");
JRadioButton rbMenu3 = new JRadioButton("Menu 3");
rbMenu2.setSelected(true);
groupe.add(rbMenu1);
groupe.add(rbMenu2);
groupe.add(rbMenu3);
// Création des cases à cocher
JCheckBox cbEntree = new JCheckBox();
JCheckBox cbPlat = new JCheckBox();
JCheckBox cbFromage = new JCheckBox();
JCheckBox cbDessert = new JCheckBox();
cbEntree.setText("Entrée");
cbPlat.setText("Plat");
cbFromage.setText("Fromage");
cbDessert.setText("Dessert");
```

Les boutons radio sont ajoutés à un `ButtonGroup`. Grâce à cela, la sélection d'un bouton provoque la désélection du bouton précédemment sélectionné.

JComboBox

Nom

`javax.swing.JComboBox`

Usage

Ce composant représente une liste déroulante, dans laquelle l'utilisateur peut choisir un item. Il est utilisé pour présenter une liste de valeurs possibles, au sein desquelles l'utilisateur effectue son choix (figure 1.25). Le composant possède deux états, fermé (figure 1.26) ou bien ouvert (figure 1.25) quant l'utilisateur a cliqué dessus pour effectuer un choix ou changé le choix actuel.

Figure 1.25 — `JComboBox` simple.

Figure 1.26 — JCombox simple, fermée.

Méthodes

Pour indiquer les items à afficher dans la liste déroulante, on peut utiliser un `Vector` ou un tableau d'objets, qu'on passe en paramètre du constructeur. On peut également ajouter des items supplémentaires à l'aide de la méthode `addItem`.

```
Vector items = new Vector();
items.add("Lundi");
items.add("Mardi");
items.add("Mercredi");
items.add("Jeudi");
items.add("Vendredi");
items.add("Samedi");
items.add("Dimanche");
JComboBox cb = new JComboBox(items);
```

Lorsque le nombre de valeurs présentées est important, un ascenseur apparaît pour permettre le défilement de la liste (figure 1.27).

Toutefois, pour présenter un grand nombre d'items, on préférera parfois passer par un composant `JList` et éventuellement fournir une possibilité de filtre et de tri.

Figure 1.27 — JCombobox avec un ascenseur.

On peut indiquer le nombre d'items à afficher de la façon suivante :

```
// Pour ne pas afficher plus de 4 lignes
cb.setMaximumRowCount(4);
```

Il est aussi possible de créer des `JComboBox` éditables (figure 1.28). Dans ce cas, le champ qui présente l'item sélectionné est éditable. Il s'agit tout simplement d'un `JTextField`.

Par défaut, une `JComboBox` est non éditable.

Figure 1.28 — `JComboBox` éditable.

```
Vector items = new Vector();
items.add("France");
items.add("Angleterre");
items.add("Allemagne");
items.add("Italie");
JComboBox cb = new JComboBox(items);
// Pour rendre la combo éditable
cb.setEditable(true);
```

1.3.4 Les capacités communes à tous les composants Swing

Activation et désactivation d'un composant

Chaque composant d'une application peut être actif ou inactif (figures 1.29 et 1.30). Un composant inactif est inaccessible à l'utilisateur. Par exemple, un élément de menu proposant une sauvegarde sera désactivé tant qu'aucun document ne sera chargé dans l'application. Un tel composant apparaîtra grisé.

Comment désactive-t-on un composant ? Grâce au polymorphisme, la manière de désactiver un composant ne dépend pas du type de composant. Ce comportement est déclaré au niveau de la classe `JComponent`.

Désactiver un `JLabel` n'a pas beaucoup de sens au niveau ergonomique. Un `JLabel` n'est pas un composant interactif.

Désactivons donc un `JButton` !

Figure 1.29 — Un bouton désactivé.

Figure 1.30 — Un bouton activé.

```
public class TestSimplissime {
     public static void main(String args[])    {
        Simplissime f = new Simplissime();
        f.SetVisible(true);
        f.desactive();
     }
}

import javax.swing.*;

public class Simplissime extends JFrame {
   protected static final String TEXTE = "Ceci est un bouton";
   protected JButton bouton = new JButton(TEXTE);

   public Simplissime() {
      getContentPane().add(bouton);
   }

   public void desactive() {
      bouton.setEnabled(false);
   }
}
```

La classe `TestSimplissime` n'est là que pour tester la classe `Simplissime`. La classe `Simplissime` est une sous-classe de `JFrame`, elle propose une méthode capable de désactiver le bouton.

Notez une différence entre cet exemple et ceux qui ont été précédemment présentés. Ainsi, dans le tout premier exemple de ce livre, la classe nommée `Bonjour` héritait directement de `Object` et se chargeait de construire l'interface graphique à l'aide d'une variable `fenetre`. Cette variable `fenetre` était une instance de `JFrame` obtenue directement :

```
fenetre = new JFrame();
```

Dans notre exemple d'activation de bouton, nous avons fait une sous-classe de `JFrame` : `Simplissime`. C'est la classe `TestSimplissime` qui instancie `Simplissime`. La variable `f` de la classe `TestSimplissime` est une instance de `Simplissime`, donc de `JFrame`. Ces deux approches sont complémentaires.

`JButton` est un composant simple. Il possède un constructeur du type `JButton(String)`. Pourquoi complique-t-on le code dans l'exemple en utilisant un attribut *static* comme paramètre ? Comme souvent, le bénéfice de cette

Pourquoi y a-t-il deux classes *TestSimplisime* et *Simplissime* ?
N'aurait-on pas pu mettre la méthode *main* dans la classe `Simplissime` ? Bien sûr, cela aurait été possible. En fait, il vaut mieux perdre l'habitude de placer dans une classe A une méthode *main* pour tester la classe A. Puisque Java est un langage orienté objet, affectons à chaque classe un *contrat de service* précis en codant deux classes A et `TestA`. Tous les utilisateurs de la classe A ont-ils besoin de cette méthode *main* ? Probablement pas. Ne « polluons » donc pas la classe A avec un savoir-faire inutile. De plus, avec ce fonctionnement, il est possible de livrer du code sans pour autant livrer les classes de test.

complexité n'apparaît pas immédiatement. C'est une vue sur le moyen terme qui est la cause de ce petit ajout. Sans remettre en cause la conception générale d'une fenêtre ou d'une boîte de dialogue, il est fréquent que des modifications mineures soient nécessaires. Si l'on a pris la peine de regrouper en début de classe et d'extraire les invariants graphiques dans des variables de classes de type `final static`, ces modifications seront grandement facilitées. Ce que nous appelons « invariant graphique », ce sont toutes ces petites choses que l'on met « en dur » dans le code sans même s'en apercevoir, comme des chaînes de caractères, des coordonnées, des tailles, etc.

Les méthodes suivantes sont proposées au niveau de la classe `Component`.

Pour tester si un composant est activé :

```
boolean isEnabled()
```

Pour activer ou désactiver un composant :

```
void setEnabled(true ou false)
```

En plus d'être actif ou inactif, un composant peut être visible ou non.

Pour positionner la visibilité :

```
void setVisible(true ou false)
```

Pour tester la visibilité :

```
boolean isVisible()
```

Vous vous demandez probablement : à quoi bon définir et gérer un composant dans une interface s'il est invisible ? Tout n'est pas statique dans une interface graphique. Il faut distinguer plusieurs cas. Une option cochée ou non peut activer ou désactiver un composant d'une interface.

Prenons l'exemple d'une boîte de dialogue d'options dans un logiciel de traitement de texte. Si l'option « sauvegarde automatique » est activée, alors

l'utilisateur peut préciser une périodicité, par exemple toutes les 10 minutes. Cette partie de la boîte de dialogue peut donc s'activer ou se désactiver, mais elle reste toujours visible.

En revanche, selon que l'on possède une version d'évaluation ou une version complète, certains composants graphiques seront visibles ou non. Par exemple, pas de sauvegarde automatique en version d'évaluation, ici l'option n'est pas visible.

Les méthodes `setVisible`, `isVisible`, `isEnabled` et `setEnabled` sont définies au niveau de la classe `JComponent` qui est une classe abstraite. Chaque composant, sous-classe de `JComponent` hérite de ces méthodes. Il est donc possible d'écrire :

```
JComponent unComposant;
unComposant = new JButton();
unComposant.setVisible(false);
```

Tailles

Pourquoi employons-nous le pluriel dans ce titre ? Il y a plusieurs notions de « taille » pour un composant. Nous verrons plus tard plus en détail comment cela est possible. Disons dès maintenant qu'un composant a une taille réelle et une taille souhaitable. Tout le problème pour une interface graphique moderne utilisant des fenêtres est que l'utilisateur peut faire varier la taille de la fenêtre ou d'une zone de la fenêtre. Il se peut donc, si le composant se redimensionne avec la fenêtre, que sa taille souhaitable ne tienne pas dans la fenêtre. Sans ce double concept de taille réelle, taille souhaitable, le composant risquerait de ne pas être entièrement visible ou bien plus simplement de ne jamais se redimensionner. Une solution pénible consisterait pour le développeur à définir par programme la stratégie de modification de taille pour chaque composant. Heureusement, Swing apporte à cela une solution solide, nous en parlerons dans le chapitre 2 consacré aux layouts.

La taille réelle s'obtient avec la méthode `getSize`, qui renvoie un objet `Dimension`.

```
Dimension getSize()
Dimension getSize(Dimension rv)
```

Les accesseurs de type `get`/`set` sont standardisés. Cet accessseur de lecture de la valeur `size` est un peu étrange : si la référence sur `Dimension rv` est nulle alors une `Dimension` est instanciée et retournée, sinon, la méthode utilise l'instance de `Dimension` *via* la référence `rv`, la modifie et la retourne. Ce mécanisme permet d'économiser des instances de `Dimension`.

`Dimension` est une classe définie dans le package `java.awt`. C'est une de ces classes que Swing utilise bien qu'elle soit définie dans AWT. Cette classe n'a pas d'impact graphique. Elle représente une dimension, avec une hauteur et une largeur :

```
double Dimension.getHeight() et
double Dimension.getWidth().
```

S'il est possible de récupérer la taille actuelle, réelle, d'une instance de `JComponent`, il doit bien être possible de lui imposer cette taille réelle. Il existe en effet une méthode `setSize`. Vous ne la trouverez pas sur la classe `JComponent`. Cela ne doit pas vous surprendre. La classe `JComponent`, comme nous l'avons vu, hérite de `Component`. Swing étend le comportement de la librairie AWT. Nous trouvons donc sur la classe `Component` une méthode `setSize` avec de multiples signatures paramétriques, dont la suivante : `void setSize(Dimension rv)`.

Voici en figure 1.31 ce que donne une interface avec des boutons dont la taille a été définie.

Figure 1.31 — Bouton de taille moyenne.

Cet exemple illustre tout à fait notre propos. Une taille « en dur » a été imposée au bouton sans tenir compte de la taille du texte contenu dans le bouton. En cherchant à imposer une taille fixe et non calculée, on va à l'encontre d'un grand principe Swing. Ce principe est que tout doit être calculé dynamiquement afin que l'interface s'adapte à toutes les situations.

Que se passe-t-il si le bouton devenait plus grand que la fenêtre (figure 1.32) ?

Notre principe de fixer une taille « en dur » est assez brutal, mais le bouton prend bien la taille que nous lui avons demandée, il ne se préoccupe pas de la taille de la fenêtre. Pour éviter le problème constaté, à savoir que le bouton ne « rentre » pas dans la fenêtre, il faudrait indiquer à la fenêtre de calculer sa taille en fonction de son contenu. Nous aborderons ce genre de comportements dans la section 1.3 traitant des fenêtres.

Figure 1.32 — Un bouton de trop grande taille.

À l'inverse, le bouton peut devenir beaucoup trop petit (figure 1.33).

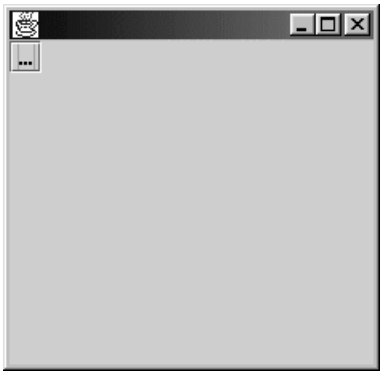


Figure 1.33 — Un bouton trop petit.

Le bouton est si petit que même la première lettre du texte ne peut tenir dans l'espace alloué au bouton.

Comment avons-nous obtenu ce comportement ? Les étapes sont les suivantes :

D'abord, instancier une fenêtre :

```
JFrame fenetre = new JFrame();
```

Instancier un bouton :

```
JButton bouton = new JButton("Ceci est un bouton");
```

Ajouter le bouton dans la fenêtre :

```
fenetre.getContentPane().add(bouton);
```

Instancier une dimension moyenne, susceptible d'être trop grande ou trop petite selon nos besoins :

```
Dimension dim = new Dimension(100, 50);
```

Positionner la taille du bouton :

```
bouton.setSize(dim);
```

Afficher la fenêtre :

```
fenetre.SetVisible(true);
```

Sans précision de votre part, le bouton prend sa taille souhaitable par défaut, mais n'oublions pas que dans tous les cas, rien ne peut garantir que la taille souhaitable sera respectée. La taille souhaitable se nomme la `preferredSize`. Il est possible d'accéder à cette valeur ou de la modifier avec les méthodes `Dimension get PreferredSize()` et `void setPreferredSize(Dimension preferredSize)`.

Positionner la `preferredSize` ne nous garantit aucunement d'observer le composant avec cette taille. La `preferredSize` n'est que la taille souhaitable, des contraintes que nous préciserons plus tard peuvent influer sur la taille souhaitable et la rendre différente de la taille réelle.

Curseurs

Il est possible de définir l'apparence que prend le curseur de la souris lorsqu'il est amené au-dessus d'un composant. Cette faculté est définie au niveau de la classe `Component` du package `java.awt`, c'est donc par héritage que nous en bénéficions au niveau de `JComponent`.

La méthode à utiliser est : `setCursor(un_curseur)`.

Le paramètre `un_curseur` doit être une instance de `Cursor`. La question qui se pose maintenant est « comment obtenir une telle instance ? ». La classe `Cursor` propose des méthodes statiques agissant comme une usine (en anglais, *factory*) à curseurs. C'est ainsi que nous trouvons comme attributs statiques de la classe `Cursor` des entiers définissant des types de curseur. De même nous trouvons deux méthodes particulières permettant d'obtenir une instance de `Cursor` à partir d'un type :

Le constructeur : `Cursor(int type)`

Une méthode statique : `Cursor getPredefinedCursor(int type)`

Par exemple, pour obtenir une instance d'un curseur en forme de croix fine, généralement utilisé pour pointer avec plus de précision, on peut utiliser la méthode statique :

```
Cursor nouveauCurseur =
Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR)
```

Ou utiliser le constructeur :

```
Cursor nouveauCurseur = new Cursor(Cursor.CROSSHAIR
➥_CURSOR)
```

Il est ensuite possible de l'affecter à un composant :

```
JComponent monComposant = new JButton();
MonComposant.setCursor(nouveauCurseur);
```

Pour que cela prenne effet, il faut que le composant soit visible et armé (*enabled*). Ensuite, le curseur change de forme suivant le composant qu'il survole. Prenons par exemple une interface composée de plusieurs boutons, chacun ayant un curseur différent (figure 1.34).

Notez que pour cet exemple nous avons utilisé une référence de type `JComponent` sur une instance de `JButton`. Cela fonctionne car `JButton` est une sous-classe de `JComponent` et parce que la méthode `setCursor` est définie au niveau de la classe `JComponent`.

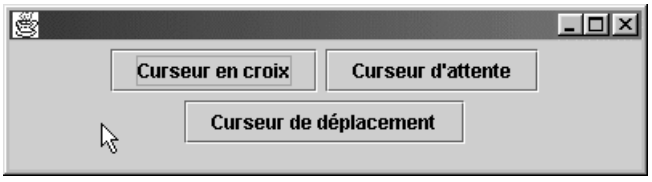


Figure 1.34 — Notre fenêtre avec trois boutons.

En dehors d'un des composants pour lesquels un curseur a été défini (figures 1.35 à 1.37), le curseur prend l'apparence standard.

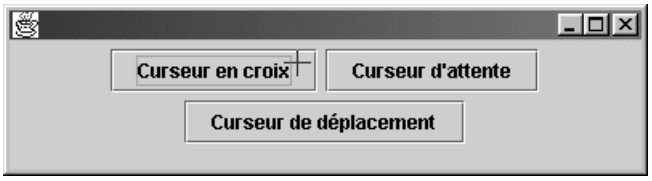


Figure 1.35 — Curseur en forme de croix.

Figure 1.36 — Curseur en forme de sablier.

Figure 1.37 — Curseur en forme de croix de déplacement.

La méthode `setCursor` ne fait que changer l'apparence du curseur, elle n'influe pas sur le comportement du curseur. En particulier, il est toujours possible de cliquer sur le bouton bien que le curseur soit en forme de sablier. Rendre cohérents l'apparence du curseur et son comportement demande une programmation spécifique utilisant les événements que nous étudierons au chapitre 3.

Tooltips, ou bulles d'aide

Sur tous les composants Swing, il est possible d'ajouter un tooltip (figure 1.38).

> Un **tooltip** est une bulle d'information qui apparaît lorsque l'utilisateur laisse sans bouger le curseur de la souris au-dessus d'un composant.

La façon de procéder pour l'ajout d'un tooltip est on ne peut plus simple. Il suffit d'utiliser la méthode `setToolTipText` définie sur la classe `Jcomponent`.

Figure 1.38 — Un premier tooltip.

Suivant le *look end feel* utilisé, le tooltip n'apparaît pas de la même façon. En figure 1.38, il s'agit d'une copie d'écran avec un *look end feel* Windows, le tooltip est sur fond jaune clair.

Le code qui permet l'ajout du tooltip est le suivant :

```
JFrame f = new JFrame("JButton");
JButton bt = new JButton("Surprise ! ");
bt.setToolTipText("Cliquez ici pour avoir une surprise");

f.getContentPane().setLayout(new FlowLayout());
f.getContentPane().add(bt);
f.setSize(400, 100);
f.setVisible(true);
```

Certains composants proposent une API spécifique pour l'ajout de tooltips. Il s'agit en particulier du panneau à onglets qui permet d'associer un tooltip différent à chaque onglet, en précisant un index en paramètre de la méthode `setToolTipTextAt` (figure 1.39).

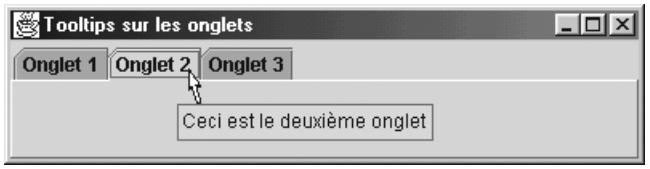


Figure 1.39 — Tooltip sur des onglets.

```
JFrame f = new JFrame("Tooltips sur les onglets");
JTabbedPane tabPan = new JTabbedPane();
tabPan.addTab("Onglet 1", new JPanel());
tabPan.addTab("Onglet 2", new JPanel());
tabPan.addTab("Onglet 3", new JPanel());

tabPan.setToolTipTextAt(0, "Ceci est le premier onglet");
tabPan.setToolTipTextAt(1, "Ceci est le deuxième onglet");
tabPan.setToolTipTextAt(2, "Ceci est le troisième onglet");

f.getContentPane().add(tabPan, BorderLayout.CENTER);
f.setSize(400, 100);
f.setVisible(true);
```

Il est également possible de mettre du texte au format HTML dans un tooltip (figure 1.40). Cela permet bien sûr d'avoir une mise en forme agréable, mais surtout, cette nouvelle possibilité pallie un manque de Swing dans la version précédente : il n'était pas possible de faire des tooltip multilignes. Or cela était problématique, car dans le cas d'un texte à afficher assez long, le tooltip se retrouvait décalé dans la fenêtre pour être affiché en entier.

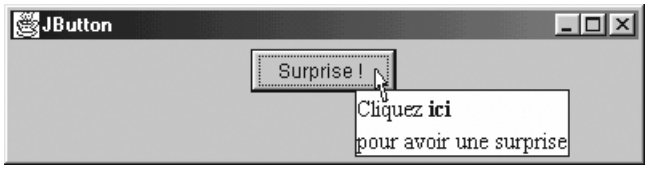


Figure 1.40 — Un tooltip mis en forme.

Pour mettre en forme un tooltip avec le format HTML, il suffit de passer du code HTML en paramètre de la méthode `setToolTipText`. La chaîne devra obligatoirement débuter par une balise `<html>` et se terminer par la balise fermante correspondante `</html>`.

Pour obtenir l'exemple présenté figure 1.40, il faut donc l'instruction suivante :

```
bt.setToolTipText("<html>Cliquez <b>ici</b> <br>pour
➥avoir une surprise</html>");
```

D'autre part, il est aussi possible de paramétrer finement le fonctionnement des tooltips grâce à la classe `ToolTipManager`. Cette classe est un singleton : on accède à l'instance unique grâce à la méthode `sharedInstance`. Cette instance a été créée automatiquement. Il existe des méthodes à utiliser sur cet objet pour modifier le temps d'attente avant qu'un tooltip n'apparaisse ou bien le temps au bout duquel le tooltip disparaît.

Ainsi, pour avoir une interface très réactive, où les tooltips apparaissent quasiment instantanément et disparaissent après deux secondes, il faut utiliser les instructions suivantes :

```
ToolTipManager.sharedInstance().setInitialDelay(200);
ToolTipManager.sharedInstance().setDismissDelay(2000);
```

Couleurs

La couleur des composants graphiques peut être paramétrée facilement grâce aux méthodes `setBackground` et `setForeground`, qui permettent de régler respectivement la couleur de fond et la couleur de premier plan.

Figure 1.41 — Une interface avec des couleurs particulières.

Pour obtenir l'interface de la figure 1.41, nous avons paramétré la couleur du panneau, la couleur de fond du bouton et la couleur de premier plan du bouton. Les méthodes `setBackground` et `setForeground` prennent en paramètre un objet de la classe `Color`. Comment crée-t-on une couleur ?

La classe `Color` se trouve dans le package `java.awt`. En effet, cette notion de couleur date des premières versions du JDK. Elle permet d'encapsuler une couleur dans l'espace de couleur standard, appelé sRGB, pour « standard Red-Green-Blue ». Une instance de `Color` n'a pas d'impact graphique tant qu'elle n'est pas affectée à un composant.

Pour construire une couleur, on peut utiliser le constructeur :

```
public Color(int r, int g, int b) ;
```

où l'on précise les trois composantes de la couleur, rouge, vert, bleu, par des entiers compris entre 0 et 255.

On peut également utiliser des variables de classes qui correspondent à des couleurs courantes : blanc, noir, les couleurs primaires, différentes nuances de gris, etc.

Ainsi, pour obtenir l'interface présentée ci-dessus :

```
JFrame f = new JFrame("Les couleurs");
JPanel pan = new JPanel();
JButton bt = new JButton("Blanc sur noir");

pan.setBackground(Color.lightGray);
bt.setBackground(Color.black);
bt.setForeground(Color.white);

pan.add(bt);
f.getContentPane().add(pan);
f.setSize(400, 100);
f.setVisible(true);
```

Il est également possible de préciser un degré de transparence, couramment appelé la composante alpha. Lorsqu'on ne précise pas cette composante, les couleurs créées sont opaques.

Comparez en figure 1.42 la différence apportée par la composante alpha.

Figure 1.42 — Un bouton avec et sans transparence.

Les deux boutons ont les mêmes composantes RGB, mais le deuxième bouton a en plus une composante alpha de 100. De ce fait, le gris clair du panneau en dessous apparaît mélangé à la couleur initiale qui est un rouge vif.

Lorsqu'on ne précise rien, la composante alpha vaut 255, ce qui équivaut à une couleur parfaitement opaque. Les deux boutons sont créés comme cela :

```
JButton bt = new JButton("Couleur opaque");
bt.setBackground(new Color(230,50,50));
bt.setForeground(Color.white);

JButton bt2 = new JButton("Degré de transparence");
bt2.setBackground(new Color(230,50,50,100));
bt2.setForeground(Color.white);
```

Certains composants sont par défaut non opaques. Ils ne sont pas impactés par le changement de la couleur de fond, à moins qu'on ne les définisse comme opaques. C'est le cas en particulier du composant `JLabel`.

Figure 1.43 — Deux `JLabels`, transparent et opaque.

La couleur de fond du premier `JLabel` (figure 1.43) a été positionnée à noir, mais comme le composant est transparent, c'est la couleur du panneau sous-jacent qui apparaît.

```
JFrame f = new JFrame("Les couleurs");
JPanel pan = new JPanel();
pan.setLayout(new BoxLayout(pan, BoxLayout.Y_AXIS));
pan.setBackground(Color.lightGray);

JLabel lb = new JLabel("Texte blanc sur fond noir");
lb.setBackground(Color.black);
lb.setForeground(Color.white);

JLabel lb2 = new JLabel("Cette fois avec l'opacité");
lb2.setBackground(Color.black);
lb2.setForeground(Color.white);
lb2.setOpaque(true);

pan.add(lb);
pan.add(lb2);
f.getContentPane().add(pan);
f.setSize(400, 100);
f.setVisible(true);
```

Bordures

L'association d'une bordure, ou *border* en anglais, est un autre exemple de fonctionnalité définie au niveau de `JComponent`. Un `Border` est une sorte de cadre de différente nature qui peut entourer le composant. Les `Border` sont souvent utilisés dans une interface graphique pour grouper les composants par zones distinctes.

Ajoutons un `Border` à notre `JLabel` !

Présentons en premier lieu le code nécessaire à la réalisation de cette interface qui affiche successivement différents types de `Border` (figures 1.44 à 1.46).

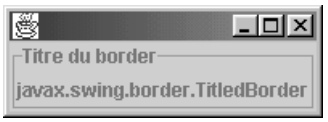


Figure 1.44 — `TitledBorder`.

Figure 1.45 — `EtchedBorder`.

Figure 1.46 — `BevelBorder`.

```
import java.util.Vector;
import java.util.Enumeration;
import javax.swing.*;
import javax.swing.border.Border;

public class Fenetre extends JFrame {

   protected static final long DELAI = 700;
   protected static final String TEXT_DU_BORDER = "Titre
   ➥du border";
   protected JLabel label = new JLabel();
   protected Vector borders = new Vector();
   protected Enumeration enum;

   public Fenetre() {
      getContentPane().add(label);
   }

   public void borderSuivant() {
      Border borderCourant;
      // Si l'énumération est vide ou si tous les éléments
      // ont été parcourus, alors on initialise l'énumération.
      if (enum == null || !enum.hasMoreElements())  {
         enum = borders.elements();
      } else {
         borderCourant = (Border)enum.nextElement();
         label.setBorder(borderCourant);
         label.setText(borderCourant.getClass().getName());
      }
   }

   public void alternerBorder() {
      while (true)  {
         try {
             Thread.sleep(DELAI);
         } catch (InterruptedException e) {}
         borderSuivant();
      }
   }
```

```
    public void remplirListe() {
       borders.add(BorderFactory.createEmptyBorder());
       borders.add(BorderFactory.createEtchedBorder());
       borders.add(BorderFactory.createLoweredBevelBorder());
       borders.add(BorderFactory.createRaisedBevelBorder());
       borders.add(BorderFactory.createTitledBorder(TEXT_DU
       ➥_BORDER));
    }
}
```

Décortiquons le code :

Les imports

Importation du package `java.util`

Nous nous servirons des collections du type `Vector` pour stocker les différentes instances de `Border` et nous itérerons sur cette collection pour affecter successivement les instances de `Border` à notre composant.

Importation de la classe `javax.swing.border.Border`

Le but de notre exemple est notamment d'illustrer l'utilisation des `Border`. `Border` est une interface Java dont nous avons besoin.

Pourquoi écrire `import x.y.classe` plutôt que `import x.y.*` ?
Il n'y a pas de considérations de performances : la compilation peut être ralentie par l'usage de *, mais dans des proportions minimes.

Cependant, il peut être utile dans le cadre d'un projet d'importance d'avoir une vision précise des dépendances entre packages. S'il est nécessaire de déplacer des classes d'un package vers un autre, éventuellement nouveau, mesurer l'impact de ce déplacement sera plus aisé avec des imports explicites. En conclusion, utiliser `import x.y.z.*` pour des packages Java ne présente pas d'inconvénient technique. En revanche pour des packages « applicatifs », qui ne font pas partie du JDK, l'usage de l'étoile peut diminuer la vision sur les dépendances entre packages.

L'algorithme utilisé

La méthode `remplirListe()`

Notre classe dispose d'une liste de `Border`. Cette liste est implémentée par un `Vector`. La méthode `remplirListe()` va simplement ajouter au vecteur des instances de `Border`. Ces instances sont obtenues *via* la classe `BorderFactory` et ses nombreuses méthodes statiques du type `create<nom du border>Border()`. Le rôle de la `BorderFactory`, conformément à un pattern classique, a pour rôle de fournir des instances de `Border` sur demande. Il s'agit ici d'une facilité prévue

par le JDK, il n'y a pas de notion de factorisation d'instance. En d'autres termes, les différents `Border` ne sont pas des *singletons*.

La méthode `borderSuivant()`

Cette méthode itère sur la collection. Une énumération est ouverte sur la collection (ici `Vector`) et `borderSuivant()` affecte au label la prochaine instance de `Border` que fournit l'itérateur (ici `Enumeration`). Si l'itérateur est vide, un nouvel itérateur est instancié de sorte que l'on puisse boucler sans fin sur la collection.

La méthode `alternerBorder()`

Cette méthode est constituée d'une boucle sans fin — `while(true)` — et la méthode `borderSuivant()` est appelée à l'intérieur de cette boucle. Une attente passive est provoquée par `Thread.sleep()` pour que l'utilisateur ait le temps de percevoir les changements.

Afin de faciliter notre découverte des `Border`, nous avons choisi d'utiliser un `JLabel` central pour afficher le nom de la classe de `Border` en cours, au lieu d'afficher un texte constant.

Pour affecter un `Border` au `JLabel` nous utilisons la méthode `setBorder (Border)` définie sur `JComponent`. Pour changer le texte affiché par un `JLabel`, nous utilisons la méthode `JLabel.setText(String)`.

Le nom de la classe est obtenu comme suit :

```
Border borderCourant ;
[…]
String nomClasse = borderCourant.getClass().getName();
```

Digression sur le métamodèle

La méthode `getClass()` n'est pas une méthode de classe car elle s'applique sur une instance de `Border`. Mais sur quoi s'applique la méthode `getName()` ? Si vous manipulez bien les concepts objet, cette question peut vous sembler saugrenue, mais dans le cas contraire, ce petit paragraphe peut vous ouvrir des horizons. `getClass()` est une méthode définie sur la classe `Object`. Elle s'applique donc à toute instance. Le type retour de `getClass()` est `Class`. La méthode `getClass()` retourne donc une instance de la classe `Class`. La classe `Class` possède une méthode `String getName()` capable de retourner le nom d'une instance de la classe `Class`.

Prenons un petit exemple : supposons que nous disposons d'une classe `Point`.

```
Point p1;
// p1 est une référence de type Point
p1 = new Point();
// p1 est maintenant une référence sur une instance de
➥Point,
Class classe = p1.getClass();
```

```
// classe est une instance de la classe Class, cette
➥instance c'est Point.
String nom = classe.getName();
// Ici nom vaut " Point ".
```

À quoi cela peut-il bien servir ? Ce que nous venons de voir sont les bases du métamodèle. La classe `Class` est le point d'entrée de *l'introspection*. Ce terme désigne la capacité autodescriptive de certains objets. La classe `Class` contient de nombreuses méthodes liées à l'introspection, comme la méthode qui renvoie la liste des attributs.

Les classes de type `Border`

Revenons aux `Border`. Pourquoi chercher à afficher le nom de la classe du `border` que nous utilisons, n'est-ce pas `Border` tout simplement ? Les copies d'écran de l'exemple montrent que ce n'est pas le cas. `Border` est une interface Java que doivent implémenter toutes les classes graphiques qui correspondent à un cadre affichable autour d'un `JComponent`.

Ainsi, en tant qu'utilisateur externe des `Border` nous n'avons pas à savoir le nom de la classe qui implémente réellement l'affichage d'un `border`. Cela est rendu possible par le fait que toutes les méthodes de `JComponent` traitant des `borders` ont une signature paramétrique ne faisant intervenir que l'interface `Border`.

Le diagramme UML de la figure 1.47 montre l'organisation des classes du package `javax.swing.border`. Nous retrouvons l'interface `Border`. Une interface plus complète héritant de `Border` sert de base à toutes les classes qui contiennent le code d'affichage nécessairement spécifique à chaque type de border.

L'association entre `TitledBorder` et l'interface `Border` montre que la bordure avec titre utilise une autre instance de `Border` : il est possible de combiner un `TitledBorder` avec n'importe quelle autre classe de type `Border`. La classe `BorderFactory` dont nous avons parlé précédemment ne se trouve pas dans le package `javax.swing.border` mais dans le package `javax.swing`.

Programmation d'une attente

Notre exemple permet d'afficher successivement différents types de `Border`. Nous devons provoquer un temps d'arrêt entre chaque nouvel affichage sinon notre œil ne percevra rien. Comment programmer une attente ? Une première hypothèse à bannir absolument consiste à écrire une boucle vide. Un délai entre chaque affichage est alors provoqué par le temps nécessaire au processeur pour effectuer *n* « tours » de boucle à vide. Cette solution est très déconseillée, car elle utilise le processeur inutilement. Pour réaliser une tâche d'attente, il est préférable de « ne rien faire » plutôt que de « faire rien » ! D'autre part, le délai d'attente sera plus ou moins long suivant la puissance de la machine sur laquelle est exécuté le programme.

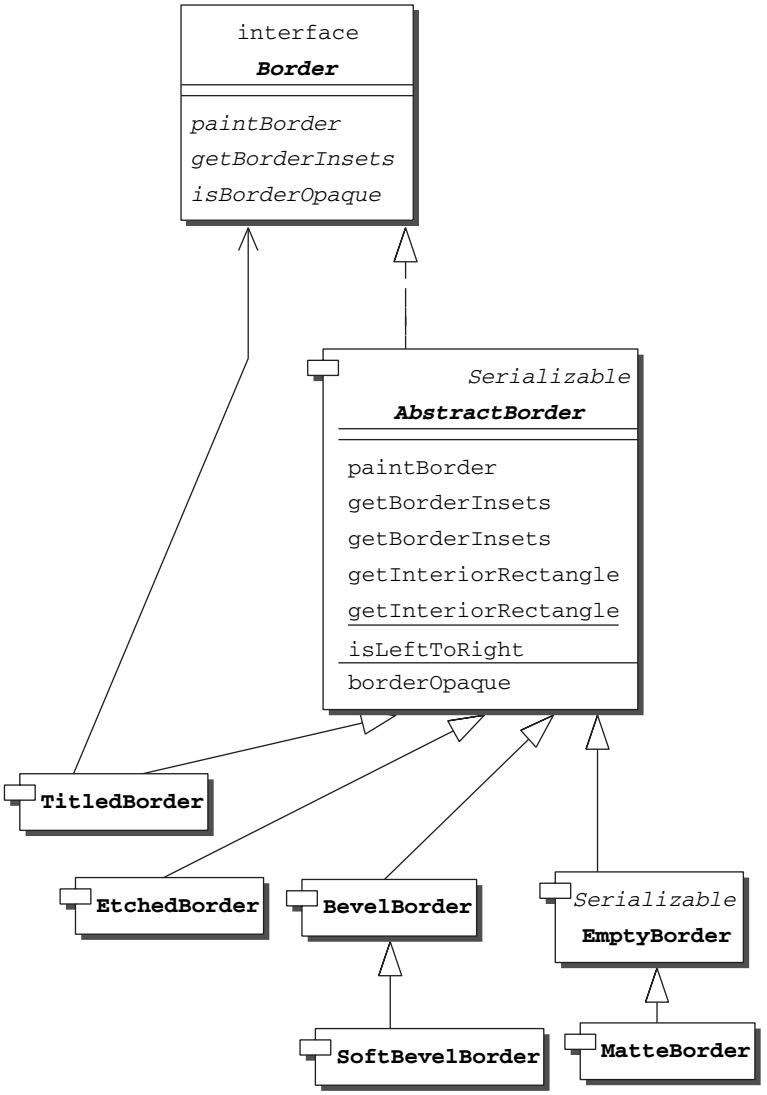


Figure 1.47 — Diagramme UML du package `javax.swing.border`.

Une seconde solution, bien plus appropriée, consiste à dire au CPU de ne rien faire. Pour cela, nous utiliserons la méthode `Thread.sleep(long t)` qui permet de suspendre le *thread* courant pendant `t` millisecondes. Il s'agit d'une méthode de classe.

Un *thread* pouvant être interrompu malencontreusement avant que la durée du `sleep` ne se soit écoulée, il est nécessaire de gérer une exception du type `InterruptedException`. En ce qui concerne notre exemple, cette interruption ne serait pas bien grave, c'est pourquoi il n'y a pas de code dans la section de traitement de l'exception.

1.4 LA BASE DES IHM : LES FENÊTRES

Reprenons notre tout premier exemple « Bonjour tout le monde ! ». La fenêtre graphique est représentée dans la classe `Bonjour`, par une instance de `JFrame`. `JFrame` est une classe représentant une fenêtre évoluée qui peut s'afficher sur le bureau graphique d'un système d'exploitation. Cette classe est évoluée car elle possède une barre de titre, elle est capable de se réduire, de se fermer et de se retailler.

Une instance de `JFrame` représente une fenêtre principale d'application. Les boîtes de dialogues sont des instances de `JDialog`. Une des différences entre ces deux classes est la possibilité pour `JDialog` de s'afficher de manière modale par rapport à une autre fenêtre.

Qu'est-ce qu'un affichage modal ?
Une fenêtre B est modale par rapport à une fenêtre A si l'affichage de B empêche l'accès — et non pas la vue — à la fenêtre A. Cela suppose que la fenêtre B s'affiche après la fenêtre A. Une fenêtre de login doit être modale par rapport à la fenêtre principale de l'application. Dans le cas contraire, le login serait absolument inutile !

Une fenêtre principale n'est donc jamais modale, les boîtes de dialogue peuvent l'être. Les fenêtres de différentes applications ne sont jamais modales les unes par rapport aux autres car il serait alors impossible d'exécuter plusieurs applications en même temps.

1.4.1 `JFrame` et `JDialog`

Peu de choses différencient les deux classes `JFrame` et `JDialog`. `JFrame` est une fenêtre principale alors qu'une `JDialog` est une fenêtre secondaire qui peut être modale par rapport à une autre `JDialog` ou à une `JFrame`.

`JDialog` semble être une fenêtre avec un comportement particulier. On s'attend donc à ce que la classe `JDialog` soit une sous-classe de `JFrame`. Ce n'est pas le cas, mais ces deux classes ont un ancêtre commun : la classe `Window`.

`Window` est une classe du package AWT qui représente une fenêtre sans bordure ni barre de titre, c'est l'équivalent de `JWindow` que nous venons de voir.

La classe `Dialog` correspond à la version AWT d'une boîte de dialogue. De même, la classe `Frame` représente la version AWT d'une fenêtre principale.

Voici en figure 1.48 un diagramme UML qui présente les relations entre ces classes.

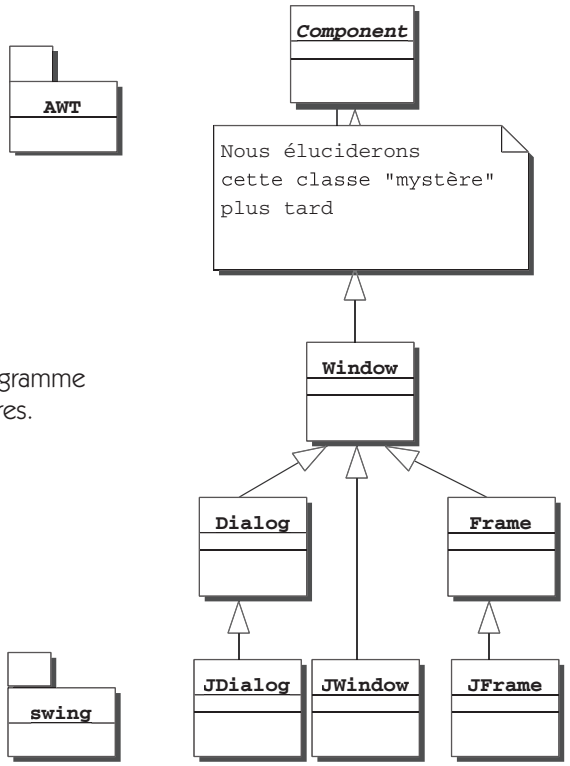


Figure 1.48 — Diagramme UML, les fenêtres.

Que peut-on faire avec une fenêtre ?

Pour afficher une fenêtre, on utilise la méthode `void SetVisible(true)`. Pour cacher une fenêtre, on utilise la méthode `void SetVisible(false)`.

Attention toutefois à ne pas confondre fermeture graphique d'une fenêtre et libération mémoire. La méthode `SetVisible(false)` ne libère pas la mémoire.

Pour libérer la mémoire occupée par une fenêtre et tout ce qu'elle référence, il faut utiliser la méthode `void dispose()`. La libération fonctionne avec le ramasse-miettes, *garbage collector*, de Java. Supposons une classe d'objet métier `OM`, suivons une instance `om` de cette classe dans l'exemple suivant :

Une méthode `afficheObjetMetier` :

```
afficheObjetMetier(OM objet) {
     JFrame fen = new JFrame();
     fen.ajoutObjetMetier(objet);
     fen.SetVisible(true);
     <…>
     fen.SetVisible(false);
     fen.dispose();
}
```

Supposons le code suivant :

```
OM om = new OM();
afficheObjetMetier(om);
```

À ce stade, l'instance référencée par `om` est toujours en mémoire bien que nous ayons appelé la méthode `dispose()` sur `fen` car `om` référence toujours l'instance de notre objet métier.

Qu'une fenêtre soit une instance de `JDialog` ou de `JFrame`, il est possible de définir un comportement lorsque l'utilisateur clique sur l'icône de fermeture. La méthode `setDefaultCloseOperation(int operation)` permet d'indiquer un comportement choisi parmi un ensemble prédéfini. Chaque comportement possible est spécifié par une constante.

Voici deux comportements simples :

- Ne rien faire :

```
setDefaultCloseOperation(
➥WindowConstants.DO_NOTHING_ON_CLOSE)
```

- Sortir de l'application par un `System.exit[…]` :

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
```

Nous verrons comment implémenter notre propre comportement dans le chapitre 3 sur les événements.

Il est possible de forcer une fenêtre à passer au premier plan avec la méthode `void toFront()`. De même, il est possible de la passer à l'arrière-plan avec la méthode `void toBack()`.

La taille d'une fenêtre se positionne comme pour un composant avec la méthode `void setSize(int largeur, int hauteur)`. Cette taille sera modifiable par l'utilisateur, sauf si la fenêtre a été spécifiée non redimensionnable à l'aide de la méthode `void setResizable(boolean b)`.

Pour positionner le texte contenu dans la barre de titre d'une `JDialog` ou d'une `JFrame`, on utilise la méthode `void setTitle(String titre)`.

1.4.2 Les menus

Une caractéristique importante de la `JFrame` est sa faculté de gérer une barre de menus (une instance de `JMenuBar`). Une barre de menus se compose comme indiqué figure 1.49.

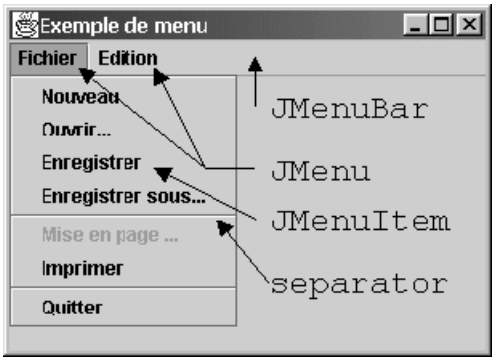


Figure 1.49 — Une barre de menu.

Comment est constitué un menu ?

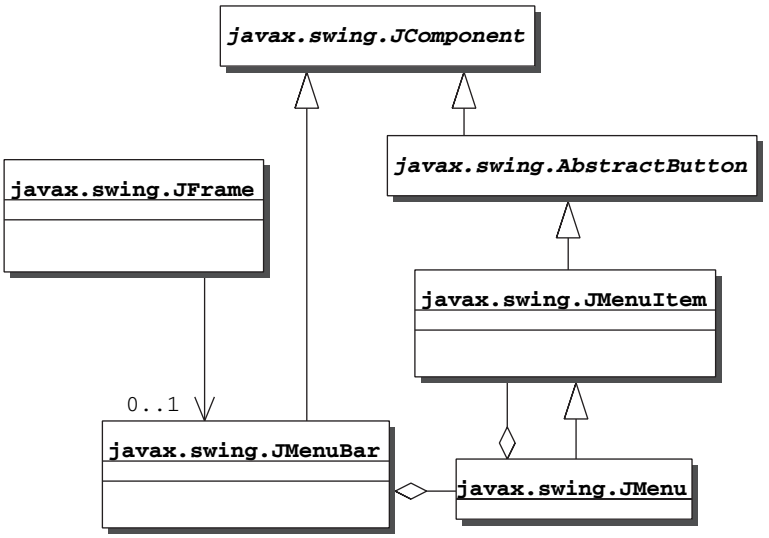


Figure 1.50 — Les classes Swing pour les menus.

Une instance de `JFrame` ne peut contenir qu'une seule instance de `JMenuBar`. En d'autres termes, une fenêtre n'a qu'une seule barre de menus. Une instance de `JMenuBar` est composée de plusieurs instances de `JMenu`. Ces instances représentent les éléments visibles directement dans la barre de menus. Enfin, chaque instance de `JMenu` peut contenir plusieurs instances de `JMenuItem`. Ces instances représentent les éléments visibles quand l'utilisateur clique sur un menu dans la barre de menus.

Notons que `JMenuBar`, `JMenuItem` et `JMenu` héritent de `JComponent`. De même les éléments de menu activables par l'utilisateur, `JMenu` et `JMenuItem`,

sont des sous-classes de `AbstractButton` de sorte qu'ils ont été conçus comme des « cas particulier de boutons ».

Techniquement, pour constituer un menu, on emboîte des instances les unes dans les autres. `JMenuBar` joue le rôle de container de `JMenu` et ce dernier joue le rôle de container de `JMenuItem` et de séparateurs.

La classe `JFrame` propose deux accesseurs pour son unique barre de menus :

```
public JMenuBar getJMenuBar()
public void setJMenuBar(JMenuBar menubar)
```

Ensuite, pour ajouter un `JMenu` à l'instance de `JMenuBar`, on utilise la méthode `add` :

```
public JMenu add(JMenu c)
```

Enfin, pour ajouter un `JMenuItem` à une instance de `JMenu` on utilise la méthode `add` :

```
public JMenuItem add(JMenuItem menuItem)
```

Pour ajouter un séparateur à une instance de `JMenu`, on utilise la méthode `addSeparator` :

```
public void addSeparator()
```

Voici le code permettant de réaliser la figure 1.49 :

```
import javax.swing.*;

public class SimplissimeMenu extends JFrame {

     public SimplissimeMenu() {
          setJMenuBar(getMenu());
          setDefaultCloseOperation(EXIT_ON_CLOSE);
          setTitle("Exemple de menu");
     }

     protected JMenuBar getMenu() {
          // Le menu fichier
          JMenuBar barreDeMenu = new JMenuBar();
          JMenu fichier = new JMenu("Fichier");
          JMenuItem nouveau = new JMenuItem("Nouveau");
          fichier.add(nouveau);
          […]
          fichier.addSeparator();
          JMenuItem miseEnPage = new JMenuItem("Mise
          ➥en page…");
          miseEnPage.setEnabled(false);
          fichier.add(miseEnPage);
          […]
          // Le menu édition
          JMenu edition = new JMenu("Edition");
```

```
            […]
            edition.addSeparator();
            JCheckBoxMenuItem retourLigne =
                new JCheckBoxMenuItem("Retour à la ligne
                ➥automatique");
            retourLigne.setState(true);
            edition.add(retourLigne);

            // La barre de menu
            barreDeMenu.add(fichier);
            barreDeMenu.add(edition);
            return barreDeMenu;
        }
    }
```

Pour ne pas surcharger le constructeur de la classe `SimplissimeMenu`, nous avons utilisé une méthode dédiée à la construction du menu. La méthode `getMenu()` retourne directement une instance de `JMenuBar` prête à être ajoutée à la fenêtre.

Notons un élément de menu particulier : `JCheckBoxMenu`. Cet élément présente une case à cocher et mémorise son état. Chaque clic de l'utilisateur inverse l'état de la case. Deux accesseurs gèrent l'accès à l'état de la case :

```
void setState(boolean etat);
boolean getState();
```

Voici figure 1.51 notre interface et son menu contenant une case à cocher.

Figure 1.51 — `JcheckBoxMenu`.

1.4.3 Un cas à part : `JWindow`

Reprenons par exemple le programme permettant de changer l'apparence du curseur. Que se passerait-il si nous changions la `JFrame` en `JWindow` (figure 1.52) ?

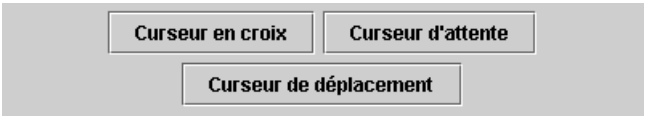


Figure 1.52 — Une `JWindow`.

L'intérieur de la fenêtre n'a pas changé. En revanche, la partie de la fenêtre dessinée par le système d'exploitation a disparu. Une `JWindow` est une sorte de `JFrame` sans « le pourtour ». Attention aux conséquences : cette fenêtre n'est ni redimensionnable ni déplaçable par l'utilisateur. Le plus troublant, d'un point de vue ergonomique, est qu'il n'y a pas de bouton système de fermeture de l'application.

C'est pour ces raisons qu'une `JWindow` ne doit pas être employée comme fenêtre principale d'application. Vous vous demandez peut-être à quoi sert donc cette classe ? `JWindow` est très utilisée pour coder un *spash screen*, cette image, parfois dotée d'une barre de progression, qui s'affiche au lancement d'une application. Il n'est pas possible de fermer cette fenêtre, mais comme l'application n'est pas encore initialisée, rien ne presse !

L'utilisation d'une `JWindow` demande quelques précautions. En particulier, il faut coder explicitement la taille et la position de la fenêtre car elle n'est ni déplaçable ni redimensionnable. D'autre part, la gestion des erreurs est plus importante que de coutume car il n'est pas possible de fermer une `JWindow`. Si l'application venait à se figer, l'utilisateur serait contraint de tuer le processus *via* le système d'exploitation.

La taille et la position d'une fenêtre se positionnent comme pour un composant avec les méthodes de la classe `Component`.

Pour la taille :

```
void setSize(int largeur, int hauteur)
```

Pour la position :

```
void setLocation(int x, int y)
```

ou

```
void setLocation(Point p)
```

Il n'est pas souhaitable de coder ces valeurs « en dur » car la taille et la résolution de l'écran peuvent varier d'une configuration à l'autre.

Il est possible d'obtenir la taille de l'écran en utilisant la classe `Toolkit`. Cette classe se trouve dans le package `java.awt`.

Première étape, obtenir l'instance par défaut :

```
Toolkit tk = Toolkit getDefaultToolkit();
```

Deuxième étape, obtenir la résolution de l'écran :

```
Dimension taille = tk.getScreenSize();
```

En calculant une taille et une position proportionnelles à la taille de l'écran, on obtient une fenêtre idéale quel que soit le contexte.

1.5 LES CONTAINERS

Nous avons vu les fenêtres qui sont à la base de l'interface, mais il nous manque une couche intermédiaire pour disposer les composants sur cette fenêtre, il s'agit, d'une façon très générale, des containers.

Finalement, une interface graphique Swing repose sur les concepts de bases suivants :

- Composants ;
- Fenêtre ;
- Container.

Mais qu'est-ce qu'un container ?

1.5.1 Qu'est-ce qu'un container ?

Pour répondre à cette question, commençons par une autre question. Comment avons-nous indiqué à la fenêtre l'existence du composant de type `JLabel` ?

L'instruction suivante s'en est chargée : `getContentPane().add(label);`

En traduisant presque mot à mot, nous avons ajouté un composant label à la fenêtre. Une fenêtre serait donc une sorte de boîte à composants ? Oui, le diagramme UML de la figure 1.53 peut nous le confirmer, `JFrame` hérite indirectement de `Container`.

Un container est donc un composant particulier (`Container` hérite de `Component`) dont le rôle est de contenir, comme son nom l'indique, d'autres composants. C'est à ce titre qu'une fenêtre est un container.

Notre classe « mystère » était donc `Container`.

Pourquoi n'y a-t-il pas de classe `JContainer` qui serait l'équivalent Swing de `Container` proposé par AWT ?

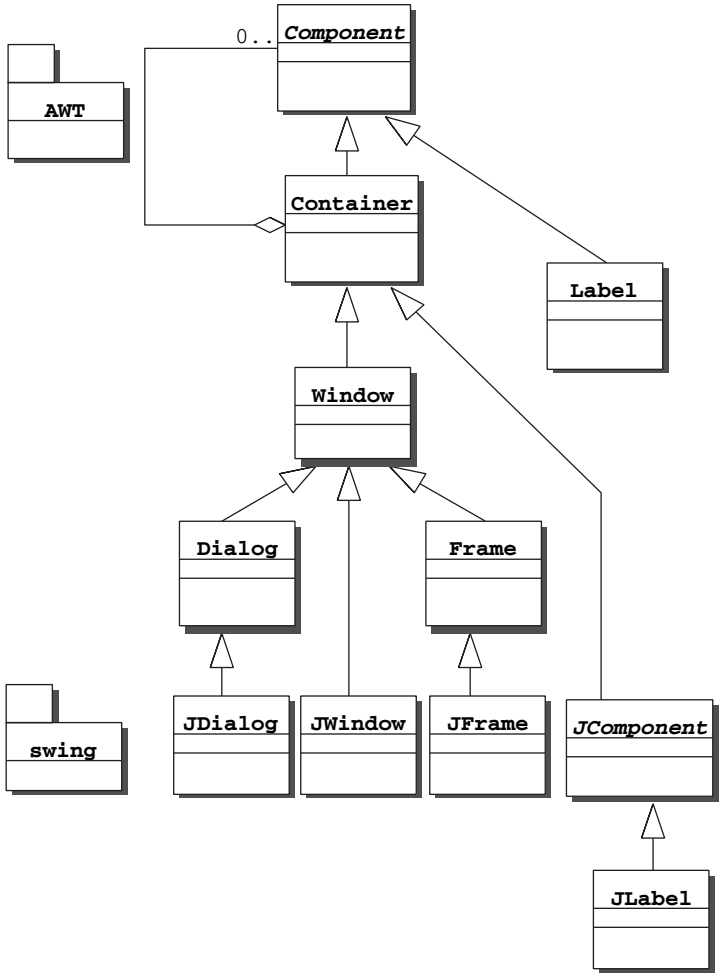


Figure 1.53 — Diagramme UML complet de composant, fenêtre et container.

La classe `JWindow` hérite de `Container`, le comportement (les méthodes) et les caractéristiques (les attributs) d'un container sont donc pleinement actifs dans une `JWindow` grâce à l'héritage. Notre éventuelle classe `JContainer` n'a donc pas lieu d'être.

La classe `Container` est très générique mais non abstraite et doit être capable de gérer une collection de `Component`.

Qu'en pensez-vous ? Ne restons pas dans le doute, allons voir ! Il est possible de consulter les sources de Java ! Voici quelques attributs qui nous intéressent de la classe `Container` issus des sources du JDK1.3 :

```
/**
 * The number of components in this container.
 * This value can be null.
 * @serial
 * @see getComponent()
 * @see getComponents()
      * @see getComponentCount()
      */
    int ncomponents;

    /**
     * The components in this container.
     * @serial
     * @see add()
     * @see getComponents()
     */
    Component component[] = new Component[4];
```

Java est livré avec ses sources. C'est une mine d'informations, prises, comme leur nom l'indique, à la source ! Nous vous recommandons vivement d'aller y puiser des informations.
Remarquez le nombre de lignes dans ces extraits de code : Deux ? Non. Douze ! Il y a bien 12 lignes de codes. Gardez toujours en tête que les commentaires sont aussi vitaux pour une application que les instructions Java.

On trouve un tableau de `Component` ainsi qu'un entier pour gérer la taille du tableau.

Ainsi, s'il existe une méthode `add` permettant l'ajout d'un composant dans un container, il doit bien exister une méthode pour retirer un composant d'un container. Certes, c'est la méthode `remove`. Cette méthode possède deux signatures paramétriques.

```
void remove(int)
void remove(component)
```

La première retire un composant à partir de son indice dans le tableau interne du container.

La seconde retire du tableau interne au container le composant dont l'instance est passée en paramètre.

La méthode `add()` existe avec de nombreuses variantes. Nous avons utilisé `add(Component)`, mais il existe une version, comme pour `remove()`, où l'on peut préciser la position du component dans le tableau du container :

```
Component : add(Component comp, int index)
```

Notons que la méthode `add` retourne toujours le composant que l'on vient d'ajouter.

La classe `Container` est un composant car elle hérite de `Component`. De ce fait, parmi l'ensemble des composants que contient un `Container`, il peut se trouver une instance de `Container`. Ainsi, il est possible de créer en mémoire des arborescences de composants à la manière du design pattern *composite*.

Définition
Un design pattern, ou modèle de conception, décrit une solution satisfaisante d'un point de vue objet (évolutive, réutilisable, intelligible) à un problème général de conception objet.

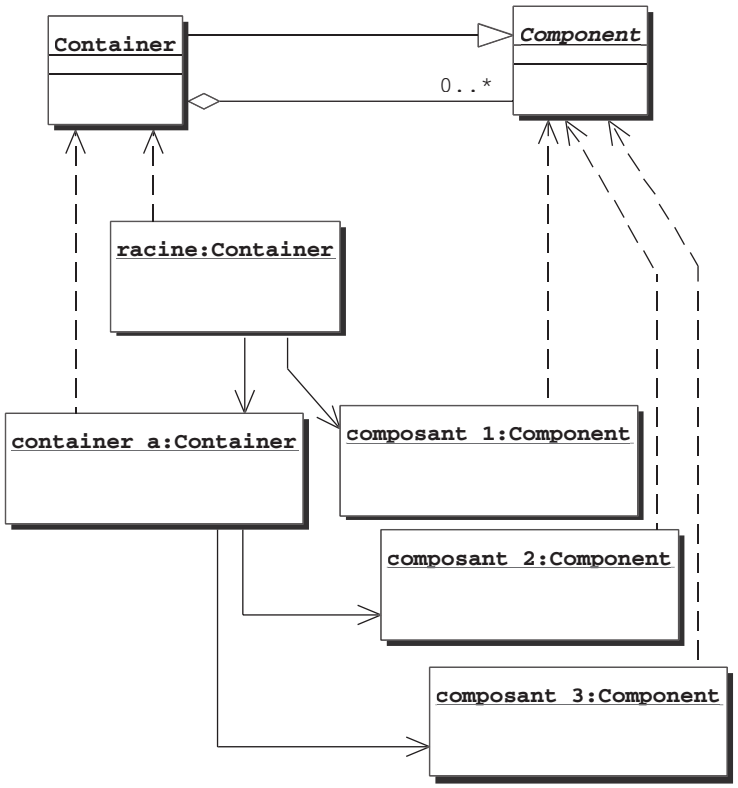


Figure 1.54 — Diagramme UML montrant des instances de `Container` et `Component`.

Racine et *container a* sont des instances de `Container`.

Composant 1 est une instance de `Component` (il peut être un `JTextField` ou un `JLabel` par exemple). Cette instance est liée à *racine*. Cela correspond dans le code à une ligne du type :

```
Racine.add(composant 1);
```

De même *container a*, en tant qu'instance d'une sous-classe de `Component`, est liée à *racine*. De même que pour *composant 1*, cela correspond dans le code à une ligne du type :

```
Racine.add(container a);
```

Ainsi de suite pour *composant 2* et *composant 3* qui sont des instances de `Component` liées à *container a*.

La classe `Container` est une classe non abstraite mais dont le niveau de généricité fait qu'elle n'est jamais instanciée pour être utilisée directement.

1.5.2 Le mécanisme de construction

La conception d'une interface graphique en Java ressemble un peu à un jeu de construction. En effet, cela consiste notamment à élaborer une arborescence de composants à l'aide de containers jusqu'à obtenir l'interface souhaitée. Ce principe n'est pas spécifique à Swing, il existait déjà avec AWT. Voyons comment procéder pour obtenir l'interface de la figure 1.55 :

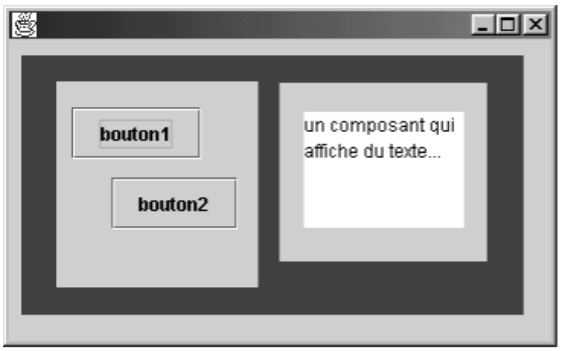


Figure 1.55 — La construction d'une interface quelconque.

Nous voyons sur cette fenêtre que plusieurs panneaux de différentes couleurs ont été disposés. Ces panneaux contiennent soit d'autres panneaux, soit des composants (figure 1.56). Les composants sont au nombre de trois sur cette interface : deux boutons et un composant texte. Ces composants ne peuvent pas contenir d'autres composants.

La figure 1.57 propose une représentation schématique de l'arborescence des composants.

Les panneaux sont donc des containers, des instances de `JPanel`. Comment aurions-nous codé cette arborescence ? Supposons que nous codions le constructeur de la fenêtre et que nous ayons les références suivantes : `bouton_1`, `bouton_2`, `panneauGauche`, `panneauDroite`, `texte`, `panneauGris`.

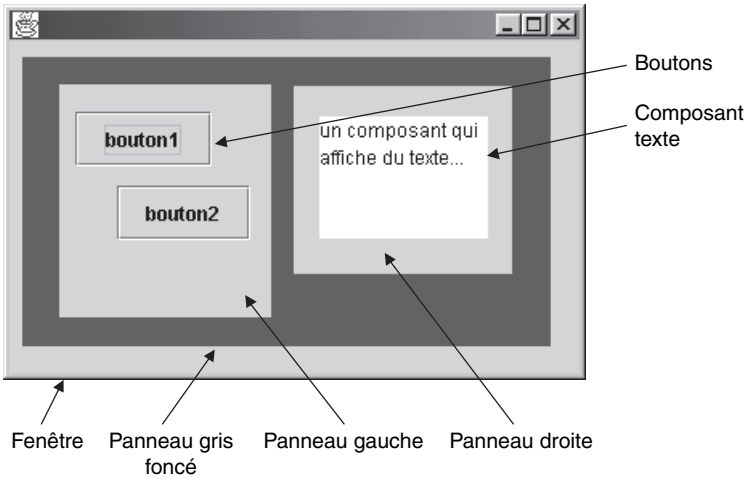


Figure 1.56 — Description de l'interface.

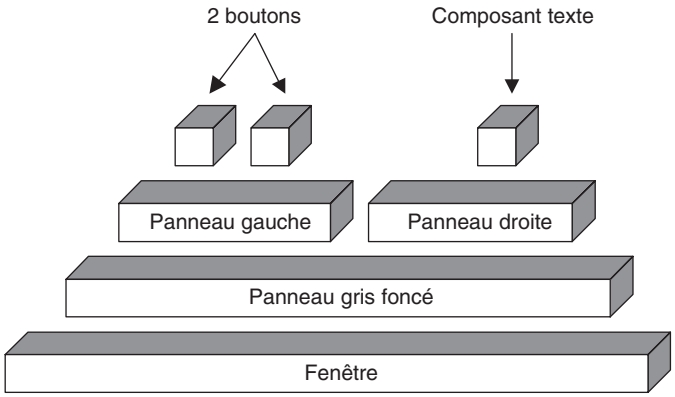


Figure 1.57 — Représentation de l'empilement des composants et conteneurs.

Voici comment nous construirions l'arborescence des composants :

```
//Commençons par la branche de gauche
panneauGauche.add(bouton_1);
panneauGauche.add(bouton_2);
// l'autre branche maintenant
panneauDroite.add(texte) ;
// Le panneau fédérateur
panneauGris.add(panneauGauche);
panneauGris.add(panneauDroite);
// La racine
getContentPane().add(panneauGris);
```

1.5.3 Différents types de container

Un container racine : `JFrame`

Une interface graphique est donc une arborescence de composants dont les feuilles sont des composants graphiques du type `JComponent` et les nœuds des containers. La racine est forcément un container particulier : une fenêtre ou une boîte de dialogue.

Pour une fenêtre principale Swing : `JFrame`, le container racine est le `contentPane`. C'est pourquoi pour ajouter le `JLabel` de notre exemple `Bonjour`, la ligne de code est :

```
getContentPane().add[…]
```

La méthode `getContentPane()` est donc l'accesseur au container racine que possède la fenêtre.

Rentrons maintenant dans les détails. Une fenêtre est un composant plus complexe qu'un simple container qui serait seulement un tableau de composants. Ainsi, une fenêtre `JFrame` peut avoir un menu. Ce menu, `JMenuBar` composé de `JMenu`, est un composant lui aussi. Quel est son container ? Certainement pas le `contentPane` dont le rôle est de contenir les composants d'une fenêtre, de plus le menu doit se maintenir à une position bien précise dans la fenêtre : en haut. Une fenêtre `JFrame` est en fait composée de plusieurs containers : le `rootPane`, le `contentPane` et le `glassPane`. Nous avons déjà évoqué le `contentPane`. Le `rootPane` est le véritable container racine de la fenêtre. C'est lui qui contient les autres containers dans une arborescence construite en mémoire sur le même principe que le pattern *composite*. Le `glassPane` est un container particulier qui se tient devant le `contentPane`.

Nous avons vu qu'une fenêtre ou une boîte de dialogue sont notamment des containers racine. Nous allons voir maintenant des exemples de containers « nœud ».

Une classe incontournable : `JPanel`

Un container de base abondamment utilisé est `JPanel`. Cette classe hérite de `JComponent` et à ce titre est un container (`JComponent` hérite indirectement de `Container`). L'aspect composant d'un `JPanel` est réduit au strict minimum au sens où une instance de `JPanel` n'a que peu d'impact graphique. `JPanel` est donc le nœud idéal dans une arborescence de composants.

La représentation graphique d'un `JPanel`, bien que peu importante, n'est tout de même pas inexistante. En particulier, un `JPanel` peut être perçu conceptuellement comme une sorte de rectangle, parfois invisible graphiquement, dans lequel on ajoute des composants.

Il est possible de modifier la couleur de fond d'un `JPanel`, sa taille, sa position, son état : visible ou invisible, désarmé ou armé (`disable`). Nous connaissons déjà

les méthodes qui permettent ces modifications : ce sont celles que nous avons exposées sur `JComponent` – `JPanel` hérite de `JComponent`.

Comment ajouter un composant dans un `JPanel` ? Cela se fait très simplement, à l'aide de la méthode `add[…]` héritée de la classe `Container`.

Un container avec défilement : `JScrollPane`

`JPanel` n'est pas le seul container fourni avec le package `javax.swing`. La classe `JScrollPane` est un container très utile dès lors que le composant qu'il contient a une taille telle qu'il n'est pas affichable dans sa totalité. Un exemple typique est un document textuel ou graphique dont la taille peut aisément excéder la taille d'une fenêtre ou même d'un écran. C'est le cas du document dans un logiciel de traitement de texte ou d'une image dans un logiciel de traitement graphique.

Ainsi le *scrolling*, ou déplacement dans une zone, n'est pas à la charge du composant, mais du container. Le container `JScrollPane` construit une vue, qui est fixe, sur un document constitué par un composant pouvant être plus grand que la vue. Souvent, le composant qui est placé dans un `JScrollPane` est lui-même un container.

Prenons l'exemple d'un `JLabel`, que nous utiliserons avec du texte au format HTML sur plusieurs lignes. Nous imposerons à la fenêtre, `JFrame`, une taille volontairement inférieure à la taille du composant, mais nous insérerons celui-ci dans un `JScrollPane`. Il sera donc nécessaire d'utiliser les ascenseurs pour faire déplacer le `JLabel`, le `JScrollPane` proposant ainsi une vue sur le `JLabel`.

Instancions nos deux composants, le `JLabel` et le `JscrollPane` :

```
JFrame fenetre = new JFrame();
MonLabel label = new MonLabel(texte);
JScrollPane sp = new JScrollPane();
```

Ensuite, il suffit d'affecter à la fenêtre le `JscrollPane` :

```
fenetre.getContentPane().add(sp);
```

et enfin, indiquer au `JScrollPane` qu'il doit former une vue sur le `JLabel` :

```
sp.setViewportView(label);
```

Après affichage de la fenêtre, nous pouvons constater le rôle que joue le `JScrollPane` si la fenêtre est plus petite que le `JLabel` (figure 1.58).

Quelques remarques sur les particularités de la classe `JScrollPane` en tant que container. Contrairement au `JPanel`, un `JScrollPane` n'accepte qu'un seul composant, mais si celui-ci est lui-même un container, par exemple un `JPanel`, il est possible de faire défiler dans la vue un ensemble de composants. Enfin, le `JScrollPane` a un impact graphique plus important que le `JPanel` puisqu'il affiche des ascenseurs.

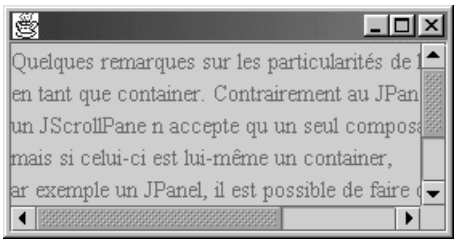


Figure 1.58 — Un exemple de `JScrollPane`.

Certains composants ont une prédisposition à être contenus dans un `JScroll Pane`. Nous aborderons ultérieurement les composants texte qui représentent fréquemment plusieurs lignes de texte, formaté ou non. `JLabel` est un composant qui est rarement utilisé dans un `JScrollPane`. Ainsi, devons-nous lui adjoindre cette capacité pour notre exemple.

Pour être géré par un `JscrollPane`, un composant doit implémenter l'interface `Scrollable` définie dans le package `javax.swing`.

Les composants dont nous évoquions la prédisposition à s'afficher dans un `JScrollPane` implémentent en standard cette interface. Dans notre exemple nous utiliserons une sous-classe de `JLabel`, `MonLabel`, dont la seule plus-value par rapport à `JLabel` est l'implémentation de `Scrollable`.

Étudions de plus près les méthodes que nous propose l'interface `Scrollable`.

```
Dimension getPreferredScrollableViewportSize ()
```

Le but de cette méthode pour un composant qui implémente l'interface `Scrollable` est de retourner la taille idéale du composant à l'intérieur du `JScrollPane`. Selon que cette taille est inférieure ou supérieure à la taille réelle du `JScrollPane`, le composant interne, aussi appelé la vue, sera entièrement visible ou non. Dans ce dernier cas, les ascenseurs seront actifs et permettront un déplacement de la vue.

Cette vue est en fait une instance de `JViewport`. Mais dans la majorité des cas, vous n'aurez pas à manipuler vous-mêmes cette classe.

Dans notre cas, nous renvoyons la taille idéale du `JLabel`. L'implémentation de cette méthode dans `MonLabel` est donc très simple :

```
return this.getPreferredSize();
```

Une autre méthode de l'interface Scrollable est :

```
int getScrollableBlockIncrement(Rectangle visibleRect,
➥int orientation, int direction)
```

Cette méthode a pour objectif de permettre au composant géré par le `JScroll Pane` d'intervenir sur l'ampleur du déplacement par bloc lorsque l'utilisateur clique entre la cage de l'ascenseur et les flèches se trouvant à l'extrémité (figure 1.59).

Figure 1.59 — Déplacement par bloc.

Pour permettre au composant de calculer le nombre de pixels à déplacer, la méthode fournit des informations. Ces informations peuvent nous sembler superflues dans le cas de notre sous-classe de `JLabel`, mais si nous avions une représentation graphique avec des colonnes dont la taille peut varier, comme une page d'un tableur, il serait nécessaire de recalculer à chaque clic la quantité de pixel pour chaque déplacement. Il serait ainsi possible de se déplacer colonne par colonne même si chaque colonne a sa propre largeur.

Le paramètre `visibleRect` est une instance de la classe Rectangle qui nous donne la taille réelle de la vue. Nous avons déjà vu les classes `Point` et `Dimension`. Conceptuellement un rectangle est un point et une dimension. La classe `Rectangle` représente, non graphiquement, un rectangle, c'est-à-dire une origine, le point en haut à gauche, ainsi qu'une longueur et une largeur. La classe dispose de méthodes très utiles, comme le calcul du rectangle résultant du chevauchement de deux rectangles.

Les informations sur le composant lui-même peuvent être obtenues *in situ* car cette méthode est implémentée par le composant.

Le paramètre orientation peut prendre l'une des deux valeurs suivantes : `SwingConstants.VERTICAL ou SwingConstants.HORIZONTAL`

Quant au paramètre direction, il est :

- positif si le déplacement s'effectue vers le bas ou vers la droite, selon la valeur de `orientation` ;
- négatif si le déplacement s'effectue vers le haut ou vers la gauche, selon la valeur de `orientation`.

Ainsi, il est possible de calculer l'incrément du déplacement par bloc en connaissant, avant que le déplacement ne soit effectif, la nature du déplacement demandé par l'utilisateur.

En ce qui concerne notre `JLabel`, fixons arbitrairement cet incrément à 10 pixels :

```
return 10;
```

Une autre méthode est :

```
int getScrollableUnitIncrement(Rectangle visibleRect,
➥int orientation, int direction)
```

Cette méthode est tout à fait comparable à la précédente. Son objectif est de permettre de fixer le nombre de pixels pour un déplacement unitaire, lorsque l'utilisateur clique sur l'une des deux flèches de la barre d'ascenseur (figure 1.60).

Figure 1.60 — Déplacement unitaire.

En ce qui concerne notre `JLabel`, fixons arbitrairement cet incrément à 1 pixel :

```
return 1;
```

Les deux méthodes suivantes ont pour objectif de définir une stratégie d'affichage lorsque la taille de vue est plus importante que la taille du composant géré par le `JScrollPane` :

```
boolean getScrollableTracksViewportHeight()
boolean getScrollableTracksViewportWidth()
```

Si l'une des méthodes retourne `true`, alors le composant se voit proposer par le container (`JScrollPane`) un espace dans lequel il peut s'étendre si cela fait partie de ses capacités. Dans le cas de notre `JLabel`, cette faculté ne nous intéresse pas, ces méthodes retournent donc `false` dans la classe `MonLabel`.

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

public class MonLabel extends JLabel implements Scrollable {

     public MonLabel(String texte)   {
        super(texte);
     }

     public Dimension getPreferredScrollableViewportSize(){
      return this.getPreferredSize();
     }

     public int getScrollableBlockIncrement(Rectangle
     ➥visibleRect, int orientation, int direction)   {
         return 10;
     }

    public boolean getScrollableTracksViewportHeight()  {
      return false;
```

```
        }

        public boolean getScrollableTracksViewportWidth()  {
          return false;
        }

        public int getScrollableUnitIncrement(Rectangle
        ➥visibleRect, int orientation, int direction)  {
          return 1;
        }
}
```

Un panneau à onglets : `JTabbedPane`

Le container `JTabbedPane` permet d'afficher plusieurs panneaux qui partagent tous le même espace. On accède à chaque panneau à l'aide d'onglets souvent situés en haut du container. L'utilisation de ce container se fait de la façon suivante :

```
JTabbedPane panOnglets = new JTabbedPane() ;
```

Il est possible de préciser en paramètre du constructeur la position des onglets à l'aide d'une constante parmi `TOP`, `BOTTOM`, `LEFT` ou `RIGHT`.

Les différents panneaux sont ensuite ajoutés au container à l'aide de la méthode `addTab`. On indique également la chaîne de caractères à afficher sur l'onglet et éventuellement une icône :

```
panOnglets.addTab("Onglet 1", pan1) ;
```

`JSplitPane`

Le container `JsplitPane` permet de séparer son contenu en deux zones distinctes dont les surfaces respectives peuvent changer dynamiquement. L'espace total alloué aux deux zones reste constant, mais la proportion qu'occupe chaque zone est variable. Ce container s'utilise tout à fait classiquement, il peut contenir n'importe quel composant, mais il ne dispose que de deux « places ». Nous l'avons utilisé dans l'exemple sur les signets pour séparer l'arborescence à gauche du panneau d'information à droite (figure 1.61). Ce container est souvent utilisé à cette fin, pour implémenter une sorte de « browser ».

Le `JSplitPane` se trouve dans le package `javax.swing`. Il dispose de deux orientations, verticale et horizontale. Voici le code d'initialisation de ce container dans l'exemple d'application de gestion de signets :

```
splitPanel.setLeftComponent(panArbre);
splitPanel.setRightComponent(panDetail);
splitPanel.setContinuousLayout(true);
splitPanel.setOneTouchExpandable(true);
```

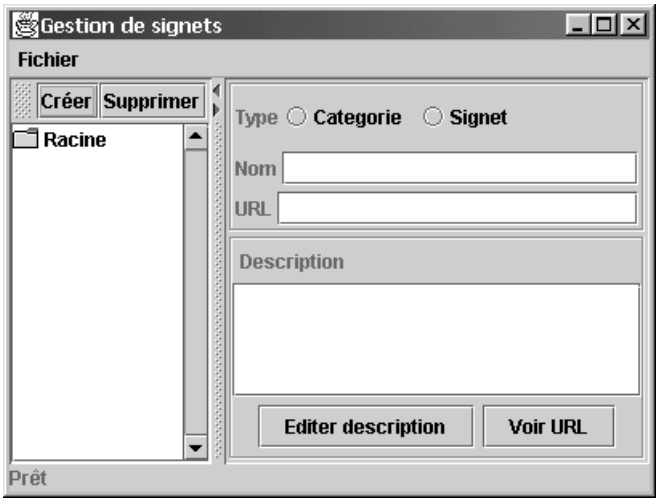


Figure 1.61 — L'usage de `JSplitPane` dans l'application de gestion de signets.

Deux panneaux sont positionnés respectivement à gauche et à droite, à l'aide des méthodes `setLeftComponent` et `setRightComponent`. C'est ainsi que le container définit son orientation. Il existe donc deux autres méthodes, non utilisées ici, `void setTopComponent(Component comp)` et `void setBottomComponent (Component comp)`.

Le paramètre `oneTouchExpandable` permet de conditionner l'existence de deux petits triangles en haut de la barre de séparation. Ces touches permettent d'étendre la partie gauche ou droite d'un seul coup.

Enfin, le paramètre `continuousLayout` permet de modifier la taille des parties gauche et droite dynamiquement, en même temps que l'utilisateur bouge la barre de séparation.

Un container à part : `JDesktopPane`

`JDesktopPane` est un container de haut niveau. Comme son nom l'indique en anglais (il signifie bureau), il permet d'obtenir un effet semblable au bureau d'un système d'exploitation graphique. Il ne contient donc que des fenêtres particulières qui peuvent évoluer en son sein. On entend par « évoluer » le fait de se déplacer, de s'icônifier, de s'agrandir… Ce container se trouve dans le package `javax.swing` ainsi que son composant de prédilection : `JInternalFrame`.

L'utilisation d'un `JDesktopPane` est assez simple, pour ajouter une fenêtre interne il suffit d'utiliser la méthode `add` (figure 1.62).

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

public class Fenetre extends JFrame {
      JDesktopPane bureau = new JDesktopPane();
      JInternalFrame f1 = new JInternalFrame
      ➥("Bloc-notes", true, true, true, true);
      JInternalFrame f2 = new JInternalFrame
      ➥("Calculatrice", false, true, false, true);
      JInternalFrame f3 = new JInternalFrame("Jeux de
      ➥taquin", true, true, true, true);

      public Fenetre() {
            getContentPane().setLayout(new BorderLayout());
            getContentPane().add(bureau, BorderLayout
            ➥.CENTER);
            setSize(450, 500);
            bureau.add(f1);
            bureau.add(f2);
            bureau.add(f3);
            f1.setVisible(true);
            f2.setVisible(true);
            f3.setVisible(true);
            f1.setSize(250, 300);
            f1.setBackground(Color.lightGray);
            f2.setSize(200, 200);
            f2.setBackground(Color.white);
            f3.setSize(200, 200);
            f3.setBackground(Color.black);
            setTitle("JBureau à tout faire");
      }

      public static void main(String args[]) {
            Fenetre f = new Fenetre();
            f.SetVisible(true);

      }
}
```

Une fenêtre d'application `JFrame` est sous-classée et le `contentPane` est positionné avec un `JDesktopPane`. Ce container est dédié à des composants de type fenêtres internes : `JInternalFrame`. Cette classe de fenêtre ressemble à une `JFrame`. Elle dispose de méthodes de placement sur l'axe des z, c'est-à-dire qu'il y a une gestion de plan dans le container `JDesktopPane`. Il est donc possible de placer une fenêtre au premier plan en utilisant la méthode `toFront()` ou à l'arrière-plan avec la méthode `toBack()`. De même qu'une `JFrame`, la classe `JInternalFrame` possède une gestion d'état : icônifiée, taille quelconque ou maximisée. Ces possibilités sont activables avec les méthodes `void setResizable (boolean b)`, `void setMaximizable(boolean b)` et `void setIconifiable (boolean b)`. Le constructeur de `JInternalFrame` permet de positionner ces valeurs à l'aide de booléens.

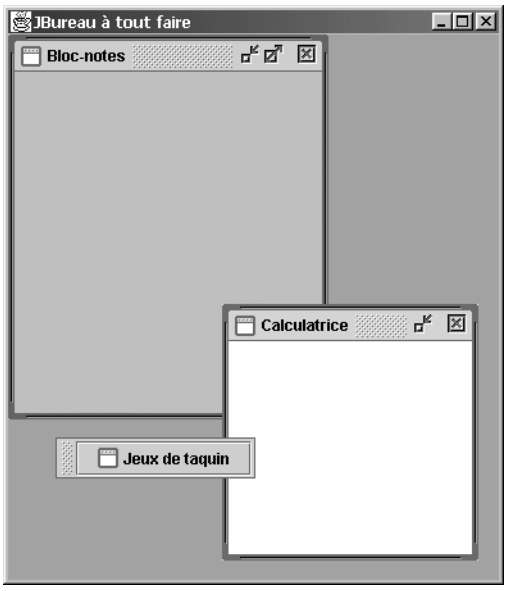


Figure 1.62 — Exemple d'utilisation d'un `JDesktopPane`.

1.5.4 Performances

Une des étapes les plus coûteuses lors de l'exécution d'un programme Java est l'instanciation. On doit tenir compte de cette constatation dans le développement des interfaces graphiques car les ressources mobilisées par le graphisme sont importantes. De ce fait, il est primordial de prendre cela en compte assez tôt. Une stratégie qui a fait ses preuves consiste à réutiliser les instances dès que cela est possible.

Le problème est souvent de déterminer les instances à réutiliser. Il faut définir une granularité correcte. Par exemple, placer la granularité de réutilisation des instances au niveau d'un composant simple comme un `JTextField` est très probablement trop fin. Les containers en tant que nœuds sont de bons candidats. En effet, conserver une instance d'un container maintient en mémoire l'arborescence dont il est la racine locale. Nous avons vu qu'il est possible de rendre visible ou invisible un container. Dès qu'un container est rendu invisible, c'est toute l'arborescence qu'il contient qui devient invisible. Naturellement, il ne faut pas systématiser ce système et conserver en mémoire tous les containers de l'application.

Réutiliser un container signifie changer son contexte de données. Il faudra donc prévoir de conserver en mémoire des méthodes d'initialisation et de paramétrage qui soient en dehors du constructeur.

Ce système peut présenter l'inconvénient de consommer plus de temps lors de la première utilisation du container car c'est à ce moment que se déroulent les instanciations en cascades des composants gérés par le container. Dans ce cas, il est toujours possible de préinstancier certains containers pendant l'écran d'affichage du logo de l'application. Tout ceci concerne aussi les boîtes de dialogue, car ce sont aussi des containers.

Il faut donc gérer un compromis désormais classique dans le domaine des performances : le compromis taille mémoire/performance. Pour économiser la mémoire il est possible, si deux boîtes de dialogues ne diffèrent que par quelques composants, de n'en coder qu'une seule avec des composants visibles ou invisibles selon son type.

Un autre aspect lié aux performances est la volumétrie des informations à afficher. Ce peut être, par exemple, une boîte de dialogue comportant une liste de milliers d'éléments ou une arborescence constituée de nombreux nœuds. Dans ces situations, il faut utiliser la ruse. Pourquoi charger, traiter et afficher toutes les informations d'un coup alors que l'utilisateur ne percevra que ce qui est visible ? La volumétrie des données est telle que l'utilisateur n'aura qu'une vue sur cet ensemble de données. Cette vue sera probablement manipulable par des ascenseurs. Il est donc possible de charger très vite la première « page » de données puis en tâche de fond, de s'occuper du reste. On peut aussi envisager d'avoir toujours une page d'avance. Par exemple si l'utilisateur consulte la page *i*, on va charger en mémoire les pages $i - 1$, i et $i + 1$. Dans le cas d'une arborescence, seuls les nœuds visibles sont chargés et non pas l'arbre entier dans toute sa profondeur ou sa largeur.

1.6 GESTION DE L'APPARENCE

1.6.1 La classe `Graphics`

`Graphics` est une classe qui permet de dessiner. Nous avons vu jusqu'à maintenant comment utiliser des composants du package `javax.swing`. Il est possible de paramétrer suffisamment les composants Swing pour subvenir aux besoins les plus courants. Tous les besoins ne peuvent cependant se satisfaire des composants existants. En particulier, comment tracer un trait, un cercle, un arc de cercle… dans une interface ? Il n'y a pas de composant *`JZoneDeDessin`*. La classe `Graphics` répond à ce besoin. Vous l'aurez compris, il n'est pas question d'interactions avec l'utilisateur, la classe `Graphics` va nous permettre de programmer des dessins, mais ces derniers ne seront pas des composants. Il n'est par exemple pas possible d'ajouter un cercle dessiné avec `Graphics` dans un container.

Nous verrons comment utiliser `Graphics`, mais comment récupérer une instance de `Graphics` ? C'est en effet la première étape.

Chaque composant doit savoir se dessiner. Même si le processus de dessin fait appel à des classes abstraites pour factoriser du code, il arrive un moment où il faut tracer des rectangles, les remplir de couleur… Cela se fait en utilisant `Graphics`. Chaque composant possède donc une instance de `Graphics`. Cette instance de `Graphics` possédée par un composant peut être vue comme une zone « vide », sur laquelle il est possible de dessiner des formes élémentaires comme des rectangles… Pour obtenir une instance de `Graphics` sur une instance de `Component`, il faut utiliser la méthode `Graphics getGraphics()`.

Nous voici donc munis d'une instance de `Graphics`. Cette instance représente un rectangle dans lequel il est possible de dessiner. La taille de ce rectangle est la taille réelle du composant. Peut-être vous demandez-vous comment dessiner sur toute la surface d'une interface ? N'oubliez pas que les containers sont des composants. Ainsi, si vous définissez une interface comme étant une `JFrame` munie d'un `JPanel` occupant toute la surface de la fenêtre, alors vous pouvez obtenir une instance de `Graphics` auprès de ce `JPanel`.

`Graphics` propose un système de coordonnées géométriques orthogonales dont l'origine se situe en haut et à gauche du rectangle dont nous venons de parler.

Traçons un cercle de rayon 50 pixels.

Une première version :

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

public class Testgraphics {
     public static void main(String args[]) {
           JFrame fenetre = new JFrame();
           fenetre.setSize(300, 300);
           JPanel pan = new JPanel();
           fenetre.getContentPane().add(pan);
           fenetre.SetVisible(true);
           Graphics g = pan.getGraphics();
           g.drawOval(10, 10, 50, 50);
     }
}
```

Les méthodes fournies par la classe `Graphics` sont d'assez bas niveau et peu pratiques pour le développeur. Par exemple, il existe une classe `Point` permettant de représenter un point, mais les méthodes de `Graphics` ne prennent pas d'instance de `Point` en paramètres. Ces méthodes prennent souvent quatre entiers en remplacement de `Point`.

Il n'y a pas non plus de méthode directe pour dessiner un cercle, il faut dessiner un ovale dont les deux diamètres sont égaux. Ce sont les deux derniers paramètres qu'attend la méthode `drawOval`. Les deux premiers paramètres précisent les coordonnées supérieures gauches du rectangle, invisible, qui encadre le losange. Ainsi notre cercle aura un diamètre de 50 pixels et sera inscrit dans un carré dont le point en haut à gauche se situera en (10, 10) dans le système de coordonnées du `Graphics`, et ayant des côtés de 50 pixels.

Tout compile bien, mais on ne voit pas de cercle ! En fait, en faisant bien attention, on peut observer fugitivement un cercle. Que s'est-il passé ? Nous avons négligé Swing. Les composants sont rafraîchis par Swing et la méthode `main` n'est bien sûr pas réexecutée à des fins de rafraîchissement ! Il faut garder le principe du dessin, mais procéder différemment. Nous allons faire une sous-classe de `JPanel` et redéfinir la méthode qui est appelée par Swing pour dessiner le composant qu'est notre `JPanel`. Cette méthode, c'est `paintComponent(Graphics g)`.

Deuxième version, voici la sous-classe de `JPanel` :

```
public class MonPanel extends JPanel {
     public void paintComponent(Graphics g) {
           super.paintComponent(g);
           g.drawOval(10, 10, 50, 50);
     }
}
```

Voici la classe de test :

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

public class Testgraphics {

     public static void main(String args[]) {
           JFrame fenetre = new JFrame();
           fenetre.setSize(200, 150);

           JPanel pan = new MonPanel();
           fenetre.getContentPane().add(pan);
           fenetre.SetVisible(true);
     }
}
```

Notre sous-classe redéfinit la méthode `paintComponent`, mais, par précaution, elle utilise tout de même la méthode `paintComponent` de `JPanel` : `super.paint(g)`. Ensuite, on dessine le cercle comme précédemment (figure 1.63).

Cette fois, nous obtenons le résultat escompté car chaque fois que Swing décide de rafraîchir tout ou partie de l'affichage et que le `JPanel` est redessiné, notre méthode `paintComponent` est appelée.

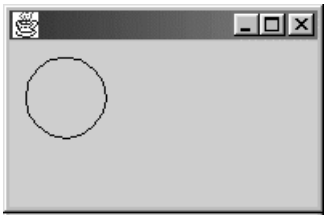


Figure 1.63 — Dessin d'un cercle avec `Graphics`.

La classe `Graphics` permet donc d'accroître l'ouverture de Swing. Avec une sous-classe de `JComponent` et `Graphics`, vous pouvez construire vos propres composants si sous-classer ceux qui existent déjà se révélait insuffisant.

D'un point de vue infographique, la classe `Graphics` est cependant assez limitée. Sun a donc développé des classes supplémentaires : `Java2D`. Le « point d'entrée » de cette librairie est la classe `Graphics2D` qui hérite de `Graphics`.

Encore une fois la question est : comment obtenir une instance de `Graphics2D` ? La réponse est simple : en forçant le type `Graphics2D`.

```
Graphics g = […]
Graphics2D g2 = (Graphics2D)g;
```

Nous n'étudierons ici qu'un seul aspect de `Java2D` : les transformations affines. Ces transformations d'un point de vue géométrique se définissent comme conservant le parallélisme et les proportions, comme les homothéties et les translations.

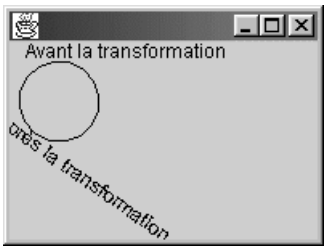


Figure 1.64 — Dessins avec transformation.

Il est possible de construire une transformation affine en instanciant la classe `AffineTransform` et en fournissant au constructeur toutes les valeurs d'une matrice mathématique qui défini la transformation. Mais il y a un moyen plus simple pour les besoins courants. Il existe des méthodes statiques sur la classe `AffineTransform` elle-même qui fournissent des instances de `AffineTransform` toutes prêtes pour des transformations courantes, comme des rotations… Nous n'essayerons pas de faire subir une rotation à notre cercle, nous ne verrions pas le résultat ! Ajoutons donc un texte (figure 1.64). Pour dessiner une chaîne de caractères on utilise la méthode `drawString(String texte, int x, int y);`

La sous-classe de `JPanel` :

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

public class MonPanel extends JPanel {
     private static final AffineTransform trans =
    ➥ AffineTransform.getRotateInstance((3.14/5), 30, 30);

     public void paintComponent(Graphics g) {
          super.paintComponent(g);
          Graphics2D g2 = (Graphics2D)g;
          g2.drawString("Avant la transformation", 10, 10);
          g2.setTransform(trans);
          g2.drawString("Après la transformation", 10, 70);
          g2.drawOval(10, 10, 50, 50);
     }
}
```

La classe de test :

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

public class Testgraphics {

     public static void main(String args[]) {
          JFrame fenetre = new JFrame();
          fenetre.setSize(200, 150);

          JPanel pan = new monPanel();
          fenetre.getContentPane().add(pan);
          fenetre.SetVisible(true);
     }
}
```

Notez le nouvel import : `java.awt.geom`. Ce package contient la classe `AffineTransform` dont nous avons besoin.

Nous avons défini notre transformation comme un attribut de la classe `TestGraphics` pour éviter de la recalculer à chaque fois que Swing appelle la méthode `paintComponent`. Cette transformation est obtenue avec la méthode statique `getRotateInstance`. Le premier paramètre est l'angle de rotation en radian, les deux suivants sont les coordonnées du point autour duquel la rotation doit s'effectuer. Notez que l'ordre est important dans la méthode `paintComponent`. La transformation ne s'applique qu'après `setTransform`.

1.6.2 Paramétrer le look and feel

Un *look and feel* peut se définir comme un type de rendu graphique. En d'autres termes, une même interface aura une apparence différente selon le *look and feel* qu'elle utilise. Un changement de *look and feel* n'impacte ni la nature, ni la position des composants, mais seulement le rendu visuel de chaque composant. Cette fonctionnalité n'a pas d'équivalent AWT, c'est une des principales différences avec Swing.

Changer de *look and feel* est particulièrement simple. Une classe est responsable du *look and feel* : `UIManager`, elle se trouve dans le package `javax.swing`. Les méthodes à utiliser pour positionner un *look and feel* sont : `setLookAndFeel (String nomDeLaClasse)`, `setLookAndFeel(LookAndFeel newLookAndFeel)`, ce sont des méthodes statiques de `UIManager`.

Deux autres méthodes statiques de `UIManager` permettent d'obtenir respectivement le nom de la classe de *look and feel* portable de Java, aussi appelé *metal*, et celui propre au système.

Pour obtenir le nom de la classe de *look end feel* système :

```
String UIManager.getSystemLookAndFeelClassName()
```

Pour obtenir le nom de la classe de *look end feel* portable :

```
String UIManager.getCrossPlatformLookAndFeelClassName()
```

Comment est déterminé le *look and feel* à utiliser s'il n'a pas été positionné explicitement, comme nous l'avons vu plus haut ?

Dans ce cas fréquent, le *look and feel* est défini dans le fichier `swing.properties` que le `UIManager` va chercher dans le répertoire d'installation du JDK. Appelons ce répertoire `JAVA_HOME`. Le fichier doit être dans `JAVA_HOME/lib`. La clé suivante détermine le *look and feel* chargé par défaut : `swing.defaultlaf`. Par exemple, pour imposer un *look and feel* MOTIF, le fichier doit contenir la ligne suivante :

```
swing.defaultlaf=
➥com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Si rien n'est spécifié dans le fichier ou s'il est inexistant, alors le *look and feel* Java (*metal*) est chargé.

Les classes de *look and feel* les plus courantes sont :

```
String motif =
➥"com.sun.java.swing.plaf.motif.MotifLookAndFeel";
```

```
String win =
➥"com.sun.java.swing.plaf.windows.WindowsLookAndFeel";
String metal = "javax.swing.plaf.metal.MetalLookAndFeel";
```

Le paramètre de la méthode `setLookAndFeel` doit vous rappeler une digression sur le métamodèle présenté plus haut dans ce chapitre. En effet, cette méthode attend comme paramètre le nom d'une classe et non pas une instance.

Grâce au métamodèle, il est possible de créer une instance d'une classe en ne manipulant que son nom. Deux phases se succèdent pour obtenir ce résultat :

- chargement de la classe dans la machine virtuelle Java ;
- instanciation.

Le premier point utilise la méthode `Class.forName(String nomClasse)`, où `nomClasse` est le nom de la classe à chercher dans le classpath et à charger dans la machine virtuelle Java :

```
Class maNouvelleClasse =
➥Class.forName(" com.toto.titi.maClasse ");
```

Le deuxième point n'est pas plus complexe, il utilise la méthode `newInstance()` :

```
Object monInstanceDeMaNouvelleClasse =
➥maNouvelleClasse.newInstance() :
```

Tout cela montre que la classe `UIManager` fonctionne comme un chargeur de classes particulier. Cela pourrait permettre de modifier le *look and feel* sans recompiler l'application et en utilisant des classes de *look and feel* qui n'étaient pas connues *a priori*.

Cette architecture n'a de sens que si le composant délègue le dessin à une classe externe à la manière du pattern *state*.

Le schéma de principe de la figure 1.65 montre comment marche la délégation.

Une instance de container possède une collection, ou plus précisément un tableau, de `JComponent`. Une méthode définie sur `JComponent` permet de déclencher le dessin, elle est appelée en boucle par le container sur les composants qu'il gère à chaque fois qu'il faut effectuer un affichage. Nous avons vu au paragraphe précédent que ce mécanisme utilise la classe `Graphics`. Cette méthode de dessin est ensuite redéfinie au niveau des sous-classes, comme sur `JButton`. Elle calcule des paramètres, comme des couleurs, des positions, des polices, … puis elle appelle `getUI().dessin(this)`, ce qui déclenche la méthode `dessin` sur l'instance de `ChouetteBoutonUI` retournée par `getUI()`.

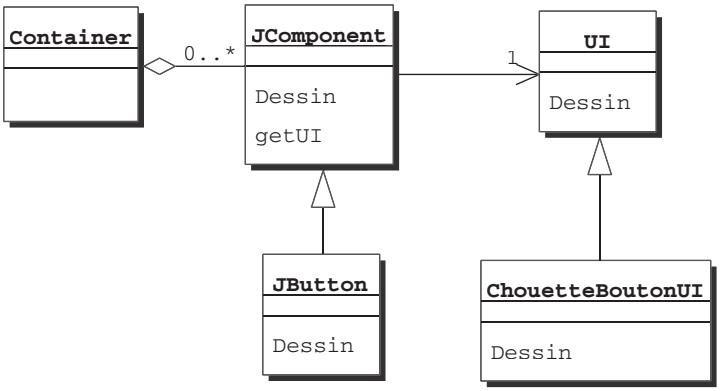


Figure 1.65 — Délégation du dessin.

Ce système, variant du pattern *state*, permet de déléguer le dessin à des sous-classes de `UI`. La méthode `dessin` de la classe `ChouetteUI` a un paramètre de type `JButton` – nous avions passé à cette méthode une instance du composant à dessiner. Dans le code de cette méthode `dessin` de `ChouetteBoutonUI`, on accède aux données qui paramètrent le graphisme et on effectue le dessin proprement dit en utilisant notamment la classe `Graphics`.

Naturellement, il faut autant de sous-classes qu'il y a de composants à dessiner.

En réalité, les choses sont un peu plus complexes. Voici présenté figure 1.66 le diagramme réel mais nous n'envisagerons ici qu'une explication de principe sur la délégation. Il est intéressant d'en comprendre les principes car c'est sur la délégation que repose un des points forts de Swing dont nous parlions dans l'introduction de ce chapitre : la portabilité graphique.

La partie gauche du diagramme montre une arborescence qui nous est familière : `JButton` hérite indirectement de `JComponent`.

La partie droite nous montre l'arborescence d'héritage d'un *look and feel* classique : *metal*. Ce *look and feel* est lié à un ensemble de `ComponentUI`. La classe `MetalLookAndFeel` bénéficie de cette relation portée par la classe `LookAndFeel` par héritage. De fait, `MetalLookAndFeel` est notamment composé de `MetalButtonUI`. C'est cette dernière qui est responsable du dessin d'un `JButton` dans le *look and feel metal*.

Comme nous l'avions dit, c'est un peu plus complexe que le principe que nous avons évoqué précédemment, mais le principe est là. Finalement, charger un *look end feel* « truc » revient à :

- charger la classe `TrucLookAndFeel` dans la machine virtuelle Java ;
- initialiser cette classe et charger les classes du type Truc<composant>UI ;
- enfin, déporter le dessin d'un `JButton` sur la classe `TrucButtonUI`.

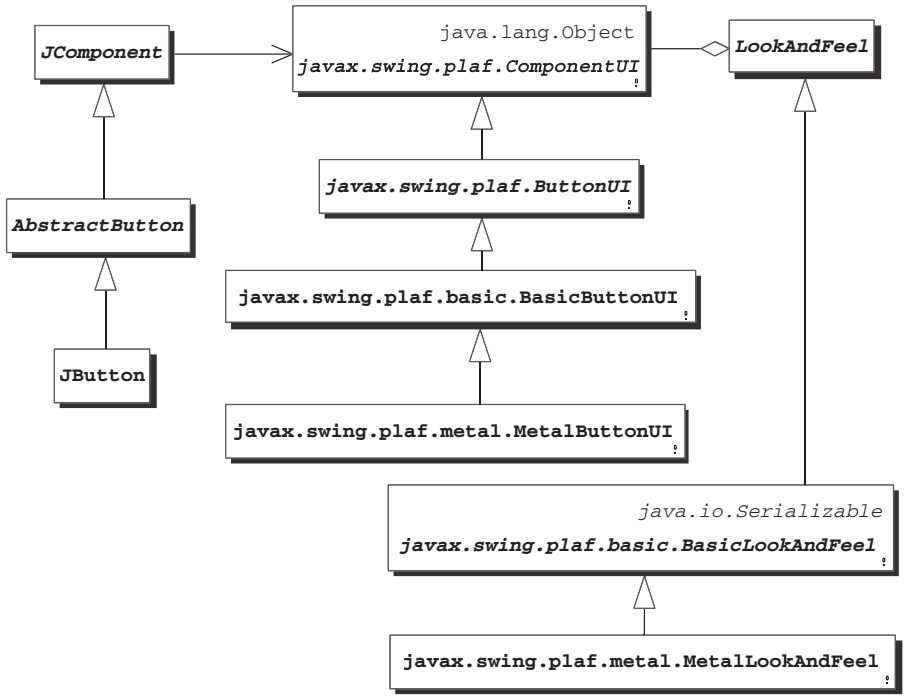


Figure 1.66 — Délégation complète du rendu graphique d'un `Jcomponent`.

1.7 UN EXEMPLE COMPLET : UNE APPLICATION DE GESTION DE SIGNETS

Tout au long de cet ouvrage, nous allons appliquer les notions présentées à un exemple complet. Cet exemple consiste en une application pour gérer les signets (aussi appelés favoris, ou en anglais *bookmarks*) d'un utilisateur.

En effet, vous avez sans doute constaté qu'il est peu commode de récupérer des signets définis sur un poste avec un navigateur donné, depuis un autre poste, éventuellement sous une autre plate-forme ou un autre navigateur.

Pour répondre à ce manque, nous avons imaginé vous faire construire cette petite application conjointement à la lecture de l'ouvrage. Cette application a pour objectif de permettre la sauvegarde de signets dans un fichier non dépendant d'une plate-forme, et donc la récupération de la liste des signets de n'importe où.

Dans ce chapitre, vos connaissances étant encore peu poussées, nous allons nous contenter de décrire précisément le cahier des charges, de construire le cadre général de l'interface et de réaliser la couche non graphique de l'application.

1.7.1 Présentation du cahier des charges

Le but est de pouvoir sauvegarder une liste de signets, ainsi que leur organisation en catégories et leurs descriptions.

Pour cela, nous avons imaginé une interface structurée comme en figure 1.67.

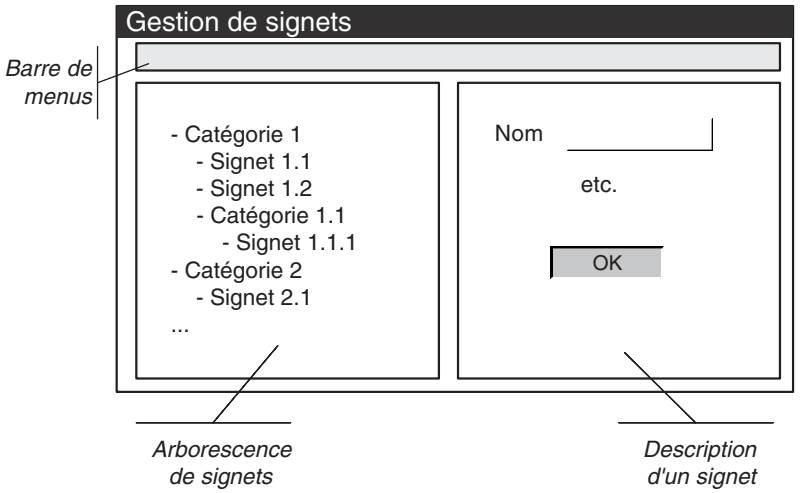


Figure 1.67 — Structure de l'application de gestion de signets.

La sélection d'un signet ou d'une catégorie dans l'arbre devra provoquer l'affichage de la description dans le panneau de droite. Un bouton permettra le lancement de cette page HTML dans un interpréteur afin de visualiser au moins la première page du site favori.

Il sera possible de créer et de supprimer des catégories ou des signets. Il sera aussi possible de modifier l'arborescence des signets.

Toutes ces informations seront stockées dans un fichier particulier sur le disque. Un menu permettra d'ouvrir une liste de signets depuis un fichier choisi sur le disque, et de sauver l'ensemble des signets dans un fichier.

1.7.2 Organisation du code en packages

Étant donné l'ampleur de cet exemple, nous allons organiser proprement notre code. En particulier, nous allons disposer les classes dans des packages, afin de ne pas mélanger des classes de natures complètement différentes.

Nous allons créer trois packages (figure 1.68) :

- Les classes directement liées à l'IHM : package `ihm` ;
- Les classes correspondant à la couche métier : package `metier` ;
- Les classes implémentant des traitements applicatifs : package `applicatif`.

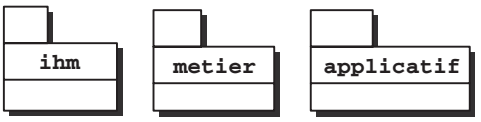


Figure 1.68 — L'organisation du code en packages.

1.7.3 Les composants graphiques à utiliser

En l'état actuel de nos connaissances, nous ne sommes pas en mesure de construire cette interface. Nous pouvons cependant lister les composants que nous savons devoir utiliser (figure 1.69).

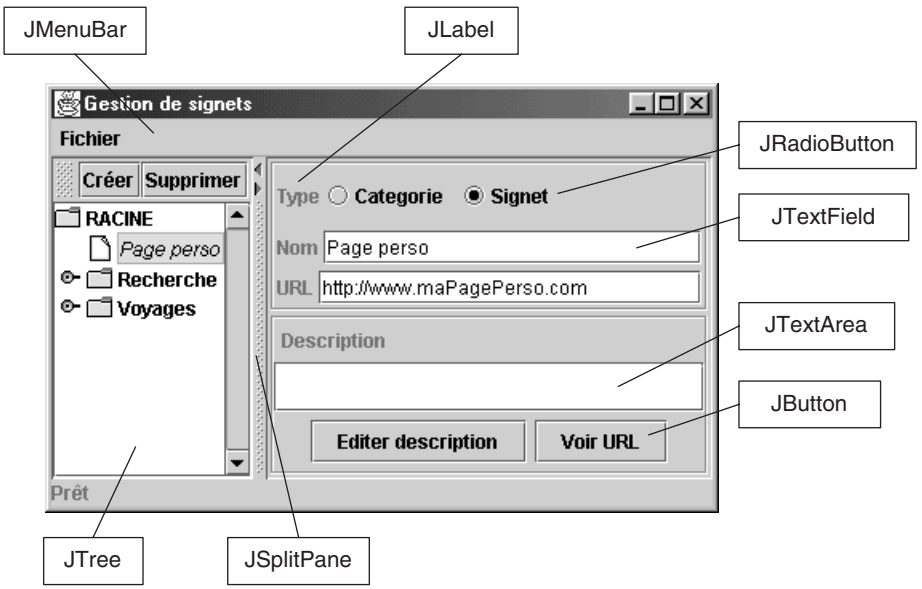


Figure 1.69 — Composants à utiliser.

Nous pouvons d'ores et déjà construire la fenêtre principale et la barre de menus. Pour cela, mettons en application ce que nous avons vu précédemment.

Il faut écrire une classe contenant la méthode `main` qui est appelée au lancement du programme et qui instancie notre fenêtre personnalisée. Il faut créer une classe héritant de `JFrame`, que nous allons nommer `FenetrePrincipale`.

La classe de lancement de l'application se trouve dans le package par défaut, et son code est le suivant :

```
import java.awt.Dimension;
import ihm.FenetrePrincipale;
import applicatif.Test;

/**
 * Classe de lancement de l'application
 */
public class Application {

   public static void main(String[] args) {
      FenetrePrincipale fenetre =
      ➥FenetrePrincipale.getInstance();
      fenetre.setSize(new Dimension(400,200));
      fenetre.SetVisible(true);
   }
}
```

La classe `FenetrePrincipale` n'est utilisée qu'une seule fois. Nous pouvons en faire un singleton, c'est-à-dire une classe qui ne peut avoir qu'une seule instance.

Dans la classe `FenetrePrincipale`, il faut créer un menu. Le menu « Fichier » est composé des items suivants :

- Ouvrir ;
- Enregistrer ;
- Quitter.

Voici donc le code que nous pouvons dès maintenant écrire :

```
package ihm;

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

```
import applicatif.*;
import javax.swing.tree.*;

/**
 * Fenêtre principale de l'application.
 * Cette fenêtre comprend un panneau divisé en 2 parties :
 * arbre des signets et description d'un signet.
 * Elle comprend aussi une barre de menus et une barre
 * d'état.
 *
 * Cette classe est un singleton : elle possède au maximum
 * une seule instance.
 */
public class FenetrePrincipale extends JFrame {

   // Variables statiques
   private static final String TITRE = "Gestion de signets";
   // Instance unique de la classe
   protected static FenetrePrincipale instance = null;

  /**
   * Cette méthode renvoie l'instance unique si elle existe
   * et la crée si nécessaire.
   */
   public static FenetrePrincipale getInstance() {
     if (instance == null)  {
              instance = new FenetrePrincipale();
     }
     return instance;
  }

  /**
   * Constructeur protégé
   */
  protected FenetrePrincipale() {
     initialiserMenus();
     setTitle(TITRE);
  }

  /**
   * Crée et affecte les menus à la fenêtre.
   *
   */
  protected void initialiserMenus() {
     JMenu menuFichier = new JMenu("Fichier");
     // menu ouvrir : pour charger depuis un fichier
     ➥l'arbre des signets
     JMenuItem ouvrir = new       JMenuItem("Ouvrir");
     menuFichier.add(ouvrir);
```

```
        // menu enregistrer : pour enregistrer dans un fichier
         ➥l'arbre des signets
        JMenuItem enregistrer = new  JMenuItem("Enregistrer");
         menuFichier.add(enregistrer);
         // menu quitter
         JMenuItem quitter = new      JMenuItem("Quitter");
         menuFichier.addSeparator();
         menuFichier.add(quitter);
         JMenuBar mb = new JMenuBar();
         mb.add(menuFichier);
         setJMenuBar(mb);
    }
}
```

1.7.4 La couche métier

Même si ce sujet n'entre pas vraiment dans le cadre de l'ouvrage, nous allons présenter la conception des objets métier. En effet, cela est nécessaire pour pouvoir implémenter la suite de l'exemple.

La description du besoin fonctionnel nous amène à penser que deux types principaux d'objets métier existent : les signets et les catégories.

Les signets et les catégories sont caractérisés par un nom et une description. Les signets possèdent en plus une URL (figure 1.70).

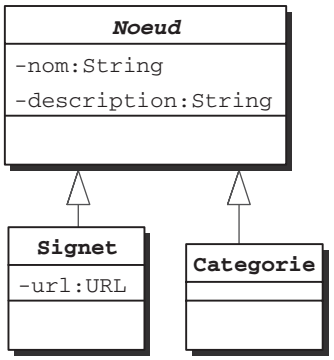


Figure 1.70 — Diagramme de classe des objets métier.

Dans cette application, l'organisation des signets entre eux (le fait qu'un signet soit situé dans telle ou telle catégorie) est représentée grâce à l'objet graphique « arbre ».

Nous allons modifier légèrement cette hiérarchie pour répondre à la problématique de la sérialisation.

La sérialisation est l'opération qui consiste à faire passer un graphe d'objets — c'est-à-dire plusieurs objets ayant des relations entre eux — dans un flux d'octets. Ici, nous allons utiliser ce mécanisme de sérialisation pour écrire dans un fichier les informations à sauvegarder. L'opération inverse, la désérialisation permet de réinstancier des objets à partir d'un flux.

Pour sauvegarder toutes les informations nécessaires, nous avons le choix entre :

- sérialiser les objets métier ;
- sérialiser les objets attachés à l'objet graphique « arbre » (ce sont des objets de type `TreeNode`, du package `javax.swing.tree`)

Il est toujours mieux de ne sérialiser que l'information nécessaire et suffisante. La sérialisation des objets du JDK peut causer des problèmes car elle ne stocke pas la définition de la classe. Le modèle des données n'est pas stocké avec les données. Par conséquent :

- pour les classes du JDK, les données sérialisées avec un JDK peuvent ne pas se désérialisées avec un autre JDK ;
- pour les classes utilisateurs, les données sérialisées sont susceptibles d'être perdues entre deux compilations. Pour une sérialisation, une compilation correspond à une modification du modèle de données.

Finalement, sérialiser nos classes nous permet d'exécuter l'application avec différentes versions du JDK, cela n'impactera pas la persistance des données.

La première solution semble donc plus satisfaisante. Toutefois, si nous choisissons de sérialiser des objets métier, cela nous oblige à stocker l'information concernant l'arborescence au niveau des objets métier. Or, cette information est obligatoirement présente dans les objets graphiques `TreeNode`. Devons-nous maintenir cette information en double ? Non, bien sûr. La solution que nous vous proposons, et qui reste simple tout en remédiant à ces inconvénients est la suivante :

1. Nous allons manipuler des objets métier `Signet` et `Categorie` sans aucune relation hiérarchique entre eux.
2. Lors de la sérialisation, nous obtenons des objets métier à partir de l'arbre, mais cette fois, nous restaurons des objets portant eux-mêmes l'information sur la hiérarchie.
3. Nous sérialisons les objets métier.

Pour ce faire, nous avons donc le diagramme des classes métier présenté figure 1.71.

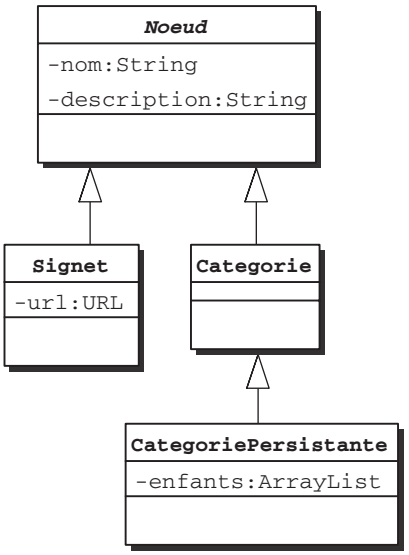


Figure 1.71 — Diagramme des classes métier.

Les opérations de sérialisation et de désérialisation s'effectuent à l'aide d'une classe utilitaire que nous avons appelée `GestionnaireNoeuds`. Cette classe est un singleton.

L'opération de sérialisation s'effectue en plusieurs temps :

1. Obtention du nœud racine de l'arbre.
2. Parcours de l'arborescence à partir de ce nœud racine. Modification des objets métier associés pour leur donner l'information sur cette arborescence. Cela est fait grâce à la méthode `enregistrer`, qui est appelée de façon récursive.
3. Sérialisation de l'objet métier `CatégoriePersistante` qui est racine de l'arbre.
4. Tous les nœuds sont sérialisés automatiquement car ils sont reliés entre eux par une relation de type « enfant ».

Le code complet de l'application est disponible sur le site de Dunod (www.dunod.com).

2

Les layouts

Reprenons la définition d'une interface graphique que nous avions donnée au début de l'ouvrage. Visuellement, une interface graphique est essentiellement une combinaison d'éléments graphiques comme des boutons, des listes ou des champs.

Nous avons vu quelques composants Swing dans le chapitre précédent. Une question importante maintenant est « comment positionner les composants les uns par rapport aux autres ? » Plus précisément, les composants sont placés logiquement dans un container, mais : « comment un container agence-t-il visuellement les composants qu'il contient ? »

En effet, jusqu'à présent, nos interfaces graphiques ne comportaient qu'un seul composant. Le rôle des layouts est précisément la gestion de la position des composants.

Une question se pose souvent à ce stade : pourquoi la gestion de la position des composants est-elle déléguée à un layout, alors que le positionnement semble si simple ? Détrompez-vous : le positionnement est tout sauf simple. Que se passe-t-il par exemple si j'agrandis ma fenêtre ? Les composants doivent-ils s'agrandir ? Si oui, doivent-ils tous s'agrandir ? Si non, où ajoute-t-on de l'espace ? On pressent la nécessité de définir des règles concernant le positionnement. Les layouts implémentent une stratégie de positionnement.

Un abus de langage fréquent consiste à parler de composants « dans » un layout. Il faut être conscient que cela ne reflète pas la réalité : un composant est contenu dans un container. Ce dernier s'attache les services d'un layout pour le positionnement des composants dont il a la charge.

Les différentes stratégies de positionnement doivent prendre en compte la taille et la position des composants qui lui sont confiés. Rappelons qu'il y a deux tailles pour un composant. La taille effective : *size* et la taille idéale : *preferredSize*. Un layout va essayer de rendre la taille réelle la plus proche possible de la taille idéale compte tenu des contraintes dues à sa stratégie et dues à la taille réelle du container. Par exemple, si la taille du container est plus petite que la somme des tailles des composants, il y aura des conflits à résoudre. De même, si la taille du container est plus grande que la somme des tailles des composants, que fait-on de l'espace disponible ? Comment le répartir ?

De multiples stratégies sont implémentées par les différents layout disponibles.

2.1 LES LAYOUTS LES PLUS COURANTS

2.1.1 FlowLayout

Présentation

Une stratégie simple et très utile est un placement des éléments graphiques les uns à côté des autres, de gauche à droite.

Dans un `FlowLayout`, les composants n'auront jamais une taille réelle plus grande que leur taille idéale. Autrement dit, si la place disponible est trop faible pour contenir tous les composants, alors le `FlowLayout` va retailler chaque composant pour qu'ils soient tous visibles. En revanche, s'il y a trop de place, le `FlowLayout` va tout simplement laisser de l'espace libre, sans retailler les composants.

Un `FlowLayout` est paramétrable, les composants qui lui sont confiés peuvent être alignés à gauche, à droite ou centrés. Cela se définit dans le constructeur du `FlowLayout` qui prend une constante entière dont les valeurs valides sont :

```
FlowLayout.LEFT
FlowLayout.RIGHT
FlowLayout.CENTER
```

On instancie un `FlowLayout` comme suit :

```
FlowLayout monLayout = new FlowLayout(FlowLayout.CENTER);
```

Supposons que mon container soit un `JPanel`. Comment lui indiquer la stratégie de positionnement choisie ?

```
JPanel monPanel = new JPanel();
monPanel.setLayout(monLayout);
```

Dorénavant, les composants contenus dans le container `monPanel` se positionneront avec la stratégie implémentée par `monLayout`, un `FlowLayout`.

L'ajout des composants dans le container se fait toujours par la méthode `add`. Par exemple, pour ajouter deux boutons :

```
JButton bouton1 = new JButton[…];
JButton bouton2 = new JButton[…];
monPanel.add(bouton1);
monPanel.add(bouton2);
```

Du fait du layout, l'ordre dans lequel les composants sont ajoutés est important. Dans notre cas, le `bouton1` sera à gauche du `bouton2` et les deux boutons seront au centre du container.

Essayons de remplir un `JPanel` associé à un `FlowLayout` avec des `Jbutton` :

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

public class Fenetre extends JFrame {
     // Nombre de boutons dans la fenêtre
     protected static final int NB_COMP = <un nombre>;
     protected static final int XMAX = 400;
     protected static final int YMAX = 100;
     protected FlowLayout layout =
     ➥ new FlowLayout (FlowLayout.CENTER);

     public Fenetre() {
           getContentPane().setLayout(layout);
           ajouteComp(NB_COMP);
           setSize(XMAX, YMAX);
     }

     protected void ajouteComp(int nombre) {
           for (int i = 0; i < nombre; i++)  {
                 getContentPane().add(
                 ➥ new JButton("Ceci est le
                 ➥ bouton " + i));
           }
     }
}
```

La classe `Fenetre` est une `JFrame` dont on impose la taille : (400, 100). Une boucle crée des instances de `JButton` et les ajoute dans le container de base d'une `JFrame` : le `contentPane`. Le `contentPane` est le `JPanel` de haut niveau d'une `JFrame`. On en obtient une référence avec la méthode `getContentPane()`. Notons comment l'on précise au `JPanel` sa stratégie de positionnement :

```
getContentPane().setLayout(layout);
```

Dans une première série de tests, nous ferons varier le paramétrage de justification du `FlowLayout`. Nous ne remplirons le container qu'avec deux boutons, alignés au centre (figure 2.1).

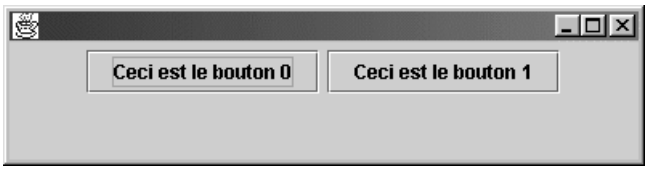


Figure 2.1 — `FlowLayout` avec alignement au centre.

Voici figure 2.2 la version alignée à gauche.

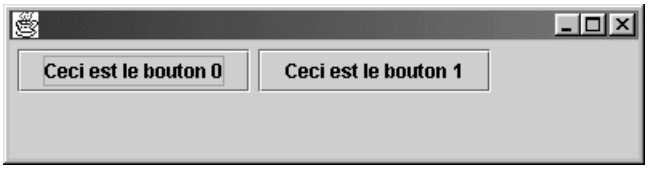


Figure 2.2 — `FlowLayout` avec alignement à gauche.

La seule différence se trouve lors de l'instanciation du FlowLayout :

```
layout = new FlowLayout(FlowLayout.LEFT);
```

Que se passe-t-il si le nombre de composants est tel qu'ils ne puissent tenir sur une ligne ? Voici figure 2.3 un exemple avec 12 `JButton` alignés à gauche.

Figure 2.3 — `FlowLayout` avec beaucoup de composants.

Le `FlowLayout` a résolu le conflit en faisant passer des composants à la ligne suivante.

Remarquons que ce qui caractérise la stratégie du `FlowLayout` est d'occuper toute la place disponible horizontalement et, au contraire, d'occuper aussi peu de

place que possible verticalement. Il reste cependant un espace inutilisé à droite car le bouton 2 ne tient pas dans cet espace. Le `FlowLayout` ne sait pas « déborder » de la fenêtre horizontalement.

Si nous agrandissons la fenêtre de sorte que l'espace restant à droite puisse accueillir un bouton, le `FlowLayout` va réorganiser la disposition des composants afin de disposer le plus de boutons possible sur une ligne (figure 2.4).

Figure 2.4 — `FlowLayout` avec une fenêtre plus grande.

Réalisation d'un formulaire

Le `FlowLayout` est donc utile pour placer des composants horizontalement. Il est par exemple très pratique pour un formulaire souvent composé de couple label/champ éditable, c'est-à-dire de `JLabel/JTextField`.

Supposons que nous souhaitions implémenter un formulaire de saisie d'informations personnelles telles que le nom et le prénom d'une personne (figure 2.5).

Figure 2.5 — Formulaire avec un unique `FlowLayout`.

Pour être réutilisable plus aisément, la classe `FichePersonne` qui implémente le formulaire dérive de `JPanel`. Nous pourrons ainsi fournir un container tout prêt qu'il sera ensuite possible d'ajouter dans n'importe quelle interface graphique.

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

public class FichePersonne extends JPanel {
     protected FlowLayout layout =
     ➥new FlowLayout(FlowLayout.LEFT);
```

```
        protected JLabel lbNom = new JLabel("Nom");
        protected JLabel lbPrenom = new JLabel("Prenom");
        protected JTextField tfNom = new JTextField();
        protected JTextField tfPrenom = new JTextField();

        public FichePersonne() {
             setLayout(layout);
             add(lbNom);
             add(tfNom);
             add(lbPrenom);
             add(tfPrenom);
        }
}
```

Le résultat visuel n'est pas très satisfaisant. Les champs de saisies sont minuscules. Pourquoi ? Le layout utilisé est un `FlowLayout`. Il se base sur la `preferred Size` des composants pour les agencer. N'oublions pas que la `preferredSize` d'un `JTextField` dépend du texte qu'il contient. Un `JTextField` vide a donc une `preferredSize` presque nulle. C'est pourquoi nos champs de saisies sont si petits. Que peut-on faire ? Voici quelques solutions envisageables :

- changer de layout ;
- positionner nous-mêmes une `preferredSize` ;
- mettre un texte par défaut dans le `JTextField`.

Nous aborderons la première solution dans le paragraphe suivant. Voyons ce qu'il est possible de faire en gardant un `FlowLayout`.

La deuxième solution présente un inconvénient majeur, qui est de devoir imposer une taille en pixels, ce qui est toujours à éviter. En effet, avec les layouts, le positionnement est évalué dynamiquement. Il n'est pas recommandé de mettre dans le code des contraintes statiques qui risqueraient d'aller à l'encontre de la résolution dynamique faite par le layout.

Nous choisirons donc la troisième solution qui, même si elle n'est pas parfaite, apporte une amélioration. Il suffit d'ajouter dans le constructeur des `JTextField` un texte par défaut (figure 2.6).

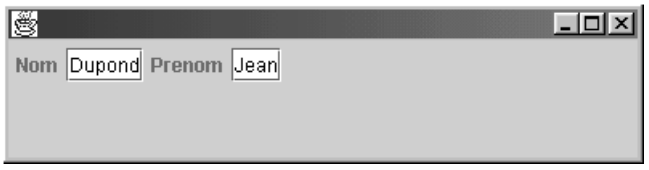


Figure 2.6 — Le formulaire avec un texte par défaut.

Le résultat est meilleur. Les composants sont bien alignés de gauche à droite et dans l'ordre des ajouts. Une amélioration importante consisterait à constituer des lignes. Une ligne pour le nom et une autre pour la saisie du prénom. Voyons maintenant si nous pouvons faire cela à l'aide d'un `GridLayout`.

2.1.2 GridLayout

Le `GridLayout` implémente une stratégie simple et efficace. L'espace du container est découpé en une grille. Le nombre de colonnes et de lignes composant cette grille est fixé lors de l'instanciation de `GridLayout` :

```
GridLayout monLayout = new GridLayout(int nbLigne, int
➥nbColonne);
```

À nouveau, l'ordre dans lequel les composants sont ajoutés est très important. La grille se remplit de gauche à droite et de haut en bas.

Création d'un nuancier de couleur

Pour illustrer l'utilisation du `GridLayout`, nous allons réaliser un nuancier de couleur. Cela revient à une matrice où chaque case a une couleur. Toutes les cases doivent avoir la même taille afin de ne pas « favoriser » une couleur particulière (figure 2.7).

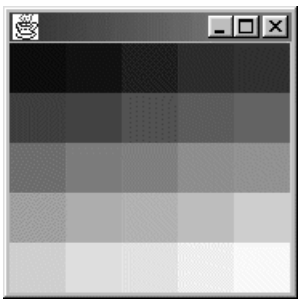


Figure 2.7 — Le nuancier.

Encore une fois, nous dériverons notre classe `Nuancier` de `JPanel`. Ce container pourra ainsi être facilement réutilisé :

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

public class Nuancier extends JPanel {
     protected GridLayout layout;

     public Nuancier(int nbLigne, int nbColonne) {
```

```
            layout = new GridLayout(nbLigne, nbColonne);
            setLayout(layout);
            int composanteRouge = 0;
            int composanteVerte = 0;
            int composanteBleue = 0;
            Color couleur;
            for (int l = 1; l <= nbLigne; l++)  {
                  for (int c = 1; c <= nbColonne; c++)  {
                    composanteVerte +=
                    ➥(255 / (nbLigne * nbColonne));
                    composanteBleue +=
                    ➥(255 / (nbLigne * nbColonne));
                    composanteRouge +=
                    ➥(255 / (nbLigne * nbColonne));
                    couleur = new Color(composanteRouge,
                                        composanteVerte,
                                        composanteBleue);
                    JPanel pan = new JPanel();
                    pan.setBackground(couleur);
                    add(pan);
                  }
            }
      }
}
```

Chaque case doit contenir un `JComponent`. Le plus pratique pour représenter une zone colorée est d'utiliser un `JPanel` et de positionner une couleur de fond avec la méthode `setBackground(Color c)`.

Il n'est pas nécessaire de garder une référence sur chaque instance de composant qui est ajoutée à chaque case.

La classe `Nuancier` prend en paramètre du constructeur la taille de la matrice en terme de nombre de lignes et de colonnes. C'est la classe qui utilise le nuancier qui déterminera ces paramètres.

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

public class TestNuancier {

      public static void main(String args[]) {
            JFrame f = new JFrame();
            Nuancier nuance = new Nuancier(5, 5);
            […]
            f.getContentPane().add(
            ➥nuance, BorderLayout.CENTER);
            f.setSize(400, 100);
            f.SetVisible(true);
      }
}
```

Une taille par défaut est donnée à la fenêtre. Que se passe-t-il si la fenêtre change de taille, comment va agir le `GridLayout` (figures 2.8 et 2.9) ?

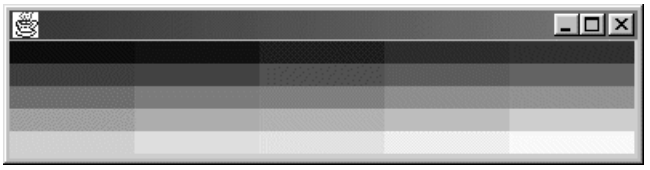


Figure 2.8 — Le nuancier dans une fenêtre de forme rectangulaire.

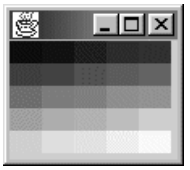


Figure 2.9 — Le nuancier dans une petite fenêtre.

La taille des cases change, mais leur nombre ne change pas. De plus, la grille occupe toujours toute la place. Le `GridLayout` ne réagence jamais la position occupée par les cases, seule leur taille peut varier en fonction de la taille du container.

Réalisation du formulaire

Utilisons un `GridLayout` pour former les lignes de notre formulaire, une pour la saisie du nom et l'autre pour le prénom. Chaque ligne devra être composée d'un label et d'un champ. Nous allons donc opter pour une matrice de deux lignes et une colonne. Chaque case contiendra un `JPanel` associé à un `FlowLayout` pour aligner le label et le champ.

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

public class FichePersonne extends JPanel {
     protected GridLayout layout = new GridLayout(2, 1);
     protected JLabel lbNom = new JLabel("Nom");
     protected JLabel lbPrenom = new JLabel("Prenom");
     protected JTextField tfNom = new JTextField("Dupond");
     protected JTextField tfPrenom = new JTextField("Jean");
     protected JPanel pNom = new JPanel();
     protected JPanel pPrenom = new JPanel();
     protected FlowLayout lNom =
     ➥new FlowLayout(FlowLayout.LEFT);
     protected FlowLayout lPrenom =
     ➥new FlowLayout(FlowLayout.LEFT);
```

```
        public FichePersonne() {
              pNom.add(lbNom);
              pNom.add(tfNom);
              pNom.setLayout(lNom);
              pPrenom.add(lbPrenom);
              pPrenom.add(tfPrenom);
              pPrenom.setLayout(lPrenom);
              setLayout(layout);
              add(pNom);
              add(pPrenom);
        }
  }
```

Quels sont les changements apportés au code par rapport à la version précédente où nous utilisions seulement un `FlowLayout` ?

Le layout général de la classe `FichePersonne` est maintenant un `GridLayout`. Deux `JPanel` apparaissent : `pNom` et `pPrenom` ainsi que leurs deux layouts associés, deux instances de `FlowLayout`. Nous utilisons des `FlowLayout` pour aligner le champ à la droite du label et aligner l'ensemble à gauche dans les `JPanel` `pNom` et `pPrenom`.

Voyons le résultat sur la figure 2.10.

Figure 2.10 — Le formulaire avec des lignes formées par un `GridLayout`.

À première vue, l'interface est encore améliorée par rapport à la version précédente. Les deux lignes rendent la lecture du formulaire plus facile.

Que se passe-t-il en cas de modification de la taille de la fenêtre ?

Voici figure 2.11 le résultat si nous réduisons trop la taille de la fenêtre .

Figure 2.11 — Le formulaire avec des lignes formées par un `GridLayout` dans une petite fenêtre.

Voici figure 2.12 la même fenêtre qui a été agrandie .

Figure 2.12 — Le formulaire avec des lignes formées par un `GridLayout` dans une grande fenêtre.

Ce résultat n'est pas satisfaisant. Le `GridLayout` sépare l'espace en cases, mais les composants déposés ne gardent pas leur `preferredSize` et sont étirés pour occuper la totalité de la surface de leur case. On observe cependant que les `JPanel pNom` et `pPrenom` semblent rester au centre des cases au fur et à mesure que les cases grandissent verticalement. Cela est dû au fait que ce ne sont pas les `JPanel pNom` et `pPrenom` que l'on voit rester au centre, mais plutôt les labels et leur champ. Il ne faut pas oublier que le layout associé au `JPanel pPrenom` par exemple est un `FlowLayout`. En fait, les composants `JLabel` et `JTextField` que nous avons utilisés ne sont pas au centre, mais alignés en haut et à gauche de leur container, du fait que le layout associé est un `FlowLayout`. Pour nous en convaincre, nous allons ajouter la ligne suivante à la fin du constructeur de la classe `FichePersonne` :

```
pPrenom.setBackground(Color.white);
```

En appliquant une couleur de fond blanche au `JPanel pPrenom`, ce dernier se détachera de la couleur de fond de son propre container, la classe `Fiche Personne` elle-même puisqu'elle dérive de `Jpanel` (figure 2.13).

Figure 2.13 — Le formulaire avec des lignes formées par un `GridLayout`. Un des Panel est coloré.

Comme dans l'exemple du nuancier, les composants sont étirés pour occuper la taille des cases définies par un `GridLayout`. De fait, ce layout ne convient pas à notre formulaire. Nous attendons de ce formulaire que les lignes restent collées

et ne s'agrandissent pas en cas d'augmentation verticale de la fenêtre. Le propre d'un `GridLayout` est d'agrandir la taille des cases dans toutes les directions en cas d'agrandissement du container. Il ne nous convient pas.

Essayons maintenant le BoxLayout.

2.1.3 BoxLayout

Présentation

Les layouts que nous avons utilisés jusqu'à présent étaient dans le package `java.awt`. La classe `BoxLayout` est une classe du package `javax.swing`. Voilà l'occasion de préciser qu'un layout n'est pas un composant et par conséquent n'hérite ni de `Component`, ni de `JComponent`. C'est pour cette raison que les layout définis dans le package `java.awt` peuvent être utilisés, sans que cela pose problème, par des applications Swing.

Le `BoxLayout` est une sorte de `FlowLayout` vertical ou horizontal : il permet d'aligner les composants *les uns à côté des autres*, ou *les uns en dessous des autres*. Rappelez-vous que le `FlowLayout` ne permet pas d'aligner les composants verticalement. Ceci dit, même lorsque l'on choisit un alignement horizontal pour le `BoxLayout`, ces deux layouts sont très différents. Le `FlowLayout`, comme nous l'avons vu, est capable de faire passer un composant « à la ligne » si l'espace horizontal est insuffisant. C'est ainsi qu'il est possible d'agencer horizontalement des composants avec un `FlowLayout` mais de les retrouver affichés les uns au-dessous des autres si le container n'est pas assez large.

Le `BoxLayout` ne permet pas cette sorte de réarrangement. Dès lors que les composants sont alignés dans un sens, verticalement ou horizontalement, ils le resteront quelle que soit la taille du container.

Voici un exemple simple qui utilise trois `JButton` alignés horizontalement, avec un `FlowLayout` puis avec un `BoxLayout`.

La première version utilise un simple `FlowLayout` (figure 2.14).

Figure 2.14 — Comparaison `FlowLayout`, `BoxLayout` : le `FlowLayout` dans une grande fenêtre.

Comme nous le disions, si la fenêtre est trop petite verticalement, les composants peuvent être repositionnés verticalement, malgré notre intention première de les aligner horizontalement (figure 2.15).

Figure 2.15 — Comparaison `FlowLayout`, `BoxLayout` : le `FlowLayout` dans une petite fenêtre.

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

public class TestBoxLayout {

    protected JButton bouton1 = new JButton("bouton 1");
    protected JButton bouton2 = new JButton("bouton 2");
    protected JButton bouton3 = new JButton("bouton 3");
    protected FlowLayout layout =
    ➥new FlowLayout(FlowLayout.LEFT);

    public static void main(String args[]) {
          TestBoxLayout tb = new TestBoxLayout();

    }

    public TestBoxLayout() {
          JFrame f = new JFrame();
          f.getContentPane().setLayout(layout);
          f.getContentPane().add(bouton1);
          f.getContentPane().add(bouton2);
          f.getContentPane().add(bouton3);
          f.setSize(400, 100);
          f.SetVisible(true);
    }
}
```

La classe `TestBoxLayout` est des plus simples. Le constructeur initialise l'interface graphique en instanciant une `JFrame` puis en utilisant une référence pour affecter au container racine, le `contentPane`, un layout et lui ajouter les trois boutons.

Voici maintenant la deuxième version, elle utilise un `BoxLayout` horizontal (figure 2.16).

Figure 2.16 — Comparaison `FlowLayout`, `BoxLayout` : le `BoxLayout` dans une grande fenêtre.

Une différence de comportement importante du BoxLayout est que si nous réduisons la taille de la fenêtre, les composants que nous avons alignés horizontalement ne se réorganiseront pas verticalement (figure 2.17).

Figure 2.17 — Comparaison `FlowLayout`, `BoxLayout` : le `BoxLayout` dans une petite fenêtre.

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

public class TestBoxLayout {

     protected JButton bouton1 = new JButton("bouton 1");
     protected JButton bouton2 = new JButton("bouton 2");
     protected JButton bouton3 = new JButton("bouton 3");
     protected BoxLayout layout;

     public static void main(String args[]) {
           TestBoxLayout tb = new TestBoxLayout();
     }

     public TestBoxLayout() {
           JFrame f = new JFrame();
           layout = new BoxLayout(
           ➥f.getContentPane(), BoxLayout.X_AXIS);
           f.getContentPane().setLayout(layout);
           f.getContentPane().add(bouton1);
           f.getContentPane().add(bouton2);
           f.getContentPane().add(bouton3);
```

```
            f.setSize(400, 100);
            f.SetVisible(true);
        }
    }
```

Le `BoxLayout` ne s'utilise pas tout à fait comme les layouts que nous avons vus précédemment. Le constructeur du `BoxLayout` prend deux paramètres :

- une référence sur le container dont il va positionner les composants ;
- une contrainte qui détermine l'orientation du layout. Cette contrainte peut prendre deux valeurs : **`BoxLayout.X_AXIS`** ou **`BoxLayout.Y_AXIS`**, selon que l'alignement sera horizontal ou vertical.

Réalisation du formulaire

Reprenons notre formulaire. Nous allons utiliser un `BoxLayout` pour former les lignes à la place du `GridLayout`. Nous en profiterons pour ajouter une troisième ligne pour l'affichage d'un numéro de référence (figure 2.18).

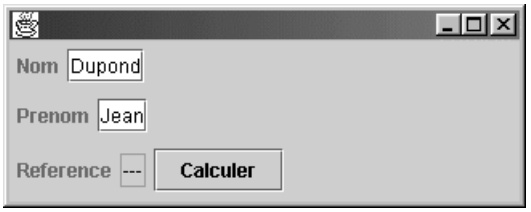


Figure 2.18 — Troisième version de notre formulaire.

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

public class FichePersonne extends JPanel {
    protected BoxLayout layout = new BoxLayout(this,
    ➥BoxLayout.Y_AXIS);
    protected JLabel lbNom = new JLabel("Nom");
    protected JLabel lbPrenom = new JLabel("Prenom");
    protected JLabel lbRef = new JLabel("Reference");
    protected JTextField tfNom = new JTextField("Dupond");
    protected JTextField tfPrenom = new JTextField("Jean");
    protected JTextField tfRef = new JTextField("---");
    protected JButton bRef = new JButton("Calculer");
    protected JPanel pNom = new JPanel();
    protected JPanel pPrenom = new JPanel();
    protected JPanel pRef = new JPanel();
    protected FlowLayout lNom =
    ➥new FlowLayout(FlowLayout.LEFT);
    protected FlowLayout lPrenom =
    ➥new FlowLayout(FlowLayout.LEFT);
```

```
        protected FlowLayout lRef =
        ➥new FlowLayout(FlowLayout.LEFT);


        public FichePersonne() {
              tfRef.setEditable(false);
              pNom.add(lbNom);
              pNom.add(tfNom);
              pNom.setLayout(lNom);
              pPrenom.add(lbPrenom);
              pPrenom.add(tfPrenom);
              pPrenom.setLayout(lPrenom);
              pRef.setLayout(lRef);
              pRef.add(lbRef);
              pRef.add(tfRef);
              pRef.add(bRef);
              setLayout(layout);
              add(pNom);
              add(pPrenom);
              add(pRef);
        }
}
```

Nous avons donc ajouté un panneau `pRef` pour la référence ainsi que son layout `lRef` de type `FlowLayout`. Imaginons que le numéro de référence est calculé par l'application. Le champ `tfRef`, un `JTextField`, n'est donc pas modifiable par l'utilisateur. Il a été rendu non saisissable par la ligne `tfRef .setEditable(false)`. De plus, nous avons ajouté un `JButton` qui déclenchera le calcul. Nous verrons comment utiliser les événements dans les chapitres suivants. On ne sait pas quelle est la taille du numéro de référence, il est même préférable de ne pas faire d'hypothèse sur sa taille, le layout associé au container `pRef` fera le positionnement approprié.

Notre interface présente un défaut esthétique : le bouton de calcul de la référence est situé immédiatement à droite du champ. Sa position est susceptible de changer suivant la longueur de la référence. Il y a plusieurs manières de résoudre ce petit problème ergonomique, sans pour autant coder « en dur » les coordonnées du `JButton`, d'autant plus que nous ignorons la longueur de la référence. Comment « caler » le bouton de calcul à droite du champ référence ?

Utilisons un `BoxLayout` associé à `pRef`. Pour cela, il suffit de remplacer la ligne suivante :

```
protected FlowLayout lRef =
➥new FlowLayout(FlowLayout.LEFT);
```

par :

```
protected BoxLayout lRef =
➥new BoxLayout(pRef, BoxLayout.X_AXIS);
```

Voici le résultat en figure 2.19.

Figure 2.19 — Utilisation du BoxLayout dans le formulaire.

Le `BoxLayout` utilise en interne la classe `Box`. Cette classe permet la création d'objets qui ne sont pas aujourd'hui des `JComponent` mais des `Component` : les glues et les structures solides. Ces objets peuvent être créés à tout moment en utilisant la classe `Box` comme une *factory*, mais ils ne seront pris en compte que si le layout est un `BoxLayout`.

Une structure solide est une sorte de composant invisible dont il faut préciser la taille en pixels. Elles peuvent servir à maintenir un espace fixe entre des composants. Cela va nous permettre de créer un espace entre le champ de référence et le bouton de calcul (figure 2.20).

Figure 2.20 — Utilisation d'une structure solide horizontale.

Nous avons inséré entre l'ajout du champ et du bouton une structure horizontale :

```
pRef.add(tfRef);
pRef.add(Box.createHorizontalStrut(10));
pRef.add(bRef);
```

Ajoutons maintenant une ligne de boutons d'actions fonctionnelles en haut du formulaire. Imaginons trois actions B1, B2 et B3. Supposons encore que l'action B3 soit « spéciale », il faut donc l'isoler. Nous utiliserons une glue pour caler ce bouton à droite. La glue est un composant qui occupe le plus d'espace possible dans une ou deux directions. Voici le résultat en figure 2.21.

Figure 2.21 — Utilisation d'une glue avec un `BoxLayout`.

Même si nous modifions la taille de la fenêtre, le bouton B3 sera toujours aligné à droite grâce à la glue. De même, l'espace entre le champ de référence et le bouton de calcul restera constant (figure 2.22).

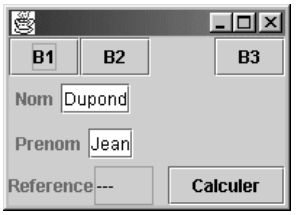


Figure 2.22 — Le formulaire version 3 complet.

La classe `Box` est une classe utilitaire qui offre des services bien utiles. En effet, cette classe propose des méthodes statiques pour créer des composants invisibles qui influent sur le layout. Elle peut être utilisée dans le cadre d'un BoxLayout ou d'un autre layout

On utilise ces méthodes lorsqu'on souhaite ajouter un espace de taille fixe entre deux composants, ou lorsqu'on souhaite contrôler la position de composants sans que leur taille change.

Par exemple, `Box` comporte une méthode de classe nommée `createGlue`. Un cas typique d'utilisation de cette méthode correspond à la création d'une barre d'outils avec l'un des boutons systématiquement aligné à droite, alors que les autres boutons sont placés les uns à côté des autres depuis le bord gauche.

Lorsqu'une telle barre d'outils est retaillée, l'espace entre les boutons de gauche et celui de droite doit s'agrandir, alors que tous les boutons gardent la même largeur.

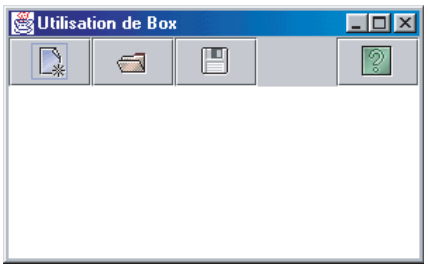


Figure 2.23 – Utilisation d'un composant « Glue » dans un BoxLayout

Pour obtenir un tel résultat, il suffit d'utiliser le code suivant :

```
JPanel barreOutils = new JPanel();
barreOutils.setLayout(newBoxLayout(barreOutils,
➥BoxLayout.X_AXIS));
```

```
barreOutils.add(boutonNouveau);
barreOutils.add(boutonOuvrir);
barreOutils.add(boutonSave);
barreOutils.add(Box.createGlue());
barreOutils.add(boutonHelp);
```

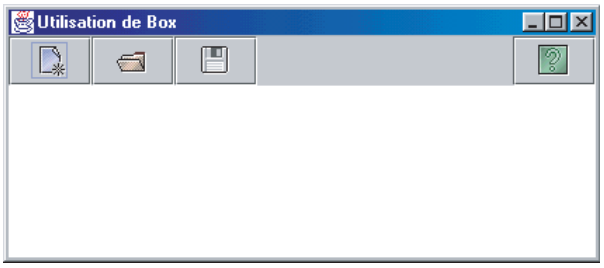


Figure 2.24 – Comportement d'un composant « Glue » lors du retaillage

2.1.4 BorderLayout

Le `BorderLayout` implémente une stratégie de positionnement tout à fait particulière et très utile. Il réserve dans l'espace de son container, cinq « cases ». Une case centrale occupe le maximum d'espace possible dans toutes les directions et quatre cases « bordent » cette case centrale. Les cases périphériques sont nommées du nom des points cardinaux. Ainsi la case qui se trouve en haut de la case centrale se nomme « Nord ».

Ce layout est particulièrement utile car cette situation est très fréquente : une barre d'état au sud, une barre d'outils au nord, une arborescence à l'ouest et un `JPanel` au centre qui prend toute la place possible. Cette petite description peut s'apparenter à un « pattern graphique » que l'on retrouve très souvent.

Pour bien illustrer les modifications de l'espace géré par un `BorderLayout` nous allons construire un exemple très simple en plaçant des boutons aux points cardinaux (figures 2.25 à 2.27) :

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

public class Fenetre extends JFrame {

public Fenetre() {
     getContentPane().setLayout(new BorderLayout());
     getContentPane().add(new JButton("Nord"),
     ➥BorderLayout.NORTH);
     getContentPane().add(new JButton("Sud"),
     ➥BorderLayout.SOUTH);
```

```
        getContentPane().add(new JButton("Est"),
        ➥BorderLayout.EAST);
        getContentPane().add(new JButton("Ouest"),
        ➥BorderLayout.WEST);
        getContentPane().add(new JButton("Centre"),
        ➥BorderLayout.CENTER);
    }
}
```

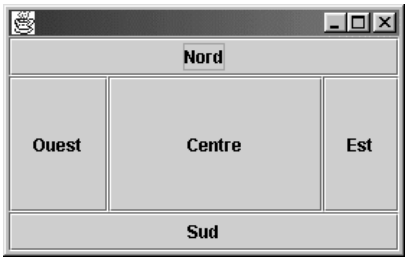


Figure 2.25 — Utilisation du `BorderLayout`, taille « normale ».

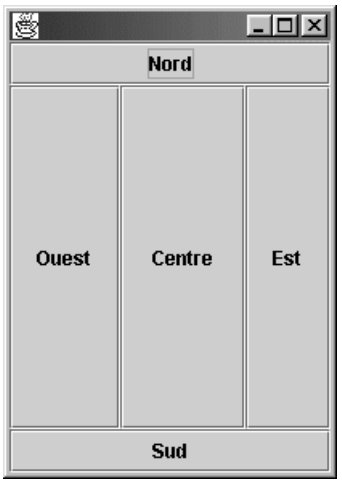


Figure 2.26 — Utilisation du `BorderLayout`, taille « allongée verticale ».

Figure 2.27 — Utilisation du `BorderLayout`, taille « allongée horizontale ».

Remarquez que le BorderLayout est le premier layout que nous utilisons qui nécessite des contraintes spécifiques lors de l'ajout des composants. De ce fait, l'ordre des `add` n'a pas d'importance. Voilà qui justifie deux des signatures paramétriques de la méthode `add` portée par la classe `Container` :

```
Component add(Component comp)
void add(Component comp, Object constraints)
```

En ce qui concerne le comportement du BorderLayout, notons que la zone centrale prend toujours le maximum de place si toutefois les composants « périphériques » ne prennent pas toute la place. Ce cas peut se produire si la fenêtre est réduite à l'excès.

Les composants placés au nord et au sud occupent le maximum de place horizontalement mais le minimum verticalement. En d'autres termes, ils ont leurs tailles souhaitables verticalement, mais sont étirés horizontalement.

C'est précisément l'inverse pour les composants placés à l'est et à l'ouest.

2.1.5 FormLayout

Le `FormLayout` est un cas à part. Ce layout ne fait pas partie du JDK mais il est si utile que nous le présentons ici. Le `FormLayout` est développé par *Karsten Lentzsch*, fondateur du site www.jgoodies.com, il est inclus dans la librairie *forms*. Comme son nom l'indique, il est particulièrement adapté pour définir des interfaces graphiques de type « formulaire de saisie ». Le `FormLayout` utilise une stratégie de positionnement d'assez haut niveau, ce qui permet de décrire un formulaire entier à l'aide d'un seul container utilisant `FormLayout`. Ainsi, on réduit le nombre de containers et on simplifie l'arborescence des composants sur une fenêtre. La mémoire, la vitesse ainsi que la maintenance s'en trouvent améliorées.

Reprenons le formulaire précédent et rendons-le redimensionnable, ce qui est bien plus confortable pour l'utilisateur.

```
public Fenetre() {
     lbPrenom.setText("Prénom");
     tPrenom.setColumns(20);
     lbNom.setText("Nom");
     tNom.setColumns(20);
     lbAdresse.setText("Adresse");
     taAdresse.setColumns(30);
     taAdresse.setRows(5);
     FormLayout layout = new FormLayout(
     ➥"pref, 5dlu, pref",
     ➥"pref, pref, pref, pref ");
     CellConstraints cc = new CellConstraints();
     JPanel formPanel = new JPanel();
     formPanel.setLayout(layout);
     formPanel.add(lbNom, cc.xy(1, 1));
```

```
        formPanel.add(tNom, cc.xy(3, 1));
        formPanel.add(lbPrenom, cc.xy(1, 2));
        formPanel.add(tPrenom, cc.xy(3, 2));
        formPanel.add(lbAdresse, cc.xy(1, 3));
        formPanel.add(taAdresse, cc.xyw(1, 4, 3));

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(formPanel, BorderLayout.CENTER);

        pack();
    }
```

`FormLayout` découpe l'espace comme une grille. Lors de l'instanciation du layout, il faut préciser les contraintes à l'aide de deux chaînes de caractères, l'une pour les colonnes et l'autre pour les lignes. Dans notre exemple, les paramètres du constructeur sont : `"pref, 5dlu, pref"`, `"pref, pref, pref, pref"`. `Cela définit une grille de 3 colonnes et 4 lignes.` Attention à ne pas confondre les virgules séparant colonnes et lignes des virgules séparant les deux paramètres du constructeur.

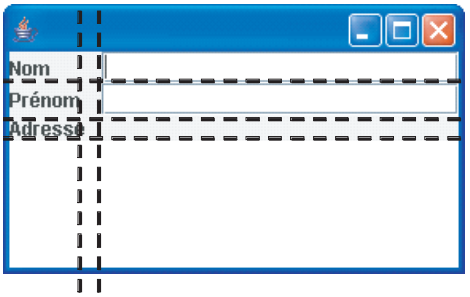


Figure 2.28 – Découpage du formulaire avec le FormLayout.

Chaque case dispose finalement de contraintes pour le composant qu'elle accueille par le croisement des contraintes de sa colonne et de sa ligne.

Établissons la grille nécessaire à notre formulaire. En ce qui concerne les colonnes, il nous faut au moins une colonne pour le libellé représenté par un `JLabel` ainsi qu'une colonne pour le champ de saisie. Idéalement, nous voudrions un espace fixe entre le libellé et le champ de saisie. Cela est possible avec le `FormLayout`. Il suffira d'indiquer la distance voulue en *dlu* (*Dialog Unit*). Le *dlu* est une unité de distance différente du pixel. Une distance de 5*dlu* varie en fonction des polices de caractères et de la résolution de l'écran de telle sorte que le rendu global de l'interface contenant des distances exprimées en *dlu* sera conservé sur différents écrans, différentes résolutions et même différents systèmes d'exploitation. En effet, rappelons qu'un même texte occupe plus ou moins de place à l'écran en fonction de la plate-forme et de certains paramétrages.

C'est d'ailleurs ce constat qui fut à l'origine des layouts, car ils sont une alternative à la définition fixe des composants au pixel près.

La notion de *dlu* permet par exemple de définir des espaces de séparation dont les proportions par rapport aux libellés affichés sont toujours respectées. Ce concept est implémenté par le FormLayout.

Une fois la grille et ses contraintes définies, il faut placer les composants dans les cases à l'aide de l'objet `CellConstraints`. Lorsqu'un composant occupe une seule case de la grille, il suffit d'appeler la méthode xy en indiquant les coordonnées de la case dans la grille : `formPanel.add(lbNom, cc.xy(1, 1))`. Il est possible d'étendre un composant sur plusieurs cellules. C'est le cas du `TextArea` contenant l'adresse qui doit occuper toute la largeur : `formPanel.add(taAdresse, cc.xyw(1, 4, 3))`. Cette fois nous avons utilisé la méthode `xyw()` qui permet de préciser le nombre de colonnes sur lesquelles le composant doit s'étendre. Le « w » de la méthode `xyw` signifie « width » (largeur).

Essayons d'agrandir la fenêtre.

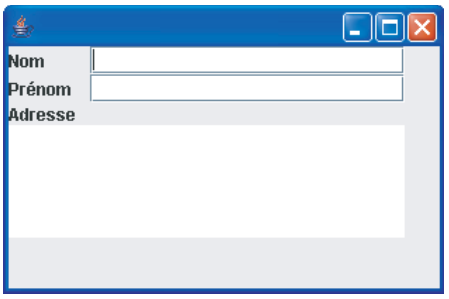


Figure 2.29 – Tentative de redimensionnement avec le FormLayout.

Nous constatons qu'il manque des contraintes : les champs de saisie n'occupent pas tout l'espace disponible. L'expression complète d'une cellule de la grille est de la forme <alignement> : <taille initiale> : <comportement au redimensionnement >. Nous n'avons jusqu'ici utilisé que la forme simple précisant la taille initiale : `"pref"` ou `"5dlu"`.

Afin d'obtenir l'effet désiré, nous devons déclarer que la troisième colonne ainsi que la quatrième ligne doivent occuper la place supplémentaire lorsque la fenêtre est agrandie (mot-clé `grow`). Par ailleurs, dès l'ouverture de la fenêtre, la quatrième ligne doit remplir tout l'espace disponible verticalement (mot-clé `fill`).

La grille est maintenant définie ainsi :

```
FormLayout layout = new FormLayout(
    "pref, 5dlu, pref:grow",
    "pref, pref, pref, fill:pref:grow");
```

Ce layout est facile d'utilisation et couvre la plupart des besoins pour les interfaces graphiques d'applications de gestion. Il est cependant malaisé de modifier un long formulaire si le code suit rigoureusement notre exemple. En effet, notez bien que chaque composant est disposé dans les cellules de la grille avec des coordonnées en dur. Si vous souhaitez maintenant ajouter un champ en haut du formulaire, vous devez modifier les coordonnées de tous les autres composants. Un tout petit changement résout se problème. Il suffit d'utiliser une variable que l'on incrémente à chaque ligne :

```
FormLayout layout = new FormLayout("pref, 5dlu, pref:grow",
➥"pref, pref, pref, fill:pref:grow");
CellConstraints cc = new CellConstraints();
JPanel formPanel = new JPanel();
formPanel.setLayout(layout);
int ligne = 1;
formPanel.add(lbNom, cc.xy(1, ligne));
formPanel.add(tNom, cc.xy(3, ligne));
ligne++;
formPanel.add(lbPrenom, cc.xy(1, ligne));
formPanel.add(tPrenom, cc.xy(3, ligne));
ligne++;
formPanel.add(lbAdresse, cc.xy(1, ligne));
formPanel.add(taAdresse, cc.xyw(1, 4, ligne));
```

Il est maintenant possible de déplacer un bloc de code et de cette position dépendra la ligne. Le code est plus facilement maintenable.

N'hésitez pas à vous organiser avec des méthodes internes qui aident à l'ajout de composant. Par exemple

```
ajout(JLabel label, JComponent comp, int ligne) {
     formPanel.add(label, cc.xy(1, ligne));
     formPanel.add(comp, cc.xy(3, ligne));
}
```

Vous pouvez noter également que nous avons utilisé la méthode `pack` dans notre constructeur. Cette méthode est très utile car elle dimensionne automatiquement la fenêtre pour que tous les composants puissent être affichés suivant leur `preferredSize` et en respectant le layout. Elle est bien sûr utilisable quel que soit le layout.

2.1.6 GridBagLayout

Ce layout fait partie intégrante du JDK et comme le FormLayout, il est destiné à réaliser des formulaires avec une disposition des composants assez recherchée.

Ce layout est relativement peu utilisé car il est assez complexe d'utilisation, mais il est en revanche très souple.

Le principe d'utilisation est le même que pour le FormLayout : le container est découpé en une grille de cellules, et les composants peuvent s'étendre sur plusieurs cellules.

Lorsqu'un composant est ajouté au container, on passe en paramètre de la méthode `add` un objet de type `GridBagConstraints`, indiquant :

- La position dans la grille (avec des coordonnées x, y).
- Le nombre de cellules utilisées en largeur et hauteur.
- Le comportement lorsque l'espace disponible est supérieur à la taille minimale du composant (le composant peut s'étendre ou non horizontalement et/ou verticalement).
- Dans le cas où le composant peut s'étendre, on peut indiquer un pourcentage afin de répartir la place supplémentaire entre plusieurs composants.
- Le point d'ancrage du composant dans son espace disponible (par exemple « FIRST_LINE_START» pour indiquer en haut à gauche).
- Des espaces de séparation (en pixels) entre le composant et la bordure de la zone qu'il occupe.

La création du panel et de l'objet layout se fait très simplement :

```
JPanel formPanel = new JPanel();
formPanel.setLayout(new GridBagLayout());
```

L'ajout du libellé « Nom » se fait de la façon suivante :

```
formPanel.add(lbNom,
     new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
     GridBagConstraints.FIRST_LINE_START,
     GridBagConstraints.NONE,
     new Insets(5, 5, 5, 5), 0, 0));
```

La construction de l'objet `GridBagConstraints` s'explique comme suit :

- Les deux premiers paramètres (0,0) indiquent que le libellé est situé dans la cellule de coordonnées (0,0), c'est-à-dire la cellule la plus en haut et à gauche. Ces paramètres correspondent aux coordonnées dans la grille.
- Les deux paramètres suivants (1,1) indiquent que le libellé s'étend sur une seule cellule en largeur et une seule cellule en hauteur.
- Les deux paramètres suivants (0.0, 0.0) correspondent au poids affecté à cette zone, et ne sont pas utiles ici puisque le libellé n'est pas destiné à s'étendre.
- Le paramètre `GridBagConstraints.FIRST_LINE_START` indique la position du composant au sein de sa cellule, ou au sein de sa zone d'affichage s'il s'étend sur plusieurs cellules. En utilisant `FIRST_LINE_START`, on

spécifie que le libellé est positionné en haut et à gauche. Ce paramètre est utile lorsqu'un composant a une taille inférieure à la zone qu'il occupe : dans ce cas, on peut choisir d'avoir le composant centré, aligné à droite, aligné à gauche…

- Le paramètre `GridBagConstraints.NONE` indique le comportement lorsque l'espace disponible est supérieur à la taille minimale nécessaire pour afficher le composant : ici, le libellé ne doit pas s'agrandir.
- Le paramètre de type `Insets` permet de définir des espaces de séparation tout autour du composant, afin d'éviter qu'il ne soit « collé » aux composants voisins.
- Les deux derniers paramètres permettent d'augmenter la taille minimale de la cellule de la grille. En effet, certains composants, comme un champ de texte vide ou une liste déroulante sans valeur ont une largeur minimale très petite. En ajoutant des valeurs pour ces deux paramètres, le composant affiché fera toujours au moins sa taille minimale plus cet espace de sécurité. Dans notre cas, puisque les champs texte s'étendent horizontalement, ces paramètres ne sont pas nécessaires.

Comparez maintenant avec l'ajout de la zone de texte pour saisir l'adresse :

```
formPanel.add(taAdresse,
     new GridBagConstraints(0, 3, 2, 1, 1.0, 1.0,
     GridBagConstraints.FIRST_LINE_START,
     GridBagConstraints.BOTH,
     new Insets(0, 5, 5, 5), 0, 0));
```

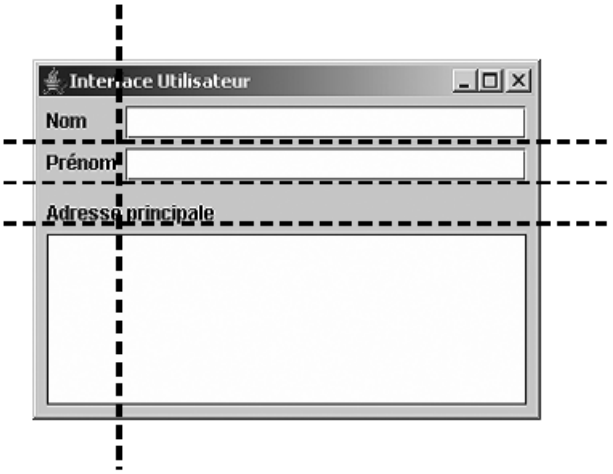


Figure 2.30 – Utilisation du `GridBagLayout`

Avec ce layout, les contraintes d'alignement et de redimensionnement sont faites au niveau de chaque composant ajouté dans la grille, alors qu'avec le FormLayout, les contraintes sont posées pour chaque ligne et chaque colonne. Cela permet donc un paramétrage très précis de l'interface graphique, mais est aussi source d'erreurs…

Voici le code du formulaire avec un `GridBagLayout`.

```
public class TestGridBagLayout extends JFrame {
   private JLabel lbNom = new JLabel();
   private JTextField tfNom = new JTextField();
   private JLabel lbPrenom = new JLabel();
   private JTextField tfPrenom = new JTextField();
   private JLabel lbAdresse = new JLabel();
   private JTextArea taAdresse = new JTextArea();
   private JPanel formPanel = new JPanel();

   public TestGridBagLayout() {
      lbNom.setText("Nom");
      lbPrenom.setText("Prénom");
      lbAdresse.setText("Adresse principale");

      formPanel.setLayout(new GridBagLayout());
      formPanel.add(lbNom, new GridBagConstraints
              ➥(0, 0, 1, 1, 0.0, 0.0
              , GridBagConstraints.FIRST_LINE_START,
              ➥GridBagConstraints.NONE, new Insets
              ➥(5, 5, 5, 5), 0, 0));
      formPanel.add(lbPrenom, new GridBagConstraints
              ➥(0, 1, 1, 1, 0.0, 0.0
              , GridBagConstraints.FIRST_LINE_START,
              ➥GridBagConstraints.NONE, new Insets
              ➥(0, 5, 5, 5), 0, 0));
      formPanel.add(tfPrenom, new GridBagConstraints
              ➥(1, 1, 1, 1, 0.0, 0.0
              , GridBagConstraints.FIRST_LINE_START,
              ➥GridBagConstraints.HORIZONTAL, new Insets
              ➥(0, 0, 5, 5), 0, 0));
      formPanel.add(tfNom, new GridBagConstraints
              ➥(1, 0, 1, 1, 1.0, 0.0
              , GridBagConstraints.FIRST_LINE_START,
              ➥GridBagConstraints.HORIZONTAL, new Insets
              ➥(5, 0, 5, 5), 0, 0));
      formPanel.add(lbAdresse, new GridBagConstraints
              ➥(0, 2, 2, 1, 0.0, 0.0
              , GridBagConstraints.FIRST_LINE_START,
              ➥GridBagConstraints.NONE, new Insets
              ➥(5, 5, 5, 5), 0, 0));
      formPanel.add(taAdresse, new GridBagConstraints
              ➥(0, 3, 2, 1, 1.0, 1.0
```

```
                , GridBagConstraints.FIRST_LINE_START,
                ➥GridBagConstraints.BOTH, new Insets
                ➥(0, 5, 5, 5), 0, 0));

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(formPanel, BorderLayout.CENTER);
    }
}
```

2.1.7 Absence de layout

Une stratégie beaucoup plus simpliste peut se résumer ainsi : « pas de stratégie ! ». Les composants sont placés dans le container d'après une position de type (x, y) dans un repère dont l'origine (0, 0) est placée en haut à gauche.

Cette stratégie s'implémente sans layout, c'est-à-dire avec une affectation du type `container.setLayout(null)`. À quoi cela peut-il bien servir ? Certaines interfaces sont si statiques qu'un layout n'est pas utile. Ce serait le cas par exemple d'un formulaire dans une fenêtre non retaillable (figure 2.31).

Figure 2.31 — Exemple de formulaire statique non retaillable.

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

public class Fenetre extends JFrame {
   protected JLabel lbNom = new JLabel();
   protected JLabel lbPrenom = new JLabel();
   protected JTextField tNom = new JTextField();
   protected JTextField tPrenom = new JTextField();
   protected JLabel lbAdresse = new JLabel();
   protected JTextArea taAdresse = new JTextArea();

   public Fenetre() {
      lbPrenom.setText("Prénom");
      lbPrenom.setBounds(new Rectangle(10, 35, 62, 17));
```

```
        lbNom.setText("Nom");
        lbNom.setBounds(new Rectangle(10, 10, 39, 17));
        getContentPane().setLayout(null);
        tNom.setBounds(new Rectangle(70, 7, 141, 21));
        tPrenom.setBounds(new Rectangle(70, 33, 141, 21));
        lbAdresse.setBounds(new Rectangle(10, 71, 62, 17));
        lbAdresse.setText("Adresse");
        taAdresse.setBounds(new Rectangle(70, 74, 141, 125));
        getContentPane().add(taAdresse);
        getContentPane().add(lbNom);
        getContentPane().add(lbPrenom);
        getContentPane().add(tNom);
        getContentPane().add(tPrenom);
        getContentPane().add(lbAdresse);
        setResizable(false);
        setSize(230, 230);
    }

    public static void main(String[] args) {
        Fenetre f = new Fenetre();
        f.SetVisible(true);
    }
}
```

Dans le cas où l'on ne précise rien, c'est-à-dire que la méthode `setLayout()` n'est jamais appelée, c'est le layout par défaut qui implémente la stratégie de placement dans le container. Pour un `JPanel`, le layout par défaut est `FlowLayout`.

Dans le cas d'un `setLayout(null)`, c'est à nous de placer chaque composant. Pour cela, il est possible de positionner la taille d'un composant avec la méthode `setSize(largeur, hauteur)` et sa position avec la méthode `setLocation (x, y)`, ou les deux en même temps avec la méthode `setBounds (x, y, largeur, hauteur)`. Cette dernière méthode existe avec une autre signature paramétrique : `setBounds(Rectangle r)`.

Que se passe-t-il si l'on agrandit la fenêtre et donc la taille du container ? Il n'y a pas de layout donc il ne se passe rien. Les composants gardent leur place. Si la taille de la fenêtre diminue, certains composants risquent de ne plus apparaître. Si au contraire, la taille de la fenêtre s'agrandit, un espace vide s'amplifie vers la droite et vers le bas.

Pour éviter cela et garantir que les composants seront toujours visibles, on pourrait chercher à être prévenu. Il serait possible d'envisager une solution événementielle à ce problème (nous traiterons les événements au chapitre 3), mais cela serait particulièrement fastidieux car il nous faudrait être absolument certain de n'oublier aucun événement capable de modifier la taille d'une fenêtre et il y en a beaucoup.

Il n'y a en fait qu'une seule solution à ce problème : écrire notre propre layout.

2.2 IMPLÉMENTER NOTRE PROPRE LAYOUT

Nous avons vu comment positionner les composants à la main en l'absence de layout. Si nous utilisions le même principe dans une classe différente que celle du container, nous ne serions pas très loin d'avoir fait nous-mêmes un layout. Essayons.

Pour concevoir un layout valide, il suffit d'implémenter l'interface `LayoutManager`. Le container s'adresse à son layout à des moments clés pour positionner ses composants. C'est donc la librairie Swing qui décide quand il est nécessaire de repositionner les composants.

Voici un exemple simple de layout qui alignera les composants en diagonale.

La classe de test est très simple, elle ajoute 5 `JButton` dans le `contentPane` d'une fenêtre `JFrame`. Naturellement, le `contentPane` est associé à notre layout.

```
import javax.swing.*;

public class TestLayout {
     protected static final int NB = 5;
     public static void main(String[] args) {
        JFrame fenetre = new JFrame();
          for(int i = 0; i < NB; i++) {
               fenetre.getContentPane().add(
                ➥new JButton ("Bouton N°"+i));
          }
          fenetre.getContentPane().setLayout(new Layout());
          fenetre.setSize(400, 200);
          fenetre.SetVisible(true);
     }
}
```

Voici la classe `Layout`. Elle hérite `d'Object` et implémente l'interface `LayoutManager` :

```
import java.awt.*;

public class Layout implements LayoutManager {

    public void addLayoutComponent(String name,
    ➥ Component comp) {
    }

    public void layoutContainer(Container parent) {
            Component[] composants = parent.getComponents();
            int x = 0;
            int y = 0;
            for (int i = 0; i < composants.length; i++) {
                 composants[i].setBounds(x, y,
                ➥composants[i].getPreferredSize().width,
                ➥composants[i].getPreferredSize().height);
```

```
                x+=composants[i].getSize().width/2;
                y+=composants[i].getSize().height;
            }
        }

        public Dimension minimumLayoutSize(Container parent) {
            return parent.getSize();
        }

        public Dimension preferredLayoutSize(Container parent) {
            return parent.getSize();
        }

        public void removeLayoutComponent(Component comp) {
        }
    }
```

La méthode `addLayoutComponent` est appelée par le container quand on lui ajoute un composant avec la méthode `add`.

La méthode la plus importante est `LayoutContainer`. Cette méthode est appelée par la librairie Swing pour déclencher un repositionnement de tous les composants gérés par le container qui est passé en paramètre.

Il n'est pas nécessaire la plupart du temps de conserver des références sur les composants ajoutés au container car il est possible de les lui demander en temps voulu. Ainsi le code de la méthode `layoutContainer(Container parent)` commence par obtenir le tableau des composants du container passé en paramètre. Nous ne coderons donc pas la méthode `addLayoutComponent`. Il en est de même pour la méthode `removeLayoutComponent` qui est appelée lorsqu'un composant est retiré du container par la méthode `remove`.

Les méthodes `minimumLayoutSize` et `preferredLayoutSize` sont appelées quand il est nécessaire de rafraîchir l'affichage. Elles permettent de positionner la taille du container qui est passé en paramètre, c'est-à-dire celui qui est associé à ce layout par la méthode `setLayout`.

C'est donc dans la méthode `layoutContainer` que nous pouvons placer les composants au sein du container. L'algorithme utilisé consiste à demander au container la liste des composants qu'il gère, puis itérer sur cette liste et placer les composants les uns après les autres. Voici le résultat figure 2.32.

Notre layout est en fait assez mauvais car il ne tient pas compte de la taille du container. Aussi, si nous réduisons la taille de la fenêtre, les composants ne se réajustent pas.

Nous constatons également un autre défaut, qui, lui, est facile à corriger : dans la classe de test, nous allons positionner un des boutons comme étant invisible, par exemple le numéro 2. Ajoutons une ligne de code du type `bouton2.setVisible(false)`. Voici le résultat figure 2.33.

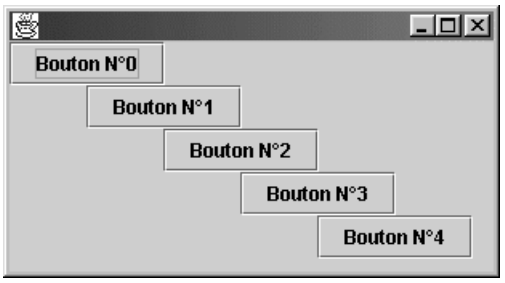


Figure 2.32 — Notre layout.

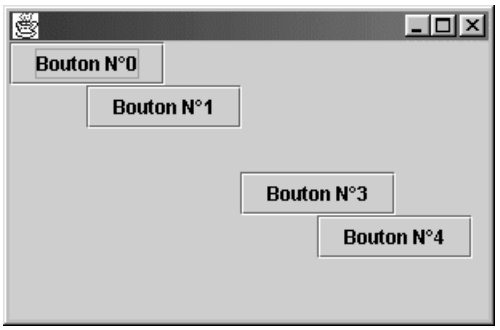


Figure 2.33 — Notre layout avec un composant invisible.

Il ne faut pas oublier qu'un composant invisible existe toujours. Il possède toujours une taille, une position… Il faut donc modifier notre layout pour qu'il ne prenne en compte que les composants visibles. La boucle devient donc :

```
for (int i = 0; i < composants.length; i++) {
     if (composants[i].isVisible()) {
           composants[i].setBounds(x, y,
                composants[i].getPreferredSize().width,
                composants[i].getPreferredSize().height);
           x+=composants[i].getSize().width/2;
           y+=composants[i].getSize().height;
     }

}
```

Voici le résultat figure 2.34.

Il ne faut pas oublier qu'en développant un layout notre seul rôle est le positionnement des composants. En particulier, nous ne devons pas modifier la nature des composants, mais uniquement leurs positions. Ainsi il n'est pas recommandé de changer la visibilité, la couleur des composants ou toute autre caractéristique.

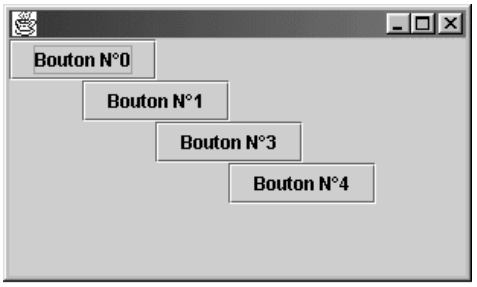


Figure 2.34 — Notre layout tient compte des composants invisibles.

2.3 QUEL LAYOUT CHOISIR ?

Prenons une série d'exemples et voyons comment nous pourrions organiser les composants, les containers et les layouts.

Nous n'essayerons pas de développer des applications, mais seulement la structure passive de l'interface graphique. Pour l'illustrer plus aisément qu'avec de longs extraits de code, nous utiliserons le formalisme présenté figure 2.35.

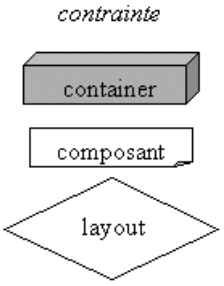


Figure 2.35 — Le formalisme de description des arborescences de composants.

2.3.1 Le bloc-notes de Windows

Ce que l'on peut « voir » c'est une zone texte qui doit bénéficier du maximum d'espace et de la totalité de l'espace libéré par un agrandissement (figure 2.36).

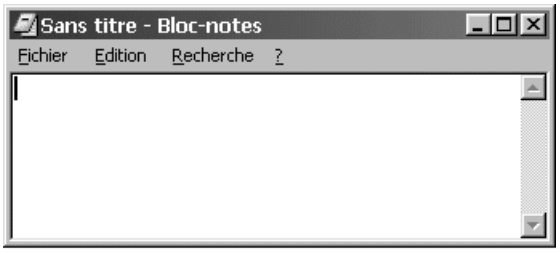


Figure 2.36 — Le bloc-notes de Windows.

Voici figure 2.37 un découpage possible pour cette interface.

Figure 2.37 — Découpage du bloc-notes de Windows.

La zone de texte

Nous utiliserons un `BorderLayout` dont la zone centrale permettra de maximiser la zone de texte. Pourquoi ce choix ? Comme nous l'avons vu, la zone centrale d'un `BorderLayout` occupe le plus de place possible, ce qui est souhaitable ici. Cette zone bénéficiera de la totalité de l'espace libéré par un éventuel agrandissement.

La barre de menus

Pourquoi la barre de menus n'est-elle pas placée au nord du `BorderLayout` ? Si nous codions un bloc-notes en Java, nous utiliserions une fenêtre sous-classe de `JFrame`. Une `JFrame` a déjà en interne une zone réservée à la barre de menus.

La barre de défilement vertical

Qu'en est-il de l'ascenseur ? Pourquoi n'est-il pas dans la zone `EAST` du `BorderLayout` ? Le fait qu'un composant possède une barre de défilement, n'est pas lié au composant lui-même. Un container spécial, `JScrollPane`, se charge de représenter un composant dans une vue. Si cette dernière est plus petite que le composant à un moment donné, alors le `JScrollPane` se charge de gérer l'affichage avec un ou deux ascenseurs (figure 2.37).

2.3.2 La boîte de dialogue d'ouverture de fichier du bloc-notes

La boîte de dialogue d'ouverture de fichier du bloc-notes est représentée figure 2.39.

Le premier réflexe à avoir ici est d'utiliser le `JFileChooser`, un composant Swing dédié aux choix de fichiers ou de répertoires. Nous n'utiliserons pas ce composant afin d'illustrer l'agencement des layouts et containers.

Voici figure 2.40 un premier découpage sommaire.

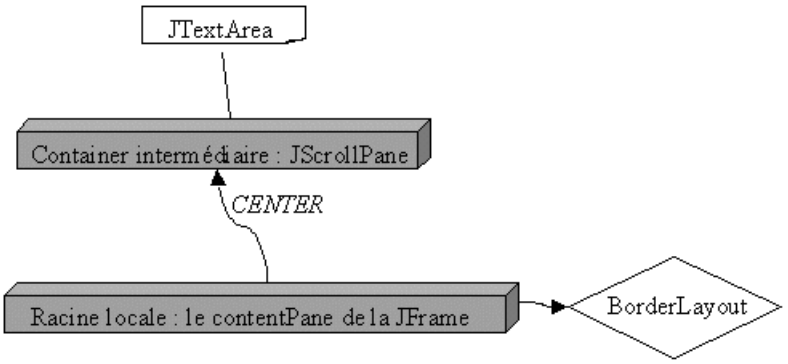


Figure 2.38 — Découpage du bloc-notes.

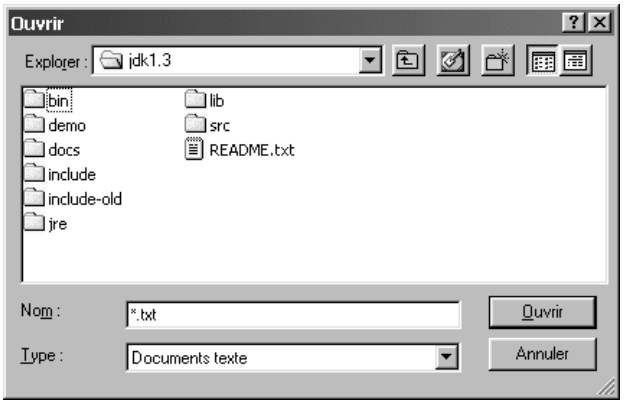


Figure 2.39 — Boîte de dialogue d'ouverture de fichier du bloc-notes.

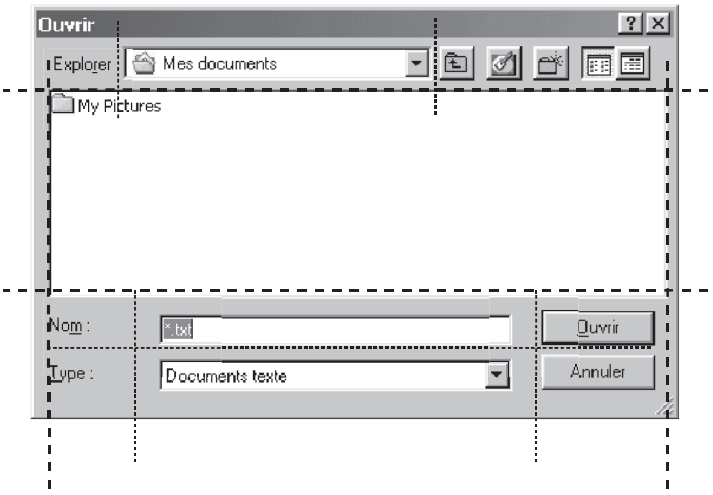


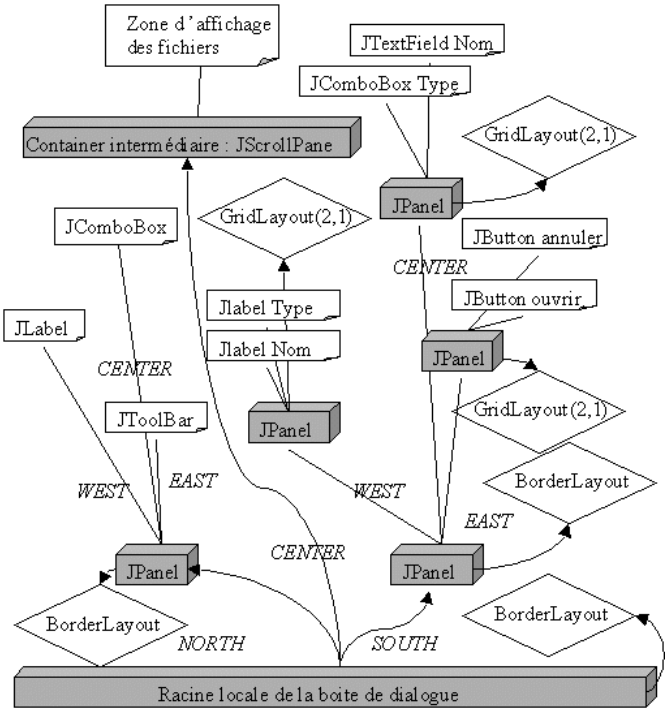
Figure 2.40 — Découpage de la boîte de dialogue d'ouverture de fichier.

On retrouve encore une sorte de « pattern graphique » à base de `BorderLayout` où la zone d'affichage des fichiers serait un composant placé dans la zone `CENTER` du `BorderLayout`. Ainsi, en cas d'agrandissement de la boîte de dialogue, cette zone bénéficierait de toute la place disponible.

La zone nord : cette zone présente une petite particularité. En tant que zone `NORTH` d'un `BorderLayout`, elle n'obtiendra pas d'espace supplémentaire verticalement, mais elle sera impactée par un agrandissement horizontal. Or dans cette situation, il serait souhaitable que la `JComboBox` qui présente le répertoire parent bénéficie totalement de cet espace. Par exemple, les icônes doivent conserver la même taille, ainsi que le `JLabel` de gauche.

Il faut donc structurer cette zone nord avec un autre `BorderLayout`. La `JComboBox` prend place dans la zone centrale, le `JLabel` dans la zone `WEST` et la zone `EAST` contient un `JPanel` car il y a plusieurs composants à placer, il faut donc un container (figure 2.41).

Figure 2.41 — Architecture de la boîte de dialogue.



Il est bon de s'exercer à ce découpage classique. Ce cas de figure est le cas typique où le `FormLayout` ou le `GridBagLayout` fera gagner un grand nombre de containers intermédiaires. Cela a ensuite un impact positif sur l'empreinte mémoire et les performances. Naturellement, les deux approches ne sont pas incompatibles. Ici, par exemple, on pourrait décider d'utiliser un `JPanel` et un `FlowLayout` pour les icônes en haut à droite. Ce container une fois initialisé pourrait ensuite remplir une cellule de la grille définie pour un `FormLayout`.

2.4 APPLICATION À L'EXEMPLE DE GESTION DES SIGNETS

Reprenons notre application de gestion de signets afin de construire la partie statique de l'interface graphique. Ce que nous souhaitons obtenir est une sorte de navigateur avec à gauche une arborescence de signets et catégories et à droite un panneau affichant des informations sur le nœud en cours de sélection dans l'arbre.

Comme dans un grand nombre de navigateurs, il doit être possible de faire varier la portion de l'interface allouée à l'arbre et celle allouée au panneau.

2.4.1 La fenêtre principale

Dès le départ, nous avons donc un découpage à l'esprit (figure 2.42) :

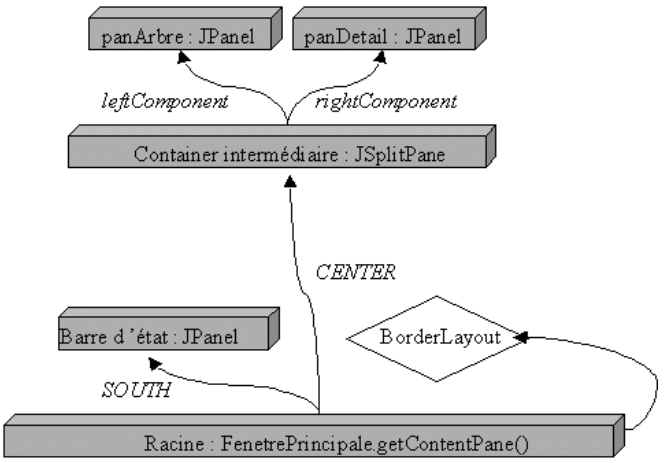


Figure 2.42 — Arborescence de base.

- une `JFrame` comme fenêtre principale,
- une zone découpée en deux parties et dont la frontière peut évoluer faisant varier la surface allouée à l'arbre ou au panneau.

Un container, le `JSplitPane` possède cette caractéristique de « frontière évolutive ».

Le `JSplitPane` contient en interne deux containers et il permet à l'utilisateur de faire varier leurs tailles à l'aide d'une barre au niveau de la « frontière ».

Pour positionner un composant dans la partie gauche ou droite on utilise la méthode :

```
setLeftComponent(Component comp)
```

ou

```
setRightComponent(Component comp).
```

La barre centrale peut être dotée de deux flèches permettant de n'afficher qu'un des deux composants. Pour activer cette fonctionnalité on utilise la méthode :

```
splitPanel.setOneTouchExpandable(boolean b).
```

Son usage dans la classe `FenetrePrincipale` ;

```
// Split panel
protected JSplitPane splitPanel = new JSplitPane();
```

Le constructeur de la classe `FenetrePrincipale` :

```
// construction du split panel
splitPanel.setLeftComponent(panArbre);
// permet l'affichage de petites flèches
splitPanel.setRightComponent(panDetail);
splitPanel.setContinuousLayout(true);
splitPanel.setOneTouchExpandable(true);
```

`panDetail` et `panArbre` sont des containers, instances de `JPanel`.

2.4.2 Le panneau de droite

Le panneau de droite représente un formulaire permettant de lire ou de modifier les informations relatives au nœud en cours de sélection.

Une zone de description est prévue pour que l'utilisateur puisse entrer des notes ou remarques. Les nœuds peuvent être des signets ou des catégories, ensembles de signets.

Pour un formulaire, la présentation classique consiste à aligner des couples de `JLabel`, `JTextField`. La difficulté est alors d'aligner sur un axe vertical la gauche de chaque `JTextField`. Les formulaires sont souvent codés sans layout, en précisant les coordonnées de chaque composant. Dans ce cas, l'alignement est parfait. De telles interfaces sont le terrain de prédilection des outils de génération d'interface graphique. Il est en effet très pénible d'entrer à la main les coordonnées de chaque composant alors qu'avec un outil, le placement se fait visuellement, comme dans un outil de dessin vectoriel.

De telles interfaces sont cependant non retaillables. Soit le développeur a désactivé cette possibilité, ce qui est de loin la solution préférable, soit les composants restent groupés en haut à gauche laissant un espace béant en bas et à droite. Des panneaux non retaillables sont souvent une cause d'exaspération pour l'utilisateur. Nous avons déjà rencontré dans un environnement de développement des champs de saisie pour le réglage des options, qui n'étaient pas retaillables. Il faut donc sans cesse utiliser les touches *Home* et *End* pour naviguer dans la ligne à éditer.

Dans notre cas, cette situation serait tout à fait détestable pour l'utilisateur pour plusieurs raisons. La plus évidente est due à cette zone de commentaires libres. Si le texte tient sur plusieurs lignes, il est souvent agréable d'agrandir la fenêtre pour voir d'un seul coup d'œil toutes les informations disponibles. Si la fenêtre n'est pas retaillable, il faut utiliser l'ascenseur.

Une autre raison vient s'ajouter : nous avons décidé de présenter notre application comme un navigateur de fichiers avec un arbre à gauche et un panneau de détail à droite. Dans une telle configuration, il y a bien souvent plusieurs types de panneaux dont l'affichage est corrélé à la nature de l'information portée par le nœud sélectionné. Avec des panneaux non retaillables il faudrait s'assurer qu'ils aient tous une taille strictement identique. Dans le cas contraire on observerait la fenêtre changer de taille pour s'ajuster au panneau en fonction du nœud en cours d'affichage, ce qui n'est pas très ergonomique, l'utilisateur n'ayant plus la maîtrise de la fenêtre.

Une dernière raison vient clore la discussion concernant notre application. Nous utilisons un container de base de type `JSplitPane` qui présente une barre centrale permettant de faire varier la taille respective des panneaux de gauche et de droite. Du point de vue du panneau, cela ne fait pas de différence avec un changement de taille de la fenêtre : il doit recalculer la position de chaque composant pour s'ajuster à une nouvelle taille globale.

Nous utiliserons donc des layouts pour construire le panneau de détail.

Voilà donc figure 2.43 ce que nous voulons obtenir.

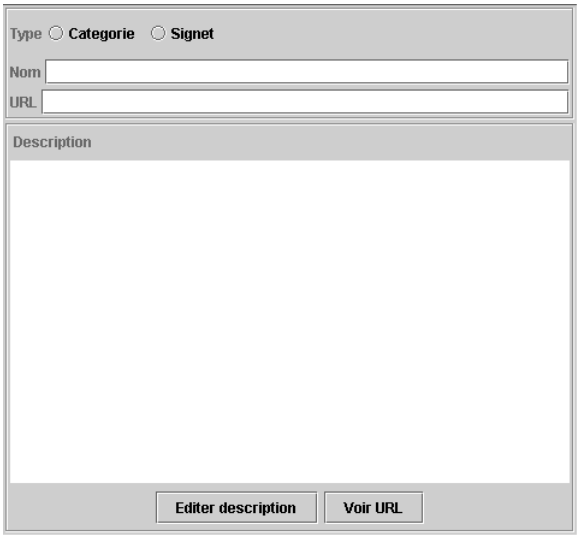


Figure 2.43 — L'interface graphique du panneau de détail.

Comment découper ce formulaire en arborescence de containers et quels layouts affecter aux containers ?

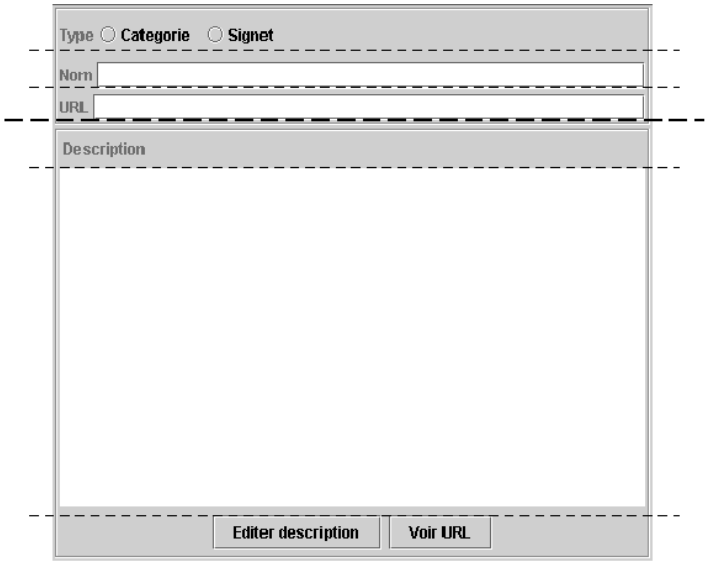


Figure 2.44 — Découpage du panneau de détails.

Pour envisager un découpage (figure 2.44), il faut en premier lieu lister les contraintes connues sur les composants. Quelles sont-elles dans notre cas ?

La zone de description doit bénéficier de tout l'espace disponible en cas d'agrandissement.

Les `JTextField` du formulaire ne doivent en aucun cas obtenir plus d'espace vertical.

Cependant, en cas d'agrandissement vers la gauche (avec la barre du `JSplit Pane`) ou vers la droite (par agrandissement de la fenêtre), les `JTextField` doivent prendre toute la place disponible. Il ne faut pas que de l'espace inutile reste sur la droite des `JTextField`.

L'idée est de découper la zone en deux grandes parties. La première en haut, contenant les `JTextField` et la seconde en bas, contenant la zone de description.

Afin que la zone du bas obtienne la totalité de l'espace disponible, nous pensons immédiatement à utiliser un `BorderLayout`. La zone du bas sera posée en partie centrale alors que la zone du haut sera posée en zone nord.

De même, si nous analysons la zone du bas, nous retrouvons aisément un autre `BorderLayout`, le label de description dans la zone nord, le composant de description (`JTextArea`) dans la zone centrale et les boutons dans la zone sud.

Naturellement, les boutons seront eux-mêmes dans un container de type `JPanel`. Ils doivent être centrés, ce qui nous amène à choisir un `FlowLayout` avec la contrainte `FlowLayout.CENTER`.

Une question peut se poser : pourquoi utiliser un container supplémentaire pour le label puisqu'il n'y a qu'un seul composant ?

En fait, nous avons une contrainte de présentation, le label doit être à gauche. Cela ne peut se faire qu'avec un layout — ici un `FlowLayout(FlowLayout.LEFT)`. C'est cette contrainte qui nous fait utiliser un container de plus (figure 2.45).

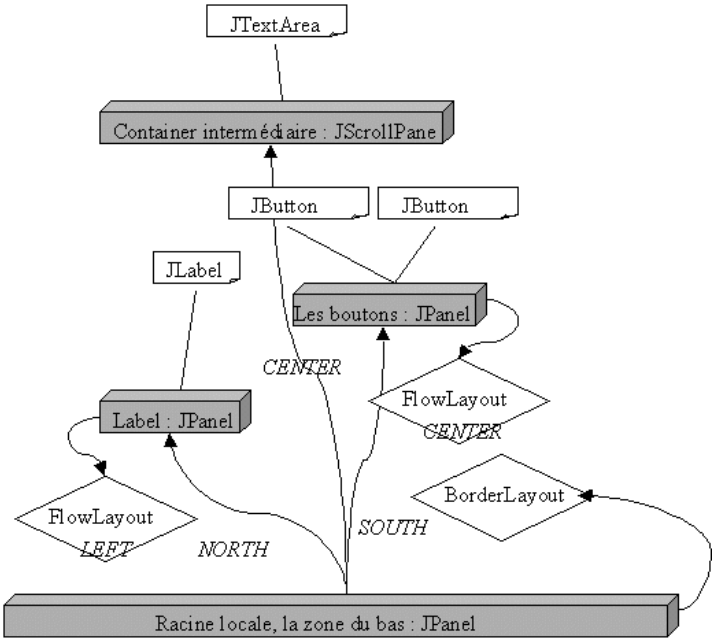


Figure 2.45 — Architecture du panneau de détails.

3

Les événements

Pour l'instant, nous n'avons vu que des interfaces « statiques », c'est-à-dire qui ne réagissent pas aux interactions de l'utilisateur. Les seules interactions possibles étaient l'utilisation des cases « système » des fenêtres (pour agrandir, iconifier…), le retaillage d'une fenêtre, ou encore la saisie de texte dans un champ et la sélection dans un menu ou une liste déroulante. Ces actions correspondent à des comportements standard et sont donc implémentées en standard par les composants Swing ou AWT. Maintenant, nous allons voir comment gérer nos propres événements et comment réagir aux événements standard…

3.1 LA GESTION DES ÉVÉNEMENTS

3.1.1 Gérer un clic bouton

Dans un premier temps, nous allons nous contenter de gérer un événement pour le moins simpliste. Nous disposons d'une fenêtre avec un bouton et un champ de texte. Lorsque l'utilisateur clique sur ce bouton, cela doit ajouter le message « Clic ! » dans le composant texte (figure 3.1).

Figure 3.1 — Un bouton simpliste.

Avant de voir comment résoudre ce problème simple, intéressons-nous à ce qu'est un événement en Java, afin de savoir comment l'intercepter.

Qu'est-ce qu'un événement ?

Toutes les interactions de l'utilisateur se traduisent par des événements en Java. Il peut s'agir d'interactions très sommaires : déplacement de la souris, obtention ou perte du focus, appui d'une touche du clavier…, ou plus élaborées : clic sur un bouton, sélection d'un item dans une liste déroulante ou dans un menu.

> Il existe aussi des événements qui ne proviennent pas de l'interaction d'un utilisateur avec une interface graphique. Nous verrons cela plus loin dans ce chapitre.

Les événements en Java sont représentés par des objets qui fournissent des informations sur l'événement lui-même et sur l'objet à l'origine de cet événement. Celui-ci est appelé la source de l'événement. Dans notre exemple, il s'agit bien sûr du bouton.

Il existe des classes d'événements spécifiques pour chaque type d'interaction de l'utilisateur. L'événement généré dépend bien sûr de la nature physique de l'interaction : est-ce qu'il s'agit d'un clic simple, du maintien du bouton de la souris enfoncé, de la frappe d'une touche au clavier ? Mais surtout, les événements Java sont différents suivant leur signification logique : le type d'objet « événement » généré n'est pas le même s'il s'agit d'un clic sur un bouton, d'un clic pour sélectionner un menu, ou encore d'un clic pour se positionner dans un champ de saisie de texte (figure 3.2).

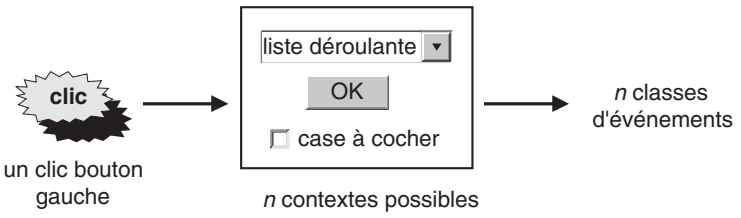


Figure 3.2 — Les événements ont une valeur logique.

La classe à la base de la hiérarchie des événements est la classe `EventObject`, qui se trouve dans le package `java.util`.

> **Attention !** Ne confondez pas avec la classe `Event` du package `java.awt`, qui était utilisée dans la première gestion d'événements proposée par Sun (JDK1.0), et qui est maintenue uniquement dans un souci de compatibilité ascendante.

Le tableau 3.1 présente quelques exemples de classes d'événements et de l'interaction utilisateur à laquelle elles correspondent.

Tableau 3.1 — Exemples de classes d'événements.

Interaction utilisateur	Événement émis
Passage du focus à un composant	`FocusEvent`
Clic sur un `JPanel`	`MouseEvent`
Frappe d'une touche clavier sur un `JPanel`	`KeyEvent`
Iconification d'une fenêtre	`WindowEvent`
Clic sur un bouton	`ActionEvent`
Ajout d'une lettre dans un `JTextField`	`DocumentEvent`
Sélection d'un item dans une `JList`	`ListSelectionEvent`

Vous remarquez donc qu'un clic souris peut donner lieu à un `MouseEvent` si le curseur se trouve sur un `JPanel`, mais à un `ActionEvent` si le clic a eu lieu sur un bouton.

Tous ces événements ont comme caractéristique commune d'encapsuler des informations concernant leur contexte d'apparition :

- le composant graphique source de l'événement (il s'agit du composant sur lequel se trouvait le curseur pour un événement « souris » ou du composant qui avait le focus pour un événement « clavier ») ;
- les coordonnées du clic pour un `MouseEvent` ;
- la valeur de la touche tapée pour un `KeyEvent`.

Comment réagir à un événement ?

L'objet qui va intercepter le clic sur notre bouton et afficher le message convenu dans le champ de texte est appelé en Java un *listener*. Ce nom est particulièrement adéquat, car le rôle principal de cet objet est d'être à l'écoute de notre bouton « Cliquez » et de se tenir prêt à intervenir en cas de clic.

Ce statut particulier d'être à l'écoute d'un clic est permis à n'importe quel type d'objet mais n'est pas automatique. Le listener doit « s'abonner » à l'événement auprès du bouton. Ce faisant, il demande au bouton de l'avertir en cas de clic (figure 3.3).

Cette étape d'abonnement est un passage obligé à la gestion des événements. Comment cela se réalise-t-il ? Sur la classe `JButton`, nous observons qu'il existe une méthode qui s'appelle `addActionListener`. Voilà précisément la méthode qui permet l'abonnement :

```
public void addActionListener(ActionListener l) ;
```

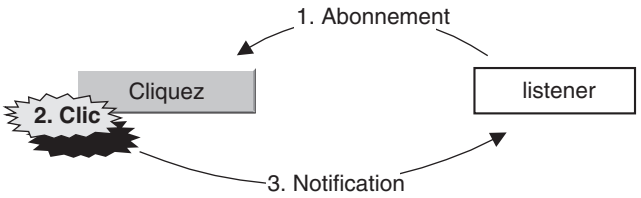


Figure 3.3 — Le principe de la gestion des événements.

Le paramètre de cette méthode est un objet du type `ActionListener`. `Action Listener` est une interface. Nous devons donc créer une classe qui implémente cette interface. Appelons notre classe `ClicListener`.

Cette interface spécifie une seule méthode à implémenter, c'est la méthode :

```
public void actionPerformed(ActionEvent e) ;
```

C'est précisément cette méthode qui va être appelée lorsqu'un clic aura lieu. Nous devons donc implémenter dans cette méthode le fait que le texte « Clic ! » s'ajoute au champ de texte. Lors du clic, le bouton « prévient » le listener qu'un clic a eu lieu. C'est ce qu'on appelle la notification. Cette action de prévenir se fait *via* l'appel à cette méthode `actionPerformed`.

La gestion des événements en Java se déroule donc en deux étapes (figure 3.4) :

- étape préalable : abonnement du listener auprès du composant graphique ;
- lors du clic : notification du listener par le composant graphique.

Finalement, l'émission d'un événement n'est qu'un simple appel de méthode. Cet appel est régi par un pattern communément appelé le pattern « `Observer` ». Le listener, « celui qui écoute », doit implémenter une méthode définie à l'avance avec l'émetteur. Le rôle de l'interface `ActionListener` est de convenir de cette méthode. En cas de survenue d'une situation que l'émetteur connaît, il appelle cette méthode, qui est déclarée dans l'interface `ActionListener`. Cela suppose que l'émetteur possède une référence vers celui qui écoute, c'est le rôle de l'abonnement.

Dans notre exemple, voyons comment implémenter la méthode `action Performed`. Dans cette méthode, il faut modifier le texte du champ de texte pour lui ajouter la chaîne de caractères « Clic ! ». Ce champ est un `JTextArea`, représenté par la variable d'instance `taResultat` définie dans la classe `Fenetre Principale`.

Pour que le listener puisse modifier cet objet, il faut lui fournir une référence sur cet objet. Par conséquent, il est nécessaire de passer en paramètre du constructeur du `ClicListener` l'objet `JTextArea` qu'il devra modifier. La méthode `action Performed` est ensuite facile à déduire, il suffit de concaténer le message « Clic ! » au texte déjà présent dans le `JTextArea`, avec la méthode `append`.

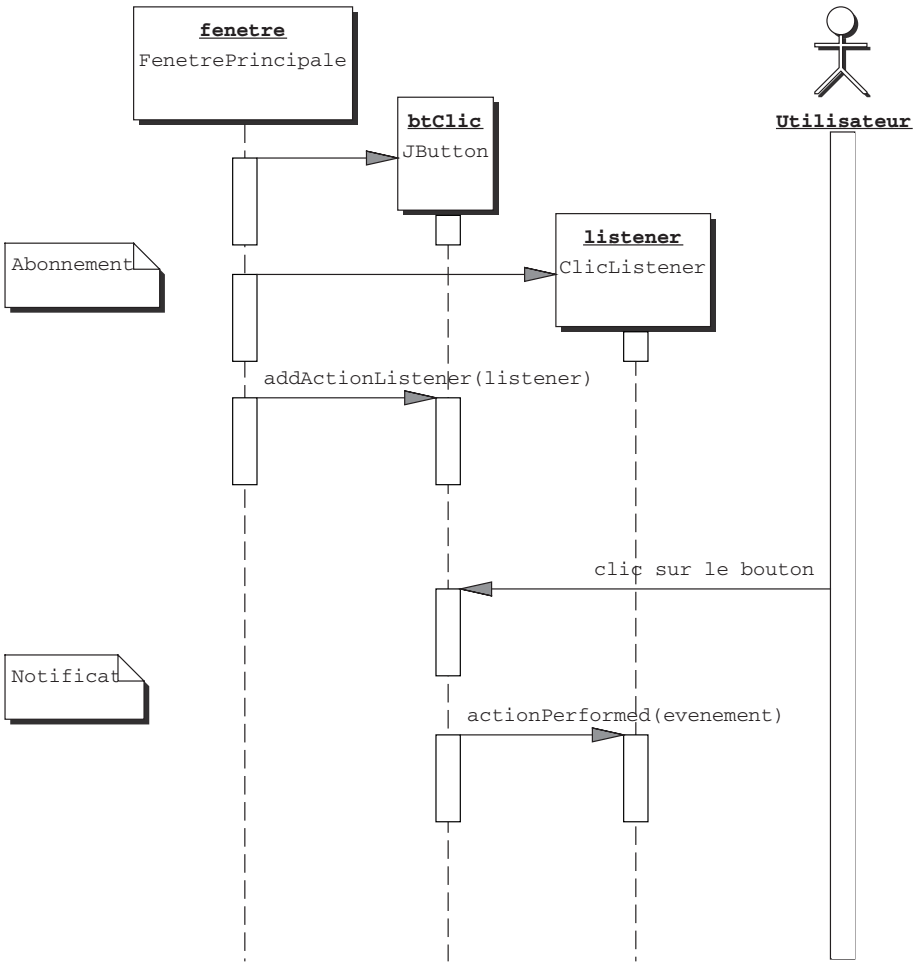


Figure 3.4 — Diagramme de séquence de la gestion des événements.

Voici le code complet de la classe `ClicListener` :

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

public class ClicListener implements ActionListener{
   public static final String msg = "Clic ! ";
   private JTextArea cible;

   public ClicListener(JTextArea t) {
     cible = t;
   }
   public void actionPerformed(ActionEvent e) {
     cible.append(msg);
   }
}
```

L'étape d'abonnement se fait dans le constructeur de la classe `Fenetre Principale`, après la création du bouton et du champ de texte :

```
btClic = new JButton();
taResultat = new JTextArea();
ClicListener listener = new ClicListener(taResultat);
btClic.addActionListener(listener);
```

3.1.2 De nombreux types de listeners

Le but ici est de créer un panneau sur lequel le déplacement de la souris provoque le dessin de points, ce qui permet de tracer des courbes (figure 3.5).

Figure 3.5 — Dessin de courbes en déplaçant la souris.

Les événements que nous devons gérer ici sont les déplacements de la souris. Comment allons-nous procéder pour trouver le listener à implémenter ?

Aide pour la gestion d'un événement

Il existe de nombreuses interfaces de type « `...Listener` ». En effet, il existe des listeners pour chaque grande catégorie d'événements. Or, comme les événements sont différents suivant l'objet source dont ils proviennent, cela multiplie les classes. Il peut donc être déroutant de trouver où écrire le code qui dessine les points sur le panneau.

Identifier le composant source de l'événement

La première question à se poser est : quel est le composant qui va générer les événements ? Ici, il s'agit bien sûr du panneau `panDessin` que nous avons ajouté à la fenêtre.

Choisir la méthode « `add...Listener` »

Nous devons ensuite regarder les méthodes qui ont pour nom `add...Listener` dans la classe du composant source. La classe `JPanel` ne définit pas de méthodes de ce style, mais elle dispose de nombreuses méthodes obtenues par héritage. Ainsi, nous voyons :

```
addPropertyChangeListener (PropertyChangeListener l)
```

```
addContainerListener(ContainerListener l)
addFocusListener(FocusListener l)
addKeyListener(KeyListener l)
addMouseListener(MouseListener l)
addMouseMotionListener(MouseMotionListener l)
…
```

Ici, il est nécessaire d'avoir un peu d'intuition pour cerner la méthode qui peut convenir. Pour notre gestion des déplacements de la souris, il est facile de deviner que la méthode `addMouseMotionListener` doit être la bonne.

De toute façon, l'étape suivante permet de se rendre compte si le listener choisi n'est pas le bon. Dans notre cas, nous allons devoir écrire un `MouseMotionListener`.

Choisir la méthode de l'interface « …`Listener`»

L'interface `MouseMotionListener` déclare les deux méthodes suivantes :

```
/**
 * Invoked when a mouse button is pressed on a component
 * and then
 * dragged.  Mouse drag events will continue to be delivered to
 * the component where the first originated until the mouse
 * button is
 * released (regardless of whether the mouse position is
 * within the
 * bounds of the component).
 */
public void mouseDragged(MouseEvent e);

/**
 * Invoked when the mouse button has been moved on a component
 * (with no buttons no down).
 */
public void mouseMoved(MouseEvent e);
```

Nous voulons gérer les déplacements simples de la souris, c'est-à-dire lorsque la souris bouge sans qu'aucun bouton soit enfoncé. C'est donc la méthode `mouseMoved` que nous devons implémenter.

Si le programmeur ne trouve pas ici de méthode correspondant à l'événement qu'il souhaite capturer, c'est probablement parce que son choix ne s'est pas orienté vers la bonne interface. Il faut regarder à nouveau les méthodes « `add…Listener` » et examiner un autre listener.

Le code de notre exemple est donc le suivant :

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

```
public class DessinListener implements MouseMotionListener {
/**
 * Le panneau dans lequel il faut dessiner
 */
  private JPanel cible;
/**
 * Constructeur qui prend le panneau en paramètre
 */
  public DessinListener(JPanel p) {
    cible = p;
  }
/**
 * Méthode qui dessine un point à l'endroit où se trouve
 * la souris
 */
  public void mouseMoved(MouseEvent e) {
     Graphics g = cible.getGraphics();
     g.drawRect(e.getX(), e.getY(),1 ,1);
  }
/**
 * Méthode vide
 */
public void mouseDragged(MouseEvent e) {
  }
}
```

Nous pouvons simplifier ce code en nous rappelant que les événements sont capables d'indiquer l'objet dont ils sont issus grâce à la méthode `getSource`. Le fait de passer le panneau cible en paramètre du constructeur est donc superflu. Voici le code simplifié :

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

public class DessinListener implements MouseMotionListener {
 /**
  * Méthode qui dessine un point à l'endroit où se trouve
  * la souris
  */
 public void mouseMoved(MouseEvent e) {
  Graphics g = ((JComponent)e.getSource()).getGraphics();
  g.drawRect(e.getX(), e.getY(),1 ,1);
 }
 /**
  * Méthode vide
  */
 public void mouseDragged(MouseEvent e) {
 }
}
```

L'étape d'abonnement se fait dans la sous-classe de `JFrame` qui construit l'interface graphique :

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

public class FenetrePrincipale extends JFrame {
   JPanel panDessin ;

   public FenetrePrincipale() {
      setTitle("Ardoise magique");
      this.setSize(new Dimension(300, 185));
      panDessin = new JPanel();
      panDessin.setBackground(Color.white);
     getContentPane().add(panDessin, BorderLayout.CENTER);
      // Abonnement
      DessinListener listener = new DessinListener();
      panDessin.addMouseMotionListener(listener);
   }
}
```

Des classes bien utiles : les `Adapters`

Toujours sur notre exemple de dessin, nous constatons le phénomène suivant : lorsque l'utilisateur ferme la fenêtre à l'aide de la case système (petite croix en haut à droite sous Windows), la fenêtre se ferme effectivement, mais notre programme n'est pas pour autant arrêté. En ligne de commande, on constate que le prompt ne revient pas. Il faut stopper l'application à l'aide des touches Ctrl+C.

Nous souhaitons gérer l'événement « fermeture de la fenêtre » afin de provoquer la fin de l'exécution du programme lorsque cet événement survient (figure 3.6).

Remarque : avec AWT, la case système de fermeture de la fenêtre n'avait pas de comportement par défaut : le programmeur devait également implémenter le fait que la fenêtre disparaisse.

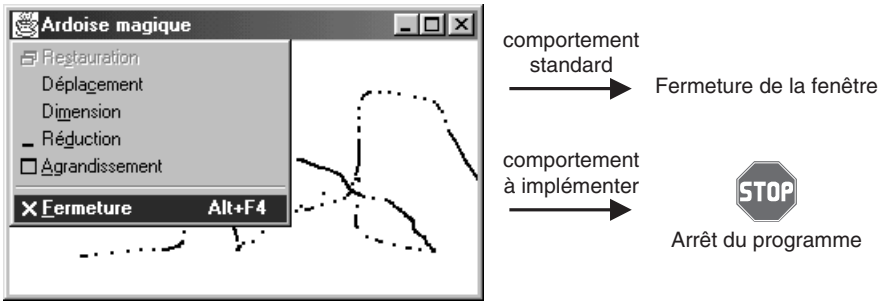


Figure 3.6 — Gérer la fermeture de la fenêtre.

Pour gérer cet événement, nous devons écrire une classe qui implémente l'interface `WindowListener` et enregistrer une instance de ce listener auprès de la fenêtre avec la méthode `addWindowListener`. L'inconvénient est que cette interface spécifie sept méthodes :

```
public void windowOpened(WindowEvent e);
public void windowClosing(WindowEvent e);
public void windowClosed(WindowEvent e);
public void windowIconified(WindowEvent e);
public void windowDeiconified(WindowEvent e);
public void windowActivated(WindowEvent e);
public void windowDeactivated(WindowEvent e);
```

La seule méthode qui nous intéresse ici est la méthode `windowClosing` qui, comme l'indique la documentation, est appelée lorsque l'utilisateur ferme la fenêtre par le menu système. (La méthode `windowClosed` est appelée lorsque la fenêtre est effectivement fermée.)

Pour nous éviter d'avoir à implémenter six méthodes vides, Java fournit des classes d'implémentation par défaut des listeners. Ce sont les `adapters`.

Ici il nous suffit donc d'hériter de la classe `WindowAdapter` et de redéfinir seulement la méthode nécessaire :

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

public class GestionnaireFenetre extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.out.println("Merci d'avoir utilisé cette
    ➥application. ");
    System.exit(0);
  }
}
```

Remarque : notez qu'il est possible d'indiquer que la fermeture de la fenêtre entraîne la fin de l'application à l'aide de la méthode `setDefaultCloseOperation`, en indiquant comme comportement `EXIT_ON_CLOSE`, comme cela a été vu dans le premier chapitre. On peut aussi court-circuiter le comportement standard pour empêcher la fermeture de la fenêtre avec `DO_NOTHING_ON_CLOSE`.

3.1.3 Où gérer les événements ?

Reprenons notre exemple du panneau sur lequel on dessine des points. Nous souhaitons ajouter une barre d'état qui affiche les coordonnées de la souris à tout instant (figure 3.7).

Figure 3.7 — Affichage des coordonnées du curseur dans une barre d'état.

Les différentes possibilités

L'implémentation qui semble la plus simple consiste à modifier le listener existant pour qu'il mette à jour le texte affiché dans la barre d'état. L'ajout de la barre d'état dans la fenêtre se fait à l'aide d'un composant `JLabel` ajouté en position `SOUTH`.

Dans le constructeur de la classe `FenetrePrincipale`, nous ajoutons donc le code suivant :

```
lbBarreEtat = new JLabel();
lbBarreEtat.setText("Coordonnées du curseur : ");
getContentPane().add(lbBarreEtat, BorderLayout.SOUTH);
```

Ensuite, pour que le listener `DessinListener` soit capable de mettre à jour le texte de `lbBarreEtat`, il faut lui fournir une référence sur cet objet. Pour cela, nous devons écrire un constructeur qui prenne en paramètre un `JLabel`. Le listener devient donc :

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

public class DessinListener implements MouseMotionListener {
  JLabel lbCible;

  /**
   * Constructeur qui prend la barre d'état en paramètre
   */
  public DessinListener(JLabel l) {
   lbCible = l;
  }
```

```
    /**
     * Méthode qui dessine un point à l'endroit où se trouve
     * la souris et qui met à jour le texte de la barre
        ➥d'état.
     */
    public void mouseMoved(MouseEvent e) {
      Graphics g = ((JComponent)e.getSource()).getGraphics();
       g.drawRect(e.getX(), e.getY(),1 ,1);
       lbCible.setText("Coordonnées du curseur : "
       + e.getX() + ", " + e.getY());
    }
    /**
     * Méthode vide
     */
    public void mouseDragged(MouseEvent e) {
    }
}
```

Cette solution est-elle satisfaisante d'un point de vue objet ?

La classe `DessinListener` ne peut plus être utilisée dans le cas d'une application qui dessine des courbes mais qui ne possède pas de barre d'état. Elle est devenue spécifique à notre application. Pour améliorer la réutilisation, nous avons intérêt à séparer les traitements qui sont liés à deux besoins distincts.

Ainsi, nous préférons la solution qui consiste à écrire deux listeners, chacun ayant un rôle bien spécifique et abonnons les deux au panneau de dessin. Nous améliorons ainsi la conception orientée objet car chaque panneau devient réutilisable en dehors de ce contexte.

Pour cela, il faut reprendre la version précédente de la classe `DessinListener` (pas de `JLabel` en paramètre du constructeur) et créer une nouvelle classe `MajCoordonneesListener`, qui, elle, ne fait que la mise à jour du texte de la barre d'état. Dans ces conditions, il faut effectuer deux abonnements : une instance de `DessinListener` et une instance de `MajCoordonneesListener` s'enregistrent auprès du panneau `panDessin`. Dans le constructeur de `Fenetre Principale`, nous avons donc :

```
// Abonnement
DessinListener listDessin = new DessinListener();
panDessin.addMouseMotionListener(listDessin);
MajCoordonneesListener listCoord = new MajCoordonnees
Listener(lbBarreEtat);
panDessin.addMouseMotionListener(listCoord);
```

Les classes de cette application sont donc celles présentées figure 3.8.

Nous constatons que pour une application très simple, nous avons une multiplication des classes de listener. Cela peut rapidement devenir difficile à gérer. On remarque également que certains listeners sont étroitement liés au composant source et ne seront probablement jamais réutilisés pour un autre composant.

Dans notre exemple, il semble difficile de réutiliser le listener `MajCoordonnees Listener` pour une autre classe de notre application. En effet, ce listener modifie le texte d'un `JLabel` pour afficher les coordonnées de la souris. Cela est assez spécifique.

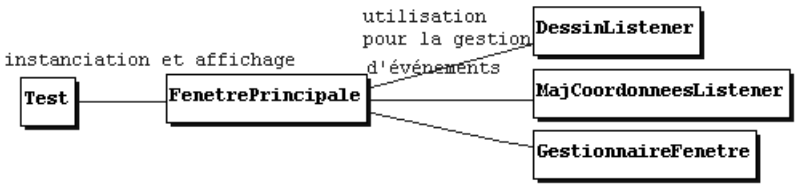


Figure 3.8 — Les classes de l'application.

Pourquoi ne pas implémenter l'interface `MouseMotionListener` directement dans la classe `FenetrePrincipale` ?
La classe `MajCoordonneesListener` pourrait être fusionnée avec la classe `FenetrePrincipale`. Une solution consiste à implémenter l'interface `Mouse MotionListener` directement dans la classe `FenetrePrincipale`. Mais d'un point de vue conceptuel, cela n'est pas conseillé, car on modifie l'interface publique d'une classe. De plus, les méthodes de l'interface du listener se trouvent « mélangées » aux méthodes de la classe. On mélange donc la partie purement graphique et la partie traitement.

Une autre solution existe pour écrire une classe intimement liée à `Fenetre Principale`, sans pour autant modifier l'interface publique de `FenetrePrincipale`. Il s'agit du mécanisme des classes incluses qui existe depuis le JDK1.1.

Les classes incluses

Depuis le JDK1.1, il est possible d'imbriquer des définitions de classes les unes dans les autres avec différentes sémantiques. Une classe définie à l'intérieur d'une autre est appelée classe incluse ou *inner class*. Toujours en suivant notre exemple de dessin de courbes, nous pouvons décliner plusieurs écritures.

Classe incluse définie au même niveau que les méthodes

Nous pouvons définir la classe `MajCoordonneesListener` dans la classe `FenetrePrincipale`. La classe incluse est indiquée « privée » pour limiter son accès à `FenetrePrincipale`.

De cette façon, le listener est un peu simplifié, car nous n'avons pas besoin de passer le composant `JLabel` en paramètre du constructeur. La classe incluse a accès aux variables, même privées, de la classe englobante.

```
public class FenetrePrincipale extends JFrame {
   [ variables d'instance ]

   public FenetrePrincipale() {
      [ construction de l'interface… ]

      // Abonnement d'un MajCoordonneesListener au panneau
      panDessin.addMouseMotionListener(
      ➥new MajCoordonneesListener());
   }

 // Classe incluse définie au même niveau que les méthodes
 class MajCoordonneesListener
 ➥implements MouseMotionListener {
    /**
     * Méthode qui met à jour le texte de la barre d'état.
     */
    public void mouseMoved(MouseEvent e) {
     lbBarreEtat.setText("Coordonnées du curseur : "
     + e.getX() + "," + e.getY());
 }
 /**
  * Méthode vide
  */
 public void mouseDragged(MouseEvent e) {
 }
 }
}
```

Remarque : si, dans cette classe incluse, nous avions défini une nouvelle variable nommée `lbBarreEtat`, comment lever l'ambiguïté entre une variable de la classe incluse et une variable de la classe englobante ? L'objet courant de la classe `FenetrePrincipale` peut être désigné par la notation : `FenetrePrincipale.this`. Ainsi, deux variables nommées `lbBarreEtat` dans la classe englobante et dans la classe incluse sont désignées respectivement par les notations : `this.lbBarreEtat` et `FenetrePrincipale.this.lbBarreEtat`. Ici, le compilateur recherche d'abord si une telle variable existe dans la classe incluse, puis, si elle n'y est pas, il appelle celle de la classe englobante.

Une classe incluse peut aussi être définie à l'intérieur d'une méthode.

La classe `MajCoordonneesListener` n'étant utilisée qu'une seule fois, il est possible de la définir directement à l'endroit où une instance de cette classe est nécessaire.

Par ailleurs, puisque cette classe est amenée à n'être utilisée qu'une seule fois, il est même inutile de lui donner un nom explicite. Nous pouvons utiliser une classe incluse anonyme.

Classes incluses anonymes

Le mécanisme des classes incluses anonymes est adapté au cas de figure où l'on a besoin à un instant donné d'une instance d'une classe, et on est certain de ne pas avoir à créer d'autres instances ailleurs, ni à référencer cette classe plus loin.

Ici, nous allons créer une instance de la classe précédemment nommée `MajCoordonneesListener`, puis abonner cette instance au panneau de dessin, et nous n'avons plus besoin de connaître cette classe `MajCoordonneesListener`.

Le titre de cette section vous a probablement donné une idée sur la caractéristique principale de la classe que nous nous apprêtons à créer : elle sera anonyme, en d'autres termes, sans nom. Comment créer une classe incluse sans lui donner un nom ? On ne peut pas utiliser l'instruction classique :

```
class XXX implements MouseMotionListener { . . . }
```

Le mécanisme consiste à appeler le constructeur et à construire en même temps la classe.

L'appel au constructeur ne nécessite pas le nom de la classe : on indique le nom de la classe héritée ou de l'interface implémentée par la classe incluse anonyme.

```
public class FenetrePrincipale extends JFrame {
   [ variables d'instance ]

   public FenetrePrincipale() {
      [ construction de l'interface… ]

     // Abonnement d'un DessinListener (classe externe) au
      // panneau
      DessinListener listDessin = new DessinListener();
      panDessin.addMouseMotionListener(listDessin);

      // Classe incluse anonyme pour la gestion des
      // coordonnées dans la barre d'état
      panDessin.addMouseMotionListener(
      ➥ new MouseMotionListener() {
       /**
        * Méthode qui met à jour le texte de la barre d'état.
        */
       public void mouseMoved(MouseEvent e) {
          lbBarreEtat.setText("Coordonnées du curseur :
          " + e.getX() + ", " + e.getY());
       }
       /**
        * Méthode vide
        */
```

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

        // Classe incluse anonyme pour gérer la fermeture de
        // la fenêtre
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.out.println("Merci d'avoir utilisé "
                  + "cette appli. ");
                System.exit(0);
            }
        });

    }
}
```

Cette portion de code contient donc deux classes incluses anonymes : l'une implémentant l'interface `MouseMotionListener` et l'autre héritant de la classe `WindowAdapter`.

Pour s'habituer à cette syntaxe un peu particulière, il est nécessaire d'avoir bien intégré les règles suivantes :

L'instruction `new MouseMotionListener() {…}` instancie un objet d'une classe qui :

- n'a pas de nom, autrement dit est anonyme ;
- dérive de `Object` ;
- implémente l'interface `MouseMotionListener`.

L'instance obtenue a été créée en utilisant le constructeur par défaut, c'est-à-dire le constructeur sans paramètre. La classe anonyme ne dispose que de ce constructeur par défaut pour la bonne raison qu'il est impossible de définir un autre constructeur dans cette classe faute de disposer d'un nom.

Une classe anonyme qui implémente une interface ne peut disposer que d'un constructeur par défaut.

Parallèlement, l'instruction `new WindowAdapter () {…}` renvoie la référence d'un objet créé dont la classe :

- est « anonyme » ;
- dérive de `WindowAdapter` ;
- n'implémente aucune interface (sauf, indirectement, celles implémentées par `WindowAdapter`).

Cette fois encore, c'est le constructeur par défaut qui a été utilisé pour l'instanciation, mais un autre constructeur de la classe mère aurait pu être invoqué.

Une conséquence de cette syntaxe particulière est qu'une classe incluse anonyme peut :

- soit hériter de `Object` et implémenter au maximum une interface ;
- soit hériter d'une autre classe et dans ce cas, elle ne peut implémenter aucune interface supplémentaire.

Utilisation des classes incluses

Les classes incluses sont largement utilisées dans les sources du JDK. Par exemple, dans la seule classe `JTable`, composant graphique correspondant à un tableau, on trouve une vingtaine de classes incluses.

Les classes incluses anonymes sont aussi utilisées dans les outils de génération de code. Ainsi, dans la plupart des environnements de développement, lorsque vous utilisez la composition visuelle d'interfaces, vous pouvez saisir directement le code de gestion de l'événement. Dans un tel cas, le code est écrit dans la méthode adéquate du listener qui est créé en tant que classe anonyme.

Toutefois, il est important de ne pas abuser des classes incluses anonymes dans la mesure où elles rendent difficile la relecture du code. Sun préconise de les utiliser uniquement si les conditions suivantes sont respectées :

- il s'agit d'une classe dont le fonctionnement n'est pas compliqué, ou qui correspond à un mécanisme classique en programmation Java (un thread, un listener) ;
- l'interface implémentée par la classe incluse ne doit pas spécifier plus de deux ou trois méthodes ;
- le nombre d'instructions dans les méthodes de la classe incluse doit être limité. Les algorithmes complexes sont à proscrire dans les classes incluses anonymes.

La classe incluse, du fait de sa portée restreinte à la classe qui la contient, ne peut pas être réutilisable. Ce fait doit donc être validé au cours de la conception.

3.1.4 Créer son propre événement

Il peut dans certains cas être utile d'implémenter son propre événement. Par exemple imaginons un champ de saisie devant émettre un événement en cas de valeur numérique et un autre événement en cas de valeur non numérique.

Pour implémenter cela, il faut prévoir toute la mécanique d'abonnement, de désabonnement et d'émission d'événements. En d'autres termes, il faut implémenter le design pattern *observer*.

Nous utiliserons ici la manière « académique » de coder ce pattern, c'est-à-dire que nous respecterons les règles de nommages standard quitte à mélanger anglais et français dans les noms des méthodes.

Nous allons appliquer les principes de la programmation objet, et isoler dans une classe tout ce qui concerne la gestion des listeners et l'émission des événements. Si nous appelons notre composant personnalisé `TOCField (TexteOuChiffre Field)`, alors nous aurons un listener nommé `TOCListener` et la classe rassemblant la gestion des événements devra se nommer `TOCListenerSupport`.

Commençons par l'interface listener :

```
public interface TOCListener {
   void isFloat(float f);
   void isNotFloat(String s);
}
```

Nous venons ainsi de définir deux événements `isFloat` et `isNotFloat`. Comme nous l'avons dit précédemment, un événement n'est en fait qu'une simple méthode. En effet, rappelons qu'un objet tel qu'une instance de `ActionEvent` représente conceptuellement l'événement utilisateur « clic sur un bouton » et contient des informations utiles sur cette action de l'utilisateur, mais programmatiquement c'est bien la méthode `actionPerformed` qui indique que cette action a eu lieu. Ici, nous n'avons pas besoin d'un objet particulier contenant des détails sur l'événement, les deux méthodes `isFloat` et `isNotFloat` avec leur paramètre sont suffisantes.

Nous verrons dans le paragraphe suivant la création d'un événement dans le cadre d'un composant JavaBean, et dans ce cas de figure, nous sentirons le besoin d'avoir une classe particulière pour encapsuler l'object « événement ».

Quel savoir faire devons-nous ajouter à notre classe support ? Elle doit permettre d'abonner et de désabonner des instances de type listener (ici `TOCListener`) et émettre l'un des deux événements vers toutes les instances qui sont abonnées. Remarquez qu'il n'est toujours pas question de composants graphiques. Nous implémentons ici un système d'événements qui sont déclenchés pour déclarer qu'une information est de type `float` ou de type `String` et ce, indépendamment de la provenance de cette information. C'est un des buts de cet exemple « académique » : la classe `TOCListenerSupport` pourrait être utilisée dans d'autres contextes que celui du champ de saisie `TOCField`.

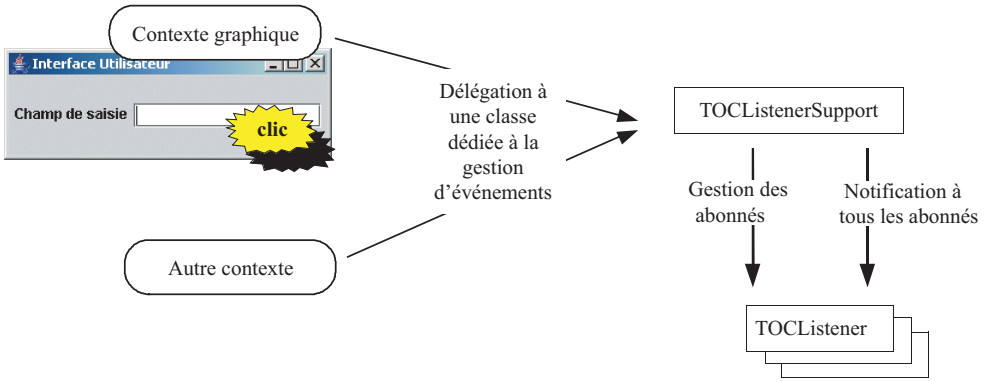


Figure 3.9 – Déléguer la gestion des événements

Voici la classe `TOCListenerSupport` :

```
public class TOCListenerSupport {
   private Collection<TOCListener> listeners =
   ➥new ArrayList<TOCListener>();

   public void addTOCListener(TOCListener listener) {
   listeners.add(listener);
   }

   public void removeTOCListener(TOCListener listener) {
   listeners.remove(listener);
   }

   public void fireIsFloat(float f) {
   for (TOCListener listener : listeners) {
      listener.isFloat(f);
      }
   }

   public void fireIsNotFloat(String s) {
   for (TOCListener listener : listeners) {
      listener.isNotFloat(s);
      }
   }
}
```

Nous avons maintenant à notre disposition une classe capable de gérer une collection de `TOCListener` et d'envoyer l'événement à tous les abonnés. La question qui se pose maintenant est « comment utiliser cette classe dans un contexte graphique » ?

Remarquez l'usage des collections permises par Java 5.
Les collections sont typées et non plus génériques. Dans les versions précédentes du langage Java (jusqu'au JDK 1.4) les collections pouvaient contenir n'importe quel type d'éléments, la seule contrainte étant d'être de type `Object`. Ici, les éléments composant la collection doivent être compatibles avec le type déclaré entre les signes < et >. Ainsi, l'instruction `Collection<TOC Listener> listeners;` déclare une variable nommée `listeners` de type `Collection` d'objets `TOCListener`. Seules des instances de classes implémentant l'interface `TOCListener` pourront être ajoutées à la collection.
De ce fait, lorsqu'on accède aux éléments de ces collections, il n'est plus nécessaire de transtyper, c'est-à-dire effectuer un cast de l'objet obtenu. Depuis Java 5, on peut aussi utiliser une version simplifiée de la boucle `for` pour parcourir une collection. Une boucle de ce type est généralement appelé boucle « for-each » car elle permet de déclarer une variable qui parcourt *chaque* élément de la collection. Ainsi, `for (TOCListener listener : listeners)` parcourt la collection `listeners` avec la variable `listener` de type `TOCListener`. En terme de performances, c'est absolument identique. Le code est plus simple à lire et plus court à écrire. De plus, le compilateur vérifie le type des instances que la collection manipule. Cela permet d'éviter les erreurs à l'exécution telles que les erreurs de transtypages (`ClassCastException`). Naturellement, il est toujours possible de déclarer une simple collection générique comme par exemple `ArrayList` qui manipule n'importe quel type d'instances. (Les méthodes d'accès aux éléments renvoient alors le type `Object`.)

Voici une sous-classe de `JTextField` qui utilise le mécanisme mis en place :

```
public class TOCField extends JTextField {

   private TOCListenerSupport listenerSupport =
   ➥new TOCListenerSupport();

   public TOCField() {
      addFocusListener(new FocusListener() {
         public void focusGained(FocusEvent e) {
         }

         public void focusLost(FocusEvent e) {
            testValeur();
         }
      });
      addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            testValeur();
         }
      });
      setColumns(20);
   }
```

```
    protected void testValeur() {
       String textCourrant = getText();
       try {
          float f = Float.parseFloat(textCourrant);
          fireIsFloat(f);
       } catch (NumberFormatException e) {
          fireIsNotFloat(textCourrant);
       }
    }

    public void addTextOuChiffreListener
    ➥(TOCListener listener) {
       listenerSupport.addTOCListener(listener);
    }

    public void removeTextOuChiffreListener
    ➥(TOCListener listener) {
       listenerSupport.removeTOCListener(listener);
    }

    protected void fireIsFloat(float f) {
       listenerSupport.fireIsFloat(f);
    }

    protected void fireIsNotFloat(String s) {
       listenerSupport.fireIsNotFloat(s);
    }
}
```

Le principe est simple : délégation à la classe de support pour la gestion des événements. Notre sous-classe utilise deux événements graphiques pour déclencher le test de la valeur. La perte du focus indique que l'utilisateur a quitté le champ pour positionner son curseur sur un autre composant, il est donc temps de tester le contenu. Une validation, déclenchée par la touche Entrée, est aussi le signe que nous devons appeler le test de la valeur.

Voici finalement comment on peut utiliser notre nouveau composant :

```
resultat = new JLabel("Pas de resultat");
texteOuchiffre = new TOCField();
texteOuchiffre.addTextOuChiffreListener
➥(new TOCListener() {
   public void isFloat(float f) {
        resultat.setText("C'est un float : " + f);
   }

   public void isNotFloat(String s) {
        resultat.setText("C'est une chaine : " + s);
   }
});
```

Il faut bien faire attention à la gestion de la mémoire lorsqu'on utilise les événements. Le *garbage collector* ne supprime que les instances qui ne sont plus la cible d'aucune référence. En l'occurrence, s'abonner à un événement revient à figurer dans une collection. Cette référence peut empêcher l'instance qui s'abonne d'être supprimée de la mémoire.

3.2 LE MODÈLE JAVABEAN

Nous allons aborder ici certaines classes d'événements qui sont directement liées au concept de JavaBean. Nous allons donc dans un premier temps nous familiariser avec le modèle JavaBean , puis nous verrons ce qui concerne les événements plus particulièrement.

Un JavaBean, souvent appelé plus simplement un Bean, est un composant logiciel réutilisable. Tous les composants Swing sont des Beans, c'est pourquoi il est intéressant d'en parler ici. En revanche, les classes d'AWT ne sont pas des Beans.

3.2.1 Qu'est-ce qu'un Bean ?

Puisque Sun a décidé d'adopter l'architecture JavaBean pour ses composants Swing, nous pouvons nous imposer de suivre cette même règle pour nos propres composants. Comment faire pour écrire des Beans ?

Prenons le cas concret suivant : nous souhaitons développer un composant graphique qui permette de saisir une heure (heure et minute) (figure 3.10). Ce composant est destiné à être utilisé dans plusieurs applications et nous allons donc en faire un Bean pour faciliter sa réutilisation par d'autres programmeurs.

Figure 3.10 — Le composant à développer.

Un Bean est constitué d'une ou plusieurs classes Java. Il n'y a aucune contrainte sur les classes à utiliser. N'importe quelle classe ou ensemble de classes peut constituer un Bean.

Avant de se lancer dans la réalisation de ce composant, il est nécessaire de s'interroger sur l'interface d'utilisation de ce composant, c'est-à-dire les membres publics que les utilisateurs de notre Bean pourront manipuler.

L'interface d'utilisation d'un tel composant doit être simple et facilement compréhensible car le but premier des Beans est la réutilisation. La construction de certaines portions d'applications peut se faire parfois uniquement en assemblant des Beans existants.

Un JavaBean peut être un composant graphique (un bouton particulier, une boîte de dialogue, etc.), ou non. Lorsqu'il s'agit d'un composant graphique, une utilisation possible est l'intégration du Bean dans un outil de développement qui propose de la composition visuelle d'interface : l'utilisateur construit son interface graphique à l'aide d'un éditeur visuel. Il choisit des composants dans des barres d'outils et les dispose à sa guise à l'écran. Le code source correspondant est alors généré automatiquement.

Il faut réfléchir à l'interface d'utilisation de notre Bean. Nous devons définir :

- les propriétés. Elles seront implémentées par des accesseurs publiques. Les attributs correspondants sont toujours internes (*private* ou *protected*). Ils ne font pas partie de l'interface ;
- les méthodes publiques ;
- les événements que peut générer le composant.

Pour notre « Panneau horaire », l'interface publique du Bean final est constituée de :

```
public int getHeure()
public int getMinute()
public void setMinute(int min)
public void setHeure(int h)
public void addPropertyChangeListener
  (PropertyChangeListener l)
public void removePropertyChangeListener
  (PropertyChangeListener l)
```

Maintenant que nous avons déterminé l'interface publique, la seconde chose à faire est de construire notre composant « Panneau horaire » à partir des composants Swing standard, comme n'importe quel programme que nous avons élaboré jusqu'à présent.

Nous avons décidé de factoriser le composant de base qui est constitué d'un `JTextField` et de deux flèches de modification. Nous aurons donc deux instances de ce panel que nous avons appelé `CaseReglable`. Cette classe permet de faire varier un entier entre deux bornes fixées lors de l'instanciation. Voici cette classe :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.ArrayList;
import java.util.Iterator;
```

```
public class CaseReglable extends JPanel {

  private class ReglageListener implements ActionListener {

   public void actionPerformed(ActionEvent e) {
     if (e.getSource() == btPlus)  {
       value++;
       if (value > max) value = min;
     } else {
       value--;
       if (value < min) value = max;
     }
     tfCase.setText(String.valueOf(value));
    }
}

   protected BorderLayout layout = new BorderLayout();
   protected JTextField tfCase = new JTextField();
   protected JPanel pReglage = new JPanel();
   protected GridLayout reglageLayout =
   ➥new GridLayout(2, 1);
   protected ImageIcon iiPlus =
   ➥new ImageIcon("plus.gif");
   protected ImageIcon iiMoins =
   ➥new ImageIcon("moins.gif");
   protected JButton btPlus = new JButton(iiPlus);
   protected JButton btMoins = new JButton(iiMoins);
   protected ReglageListener reglageListener =
                                   new ReglageListener();
     protected int value = 0;
     protected int max;
     protected int min;
     protected ArrayList listeners = new ArrayList();

     public CaseReglable(int min, int max) {
           this.min = min;
           this.max = max;
           setLayout(layout);
           add(tfCase, BorderLayout.CENTER);
           pReglage.setLayout(reglageLayout);
           pReglage.add(btPlus);
           pReglage.add(btMoins);
           // Pour régler la largeur du JTextField :
           // Le nombre de chiffres du nombre
           // le plus grand +1
           tfCase.setColumns(
           ➥(String.valueOf(max)).length()+1);
           // On impose aux boutons de prendre la largeur
           // de l'icône.
           btPlus.setPreferredSize(new Dimension(
                                   iiPlus.getIconWidth(),
                                   btPlus.getHeight()));
```

```
            btMoins.setPreferredSize(new Dimension(
                                    iiMoins.getIconWidth(),
                                    btMoins.getHeight()));
            add(pReglage, BorderLayout.EAST);
            setValue(0);
            // On ajoute les listeners
            btPlus.addActionListener(reglageListener);
            btMoins.addActionListener(reglageListener);
      }

      public int getValue() {
            return value;
      }

      public void setValue(int i) {
            value = i;
            tfCase.setText(String.valueOf(value));
      }

      public String getText() {
            return String.valueOf(getValue());
      }

      public void addClicListener(ClicCase cc) {
            listeners.add(cc);
      }

      public void removeClicListener(ClicCase cc) {
            listeners.remove(cc);
      }

      protected void clic(int oldValue) {
            Iterator ite = listeners.iterator();
            ClicCase cc;
            ClicEvent ce = new ClicEvent(this, oldValue,
            ➥getValue());
            while (ite.hasNext())  {
                  cc = (ClicCase)ite.next();
                  cc.clic(ce);
            }
      }
}
```

Nous avons voulu cette classe assez générique pour mieux illustrer le fait qu'il est possible d'implémenter de toutes pièces un système d'événement basé sur le pattern *observer*. Notre objet `CaseReglable` est capable d'envoyer un événement `ClicEvent`. Cet événement est envoyé chaque fois que la valeur de la case change. La classe d'événement n'hérite volontairement pas de la classe `EventObject`. En effet, étant donné que nous gérons nous-mêmes les événements, depuis leur création jusqu'à leur capture, nous ne sommes pas du tout dépendants des classes

existantes. Nous pouvons choisir d'hériter de `EventObject` pour des raisons de facilité, mais ce n'est en aucun cas une obligation. Voici notre classe d'événement :

```
public class ClicEvent {
     protected int oldValue;
     protected int newValue;
     protected Object source;

     public ClicEvent(Object s, int ov, int nv) {
           source = s;
           oldValue = ov;
           newValue = nv;
     }

     public Object getSource() {
           return source;
     }

     public int getNewValue() {
           return newValue;
     }

     public int getOldValue() {
           return oldValue;
     }
}
```

Sur réception de l'événement il sera donc possible de connaître la nouvelle valeur de la case ainsi que l'ancienne.

Il ne reste plus qu'à construire une interface, l'équivalent d'un listener :

```
public interface ClicCase {
     public void clic(ClicEvent cv);
}
```

Toute classe désireuse d'être informée par le biais de notre événement `ClicEvent` d'un changement dans une case devra donc implémenter cette interface et utiliser ensuite la méthode `addClicListener` de `CaseReglable` pour s'abonner à l'événement.

C'est exactement ce que fait le panneau `PanneauHoraire`. Ce dernier est une sous-classe de `JPanel` qui comporte deux instances de `CaseReglable`, une pour les heures et une pour les minutes.

La classe `CaseReglable` doit gérer une collection des objets implémentant l'interface `ClicCase`. Cette collection est maintenue à jour par deux méthodes qui permettent d'ajouter ou de supprimer une instance de la collection. Cela signifie qu'une classe s'est abonnée ou désabonnée de notre listener. Ces méthodes sont `addClicListener` et `removeClicListener`.

Voici le code de cette classe :

```
import java.awt.*;
import javax.swing.*;
import java.beans.*;
import java.util.ArrayList;
import java.util.Iterator;

public class PanelHoraire extends JPanel {
  protected class ClicListener implements ClicCase {
    public void clic(ClicEvent cv) {
      if (cv.getSource() == crHeure)  {
        propertyChange("heure", cv.getOldValue(),
        ➥cv.getNewValue());
      } else {
        propertyChange("minute", cv.getOldValue(),
        ➥cv.getNewValue());
      }
   }
  }
  protected BoxLayout layout =
  ➥new BoxLayout(this, BoxLayout.X_AXIS);
  protected CaseReglable crHeure =
  ➥new CaseReglable(0, 23);
  protected CaseReglable crMinute =
  ➥new CaseReglable(0, 59);
  protected ArrayList listeners = new ArrayList();
  protected ClicListener clicListener = new
  ➥ClicListener();

    public PanelHoraire() {
          add(crHeure);
          add(crMinute);
          crHeure.addClicListener(clicListener);
          crMinute.addClicListener(clicListener);
    }

    public int getHeure() {
          return crHeure.getValue();
    }

    public int getMinute() {
          return crMinute.getValue();
    }

    public void setMinute(int min) {
          int oldMin = getMinute();
          crMinute.setValue(min);
          propertyChange("minute", oldMin, min);
    }

    public void setHeure(int h) {
          int oldHeure = getHeure();
```

```
            crHeure.setValue(h);
            propertyChange("heure", oldHeure, h);
        }

        public void addPropertyChangeListener
        ➥(PropertyChangeListener l) {
            listeners.add(l);
        }

        public void removePropertyChangeListener
        ➥(PropertyChangeListener l) {
            listeners.remove(l);
        }

        protected void propertyChange(String name,
                                      int oldValue,
                                      int newValue) {
            Iterator ite = listeners.iterator();
            PropertyChangeListener l;
            PropertyChangeEvent evt = new
            ➥PropertyChangeEvent(this,
                                  name,
                                  new Integer(oldValue),
                                  new Integer(newValue));
            while (ite.hasNext())  {
                l = (PropertyChangeListener)ite.next();
                l.propertyChange(evt);
            }
        }
    }
```

Remarquez l'implémentation du listener en tant que classe incluse. Le listener se contente de relayer l'événement. Pour `PanneauHoraire`, un `ClicEvent` correspond à un changement d'une de ses deux propriétés : les heures ou les minutes. `PanneauHoraire` propose donc lui aussi des méthodes `add/removeProperty ChangeListener`, ce qui permet à des classes clientes de s'abonner et d'être prévenues qu'une propriété a changé. Cette fois nous nous plaçons dans le cadre du standard JavaBean, nous utilisons donc le listener `PropertyChangeListener` et la classe d'événement associée `PropertyChangeEvent`. On peut voir l'instanciation de cette classe dans la méthode `propertyChange`. Nous verrons plus loin plus de détails sur le fonctionnement de ce listener particulier.

Comme nous l'avons dit plus haut, une classe d'implémentation d'un Bean est une classe Java quelconque. Elle doit juste implémenter un constructeur sans paramètre, respecter certaines conventions de nommage et éventuellement implémenter l'interface `Serializable` ou `Externalizable`. La classe d'implémentation de notre Bean est donc bien `PanneauHoraire`. La classe `Case- Reglable` n'est pas un Bean du fait de son constructeur avec paramètres. Cette classe fait partie de l'implémentation interne de notre Bean.

Un composant Bean peut être composé de ressources, comme des images, des fichiers de paramétrages ou multimédia. Notre Bean contient par exemple deux images représentant les triangles qui sont affichés sur les boutons d'une case réglable.

Nous n'avons malgré tout pas encore un Bean. Comme nous l'avons dit, un Bean est intégrable à un environnement de développement. Pour ce faire, l'outil doit savoir trouver la classe principale qui reflète l'interface du composant parmi l'ensemble des classes qui composent le Bean. Cela n'est pas déductible automatiquement car un Bean peut tout à fait en utiliser un autre en interne. Ce serait par exemple le cas si notre classe `CaseReglable` était elle-même un Bean. Il est donc nécessaire de spécifier cette classe dans un fichier texte de description du Bean. Ce fichier fait partie du standard Bean. D'autre part, tous les éléments qui composent un Bean doivent être encapsulés dans un fichier jar. C'est donc dans le fichier jar que nous devons trouver ce fichier de description. Il doit avoir le même nom que le fichier jar et se terminer par l'extension « manifest ». C'est pourquoi on parle de « fichier manifest ». Voici notre fichier manifest :

```
Manifest-Version: 1.0
Name: PanelHoraire.class
Java-Bean: True
```

Le nom du fichier manifest (ici `paneauHoraire.manifest`) doit être le même que le nom du fichier jar. En revanche, il n'y a pas de contrainte de nommage sur le nom de la classe technique de Bean (ici `PanelHoraire`).

Nous allons maintenant détailler ces conventions de nommage.

3.2.2 Les conventions de nommage

Toujours dans l'optique de faciliter la réutilisation, Sun fournit des conventions de nommage. Il n'y a pas réellement d'obligation de respecter ces conventions, il est possible de produire un composant Bean sans respecter ces normes. Dans ce cas, vous aurez plus de travail car vous devrez indiquer l'interface publique de votre composant d'une autre façon, en utilisant l'interface `BeanInfo`.

Le fait de respecter les conventions constitue donc un gain de temps. Par ailleurs, ces conventions sont très intuitives et on les respecte généralement sans y réfléchir.

Les propriétés d'un Bean sont caractérisées uniquement par leurs accesseurs. La ou les variables d'instances qui servent à l'implémentation d'une propriété doivent être privées car elles n'intéressent pas l'utilisateur du Bean, et il n'a pas à les manipuler directement. L'existence de deux accesseurs du type :

```
public int getHeure() ;
public void setHeure(int h) ;
```

indique l'existence d'une propriété de nom `heure`.

Les noms de ces accesseurs doivent être du type `getXxx` et `setXxx`, ou bien `isXxxx` et `setXxx`, dans le cas d'une propriété de type booléen.

Dans le cas d'une propriété qui est une collection d'éléments, il faut fournir en plus des deux accesseurs habituels, des méthodes de manipulation d'un élément : `getXxx(int index)` et `setXxx(int index, <typeElement> <valeurElement>)`.

3.2.3 Les propriétés liées

Souvent, le changement de valeur d'une propriété particulière est susceptible d'intéresser d'autres composants, ou plus généralement d'autres parties de l'application.

Notre Bean comporte deux propriétés liées : les minutes et les heures car leur variation entraîne l'émission d'un événement de type `PropertyChange`.

Une propriété est dite *liée* si le changement de sa valeur génère un événement.

L'événement « changement de valeur » d'une propriété liée peut être de deux types : `PropertyChangeEvent` ou plus simplement `ChangeEvent`. La classe `ChangeEvent` est une nouvelle classe qui se trouve dans le package `javax.swing.event`, alors que la classe `PropertyChangedEvent` se trouve dans `java.beans`.

La classe `PropertyChangeEvent` encapsule plusieurs informations qui peuvent être émises lorsqu'une propriété change de valeur.

```
public PropertyChangeEvent(Object objetSource, String
nomPropriete,Object ancienneValeur, Object nouvelleValeur) ;
```

Remarque : si la valeur est de type primitif, il faut la convertir en son équivalent objet.

La classe émettrice doit implémenter les deux méthodes qui permettent respectivement l'abonnement et le désabonnement de listeners :

```
public void addPropertyChangeListener
(PropertyChangeListener l)
public void removePropertyChangeListener
(PropertyChangeListener l)
```

La classe émettrice doit également modifier l'accesseur en modification correspondant à la propriété afin d'émettre un événement. Cet accesseur doit donc :

- créer un objet `PropertyChangeEvent` ;
- appeler la méthode `propertyChange` de chaque `PropertyChangeListener` abonné en passant l'événement en paramètre, **après avoir modifié la valeur de la propriété.**

Tout le travail de gestion des listeners est une tâche de développement récurrente. Nous l'avons par exemple fait deux fois dans ce paragraphe : une fois dans `CaseReglable` pour gérer notre propre système d'événements et une fois dans la classe `PanelHoraire` pour gérer les événements de type `PropertyChangeEvent`. Il existe une classe utilitaire dans le package `java.awt.bean` qui peut faire cela pour nous. En général, on utilisera cette classe comme un attribut d'une classe susceptible d'être source d'événements. On agit ensuite par délégation sur cet attribut. La classe `PropertyChangeSupport` gère donc pour nous une collection de listeners, propose des méthodes du type `addPropertyChangeListener` et `removePropertyChangeListener` ainsi que la faculté d'envoyer l'événement aux listeners abonnés.

3.2.4 Les propriétés contraintes

Lorsque la modification d'une propriété peut être acceptée ou refusée par un listener, nous dirons que cette propriété est contrainte *(constraint property)*. Son implémentation et son utilisation ressemblent à celles d'une propriété liée.

L'événement émis reste le même que pour une propriété liée. En revanche, le listener qui doit accepter ou refuser le changement doit implémenter l'interface : `VetoableChangeListener`.

Cette interface décrit la méthode :

```
public void vetoableChange(PropertyChangeEvent evt)
   throws PropertyVetoException
```

La seule différence par rapport à une propriété liée est que l'exception `PropertyVetoException` est levée si le listener refuse la modification. Naturellement, un client qui souhaite recevoir l'événement doit tenir compte de l'exception.

La classe émettrice doit implémenter les deux méthodes :

```
public void addVetoableChangeListener
➥(VetoableChangeListener l)
public void removeVetoableChangeListener
➥(VetoableChangeListener l)
```

qui permettent l'abonnement et le désabonnement de listeners.

Enfin, elle doit modifier l'accesseur en modification correspondant à la propriété :

- créer un objet `PropertyChangeEvent` ;
- appeler la méthode `vetoableChange` de chaque `VetoableChangeListener` abonné en passant l'événement en paramètre, **avant de modifier la valeur de la propriété**.

Le package `java.awt.bean` fournit une classe de support semblable à la classe que nous avons vu précédemment, `PropertyChangeSupport`, mais qui prend en compte l'exception. Cette classe `VetoableChangeSupport` est en tout point identique à `ProperyChangeSupport`, si ce n'est l'exception `PropertyVeto Exception`.

3.3 APPLICATION À L'EXEMPLE DE GESTION DE SIGNETS

Dans cette section, nous allons poursuivre l'implémentation de notre application de gestion de signets. Pour le moment, la fenêtre principale de l'application contient des champs statiques, qui ne réagissent pas aux interactions de l'utilisateur. Nous allons ajouter la gestion des événements.

3.3.1 La fermeture de l'application

Le premier événement à gérer est la fermeture de l'application. Deux actions de l'utilisateur doivent provoquer cette fermeture : le clic sur la case système de fermeture (par exemple, la croix en haut à droite de la fenêtre sous Windows, ou le menu contextuel de l'icône de la fenêtre), et la sélection du menu « Quitter ».

Ce que notre programme devra effectuer alors, c'est l'affichage d'un message de sortie, comme « Merci d'avoir utilisé l'application Signet », puis quitter le programme. Nous pourrions aussi choisir d'enregistrer automatiquement l'arbre des signets, ou bien nous pourrions demander un message de confirmation à l'utilisateur.

Choisissons la solution minimaliste : un message de sortie et la fermeture de l'application. Ce code sera donc bref :

```
System.out.println("Merci d'avoir utilisé l'application
➥Signet");
System.exit(0);
```

Cependant, nous souhaitons pouvoir faire évoluer ce code plus tard, par exemple pour ajouter une demande de confirmation. Pour que l'évolution de l'application soit la plus aisée possible, nous devons donc partager ce code pour les deux événements à gérer.

Voyons comment réaliser cela.

Tout d'abord, identifions les composants source des événements. Ils sont deux : la fenêtre elle-même et l'item de menu « Quitter ».

Recherchons les méthodes « `add...Listener` » adéquates.

Pour la fenêtre, la méthode adéquate est :

```
public void addWindowListener (WindowListener l) ;
```

Pour l'item de menu, il s'agit de :

```
public void addActionListener (ActionListener l) ;
```

Pour les deux interfaces listener citées, les méthodes à implémenter sont respectivement :

```
public void windowClosing (WindowEvent e) ;
```

et :

```
public void actionPerformed (ActionEvent e) ;
```

La question qui se pose ensuite est de savoir où écrire ces classes listener. Étant donné que le code à écrire va être très simple et que ce listener est spécifique à notre application, nous pouvons choisir la solution des classes incluses anonymes.

Pour partager le code de sortie afin d'améliorer l'évolutivité, nous écrivons une méthode `fermerApplication` dans notre classe `FenetrePrincipale`.

Ainsi, le code (simplifié) de `FenetrePrincipale` est le suivant :

```
package ihm;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
[ autres import nécessaires ]

public class FenetrePrincipale extends JFrame {

[ Variables d'instance et de classe ]

   /**
    * Constructeur protégé
    */
   protected FenetrePrincipale() {
      [ initialisation des variables d'instances ]
      initialiserMenus();
      // fermeture de la fenêtre par la case système
      addWindowListener( new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            fermerApplication();
         }
```

```
      });
   }

   /**
    * Crée et affecte les menus à la fenêtre.
    */
   protected void initialiserMenus() {
      JMenu menuFichier = new JMenu("Fichier");
      // menu ouvrir
      JMenuItem ouvrir = new  JMenuItem("Ouvrir");
      menuFichier.add(ouvrir);
      // menu enregistrer
      JMenuItem enregistrer = new JMenuItem("Enregistrer");
      menuFichier.add(enregistrer);
      // menu quitter
      JMenuItem quitter = new JMenuItem("Quitter");
      quitter.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent eve) {
            fermerApplication();
         }
      });
      menuFichier.addSeparator();
      menuFichier.add(quitter);
      JMenuBar mb = new JMenuBar();
      mb.add(menuFichier);
      setJMenuBar(mb);
   }

   private void fermerApplication() {
      System.out.println("Merci d'avoir utilisé
      l'application Signet");
      System.exit(0);
   }
   [ autres méthodes ]
}
```

3.3.2 Gestion des menus « Ouvrir » et « Enregistrer »

Le cœur de notre application consiste à faire persister une arborescence de signets. Pour cela, nous utilisons un fichier dans lequel l'utilisateur doit enregistrer son arbre.

Les deux items de menu « Ouvrir » et « Enregistrer » servent respectivement à choisir sur le disque le fichier qui contient les signets et à choisir celui dans lequel on souhaite enregistrer l'arbre.

Pour permettre ce choix, nous allons utiliser un composant Swing particulier : un `JFileChooser` (figure 3.11).

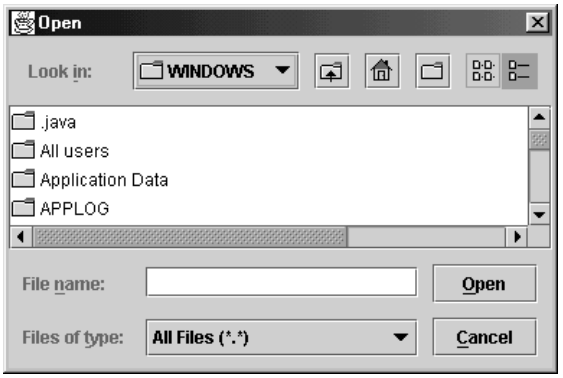


Figure 3.11 — Choix d'un fichier dans l'application de gestion de signets.

Généralités sur le `JFileChooser`

Ce composant offre un mécanisme simple pour choisir un fichier ou un répertoire sur le disque. Il existe deux manières d'utiliser ce composant :

- créer une instance et l'ajouter à un container comme on fait habituellement pour tous les composants déjà vus ;
- utiliser les méthodes statiques de `JFileChooser` qui permettent de créer directement une boîte de dialogue modale pour le choix d'un fichier.

Cette deuxième solution est la plus utilisée car elle est simple à mettre en œuvre et correspond à notre besoin pour l'application. Les méthodes statiques sont au nombre de trois, suivant si l'on souhaite ouvrir une boîte de dialogue pour l'ouverture d'un fichier, pour la sauvegarde ou pour une autre opération. Ces trois méthodes prennent en paramètre le composant « maître » de cette boîte de dialogue.

Par exemple :

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

L'entier retourné par cette méthode permet de savoir si l'utilisateur a cliqué sur le bouton « Ouvrir », ou « Annuler ». On accède ensuite au fichier choisi grâce à `getSelectedFile`.

```
JFileChooser fc = new JFileChooser();
int retour = fc.showOpenDialog(maFenetreParent);
if (retour == JFileChooser.APPROVE_OPTION) {
   File fichier = fc.getSelectedFile();
   //traitements...
}
```

Il est possible de paramétrer le répertoire sur lequel est positionné le `JFileChooser` à l'ouverture à l'aide de la méthode `setCurrentDirectory`. Si on ne précise rien, il est positionné sur le répertoire HOME de l'utilisateur.

Un problème couramment posé par les programmeurs francophones est de pouvoir afficher cette boîte de dialogue standard avec les textes des boutons en français. En ce qui concerne le bouton correspondant au choix « OK », une méthode est proposée : `setApproveButtonText`.

Pour les autres textes, il est nécessaire de changer des propriétés de l'`UIManager` de la façon suivante :

```
UIManager.put("FileChooser.cancelButtonText","Annuler");
```

Ce composant offre aussi la possibilité de définir des filtres en fonction des types de fichiers. Pour cela, il faut créer une sous-classe de `FileFilter`, dans laquelle la méthode `accept` qu'il faut implémenter, permet de déterminer si un type de fichier est accepté par ce filtre ou non. Une fois cette classe « filtre » créée, on ajoute une instance au `JFileChooser`. Notre filtre sera alors proposé dans le champ « Type de fichier » qu'on trouve habituellement dans les boîtes de dialogue standard.

```
fc.addChoosableFileFilter(monFileFilter);
```

Application à l'exemple de gestion des signets

Les deux composants source des événements à gérer sont les deux items de menu « Ouvrir » et « Quitter ». La méthode d'abonnement est donc là encore `addActionListener`, et le listener est un `ActionListener`.

Nous pouvons également utiliser ici des classes incluses anonymes. Complétons la méthode `initialiserMenus` de la classe `FenetrePrincipale` que nous avons présentée dans le paragraphe précédent.

Lorsque l'utilisateur choisit un fichier à ouvrir, nous devons appeler le gestionnaire de nœuds qui se charge de lire les objets sérialisés dans un fichier. Pour plus d'information sur ce fonctionnement, reportez-vous aux explications du premier chapitre ou consultez le code complet de l'exemple disponible sur le site de Dunod (www.dunod.com).

```
// menu ouvrir : pour charger depuis un fichier l'arbre
// de signets
JMenuItem ouvrir = new    JMenuItem("Ouvrir");
ouvrir.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent eve) {
    JFileChooser fc = new JFileChooser();
    int returnVal =
    ➥fc.showOpenDialog(FenetrePrincipale.this);
    if(returnVal == JFileChooser.APPROVE_OPTION) {
```

```
            [... appel à la méthode ouvrir du gestionnaire de
            ➥nœuds et mise à jour de l'arbre... ]
        }
      }
});
```

De la même façon, le menu « Enregistrer » provoque l'ouverture d'une fenêtre d'enregistrement d'un fichier. Lorsque l'utilisateur a effectué son choix, on déclenche l'opération de sérialisation.

```
// menu enregistrer : pour enregistrer dans un fichier
// l'arbre des signets
JMenuItem enregistrer = new   JMenuItem("Enregistrer");
enregistrer.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent eve) {
    JFileChooser fc = new JFileChooser();
    int returnVal = fc.showSaveDialog
    ➥(FenetrePrincipale.this);
    if(returnVal == JFileChooser.APPROVE_OPTION) {
      [...appel à la méthode enregistrer du gestionnaire
      ➥ de nœuds et mise à jour de l'arbre...]
    }
  }
});
```

4

Les composants plus complexes

Vous avez pu constater jusqu'ici que l'utilisation des composants Swing se révélait relativement aisée lorsque l'on connaît les principes de base présentés dans les chapitres précédents. En effet, la lecture de la documentation Java d'un composant graphique suffit généralement pour savoir comment l'employer et une connaissance approfondie du *framework* de la librairie ne semble pas à première vue nécessaire.

Il en va autrement pour les composants que nous avons choisi de qualifier de « complexes ». Cet adjectif ne doit pas vous inquiéter : on entend par « composant complexe » un composant qu'il est possible de paramétrer très finement et dont l'utilisation dans un contexte logiciel exigeant peut se révéler complexe.

Ces composants complexes peuvent très bien être employés sans originalité aucune, en appliquant tels quels les constructeurs les plus simples. C'est cette approche que nous allons montrer tout d'abord dans le premier paragraphe.

Toutefois, il est bien rare que cette utilisation sommaire suffise à composer une interface graphique efficace et attrayante. Nous allons donc ensuite parler des différents aspects de la personnalisation des composants complexes. Ces aspects vont nous amener à découvrir des notions fondamentales sur lesquelles repose la librairie Swing.

Nous traiterons tout d'abord du paramétrage de la sélection, puis de celui de l'apparence d'un composant à l'aide des *renderers*. Nous verrons ensuite comment personnaliser la façon dont un composant est édité par l'utilisateur à

l'aide des *editors*. Enfin, nous présenterons le concept qui est au cœur du *framework* de Swing : l'architecture Modèle-Vue-Contrôleur.

4.1 UNE PREMIÈRE UTILISATION DES COMPOSANTS COMPLEXES

Avant de nous lancer dans la personnalisation des composants, nous allons présenter brièvement ce que nous avons appelé ici les « composants complexes ». Ce sont :

- `JList`
- `JTable`
- `JTree`

Le composant `JComboBox` peut aussi être personnalisé selon les mêmes procédés que ces trois composants. Nous n'allons pas le présenter à nouveau car il a déjà été abordé dans le premier chapitre.

Pour pouvoir traiter de petits exemples simples avant d'aborder l'utilisation optimale de ces composants complexes, voyons tout d'abord comment les instancier de la façon la plus simple.

Pour se familiariser avec ces trois nouveaux composants, essayons de créer l'interface suivante, qui permet de sélectionner des renseignements sur un salarié d'une entreprise. Cette fenêtre contient les trois composants précités. Nous supposerons que cette fenêtre n'est qu'une partie d'une grosse application de gestion du personnel. Voici donc figure 4.1 la fiche d'affectation d'une personne.

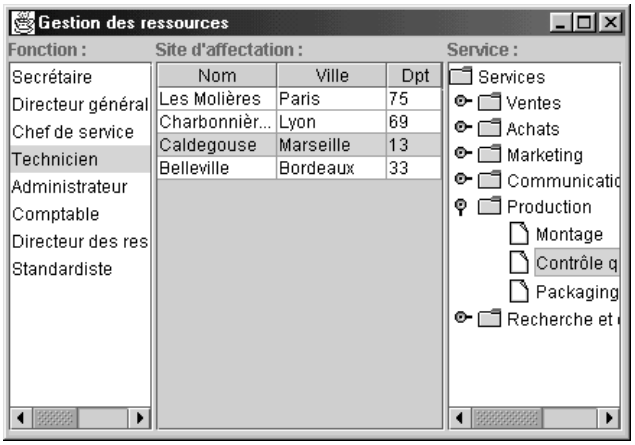


Figure 4.1 — Une interface contenant les trois composants complexes.

4.1.1 Utilisation simple d'une `JList`

Une caractéristique commune à tous ces composants est d'afficher une liste de valeurs. Pour cela, ils offrent tous la possibilité de se construire à partir d'un vecteur d'objets. Ces objets ne doivent respecter qu'une seule contrainte : implémenter une méthode `toString` qui retourne ce qui sera affiché et qui doit donc être intelligible pour l'utilisateur. Ici, on ajoute directement des objets `String` dans la liste, la méthode `toString` n'a donc pas besoin d'être redéfinie.

Ainsi la construction de notre première `JList` se fait de la façon suivante :

```
Vector vListe = new Vector();
vListe.addElement("Secrétaire");
vListe.addElement("Directeur général");
vListe.addElement("Chef de service");
vListe.addElement("Technicien");
vListe.addElement("Administrateur");
vListe.addElement("Comptable");
vListe.addElement("Directeur des ressources humaines");
vListe.addElement("Standardiste");
JList liste = new JList(vListe);
```

Dans le premier chapitre, nous avons évoqué l'interface `Scrollable` qui permet à un composant d'être affiché dans un panneau de défilement. `JList` implémente déjà cette interface. Pour obtenir l'affichage d'une liste dans un `JScrollPane`, il suffit de procéder comme suit :

```
JScrollPane jspListe = new JScrollPane(liste);
```

Vous avez pu constater la simplicité d'utilisation de ce composant. Nous verrons plus loin que le fait de passer un vecteur de chaînes de caractères en paramètre n'est vraiment pas la façon la plus optimale d'utiliser la classe `JList`.

4.1.2 Utilisation simple d'un `JTable`

Poursuivons notre exemple avec le `JTable`. À la différence de `JList`, `JTable` affiche une matrice de données. Les objets qui doivent donc être indiqués en paramètres du constructeur correspondent à des « lignes ». Chaque ligne contient plusieurs valeurs.

Le procédé le plus simple pour utiliser cette classe est donc de passer un vecteur de vecteurs en paramètre.

Nous devons également indiquer le texte des en-têtes de colonnes.

Notre exemple se fait donc très simplement de la façon suivante :

```
// Construction du vecteur des en-têtes
Vector vNomColonnes = new Vector();
vNomColonnes.addElement("Nom");
vNomColonnes.addElement("Ville");
vNomColonnes.addElement("Dpt");

// Construction des lignes
Vector ligne1 = new Vector();
ligne1.addElement("Les Molières");
ligne1.addElement("Paris");
ligne1.addElement("75");
[ lignes suivantes...]

// Construction du vecteur contenant les lignes
Vector vtableau = new Vector();
vtableau.addElement(ligne1);
vtableau.addElement(ligne2);
vtableau.addElement(ligne3);
vtableau.addElement(ligne4);

// Instanciation du composant JTable
JTable tableau = new JTable(vtableau, vNomColonnes);
```

4.1.3 Utilisation simple d'un `JTree`

La création d'un arbre est à peine plus complexe que les exemples précédents. On observe cependant une différence notable : pour véhiculer une arborescence, la notion de vecteur ne convient pas.

Certes, il existe un constructeur qui ne prend qu'un vecteur ou un tableau d'objets en paramètre, mais dans ces conditions, tous les éléments du vecteur sont affichés à un même niveau d'arborescence : un nœud racine qui n'est pas visible contient tous les éléments du vecteur passé en paramètre.

Nous allons voir dans ce paragraphe comment créer simplement des objets arborescents.

Le vocabulaire utilisé pour manipuler un arbre correspond au vocabulaire standard : on parle de **nœuds**. Les relations d'appartenance d'un nœud à un autre sont dites « parent-enfant ». Le nœud placé le plus haut dans la hiérarchie est appelé la **racine**. Les nœuds qui n'ont pas d'enfants sont des **feuilles**.

Dans un objet `JTree`, les nœuds sont représentés par des objets `TreeNode`. Sur cette classe, il existe des méthodes pour accéder au parent et à la liste des enfants.

Un constructeur de la classe `JTree` attire l'attention parce qu'il semble simple à utiliser et parce qu'il paraît correspondre à notre souhait d'indiquer une arborescence. Il s'agit de :

```
public JTree (TreeNode root)
```

Le paramètre est un objet implémentant l'interface `TreeNode`. Cette interface se trouve dans le package `javax.swing.tree`. Une classe instanciable implémentant `TreeNode` est `DefaultMutableTreeNode`. Malgré un nom peu engageant, cette classe est facile à manipuler Cet objet permet bien de manipuler un arbre : on peut lui ajouter des enfants, accéder à son parent, ainsi qu'à la liste de ses enfants et ainsi de suite.

Le constructeur qui va nous servir ici est celui qui permet d'associer cet objet « nœud » avec un objet de notre convenance. Cet objet subit la même contrainte que pour les listes et les tableaux, c'est-à-dire que la méthode `toString` sera utilisée par le `JTree` pour obtenir la chaîne de caractères à afficher.

La création des « nœuds » se fait donc de la façon suivante :

```
DefaultMutableTreeNode racine =
➥new DefaultMutableTreeNode("Services");
DefaultMutableTreeNode n1 =
➥new DefaultMutableTreeNode("Ventes");
DefaultMutableTreeNode n2 =
➥new DefaultMutableTreeNode("Achats");
DefaultMutableTreeNode n3 =
➥new DefaultMutableTreeNode("Marketing");
[ et ainsi de suite...]
```

L'organisation en arbre s'effectue grâce à la méthode `add` :

```
DefaultMutableTreeNode n5 =
➥new DefaultMutableTreeNode("Production");
DefaultMutableTreeNode n51 =
➥new DefaultMutableTreeNode("Montage");
DefaultMutableTreeNode n52 =
➥new DefaultMutableTreeNode("Contrôle qualité");
DefaultMutableTreeNode n53 =
➥new DefaultMutableTreeNode("Packaging");
n5.add(n51);
n5.add(n52);
n5.add(n53);
```

L'instanciation de l'arbre consiste juste en :

```
JTree arbre = new JTree(racine);
```

Comme précédemment, il est nécessaire d'ajouter ce composant dans un `JScrollPane` si on veut bénéficier des barres de défilement.

Une autre notion utile pour manipuler les arbres est la notion de `TreePath`. Ce type d'objets permet d'encapsuler le chemin d'un nœud dans un arbre. Un chemin est constitué des différents nœuds par lesquels il faut passer depuis la racine pour accéder à un élément.

Il est possible de forcer par programmation le fait qu'un nœud en particulier soit visible, c'est-à-dire que le panneau de défilement déplace sa « vue » sur une partie précise de l'arbre. Pour cela, on indique un `TreePath` en paramètre de `scrollPathToVisible`.

```
public void scrollPathToVisible(TreePath path)
```

Il est possible d'indiquer à un arbre si on souhaite que la racine soit visible ou non, à l'aide de la méthode `setRootVisible(boolean b)`. On peut aussi indiquer si les « poignées » qui représentent l'état « déployé » ou non d'un nœud doivent être affichées, à l'aide de la méthode `setShowsRootHandles(boolean b)`.

4.2 PERSONNALISER LA SÉLECTION

De nombreux composants Swing permettent d'afficher à l'écran plusieurs données distinctes. Il s'agit des listes, des listes déroulantes, des tableaux, des arbres, etc.

Pour ces différents composants, la sélection est le seul moyen pour l'utilisateur d'interagir avec le composant afin d'exprimer des choix.

Le mécanisme de sélection vous apparaît peut-être simple, standardisé et immuable. Il n'en est rien.

Il est important de mettre en place un mécanisme de sélection adapté au contexte : l'utilisateur doit-il sélectionner une ou plusieurs données, y a-t-il une valeur sélectionnée par défaut, est-il possible de tout désélectionner ?

Avant de nous engager dans la personnalisation complète d'un composant graphique, étudions tout d'abord ce comportement des composants : le mécanisme de sélection.

4.2.1 Le mode de sélection

Le mode de sélection représente la règle qui indique les types d'intervalles disponibles pour une sélection d'items. Par exemple, l'utilisateur a-t-il le droit de sélectionner un seul item, un ensemble d'items contigus, ou plusieurs items sans contrainte de contiguïté ?

Sur un composant `JList`, il existe plusieurs politiques de sélection possibles. Il est important de bien paramétrer cela afin de guider l'utilisateur dans sa saisie

et surtout d'empêcher des saisies aberrantes. Comparez les différentes politiques de sélection proposées figure 4.2.

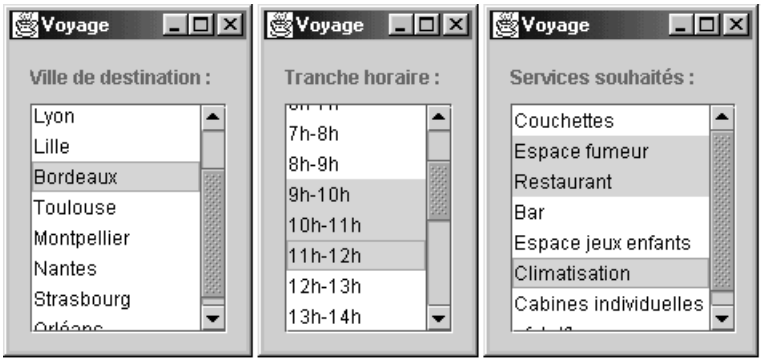


Figure 4.2 — Plusieurs politiques de sélection.

La première liste autorise une sélection unique : l'utilisateur doit choisir une et une seule ville de destination pour préparer son voyage. La seconde autorise la sélection multiple, mais seulement pour des items contigus. En effet, l'utilisateur doit indiquer la tranche horaire pour son horaire de départ : il doit choisir un intervalle d'une heure ou plus mais il doit s'agir d'un intervalle continu. La troisième liste présente des services divers : l'utilisateur est libre d'en choisir éventuellement plusieurs, sans contrainte de contiguïté.

La sélection multiple dans un programme Java se fait de façon classique en utilisant les touches « Shift » (Majuscule) et « Ctrl ». Pour sélectionner un intervalle, il faut cliquer sur un élément qui borde l'intervalle, enfoncer la touche Shift, puis cliquer sur l'autre borne. Pour sélectionner des items de façon discontinue, il faut maintenir la touche « Ctrl ».

Pour paramétrer ce comportement, il suffit d'utiliser la méthode `setSelectionMode`. Cette méthode prend en paramètre une constante parmi les trois suivantes :

- `SINGLE_SELECTION` : un seul item peut être sélectionné ;
- `SINGLE_INTERVAL_SELECTION` : un intervalle contigu d'items peut être sélectionné ;
- `MULTIPLE_INTERVAL_SELECTION` : n'importe quelle combinaison d'items peut être sélectionnée. Ce mode est le mode par défaut.

Comment utilise-t-on ces constantes ? Elles sont définies dans l'interface `ListSelectionModel`. Ce sont des constantes de classes. L'instruction ci-dessous permet donc d'indiquer que la liste des destinations, représentée par la variable `listeDestin`, est une liste à sélection unique :

```
listeDestin = new JList(v);
listeDestin.setSelectionMode(
➥ListSelectionModel.SINGLE_SELECTION);
```

Dans ces conditions, la combinaison des touches « Ctrl » et « Shift » avec la sélection n'est d'aucun effet. La sélection d'un item annule la précédente.

Quelle est cette interface dont nous venons de parler qui s'appelle `List SelectionModel` ? Comme son nom l'indique, il s'agit d'un modèle pour la sélection. Chaque composant `JList` utilise un objet qui encapsule les informations concernant la sélection : la politique choisie, les items sélectionnés à l'instant *t*…

Le modèle de sélection peut être obtenu depuis le composant graphique à l'aide de la méthode `getSelectionModel`.

Par ailleurs, le choix du mode de sélection peut se faire directement sur le modèle de la façon suivante :

```
listeDestin.getSelectionModel().setSelectionMode(
     ListSelectionModel.SINGLE_SELECTION);
```

Cette instruction est parfaitement équivalente à celle vue précédemment. En fait, lorsqu'on modifie le mode de sélection sur l'objet graphique, c'est le modèle de sélection sous-jacent qui est réellement modifié.

4.2.2 L'orientation de la sélection

Pour une liste ou un arbre, il n'y a bien sûr qu'une seule orientation possible. Pour un tableau en revanche, on peut souhaiter la sélection par lignes, ou par colonnes.

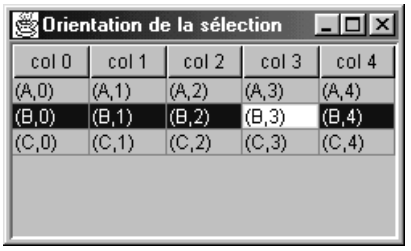


Figure 4.3 — Sélection des lignes.

Le comportement standard d'un tableau est de permettre la sélection par lignes. Pour obtenir l'interface de la figure 4.3, nous n'avons donc eu recours à aucune instruction particulière. Nous avons affiché le tableau en gris clair, afin de bien distinguer la cellule qui a le focus, qui elle, apparaît en blanc bordé de jaune avec le *look and feel* Windows.

```
tableau = new JTable(donnees, nomColonnes);
tableau.setBackground(Color.lightGray);
```

Pour obtenir une sélection colonne par colonne, il est nécessaire d'interdire la sélection des lignes, et d'autoriser celles des colonnes :

```
tableau.setColumnSelectionAllowed(true);
tableau.setRowSelectionAllowed(false);
```

Ainsi, nous obtenons le résultat de la figure 4.4.

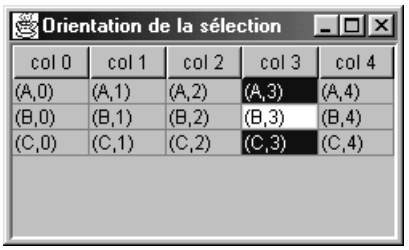


Figure 4.4 — Sélection des colonnes.

Pour autoriser cette fois la sélection des cellules (figure 4.5), il suffit d'indiquer la ligne de code suivante :

```
tableau.setCellSelectionEnabled(true);
```

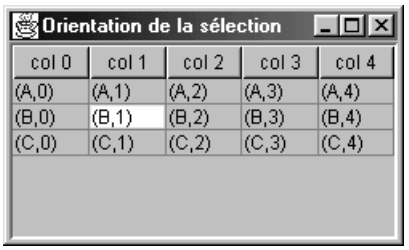


Figure 4.5 — Sélection des cellules.

4.2.3 Le contenu de la sélection

Maintenant que nous avons vu comment paramétrer la sélection, intéressons-nous à un problème plus crucial : comment savoir ce que l'utilisateur a sélectionné ?

Obtenir les lignes sélectionnées dans une liste

Reprenons notre exemple de la liste affichant les services disponibles au cours d'un voyage.

Pour obtenir les lignes sélectionnées, plusieurs solutions s'offrent à nous :

- la méthode `getSelectedIndices` renvoie un tableau d'entiers avec les index des lignes sélectionnées ;
- la méthode `getSelectedValues` renvoie directement un tableau d'objets. Ici, on va donc pouvoir récupérer les chaînes de caractères présentes dans notre vecteur de données.

Pour afficher les services qui ont été sélectionnés par l'utilisateur, il suffit donc de faire :

```
Object[] selection = listeServices.getSelectedValues();
for (int i=0; i<selection.length; i++) {
   System.out.println(selection[i]);
}
```

D'autres méthodes existent lorsqu'on est en mode sélection unique. La méthode `getSelectedIndex` (index est ici au singulier) renvoie l'index de la première ligne sélectionnée.

Le rôle du modèle de sélection

Nous avons parlé plus haut du modèle de sélection. Un des rôles du modèle est d'encapsuler la sélection courante.

Lorsqu'on utilise des méthodes sur des composants graphiques comme celles que nous venons de voir, ce sont en fait les méthodes du modèle qui sont appelées (figure 4.6).

Il existe des méthodes sur l'interface `ListSelectionModel` pour connaître les items sélectionnés. Pour savoir quels sont les premier et dernier items sélectionnés, les méthodes `getMinSelectionIndex` et `getMaxSelectionIndex` sont disponibles. Grâce à ces deux méthodes, on peut parcourir tous les index compris entre le minimum et le maximum et tester leur état à l'aide de la méthode `isSelectedIndex`.

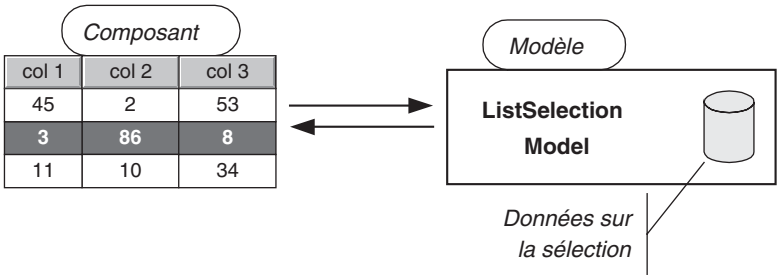


Figure 4.6 — Relation entre le composant et le modèle de sélection.

On peut aussi connaître le dernier intervalle qui a été ajouté à la sélection, grâce aux méthodes `getAnchorSelectionIndex` (première borne du dernier intervalle ajouté) et `getLeadSelectionIndex` (deuxième borne du dernier intervalle ajouté). Bien sûr, dans le cas d'une sélection discontinue avec la touche « Ctrl », le dernier item ajouté constitue un intervalle à lui tout seul (figure 4.7).

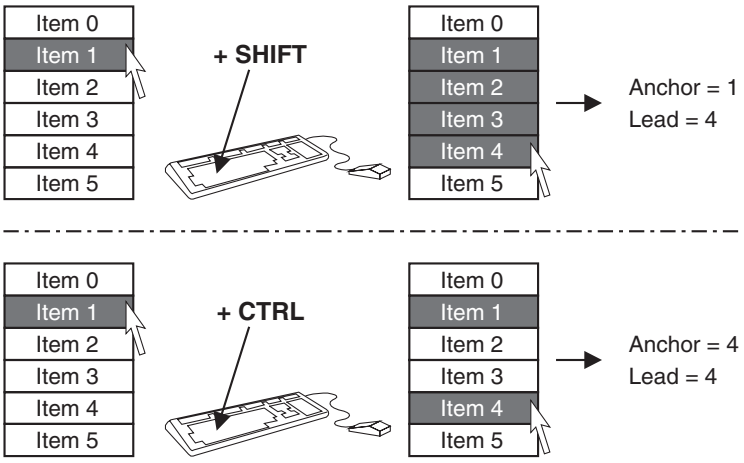


Figure 4.7 — Explications des index `anchor` et `lead`.

Si on élargit notre réflexion à d'autres composants que `JList`, on s'aperçoit que cette architecture présente quelques variantes suivant les composants :

- pour un `JTable` : si la sélection de lignes est autorisée, cela fonctionne exactement pareil, avec un `ListSelectionModel` associé au `JTable`. Si la sélection des colonnes est autorisée, on a encore un `ListSelection Model`, mais au lieu d'être associé au composant graphique, celui-ci est associé au modèle de colonnes. Nous détaillerons plus loin ce qu'est le modèle de colonnes (`ColumnModel`) ;

- pour un `JTree`, c'est une classe implémentant `TreeSelectionModel` qui est utilisée. Les méthodes sont légèrement différentes : au lieu d'obtenir la liste des objets sélectionnés, on obtient une liste de `TreePath` (chemin d'un nœud). Le fonctionnement global reste cependant le même.

On peut également forcer la sélection, par exemple pour indiquer une sélection initiale par défaut.

Grâce à l'instruction suivante, on force la sélection des items 1 à 3.

```
jList1.setSelectionInterval(1,3);
```

Détecter les changements de sélection

Dans des applications un peu dynamiques, le changement d'une sélection provoque parfois l'adaptation d'une partie de l'IHM. Par exemple, on peut avoir une fenêtre permettant la sélection d'une couleur pour l'affichage d'un texte parmi une liste de couleurs prédéfinies. Lorsque l'utilisateur sélectionne un item de la liste, le texte d'exemple est modifié pour s'afficher dans la couleur souhaitée (figure 4.8).

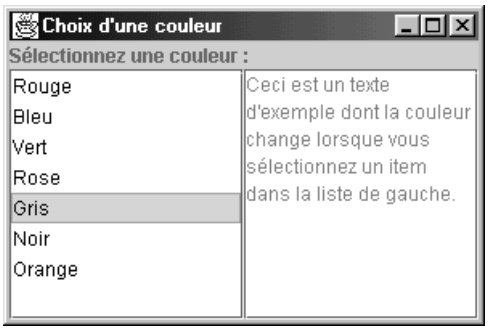


Figure 4.8 — Choix d'une couleur pour une police.

Pour que la modification du texte se fasse, il faut traiter l'événement : changement de sélection. Le composant source de l'événement est le composant `JList`. Recherchons une méthode d'abonnement sur ce composant. Nous trouvons : `addListSelectionListener`.

L'interface à implémenter est donc `ListSelectionListener`. Cette interface ne spécifie qu'une seule méthode :

```
public void valueChanged(ListSelectionEvent e)
```

Le code nécessaire à l'exemple est donc le suivant :

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

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

public class MaFenetre extends JFrame {

   public static final String TITRE_FEN =
   ➥"Choix d'une couleur";
   public static final String TEXTE_LB = "
   ➥Sélectionnez une couleur : ";
   public static final String TEXTE_EXPLE = "Ceci est un"
      + "texte d'exemple "
      + "dont la couleur change lorsque vous sélectionnez un "
      + "item dans la liste de gauche. ";

   JPanel panGeneral = new JPanel();
   JScrollPane jScrollPane1 ;
   JList listeCouleurs ;
   JLabel lbInstruction = new JLabel();
   Vector v = new Vector();
   JTextArea zoneExemple = new JTextArea();
   JScrollPane panneauTexte = new JScrollPane(zoneExemple);

   public MaFenetre() {
      initialisationVecteurDonnees();

      // Initialisation variables d'instance
      listeCouleurs = new JList(v);
      jScrollPane1 = new JScrollPane(listeCouleurs);

      // Construction interface graphique
      this.getContentPane().add(lbInstruction,
      ➥BorderLayout.NORTH);
      this.getContentPane().add(panGeneral,
      ➥BorderLayout.CENTER);
      panGeneral.setLayout(new GridLayout(1,2));
      panGeneral.add(jScrollPane1);
      panGeneral.add(panneauTexte);

      this.setTitle(TITRE_FEN);
      lbInstruction.setText(TEXTE_LB);
      zoneExemple.setText(TEXTE_EXPLE);
      zoneExemple.setLineWrap(true);
      zoneExemple.setWrapStyleWord(true);
      listeCouleurs.getSelectionModel().setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION);

      // Création et enregistrement du listener
      listeCouleurs.addListSelectionListener(new
         ListSelectionListener() {
         public void valueChanged(ListSelectionEvent e){
            int i = listeCouleurs.getSelectedIndex();
```

```
                Color c;
                switch (i) {
                   case 0 : {
                      c = Color.red;
                      break;
                   }
                   case 1 : {
                      c = Color.blue;
                      break;
                   }
                   [... et ainsi de suite ...]
                   default : c = Color.black;
                }
                zoneExemple.setForeground(c);
             }
          });
       }

       private void initialisationVecteurDonnees() {
          v.add("Rouge");
          v.add("Bleu");
          v.add("Vert");
          v.add("Rose");
          v.add("Gris");
          v.add("Noir");
          v.add("Orange");
       }
    }// fin class
```

Les événements `ListSelectionEvent` ne contiennent aucune information sur l'état de la sélection. Pour connaître le ou les items sélectionnés, il faut s'adresser au composant source ou à son modèle de sélection.

Le composant source peut être obtenu à l'aide de la méthode `getSource`.

Attention ! Il est aussi possible d'abonner le listener directement au modèle de sélection, et non au composant source.

Il existe une méthode utile sur la classe `ListSelectionEvent`, qui permet de distinguer les sélections « finalisées » : on ignore les événements successifs qui sont déclenchés pendant que l'utilisateur effectue sa sélection.

```
if ( ! e.getValueIsAdjusting() ) {
   [... actions ...]
}
```

La méthode `getValueIsAdjusting` renvoie vrai lorsqu'il s'agit d'un événement faisant partie d'une série d'événements rapides qui ont lieu lorsque l'utilisateur fait une sélection multiple et que l'on peut ignorer.

4.2.4 Créer une nouvelle politique de sélection

Il est possible de définir notre propre politique de sélection.

Par exemple, nous souhaitons proposer un modèle de sélection pour des utilisateurs non habitués à l'outil informatique. Nous craignons que la sélection multiple à l'aide de la touche « Ctrl » ne leur pose problème.

Nous allons implémenter un comportement où la sélection, même multiple, se fait en cliquant sur les items : une sélection additive (figure 4.9). Si on clique sur un item déjà sélectionné, il se désélectionne (figure 4.10).

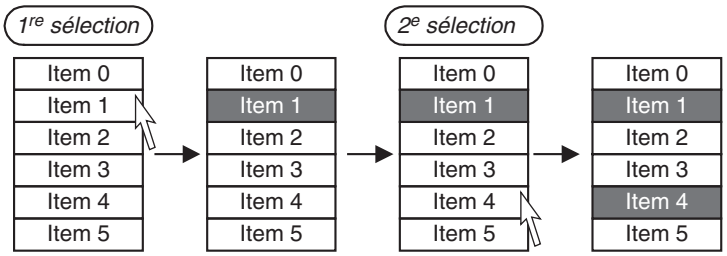


Figure 4.9 — Sélection de plusieurs items.

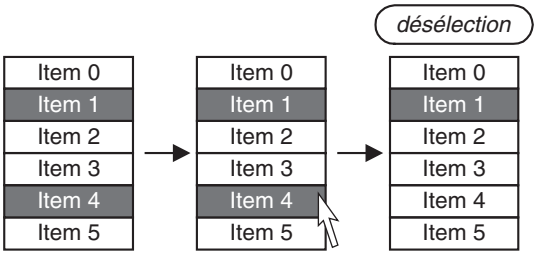


Figure 4.10 — Désélection d'un item.

Par ailleurs, si l'utilisateur sait utiliser la touche « Shift » pour la sélection d'un intervalle entier, cela devra fonctionner également.

Pour cela, il faut écrire notre propre modèle de sélection. Nous avons le choix entre implémenter directement l'interface `ListSelectionModel`, ou bien dériver d'une sous-classe concrète. La sous-classe concrète utilisée par les composants standard est la classe `DefaultListSelectionModel`.

La méthode qui gère l'ajout d'une sélection est la méthode `setSelection Interval`. Cette méthode fixe les index *anchor* et *lead* et rafraîchit la sélection en fonction. Nous n'avons donc qu'à redéfinir cette méthode uniquement. On choisit d'hériter de la classe concrète (figure 4.11).

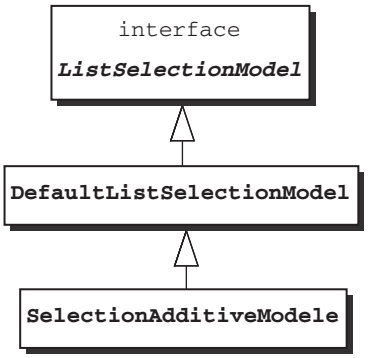


Figure 4.11 — Création d'un modèle de sélection personnalisé.

La méthode `setSelectionInterval` doit effectuer un traitement particulier lorsqu'un seul item est en train d'être ajouté. Les deux paramètres correspondent justement aux index *anchor* et *lead* d'une sélection multiple. S'ils sont égaux, cela signifie que l'utilisateur a cliqué sur un item sans utiliser la touche « Shift ».

Dans ce cas, il faut ajouter cet index à la sélection s'il n'en faisait pas déjà partie, sinon le retirer.

Si l'utilisateur a fait la sélection d'un intervalle à l'aide de la touche « Shift », le comportement est le même que celui du `DefaultListSelectionModel`.

Voici donc l'implémentation de notre classe :

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

/**
 * Modèle de sélection dans une liste : sélection additive,
 * sans la touche Ctrl
 */
public class SelectionAdditiveModele
➥extends DefaultListSelectionModel {

   public SelectionAdditiveModele() {
      setSelectionMode(ListSelectionModel.
      ➥MULTIPLE_INTERVAL_SELECTION);
}
public void setSelectionInterval(int index0, int index1) {
   if (index0 == index1) {
      // Sélection unique
      if (isSelectedIndex(index0)) {
         removeSelectionInterval(index0, index0);
      } else {
```

```
            addSelectionInterval(index0, index0);
          }
        } else {
          // Sélection multiple
          super.setSelectionInterval(index0, index1);
        }
      }
    }
```

4.3 PERSONNALISER L'APPARENCE AVEC LES RENDERERS

Imaginons le logiciel suivant : dans un but de gestion de stock, un grand magasin a mis en place une application où des employés évaluent la quantité restante de certains produits, saisissent cette quantité à un instant *t*. Ils peuvent aussi indiquer s'ils estiment qu'une commande est nécessaire.

Pour cela, nous leur fournissons un tableau représentant les saisies de tous les employés. Dans un premier temps, nous allons créer un tableau avec nos connaissances actuelles. L'aspect du tableau est celui présenté figure 4.12.

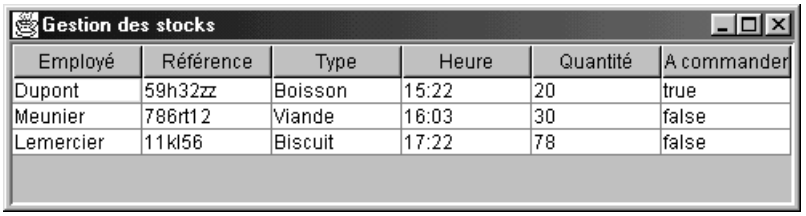


Figure 4.12 — Aspect graphique non travaillé.

Nous constatons que l'apparence du tableau n'est pas idéale. En effet, toutes les cellules ont la même apparence : celle d'un champ contenant du texte aligné à gauche. Or, pour certaines données, cela n'est pas adapté. Ainsi, la colonne « A commander » pourrait être représentée par une case à cocher, les colonnes affichant des valeurs numériques devraient plutôt présenter un texte aligné à droite.

Pour remédier à cela, nous allons nous pencher sur le mécanisme des *renderers*.

4.3.1 Qu'est-ce qu'un renderer ?

Comment le tableau dessine-t-il ses cellules ? Vous avez sans doute remarqué que les cellules ressemblent fortement à des composants simples précédemment étudiés. En effet, chaque cellule peut être considérée comme un composant graphique à part entière : `JLabel`, `JCheckBox`, etc.

On pourrait penser que chaque cellule est un composant et que le tableau n'est finalement qu'un container un peu particulier. Cette vision simpliste ne correspond heureusement pas à la réalité. Si c'était le cas, cela signifierait que lorsqu'on affiche un tableau à *n* cellules, on instancie un objet tableau et *n* composants graphiques liés à ce tableau, et que ces *n* + 1 objets sont conservés en mémoire tant qu'on dispose d'une référence sur le tableau. Cette solution serait très peu performante.

Le mécanisme des *renderers* permet de traiter chaque cellule à la façon d'un composant graphique classique, tout en empêchant l'instanciation d'une multitude de composants (figure 4.13).

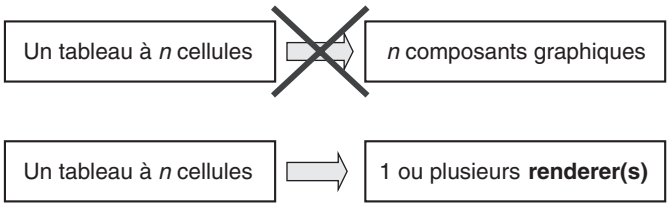


Figure 4.13 — L'intérêt des *renderers*.

On peut se représenter un *renderer* comme une sorte de « tampon » configurable pouvant être utilisé pour toutes les cellules ayant le même aspect.

Le *renderer* permet l'instanciation des cellules en tant que composants graphiques habituels, mais il permet également de libérer ces instances de la mémoire dès que les cellules ont été dessinées.

Un unique *renderer* est généralement utilisé pour dessiner toutes les cellules d'une même colonne. Souvent ce *renderer* est partagé entre toutes les colonnes contenant le même type de données. Si le tableau n'affiche que des cases à cocher par exemple, on a un seul *renderer* pour tout le tableau.

Il est important d'être conscient de ce mécanisme. Même si vous ne précisez aucune information comme on l'a fait dans la première utilisation d'un `JTable`, ce fonctionnement d'affichage à l'aide d'un *renderer* est toujours utilisé. Il existe un *renderer* par défaut qui dessine chaque cellule. Un raisonnement consistant à penser « je vais me contenter d'un affichage rustique pour ne pas ralentir mon application » n'est pas valide. Même si vous n'indiquez aucune instruction d'affichage particulière, le processus se déroule de la même façon. Alors, autant soigner l'apparence !

Le mécanisme de fonctionnement du *renderer* est très intuitif : schématiquement, on peut considérer que trois étapes se répètent autant de fois qu'il y a de cellules dans le tableau (figure 4.14).

- étape 1 : le tableau demande au *renderer* comment dessiner la cellule (i, j) ;
- étape 2 : grâce aux informations fournies par le tableau, le *renderer* crée un composant graphique adéquat pour représenter la cellule (i, j) ;
- étape 3 : il renvoie ce composant qui va être dessiné à l'écran. L'optimisation réside dans le fait que la même instance de composant est réutilisée par le *renderer*.

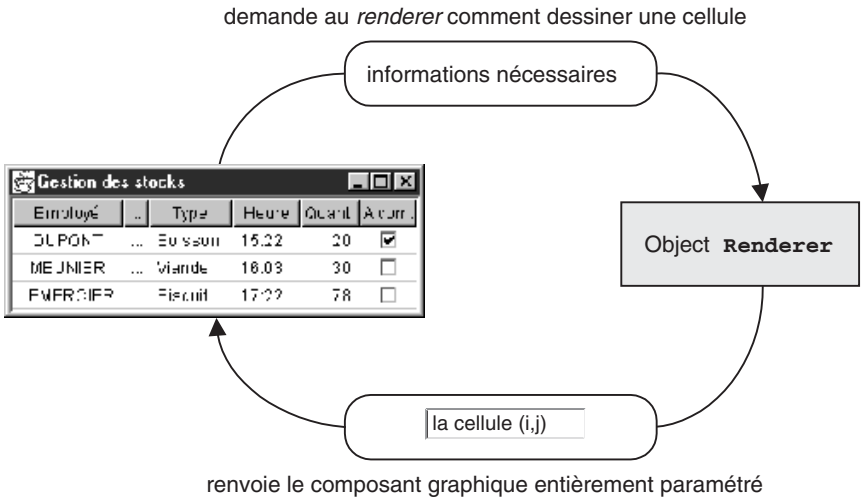


Figure 4.14 — Principe de fonctionnement du *renderer*.

Le composant renvoyé par le *renderer* est ensuite peint à l'écran, puis l'instance est « garbage collectée ». On ne peut pas manipuler après coup cet objet, il n'est pas accessible depuis le tableau. Cette constatation illustre bien la différence entre ce principe et celui des containers et composants.

4.3.2 Créer son propre renderer

Nous allons essayer d'ajouter un *renderer* pour la première colonne pour que le texte apparaisse centré et en majuscules. Par ailleurs, nous allons en profiter pour faire quelques modifications sur l'apparence générale du tableau afin qu'il soit plus agréable à lire.

Voilà figure 4.15 le résultat auquel nous souhaitons arriver.

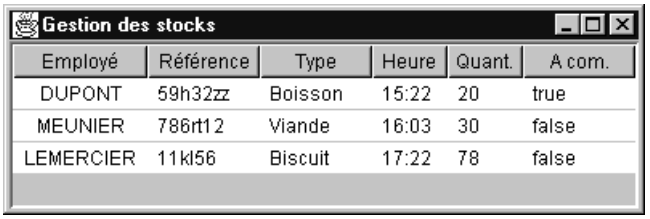


Figure 4.15 — Un *renderer* sur la première colonne.

La caractéristique principale d'un *renderer* est d'hériter d'une interface particulière. Étudions cela tout de suite.

L'interface à implémenter

Il doit implémenter une interface de type « *Renderer* ». Examinons les classes et interfaces concernées qui sont fournies par Swing (figure 4.16).

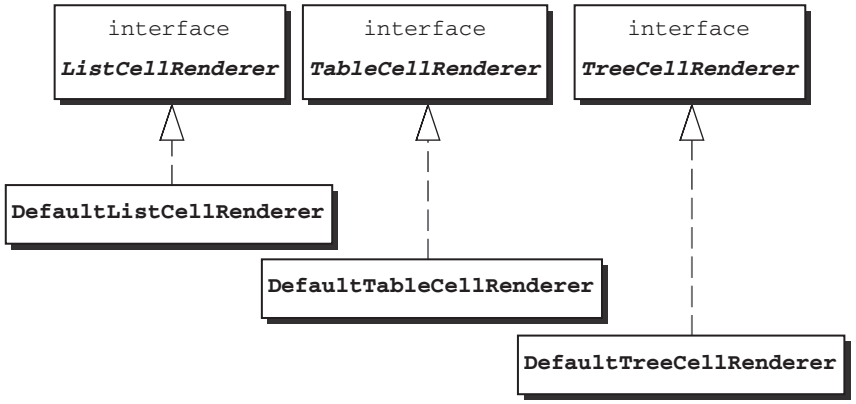


Figure 4.16 — Hiérarchie des classes « *Renderer* ».

Le package `javax.swing` contient aussi une interface qui s'appelle simplement `Renderer`, et qui n'est utilisée par aucune classe de Swing. Elle servira à un utilisateur qui souhaite implémenter son propre *renderer*, pour un composant complexe autre que `JList`, `JTable` et `JTree`.

Puisque notre *renderer* est prévu pour s'appliquer à un tableau, il va implémenter l'interface `TableCellRenderer`.

La seule méthode à implémenter est la méthode `getTableCellRenderer Component`. Cette méthode prend en paramètres toutes les informations nécessaires pour renvoyer le composant exact à afficher.

Dans le cas du tableau, la signature de cette méthode est la suivante :

```
public Component getTableCellRendererComponent
  (JTable table, Object value,boolean isSelected,
  boolean hasFocus, int row, int column)
```

Les différents paramètres correspondent aux informations suivantes :

- le composant utilisateur du *renderer* (tableau, liste…) ;
- la donnée à afficher (de type `Object`) ;
- l'état du composant à afficher (sélectionné, a le focus…) ;
- la position de la cellule dans le tableau.

Pour les autres interfaces, `ListCellRenderer` et `TreeCellRenderer`, le nom de cette méthode varie légèrement et surtout les paramètres nécessaires ne sont pas exactement les mêmes. Les informations « ligne » et « colonne » sont remplacées par un index dans le cas de la liste. Dans le cas d'un arbre, on trouve également des informations sur l'état du nœud : est-il « fermé » ou bien « étendu » ? Est-ce une feuille ?

Implémentation de la classe `RendererEmploye`

Un *renderer* doit donc créer un composant graphique adapté. Plusieurs politiques sont alors possibles :

- un *renderer* peut être lui-même un composant, c'est-à-dire hériter d'une classe « Composant », comme par exemple `Jlabel` ;
- il peut créer un nouveau composant graphique à chaque demande du tableau. Cette solution est bien sûr à bannir car elle nécessite une instanciation à chaque appel au *renderer*, or cette opération peut être coûteuse. Cette politique est pourtant parfois rencontrée ;
- il peut disposer d'une instance de composant graphique et paramétrer cette instance différemment pour chaque cellule.

On est tout à fait libre dans l'implémentation. Le seul impératif est la méthode `getTableRendererComponent`, qui doit renvoyer le composant paramétré. Si on regarde les classes de *renderers* par défaut présentées dans le diagramme UML précédent afin de s'inspirer de la façon de faire de Sun, on voit qu'ils ont choisi d'hériter d'une classe graphique. Dans ce cas, la méthode `getTableRenderer Component` renvoie la référence `this` après l'avoir paramétrée.

Notre classe va donc hériter d'un composant graphique. Étant donné que nous souhaitons représenter du texte simple, nous pouvons utiliser tout simplement la classe `JLabel`.

Notre classe va donc respecter le schéma de la figure 4.17.

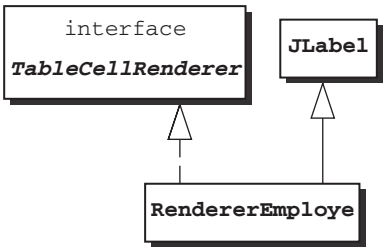


Figure 4.17 — La classe `RendererEmploye`.

Le code est le suivant :

```
public class RendererEmploye extends JLabel implements
➥ TableCellRenderer {

   public RendererEmploye() {
      super();
      // pour pouvoir afficher la couleur de fond
      setOpaque(true);
   }

   public Component getTableCellRendererComponent
   ➥(JTable table,O bject obj, boolean isSelected,
   ➥boolean hasFocus, int row, int column) {
      if (isSelected) {
         setBackground(SystemColor.textHighlight);
         setForeground(Color.white);
      }
      else {
         setBackground(Color.white);
         setForeground(Color.black);
      }
      setHorizontalAlignment(SwingConstants.CENTER);
      setText(((String)obj).toUpperCase());
      return this;
  }
}
```

Utiliser notre renderer

Maintenant que nous avons créé cette classe *Renderer*, il faut créer une instance et indiquer au tableau où il doit l'utiliser.

Dans notre exemple, nous souhaitons associer ce *renderer* uniquement à la première colonne. Pour ce faire, il est nécessaire d'accéder à l'objet « colonne » et de lui appliquer ensuite la méthode `setCellRenderer`.

```
TableColumn col0 = tableau.getColumnModel().getColumn(0);
// Il n'y a donc bien qu'une seule instance du renderer
rendererCol0 = new RendererEmploye();
```

```
col0.setCellRenderer(rendererCol0);
```

Pour obtenir l'objet `TableColumn`, on peut aussi s'adresser directement au tableau au lieu de passer par la méthode `getColumnModel`, mais dans ce cas, la colonne s'identifie par la chaîne de caractères affichée dans l'en-tête, ce qui est beaucoup moins performant.

Il est aussi possible d'associer le *renderer* à un type d'objets particulier. Ainsi, nous pourrions décider que toutes les données de type `String` seront affichées en majuscules et avec un alignement centré. Il suffit alors d'utiliser l'instruction suivante :

```
tableau.setDefaultRenderer(String.class, rendererCol0) ;
```

Attention ! Ceci ne marche que si le tableau est capable de savoir le type exact des objets qu'il affiche… On verra plus loin comment s'assurer de cela.

On peut aussi associer le *renderer* à une cellule particulière. Par exemple, pour n'appliquer ce *renderer* que sur la première cellule de la colonne « Employé », nous pouvons procéder de deux manières :

- 1re solution : dans l'implémentation du renderer, on modifie la méthode `getTableCellRendererComponent` pour renvoyer des composants différents suivant les coordonnées de la cellule ;
- 2e solution : créer une classe qui hérite de `JTable`, puis redéfinir la méthode `getCellRenderer` afin d'utiliser le *renderer* seulement pour certaines cellules :

```
public TableCellRenderer getCellRenderer (int row,
 int column) {
  if ((row==0) && (column==1))
    return new RendererEmploye() ;
  else return super.getCellRenderer(row, column) ;
}
```

Attention aux performances.
Un *renderer* peut avoir un impact profond sur les performances d'une application. Lorsque nous avons créé notre propre renderer, nous avons fait hériter notre classe `RendererEmploye` de `JLabel`. De la sorte, la méthode `getTableCellRendererComponent(…)` n'instancie aucun composant. Ce point est critique pour les performances car cette méthode sera appelée pour chaque cellule visible du tableau, aussi souvent que le gestionnaire de rafraîchissement Swing le jugera nécessaire. Il est parfois incontournable d'utiliser une ou plusieurs instances de composant dans un *renderer*. Dans ce cas, pensez à utiliser et conserver ces instances de composants dans des attributs du *renderer*. En conclusion, résistez à la tentation d'instancier des composants Swing dans des méthodes telles que `getTableCellRendererComponent(…)`.

4.3.3 Utiliser les renderers par défaut

Si l'utilisateur ne souhaite pas créer de *renderer* personnalisé, il peut s'appuyer sur les *renderers* fournis par défaut. Comment cela se passe-t-il ?

Le tableau doit connaître le type de la donnée à afficher. Or, le `JTable` ne dispose d'aucune information sur ses données, il doit donc s'adresser à l'objet qui encapsule les données : le `TableModel`.

Nous reviendrons plus longuement sur cette classe dans la suite du chapitre.

Deux cas sont possibles :

- cas 1 : il n'existe pas de `TableModel` créé par l'utilisateur. Le composant `JTable` utilise alors un `TableModel` créé par défaut ;
- cas 2 : un `TableModel` personnalisé a été implémenté.

L'utilisation des *renderers* par défaut est très différente suivant qu'on est dans un cas ou dans l'autre. Comparez ces deux cas présentés respectivement dans les figures 4.18 et 4.19.

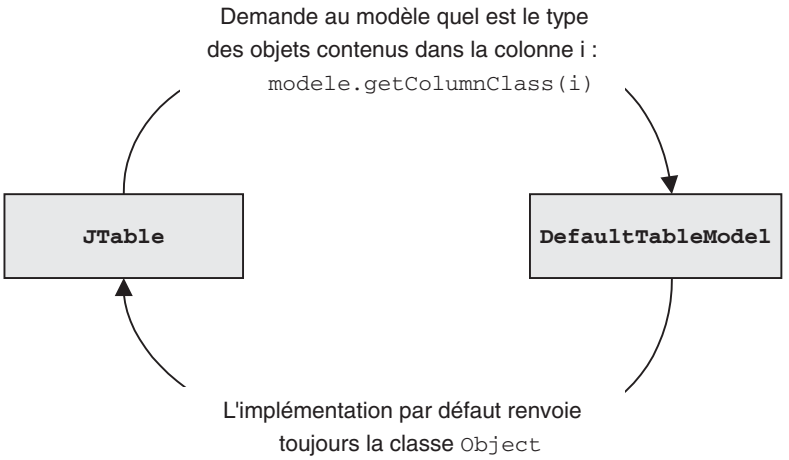


Figure 4.18 — Cas où aucun `TableModel` n'a été redéfini.

Nous venons de voir une des raisons pour lesquelles il est utile d'implémenter son propre `TableModel`. Nous verrons plus loin plus de détails sur cette implémentation.

Maintenant que le `JTable` connaît le type de la donnée à afficher, comment procède-t-il pour savoir quelle apparence lui donner ?

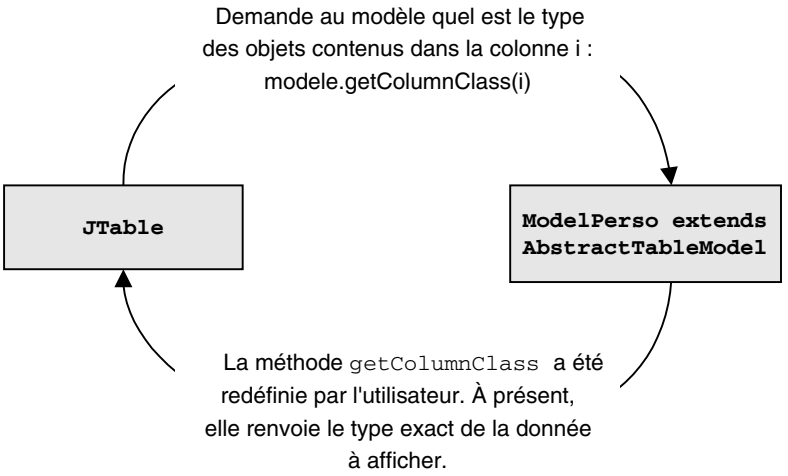


Figure 4.19 — Cas où un `TableModel` personnalisé existe.

Il recherche s'il existe un *renderer* par défaut pour ce type. Il compare la classe de l'objet à afficher avec les classes pour lesquelles un *renderer* a été enregistré. Il dispose d'un dictionnaire liant les types et leurs *renderers* associés :

```
protected transient Hashtable defaultRenderersByColumnClass;
```

Cette liste contient par défaut :

- `Boolean` (représenté par une `JCheckBox`) ;
- `Number` (représenté par un `JLabel` avec alignement à droite) ;
- `Date` (représenté par un `Jlabel` avec alignement à droite et formaté avec `DateFormat`) ;
- `ImageIcon` (représenté par un `JLabel` avec alignement centré) ;
- `Object` (représenté par un `JLabel` avec alignement à gauche qui affiche l'objet sous forme de chaîne de caractères).

Vous observez que les objets de type `Number` sont censés être alignés à droite et que les booléens sont représentés par des cases à cocher. Sur notre exemple, nous n'avons pas ce résultat car nous n'avons pas redéfini un `TableModel`.

Sans entrer dans le détail de l'implémentation qui sera clarifiée plus tard, nous pouvons d'ores et déjà affirmer la chose suivante : si on redéfinit un `TableModel` et qu'on écrit la méthode `getColumnClass` de la façon suivante :

```
// Renvoie le type de l'objet situé en colonne c, et sur
// la première ligne
public Class getColumnClass(int c) {
  return getValueAt(0, c).getClass();
}
```

Alors, sans ajouter de code concernant les *renderers*, nous obtiendrons le résultat suivant de la figure 4.20.

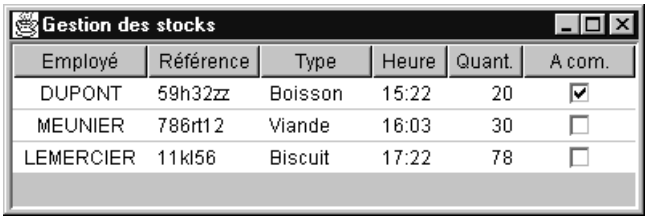


Figure 4.20 — Utilisation des *renderers* par défaut.

Les valeurs numériques (colonne « Quantité ») sont bien alignées à droite et les valeurs booléennes (colonne « À commander ») sont bien affichées à l'aide de cases à cocher.

Par ailleurs, une méthode permet de modifier le *renderer* associé avec l'un de ces cinq types de base. C'est ce dont nous avons parlé plus haut en présentant l'instruction :

```
tableau.setDefaultRenderer(String.class, rendererCol0) ;
```

Il faut bien garder à l'esprit que cela ne fonctionne que dans le cas d'un `TableModel` personnalisé avec une méthode `getColumnClass` correctement implémentée.

Cette méthode `setDefaultRenderer` permet également d'ajouter de nouvelles associations (type de donnée-*renderer*) à ce dictionnaire.

4.3.4 Utiliser les classes d'implémentation « `Default…CellRenderer` »

Après avoir vu comment créer notre propre *renderer*, comment se reposer sur les *renderers* par défaut, nous allons voir un troisième et dernier procédé d'utilisation des *renderers*. Nous allons tout simplement utiliser les classes concrètes offertes par Swing.

On souhaite ajouter le texte « Auteur de ce relevé » dans un tooltip, qui apparaît lorsqu'on positionne la souris sur l'en-tête de la colonne « Employé », afin de bien préciser aux utilisateurs quelle information on attend.

Comme on veut seulement ajouter un tooltip et qu'on ne veut pas modifier l'apparence par défaut, on peut utiliser la classe `DefaultTableCellRenderer` qui fournit un `JLabel` avec l'objet à afficher sous forme de chaîne de caractères.

Dans ces conditions, les étapes sont les suivantes :

- créer un objet de la classe `DefaultTableCellRenderer` :

```
DefaultTableCellRenderer renduEnTete =
 new DefaultTableCellRenderer();
```

- personnaliser cet objet. On dispose de toutes les méthodes de `JLabel` :

```
renduEnTete.setToolTipText("Auteur de ce relevé ");
```

- l'ajouter à l'en-tête de la colonne « Employé ». Pour ce faire, on utilise la méthode `getHeaderRenderer` de la classe `TableColumn` :

```
TableColumn col0 = tabTrains.getColumnModel().getColumn(0);
col0.setHeaderRenderer(renduEnTete);
```

Un autre cas d'utilisation de la classe `DefaultTableCellRenderer` est le suivant : lorsque nous avons créé tout à l'heure notre propre classe `Renderer Employe`, nous avons choisi d'implémenter directement l'interface `TableCell Renderer`. Nous n'avions pas hérité de `DefaultTableCellRenderer` car les paramètres graphiques ne nous convenaient pas : alignement du texte, couleurs, bordures, etc.

Imaginons un autre cas de figure : notre seul souhait est que le nom des employés apparaisse en majuscules. En dehors de cela, l'affichage proposé par défaut nous satisfait. Sommes-nous obligés de redéfinir la méthode `getTable CellRendererComponent` ? Nous pouvons l'éviter car une méthode spécifique est appelée uniquement pour personnaliser la manière dont la valeur de l'objet est affichée. Il s'agit de la méthode `setValue` qui est appelée depuis `getTable CellRendererComponent`.

Dans l'implémentation par défaut, `setValue` se contente d'afficher dans le composant `JLabel` l'objet converti en chaîne de caractères. Dans les sources du JDK, on trouve :

```
protected void setValue(Object value) {
   setText((value == null) ? "" : value.toString());
}
```

Nous pouvons très bien redéfinir cette méthode comme ceci :

```
protected void setValue(Object value) {
  setText((value == null) ? "" : value.toString()
  .toUpperCase());
}
```

4.3.5 Personnalisations supplémentaires de l'apparence

Présentons quelques suppléments afin de montrer comment obtenir exactement le même rendu que sur l'interface exposée plus haut (figure 4.20).

Tout d'abord, nous avons indiqué une taille adéquate pour le tableau. Cela se fait de la façon suivante, où `TAB_DIMENSION` est une variable statique de la classe `Dimension` définie au début de notre classe :

```
tableau.setPreferredScrollableViewportSize(TAB_DIMENSION);
```

On peut également indiquer une taille optimale pour chaque colonne :

```
// Modifier la taille souhaitée des colonnes
TableColumn col0 = tableau.getColumnModel().getColumn(0);
TableColumn col3 = tableau.getColumnModel().getColumn(3);
TableColumn col4 = tableau.getColumnModel().getColumn(4);
col0.setPreferredWidth(90);
col3.setPreferredWidth(50);
col4.setPreferredWidth(50);
// Paramétrer la hauteur des lignes
tableau.setRowHeight(20);
```

On peut choisir d'afficher ou non tout le quadrillage. Ici, nous avons opté pour l'affichage des séparations de lignes en gris clair, avec un interstice assez grand entre deux colonnes (10 pixels) :

```
// Paramétrer le quadrillage
tableau.setShowVerticalLines(false);
tableau.setGridColor(Color.lightGray);
tableau.setIntercellSpacing(new Dimension(10,5));
```

Nous pouvons aussi jouer sur la façon dont le tableau réagit lors du retaillage. Il existe cinq comportements de redimensionnement disponibles, respectivement représentés par les constantes suivantes :

- `AUTO_RESIZE_SUBSEQUENT_COLUMNS` : mode par défaut. Il indique que lorsqu'on modifie la taille d'une colonne, toutes les colonnes à sa droite sont affectées dans le but de maintenir la taille globale du tableau ;
- `AUTO_RESIZE_NEXT_COLUMN` : redimensionne seulement les colonnes situées immédiatement à droite et à gauche de la colonne manipulée ;
- `AUTO_RESIZE_OFF` : redimensionne la taille totale du tableau ;
- `AUTO_RESIZE_LAST_COLUMN` : redimensionne seulement la dernière colonne ;
- `AUTO_RESIZE_ALL_COLUMNS` : redimensionne toutes les colonnes proportionnellement pour garder la même taille totale.

Dans notre cas, nous avons choisi de retailler proportionnellement toutes les colonnes :

```
tableau.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
```

4.4 PERSONNALISER L'ÉDITION AVEC LES EDITORS

Il y a une question que nous ne nous sommes pas encore posée : l'utilisateur peut-il saisir des données dans notre tableau ? La réponse à cette question est multiple. En effet, avec la première version du tableau sans aucun *renderer*, toutes les cellules

étaient éditables. Autrement dit, il est important de retenir que par défaut, les tableaux sont éditables.

Cependant, dans la dernière version, lorsqu'on s'est efforcé d'utiliser les *renderers* par défaut, nous avions implémenté notre propre `TableModel`. Le code détaillé de cela n'a pas encore été présenté, mais il le sera dans la partie suivante. C'est uniquement grâce à cette personnalisation du modèle que nous pouvons choisir de rendre le tableau non éditable.

Maintenant que nous savons que les tableaux peuvent être éditables, l'idée qui vient naturellement à l'esprit est d'offrir une manière d'éditer chaque cellule la plus ergonomique possible.

C'est ce que nous allons voir ici avec le mécanisme des *editors*.

4.4.1 Qu'est-ce qu'un editor ?

De la même façon que le *renderer* gère l'apparence des cellules au repos, *l'editor* gère l'apparence et le comportement des cellules pendant l'édition. En fait, *l'editor* prend le contrôle sur la cellule dès que l'utilisateur commence à saisir des données (figure 4.21).

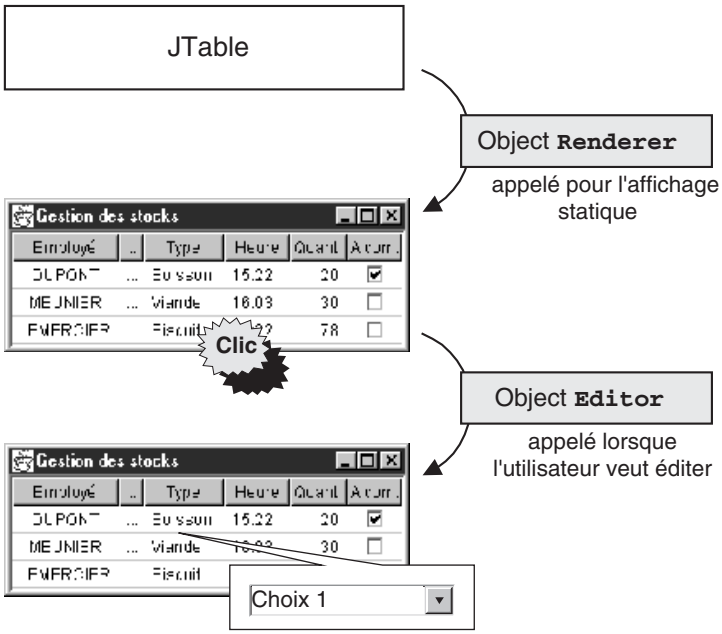


Figure 4.21 — Le relais *renderer/editor*.

Dès que l'utilisateur initie l'édition d'une cellule, l'*editor* est immédiatement sollicité. Le stimulus correspondant à l'édition d'une cellule est généralement le

double-clic sur cette cellule ou bien le fait de taper une touche au clavier (autre que les flèches ou la touche « Entrée »), lorsque la cellule a le focus.

Le fonctionnement des *editors* est similaire à celui des *renderers* (figure 4.22).

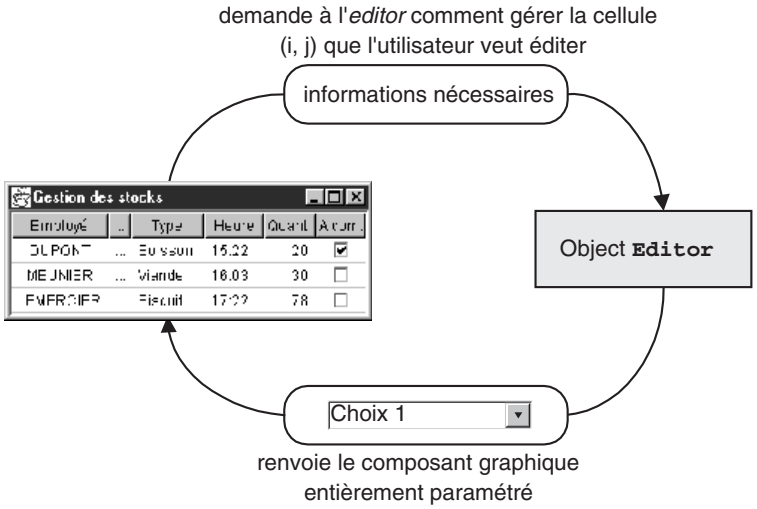


Figure 4.22 — Principe de fonctionnement d'un *editor*.

Il existe cependant une différence majeure entre un *renderer* et un *editor*, c'est que lorsque *l'editor est* appelé à un moment donné pour dessiner une cellule, une référence sur cette cellule particulière est conservée jusqu'à ce que l'édition soit terminée et que *l'editor* rende la main au *renderer*. Cela est bien logique puisqu'il faut être capable de prendre en compte la saisie de l'utilisateur.

4.4.2 Créer nos propres editors

Nous allons poursuivre notre exemple de l'application de gestion de stock et fournir des facilités d'édition : le choix d'un type de produits pourra se faire à l'aide d'une liste déroulante, la saisie de l'heure se fera avec le composant « cadran horaire » défini dans le chapitre sur les événements (figures 4.23 et 4.24).

Les classes et interfaces en présence

Comme pour les *renderers*, l'architecture proposée par Swing est très ouverte, dans le sens où on est assez libre dans l'implémentation de *l'editor*. En effet, là encore on peut choisir d'hériter d'un composant graphique, ou bien d'encapsuler ce composant dans une variable d'instance.

La seule contrainte est d'implémenter l'interface `CellEditor`. Regardons les classes proposées par Swing présentées figure 4.25.

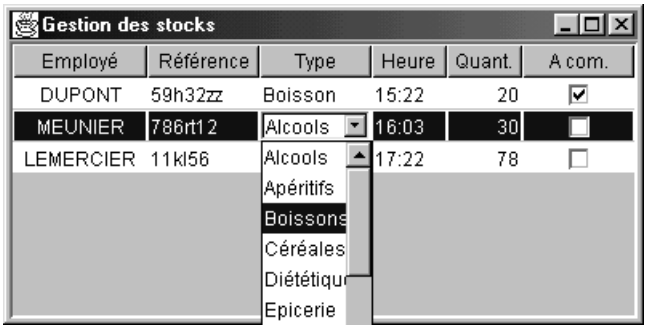


Figure 4.23 — Édition à l'aide d'une `JcomboBox`.

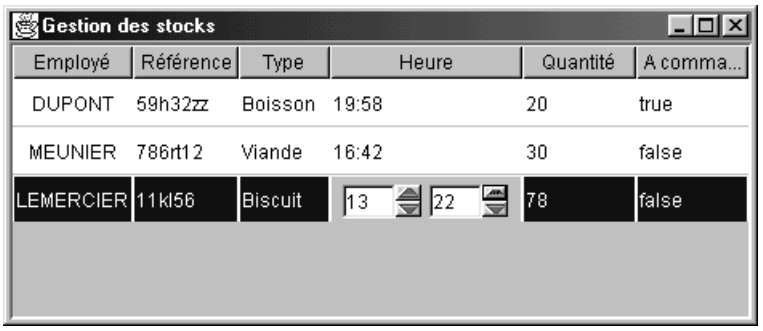


Figure 4.24 — Édition à l'aide d'un composant non standard.

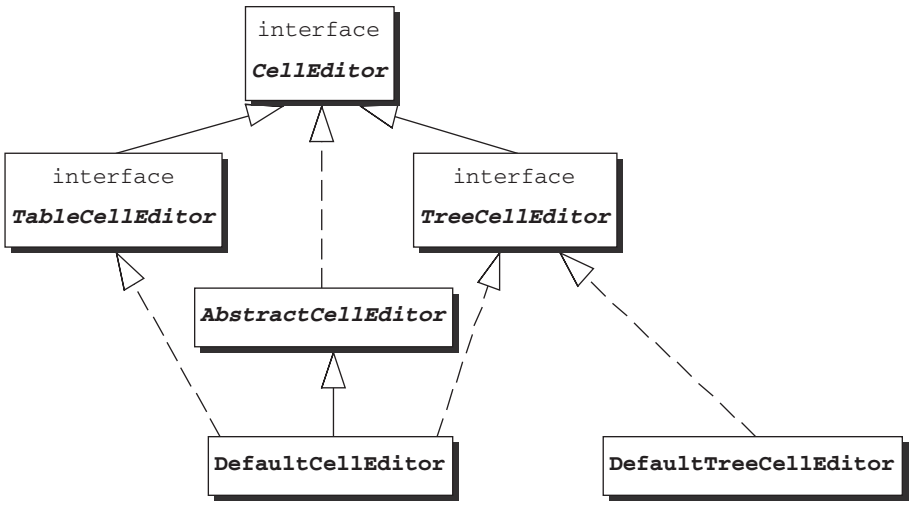


Figure 4.25 — Hiérarchie des classes *editor*.

Cette hiérarchie est un peu plus compliquée que celle des *renderers*. Une première remarque est qu'il n'y existe aucune classe du type « `ListCellEditor` ». Cela est normal car le composant `JList` ne peut jamais être éditable : son rôle est de permettre la sélection dans une liste.

Une deuxième différence est que la classe `DefaultCellEditor` semble pouvoir être utilisée aussi bien pour des tableaux que pour des arbres, car elle hérite des deux interfaces. C'est en effet le cas, même s'il existe une classe plus adaptée à l'édition d'un arbre : `DefaultTreeCellEditor`.

L'interface `CellEditor` spécifie des méthodes liées au comportement habituel d'un éditeur : gestion des abonnements et désabonnements des *listeners*, renvoi de la valeur saisie, etc.

Remarque : si vous utilisez le JDK 1.2, ne cherchez pas trop longtemps la classe abstraite `AbstractCellEditor` : il s'agit d'un ajout du JDK 1.3.

Les interfaces `TableCellEditor` et `TreeCellEditor` spécifient une méthode « `get[…]CellEditorComponent` » qui renvoie le composant graphique permettant l'édition. Cette méthode constitue le cœur de la classe.

Création de l'editor pour le type de produits

Nous allons tout de suite écrire notre premier *editor*. Une façon simple de créer un editor personnalisé est d'utiliser une instance de la classe `DefaultCellEditor`.

Elle encapsule le composant graphique servant à l'édition sous la forme d'une variable d'instance nommée `editorComponent`.

Elle fonctionne tout à fait différemment de ce que nous avons vu pour les *renderers*. Sa particularité est de n'être pas spécifique d'un composant graphique donné. On peut utiliser cette classe que l'on veuille afficher une `JComboBox`, un `JTextField` ou une `JCheckBox`. Pour cela, elle propose des constructeurs prenant en paramètres différents types de composants graphiques :

- `JTextField`
- `JCheckBox`
- `JComboBox`

La création d'une instance de `DefaultCellEditor` se fait donc de la façon suivante :

```
JComboBox listeTypes = new JComboBox(TAB_TYPES);
DefaultCellEditor editorTypes =
➥new DefaultCellEditor(listeTypes);
```

où TAB_TYPES est un tableau de chaînes de caractères défini comme variable de classe.

Comme pour les *renderers*, un *editor* peut être associé à une colonne particulière ou à un type de données particulier. Ici, on choisit d'ajouter cet *editor* à la colonne « Type » :

```
TableColumn col2 = tableau.getColumnModel().getColumn(2);
col2.setCellEditor(editorTypes);
```

Création de l'editor pour l'heure

Nous allons maintenant créer un *editor* plus complexe : il s'agit d'utiliser le composant qui permet de saisir des heures et des minutes que nous avons développé dans le chapitre 3 sur les événements, pour éditer de la quatrième colonne de notre tableau de gestion de stock.

Pour faire cela, nous n'allons pas pouvoir utiliser la classe DefaultCellEditor car elle ne dispose évidemment pas d'un constructeur prenant en paramètre un objet PanelHoraire. Créons notre propre classe *d'editor*. Plusieurs possibilités s'offrent à nous :

- implémenter directement l'interface CellEditor ;
- hériter de DefaultCellEditor ;
- hériter de AbstractCellEditor.

La première solution présente l'inconvénient d'être plus longue, car il faudra implémenter les comportements standard sur lesquels nous ne voulions pas faire de modifications.

La deuxième solution n'est pas satisfaisante d'un point de vue de la modélisation objet. On pourrait sous-classer DefaultCellEditor, redéfinir la méthode qui nous importe, mais comment écrire un constructeur sans paramètre ? Il est nécessaire d'appeler un constructeur de la classe mère, mais il n'existe que des constructeurs prenant en paramètre des composants graphiques.

La troisième solution est de loin la plus commode.

Puisque notre classe va hériter de AbstractCellEditor, nous ne pouvons pas hériter d'une autre classe. Le composant graphique *« PanelHoraire »* sera donc une variable d'instance.

La méthode getTableCellEditorComponent va renvoyer le composant graphique. Au commencement de l'édition, le panneau horaire doit prendre comme valeurs, les heures et les minutes qui étaient présentes dans la cellule. Pour cela, on utilise un objet StringTokenizer qui permet de lire une chaîne avec séparateurs pour interpréter le paramètre value de la méthode.

```
public class EditorHoraire extends AbstractCellEditor
  implements TableCellEditor {
   PanelHoraire panneau;
   public EditorHoraire() {
      panneau = new PanelHoraire();
   }

   public Component getTableCellEditorComponent
     (JTable table, Object value, boolean isSelected,
      int row, int column) {
      if (value != null) {
       StringTokenizer st = new StringTokenizer((String)
         value, ":");
        panneau.setHeure(Integer.parseInt(st.nextToken()));
        panneau.setMinute(Integer.parseInt(st.nextToken()));
      }
      return panneau;
   }

   public Object getCellEditorValue(){
      StringBuffer horaire = new StringBuffer();
      horaire.append(panneau.getHeure());
      horaire.append(":");
      horaire.append(panneau.getMinute());
      return horaire.toString();
   }
}
```

La méthode `getCellEditorValue` doit être redéfinie pour renvoyer la valeur modifiée.

4.4.3 Utiliser les editors par défaut

De la même façon que pour les *renderers*, le `JTable` dispose d'une liste *d'editors* par défaut pour les types les plus courants.

Lorsque le `JTable` doit afficher une cellule en cours d'édition, et qu'aucun *editor* n'a été défini pour cette colonne, le `JTable` s'adresse au `TableModel` pour connaître le type de la donnée à éditer. Il utilise ensuite *l'editor* par défaut associé à ce type.

Ces associations sont stockées dans le dictionnaire `defaultEditorsByColumnClass` qui est une variable d'instance de type `Hashtable` de la classe `JTable`. Il contient les associations suivantes :

- `Boolean` (représenté par une `JCheckBox`) ;
- `Number` (représenté par un `JTextField` avec alignement à droite) ;
- `Object` (représenté par un `JTextField` avec alignement à gauche).

Comme pour les *renderers*, on peut modifier les *editors* pour ces trois types de base, ou ajouter de nouvelles associations au dictionnaire à l'aide de la méthode `setDefaultEditor`.

4.5 L'ARCHITECTURE MVC

Nous venons de voir que les cellules d'un tableau peuvent être éditables. Comment récupérer les données saisies par l'utilisateur ? La méthode `getValueAt(int ligne, int colonne)` permet d'accéder à la valeur d'une cellule. Mais doit-on appeler cette méthode *n* fois, pour mettre à jour des objets métiers ou une base de données en arrière-plan de l'application ?

4.5.1 Intérêt de cette architecture

Souvent citée – et fort appréciée – par les utilisateurs du langage *Smalltalk*, l'architecture MVC consiste à **séparer les responsabilités** au sein d'un composant logiciel.

Dans le langage *Smalltalk*, elle consiste à implémenter trois classes au lieu d'une seule pour coder un élément graphique. Le rôle respectif de ces trois classes correspond à l'acronyme MVC, puisqu'il s'agit d'une classe « Modèle », d'une classe « Vue » et d'une classe « Contrôleur ».

Le modèle contient les données et fournit les méthodes pour y accéder en consultation ou en modification. La vue est responsable de la représentation visuelle d'une partie ou de la totalité des données. Le contrôleur se charge de gérer les événements en provenance de la vue : il est responsable de la cohérence entre la vue et le modèle.

Vous vous demandez peut-être l'intérêt de cette solution, qui nécessite trois classes au lieu d'une seule ?

Les intérêts sont multiples. D'une part, il est toujours intéressant de scinder le code en entités séparées, chacune ayant un rôle propre, car cela facilite l'évolution et la réutilisation du code. Le fait de séparer le modèle de la vue permet d'avoir des classes qui gèrent la représentation graphique bien distinctes des couches inférieures de l'application où sont stockées par exemple les données métier. Cette organisation en « couches » superposées est classique dans les projets informatiques. Les couches supérieures peuvent avoir accès aux couches sur lesquelles elles reposent. En revanche, une couche basse, comme la couche de persistance qui gère l'enregistrement des données, n'a aucun besoin de connaître les détails de l'IHM.

D'autre part, grâce à l'architecture MVC, il est possible d'avoir plusieurs vues sur le même modèle de données. Par exemple, si on revient à notre exemple de gestion de stocks dans une grande surface, on pourrait avoir une interface graphique « minimale » pour les employés qui effectuent la saisie, et une interface plus complète pour les gérants : ces derniers pourraient voir par exemple des informations sur la date de dernière commande ou les prix des produits. Ces renseignements n'intéressent pas l'employé : il est inutile d'encombrer son écran avec cela. Grâce à l'architecture MVC, on pourrait avoir le même modèle de données pour ces deux vues différentes. Cela offre l'avantage de ne pas dupliquer le stockage des données en mémoire. Par ailleurs, si le modèle vient à évoluer, on ne sera pas obligé de modifier du code à deux endroits différents.

4.5.2 L'implémentation de MVC dans Swing

Les principes de fonctionnement de cette architecture sont les suivants :

- un modèle peut avoir zéro ou plusieurs vues ;
- les vues se sont enregistrées auprès du modèle afin d'être averties en cas de changement d'une donnée ;
- lorsqu'un utilisateur modifie une donnée dans une vue, les mises à jour sont répercutées au niveau du modèle et les autres vues sont donc automatiquement actualisées.

L'architecture MVC de Swing est particulière. Elle doit permettre de supporter le *pluggable look and feel*. Un composant graphique met en œuvre trois objets principaux :

- un modèle ;
- un objet responsable du dessin et de la gestion des événements qu'on appelle parfois le `UIDelegate;`
- un composant graphique qui hérite de `Jcomponent`.

L'objet `UIDelegate` a été présenté rapidement dans le premier chapitre, dans le paragraphe concernant le *look and feel*. Il est responsable du dessin du composant ainsi que de la gestion des événements. Il joue donc à la fois le rôle de la vue et du contrôleur. Il possède de nombreux listeners qui sont abonnés au composant et au modèle, il est ainsi prévenu en cas de changement sur l'un ou l'autre. Ainsi, il peut gérer la concordance entre la vue et les données.

Le composant graphique, quant à lui, est l'interface proposée au programmeur pour manipuler des éléments graphiques. En effet, à part dans le cas où vous souhaitez écrire votre propre *look and feel*, vous n'aurez pas besoin de manipuler vous-mêmes le `UIDelegate`.

Vous constatez donc que l'implémentation de MVC en Swing n'est pas exactement celle de l'architecture telle qu'elle est habituellement décrite. Cependant, les principes de base sont toujours présents.

Certains préfèrent parler d'une architecture de type « Modèle-Composant ».

Jusqu'à présent, nous avons utilisé de nombreux composants graphiques, sans jamais se préoccuper de cette notion de modèle. Cela est possible car les composants Swing proposent un modèle par défaut qui est utilisé si aucun autre n'est spécifié explicitement. Les classes de composants graphiques disposent de méthodes pour accéder au modèle, et permettent à l'utilisateur de ne pas le faire manuellement.

Par exemple, dans la classe `JSlider`, composant dont on n'a pas parlé jusqu'à présent, et qui permet de définir une « règle graduée munie d'un curseur », la méthode `getMiminum` s'adresse au modèle pour lui demander l'information : `getModel().getMinimum()`.

Pour simplifier, si on ne représente que les objets que le programmeur devra manipuler — ce n'est que la partie visible de l'iceberg — on peut schématiser cela comme sur la figure 4.26 :

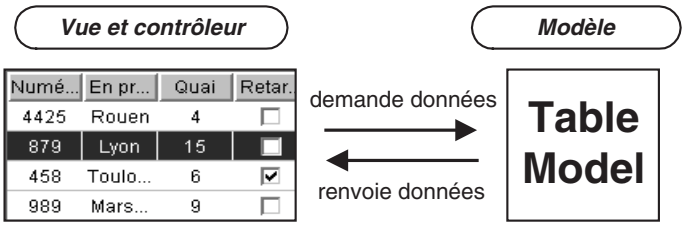


Figure 4.26 — Partie visible de MVC en Swing.

En définissant nous-mêmes un modèle de données, nous pouvons gérer le fait que les données à afficher sont extraites d'une base de données ou bien d'un pool d'objets métier se trouvant en mémoire, ou de n'importe quelle autre source. Nous sommes libres dans l'implémentation du modèle pour aller chercher les données à l'endroit adéquat. Dans le schéma figure 4.27, nous représentons le fait qu'une vue s'adresse à son modèle pour connaître les données à afficher et celui-ci peut par exemple effectuer une requête dans une base pour renvoyer les données à afficher.

Une autre utilisation du modèle est de répercuter les mises à jour de données en provenance de l'IHM dans une base de données ou dans n'importe quel niveau de stockage qui nous convient (couche métier…) (figure 4.28). Généralement, on ne code pas l'accès à la base de données dans le modèle lui-même, celui-ci peut déléguer cette tâche à une couche applicative mieux appropriée.

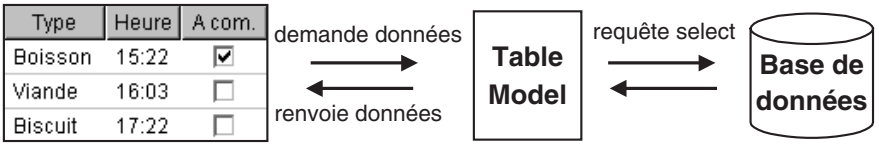


Figure 4.27 — Rechercher des données dans une base.

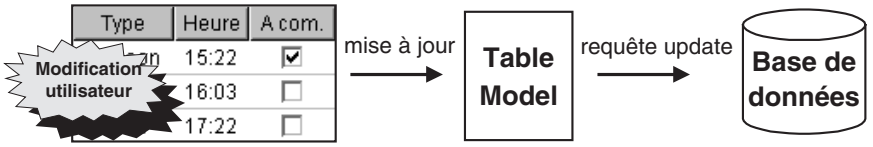


Figure 4.28 — Mettre à jour une base.

Enfin, toujours grâce à MVC, lorsque les données changent de façon externe — les données peuvent être modifiées à partir d'une autre application non graphique, ou bien elles évoluent au cours du temps sans aucune intervention extérieure — nous pouvons mettre à jour le modèle afin que toutes les vues soient rafraîchies en temps réel (figure 4.29).

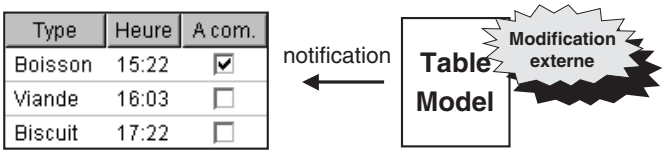


Figure 4.29 — Avertir la vue que les données ont changé.

4.5.3 Utilisation de cette architecture

Maintenant que nous connaissons les grands principes et les intérêts d'une telle architecture, voyons comment la mettre en œuvre dans notre exemple de gestion de stocks.

Les classes et interfaces proposées

Pour programmer un modèle lié à un composant `JTable`, il faut implémenter l'interface `TableModel`. Deux classes implémentent cette interface (figure 4.30).

Pour écrire notre propre modèle, trois solutions s'offrent à nous :

- implémenter directement `TableModel` ;
- hériter de `AbstractTableModel` ;

- hériter de `DefaultTableModel`.

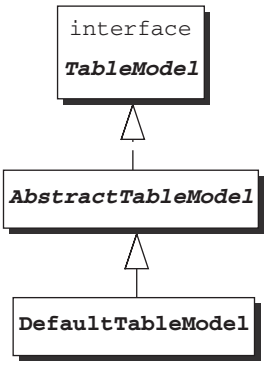


Figure 4.30 — Hiérarchie des classes et interfaces.

La classe `DefaultTableModel`, comme vous vous en doutez, est celle qui est utilisée lorsqu'on ne précise rien ; regardons l'implémentation de cette classe pour savoir si elle nous convient.

Les données sont stockées sous forme de vecteur de vecteurs. Ce choix très générique peut s'adapter à tous les tableaux. Il nous suffirait de redéfinir les méthodes dont l'implémentation ne nous convient pas. En réalité, cette façon de faire est très peu optimisée. À chaque fois que le tableau va vouloir accéder à une donnée, il va devoir parcourir le vecteur des lignes, puis dans la ligne, le vecteur contenant les éléments des différentes colonnes.

La classe `Vector` étant synchronisée pour s'affranchir des problèmes de concurrence d'accès à une instance dans deux threads, les appels sont plus lents, car ils nécessitent une pose de verrou.

Ce type d'implémentation est rapide : c'est ce que nous avons utilisé sans le savoir dans le tout premier exemple avec `JTable`, lorsqu'on passait les données en paramètres du constructeur. Mais, il ne peut convenir que dans le cas d'un petit tableau statique, dont les données n'ont pas besoin d'être mises à jour. Pour des utilisations plus conséquentes, nous vous déconseillons vivement d'utiliser cette stratégie.

La deuxième solution consiste à sous-classer `AbstractTableModel`. Pour que notre classe devienne instanciable, il est obligatoire de définir certaines méthodes. Cependant, on est plus libre dans l'implémentation. On n'est pas obligé d'utiliser un vecteur de vecteurs.

C'est la solution que nous choisissons pour notre exemple de gestion de stocks.

L'implémentation de notre modèle

Dans la classe que nous allons écrire, qui héritera de `AbstractTableModel`, on choisit de stocker une collection d'objets métier de type `Releve`, un relevé correspondant à une ligne dans le tableau.

Pour la collection, nous pouvons choisir la classe `ArrayList`, qui, à la différence de `Vector`, n'est pas synchronisée.

Les méthodes à implémenter obligatoirement sont :

- `getRowCount()` renvoie le nombre de lignes ;
- `getColumnCount()` renvoie le nombre de colonnes ;
- `getValueAt(int row, int col)` renvoie une instance de `Object` correspondant à la valeur à afficher. Cette méthode est appelée par le `JTable` à chaque fois que le tableau doit être redessiné. Il est donc important de soigner son implémentation.

D'autres méthodes peuvent être redéfinies :

- `getColumnClass(int col)` renvoie la classe des objets affichés dans la colonne `col`. Cette méthode est nécessaire pour utiliser les *renderers* et les *editors* associés à des types particuliers (entiers, chaînes de caractères, booléens…) ;
- `isCellEditable(int row, int col)` renvoie un booléen indiquant si la cellule située à la ligne `row` et à la colonne `col` est éditable ;
- `setValueAt(Object value, int row, int col)` est appelée lorsqu'une donnée du tableau est modifiée. On peut implémenter cette méthode pour mettre à jour les objets métier `Releve`.

L'implémentation de notre modèle est donc la suivante :

```
public class TableauRelevesModel extends
  AbstractTableModel {
   String[] nomsColonnes = {"Employé", "Référence", "Type",
      "Heure", "Quant.", "A com."};
    ArrayList donnees;

   public TableauRelevesModel() {
    donnees = new ArrayList();
    initialiser();
   }

   public int getColumnCount() {
      return nomsColonnes.length;
   }
```

```
    public int getRowCount() {
      return donnees.size();
    }

    public String getColumnName(int col) {
      return nomsColonnes[col];
    }

    public Object getValueAt(int row, int col) {
      // Obtention de l'objet métier Releve à l'aide
         // de la ligne de Renvoi de l'attribut souhaité
en fonction
      // du numéro de colonne
      // [ . . . ]
      // Cas simplifié d'un ArrayList contenant des
      // ArrayList :
      return ((ArrayList)donnees.get(row)).get(col);
    }

    public Class getColumnClass(int c) {
      return getValueAt(0, c).getClass();
      }

      public void initialiser() {
        // Obtention des données à partir d'une base ou
        // d'un fichier de sérialisation…
      }
}
```

Notez l'implémentation de `getColumnClass`, qui permet de renvoyer le type réel de l'objet. Nous avons vu plus haut que cette implémentation était indispensable pour bénéficier des *renderers* par défaut.

Pour rendre le tableau éditable, il suffit d'ajouter les deux méthodes suivantes :

```
public boolean isCellEditable(int rowIndex, int columnIndex){
  return true;
}
public void setValueAt(Object value, int rowIndex, int
➥columnIndex) {
  // mise à jour des objets Releve
}
```

Maintenant que notre classe de modèle est créée, il faut ajouter ce modèle au `JTable`. Cela se fait de la façon suivante au moment de l'appel au constructeur :

```
TableauRelevesModel modele = new TableauRelevesModel();
JTable tableau = new JTable(modele);
```

Cela peut aussi se faire *a posteriori* :

```
tableau.setModel(modele) ;
```

Comment gérer une modification « externe » ?

Supposons que les objets métier `Releve` puissent être modifiés autrement que par l'intermédiaire d'une vue. Comment faut-il procéder pour que toutes les vues soient impactées par cette modification ?

Le modèle dispose de méthodes pour déclencher des événements relatifs à des changements du modèle. Grâce au principe des listeners abonnés au modèle, toutes les vues seront prévenues.

Imaginons les méthodes suivantes sur notre classe `TableauRelevesModel`. Ces méthodes sont appelées lorsqu'une modification survient dans une autre partie de l'application.

```
public void ajouterReleve(Releve t) {
   //on ajoute un nouveau relevé à la fin du tableau
   [...ajout à l'ArrayList contenant toutes les lignes...]
   // x correspond au numéro de cette dernière ligne
   fireTableRowsInserted(x, x);
}

public void enleverLigne(int numeroLigne) {
  [...retrait de l'élément numeroLigne à l'ArrayList...]
   fireTableRowsDeleted(numeroLigne, numeroLigne);
}

public void modifierQuantite(int numeroLigne, int
➥nouvelleQuantite) {
   [...modification de l'élément numeroLigne de
   ➥l'ArrayList ...]
   fireTableRowsUpdated(numeroLigne, numeroLigne);
}
```

Récapitulatif des classes liées au `JTable`

Nous avons vu tout au long de ce chapitre de nombreuses classes liées au composant `Jtable` (figure 4.31). En effet, l'implémentation de Swing est très largement basée sur la séparation du code dans différentes classes ayant des rôles distincts.

4.6 APPLICATION À L'EXEMPLE DE GESTION DE SIGNETS

Reprenons notre exemple de gestion de signets. Les connaissances acquises dans ce chapitre nous permettent de compléter le code pour répondre aux besoins suivants :

- la sélection d'un nœud dans l'arbre doit provoquer l'affichage détaillé dans la partie droite de l'écran ;
- les nœuds de l'arbre doivent être affichés avec une apparence personnalisée.

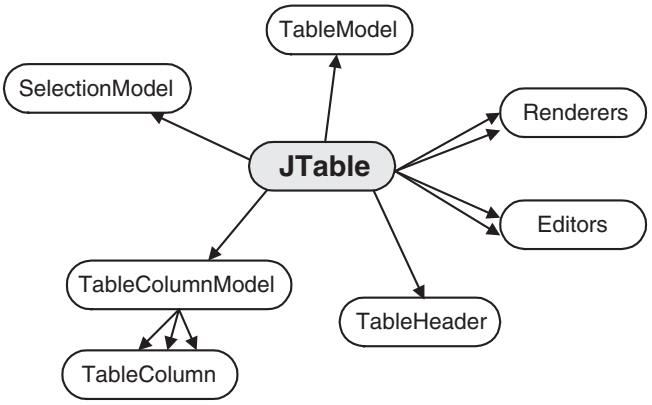


Figure 4.31 — Récapitulatif des classes liées au `Jtable`.

4.6.1 Gérer la sélection dans l'arbre

La création de l'arbre se fait dans la classe `PanneauDetail`, et plus précisément dans la méthode `initialiserArbre` qui est appelée dans le constructeur.

À l'ouverture, l'arbre contient seulement un nœud racine, qui est un objet `CategoriePersistante` de nom « Racine ». L'arbre est placé dans un `JScroll Pane`, qui dispose toujours d'une barre de défilement verticale et, si besoin, d'une barre de défilement horizontale.

En ce qui concerne la sélection, l'arbre ne doit autoriser que la sélection unique. Nous souhaitons implémenter une action lorsque la sélection change. Nous allons donc écrire un `TreeSelectionListener`.

```
/**
 * Initialisation de l'arbre à l'ouverture
 */
public void initialiserArbre() {
   CategoriePersistante racine =
   ➥new CategoriePersistante("Racine");
   DefaultMutableTreeNode nRacine =
   ➥new DefaultMutableTreeNode(racine);
   arbre = new JTree(nRacine);
   arbre.setRootVisible(true);
   arbreScrollPane = new JScrollPane(arbre);
   arbreScrollPane.setHorizontalScrollBarPolicy(
      JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
   arbreScrollPane.setVerticalScrollBarPolicy(
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
   arbre.getSelectionModel().setSelectionMode(
      TreeSelectionModel.SINGLE_TREE_SELECTION);
   arbre.addTreeSelectionListener(new
    ArbreSelectListener(fenetrePpale.getPanneauDetail()));
   }
```

Pour écrire le listener sur la sélection dans l'arbre, nous avons choisi d'écrire une nouvelle classe dans le package `ihm`. Ce listener doit disposer d'une référence sur l'instance de `PanneauDetail` qui affiche l'arbre. En effet, c'est sur cette classe que nous avons écrit les méthodes d'enregistrement et d'affichage du détail d'un nœud.

Lors d'un changement de sélection, nous devons :

- enregistrer les informations sur le nœud qui vient d'être déselectionné ;
- afficher le détail du nouveau nœud sélectionné.

Pour obtenir respectivement l'ancienne et la nouvelle sélections, nous allons utiliser les méthodes `getOldLeadSelectionPath` et `getNewLeadSelection Path` de la classe `TreeSelectionEvent`. Elles renvoient des références de type `TreePath`, à partir desquelles on peut obtenir des `DefaultMutableTreeNode`.

Voici donc cette classe `ArbreSelectListener` :

```
package ihm;

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
import metier.*;

/**
 * Ce listener a pour rôle de mettre à jour le panneau de
 * droite dans la fenêtre principale avec les informations
 * du nœud qui vient d'être sélectionné.
 * Par ailleurs, les modifications sur l'objet métier qui
 * vient d'être désélectionné sont enregistrées.
 */
public class ArbreSelectListener implements
  TreeSelectionListener {

   PanneauDetail panneauCible;

   public ArbreSelectListener(PanneauDetail pan) {
      panneauCible = pan;
   }

   /**
    * Cette méthode est appelée lorsqu'un changement de
    * sélection a lieu.
    */
   public void valueChanged(TreeSelectionEvent e) {
      // Enregistrement du nœud qui vient d'être
      // désélectionné
      TreePath chemin = e.getOldLeadSelectionPath();
```

```
            if (chemin != null) {
               DefaultMutableTreeNode noeudDeselect =
                  DefaultMutableTreeNode)chemin.
                  getLastPathComponent();
               Noeud omDeselect = (Nœud) noeudDeselect
                  .getUserObject();
               Noeud omModifie = panneauCible.enregistrerNoeud
               ➥(omDeselect);
               // Dans le cas où omDeselect a changé de nature
               // (catégorie -> signet par exemple), il faut
               // affecter ce nouveau
               // noeud à l'objet graphique TreeNode
                   noeudDeselect.setUserObject(omModifie);
            }

            // Affichage du détail du nouveau nœud sélectionné
           TreePath cheminNouveau = e.getNewLeadSelectionPath();
            if (cheminNouveau != null) {
               DefaultMutableTreeNode noeudSelect =
                  (DefaultMutableTreeNode)cheminNouveau
                  ➥.getLastPathComponent();
               if (noeudSelect != null) {
                    Nœud omSelect = (Nœud) noeudSelect.getUser
                    ➥Object();
               if (omSelect != null) {
                   panneauCible.afficherNoeud(omSelect);
                  }
               }
            }
         }
      }
```

Nous allons maintenant détailler les deux méthodes `afficherNoeud` et `enregistrerNoeud` de `PanneauDetail`.

La méthode `afficherNoeud` est simple, elle effectue les actions suivantes :

- sélectionner le radio bouton adéquat (catégorie ou signet) ;
- afficher le nom et la description du nœud ;
- afficher l'URL dans le champ de texte s'il s'agit d'un signet.

```
/**
 * Cette méthode permet d'afficher les informations
 * concernant un nœud donné.
 */
public void afficherNoeud(Nœud noeud) {
   if (noeud instanceof Categorie) {
      rbCat.setSelected(true);
      configurerIHM(CONFIG_CATEGORIE);
   }
```

```
    else if (noeud instanceof Signet) {
      rbSignet.setSelected(true);
      configurerIHM(CONFIG_SIGNET);
      URL urlNoeud = ((Signet)noeud).getUrl();
      if (urlNoeud != null) {
        tfUrl.setText(urlNoeud.toString());
      }
      else {
        tfUrl.setText("");
      }
    }
    tfNom.setText(noeud.getNom());
    taDescription.setText(noeud.getDescription());
  }
```

La méthode `enregistrer` doit obtenir les informations du panneau de détail et les affecter à l'objet `Nœud`. Avant tout enregistrement, on vérifie que le nom, qui est une information obligatoire, a bien été saisi.

La seule difficulté de cette méthode est le cas où l'utilisateur a changé la nature du nœud (catégorie en signet ou inversement). Dans ce cas, il faut créer un nouvel objet adéquat.

```
  /**
   * Cette méthode permet d'enregistrer les informations
   * saisies dans l'objet métier affiché.
   *
   * @param noeud le nœud à enregistrer
   * @return le nœud modifié
   */
  public Noeud enregistrerNoeud(Nœud noeud) {
    // Récupération du champ nom. Si ce champ est vide,
    // l'enregistrement ne peut avoir lieu.
    String n = tfNom.getText();
    if (n == null || n.equals("")) {
      JOptionPane.showMessageDialog(this,
        "Enregistrement impossible : un noeud doit avoir "
        + "un nom", "Enregistrement impossible",
      JOptionPane.ERROR_MESSAGE);
      return null;
    }
    // Cas où l'utilisateur a transformé la nature du nœud :
    // signet -> catégorie ou inversement
    if (noeud instanceof Signet && rbCat.isSelected()) {
      noeud = new CategoriePersistante(n);
    }
    else if (noeud instanceof Categorie &&
    Signet.isSelected()) {
      noeud = new Signet(n);
    }
    else {
```

```
            noeud.setNom(n);
            noeud.setDescription(taDescription.getText());
         }
         // Dans le cas d'un signet, on enregistre l'URL
         if (noeud instanceof Signet) {
            String url = tfUrl.getText();
            if (url != null) {
               try {
                  ((Signet)noeud).setUrl(new URL(url));
               } catch (MalformedURLException e) {
                  e.printStackTrace();
               }
            }
         }
         return noeud;
      }
```

4.6.2 Paramétrer l'apparence de l'arbre

À la suite de ce chapitre, nous pouvons apporter une nouvelle amélioration à notre exemple, qui commence à prendre forme.

Nous allons implémenter un *renderer* particulier pour l'arbre, afin que la distinction entre catégorie et signet se fasse au premier coup d'œil.

Notre cahier des charges est le suivant : le texte visible sur les nœuds de l'arbre correspond au nom du signet ou de la catégorie. Les catégories doivent toujours utiliser l'icône représentant les dossiers, et cela même si une catégorie ne contient aucun nœud. Les catégories sont toujours affichées en gras et les signets en italique.

Notre *renderer* repose sur la classe `DefaultTreeCellRenderer`. En particulier, nous allons utiliser la variable d'instance de type `JLabel` utilisée dans cette classe mère. Dans la méthode qui doit renvoyer le composant graphique paramétré, nous appelons la méthode de la classe mère et nous modifions l'instance de `JLabel` retournée.

Afin d'obtenir l'icône spécifique des « dossiers » pour toutes les catégories, y compris celles qui n'ont aucun fils, il suffit d'appeler cette méthode avec la valeur `false` pour le paramètre `isLeaf`.

```
package ihm;

import javax.swing.*;
import javax.swing.tree.*;
import java.awt.*;
import metier.*;
```

```
public class ArbreRenderer extends DefaultTreeCellRenderer
{

  public Component getTreeCellRendererComponent(
    JTree arbre,
    Object valeur,
    boolean selectionne,
    boolean etendu,
    boolean feuille,
    int ligne,
    boolean avecFocus) {

    JLabel label = null ;
   if ( ((DefaultMutableTreeNode)valeur).getUserObject()
      instance of Categorie) {
      label = (JLabel) super
      ➥.getTreeCellRendererComponent(arbre, valeur,
      ➥selectionne, etendu, false, ligne, avecFocus);
   label.setFont( new Font("Dialog", Font.BOLD, 12));
  }
  else if ( ((DefaultMutableTreeNode)valeur)
  ➥ .getUserObject()
    instanceof Signet) {
    label = (JLabel) super
    ➥.getTreeCellRendererComponent(arbre, valeur,
    ➥selectionne, etendu, true, ligne, avecFocus);
    label.setFont( new Font("Dialog", Font.ITALIC,
    ➥12));
  }
  else {
    label = (JLabel) super
    ➥.getTreeCellRendererComponent(arbre, valeur,
    ➥selectionne, etendu, feuille, ligne, avecFocus);
  }
  return label;
 }
}
```

5

Les composants texte

Les composants texte sont très utilisés dans la plupart des applications, car la saisie de texte par l'utilisateur reste le moyen le plus intuitif pour sauvegarder des données, indiquer un choix (par exemple la saisie d'un mot pour un moteur de recherche), s'identifier (par exemple la saisie d'un *login* et d'un mot de passe), etc. Il est donc indispensable d'être en mesure de déterminer le composant qui correspond à ce que l'on souhaite faire et ensuite de savoir l'utiliser et l'adapter à son besoin.

Dans ce chapitre, nous verrons tout d'abord une utilisation simple de chacun des composants texte qui existent, puis, nous montrerons l'utilisation plus poussée que l'on peut en faire, avec la personnalisation des modèles de documents, l'utilisation des actions standard et l'implémentation de l'annulation.

5.1 PRÉSENTATION GÉNÉRALE DES DIFFÉRENTS COMPOSANTS TEXTE

La problématique du choix des composants texte intervient très tôt dans la réalisation d'une application, car on commence souvent à les utiliser dès la toute première fenêtre lorsque l'identification d'un utilisateur est nécessaire.

Nous commencerons donc par cet exemple trivial. Voici figure 5.1 la fenêtre que nous souhaitons obtenir.

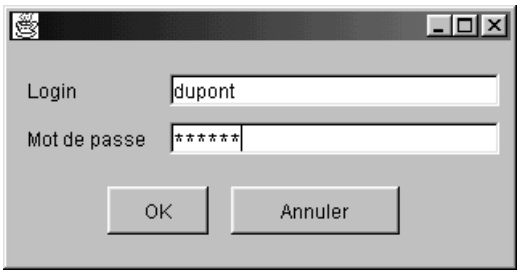


Figure 5.1 — Fenêtre d'identification.

5.1.1 Choix d'un composant texte

Pour déterminer les composants à utiliser pour notre fenêtre d'identification, consultons la documentation de Java pour connaître les composants qui existent.

Swing fournit cinq composants texte qui répondent aux besoins classiques des applications (figure 5.2).

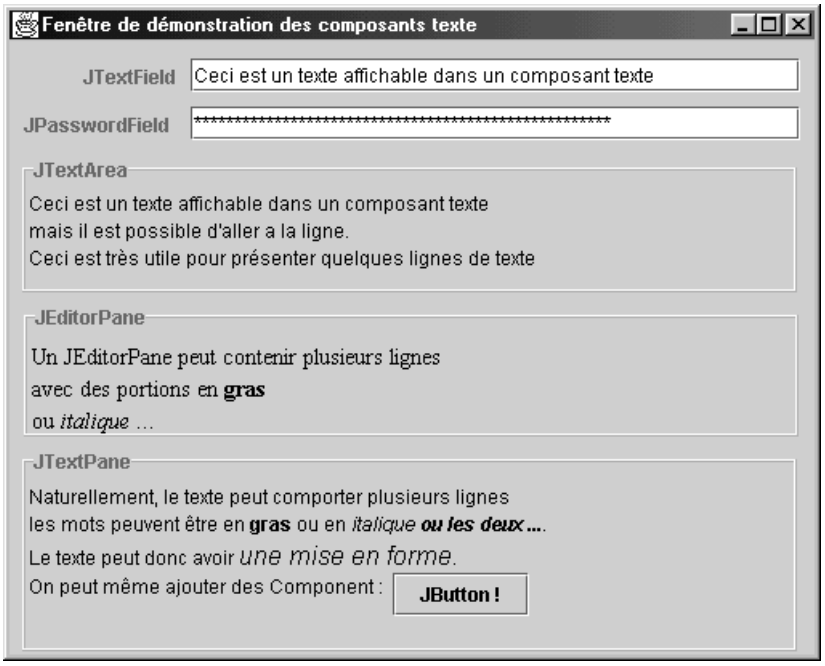


Figure 5.2 — Les 5 composants texte de Swing.

Les caractéristiques de ces composants sont résumées dans le tableau 5.1.

Tableau 5.1 — Les caractéristiques des composants texte de Swing.

	Texte sur plusieurs lignes	Utilisation de plusieurs styles de texte	Possibilité d'inclure des images et des composants
JTextField	Non	Non	Non
JPasswordField	Non	Non	Non
JTextArea	Oui	Non	Non
JEditorPane	Oui	Oui	Non
JTextPane	Oui	Oui	Oui

Les composants texte se trouvent dans le package `javax.swing`. Ils héritent tous d'une classe abstraite `JTextComponent`, située dans le package `javax.swing.text`. Cette classe supporte les fonctionnalités de base des composants texte, que nous détaillerons plus tard (figure 5.3).

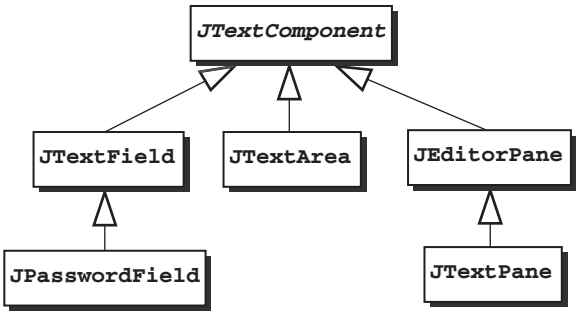


Figure 5.3 — Hiérarchie des composants texte.

D'après les caractéristiques des différents composants, nous constatons que `JTextField` et `JPasswordField` sont tout à fait adaptés à notre besoin. En effet, il s'agit de saisir du texte sur une seule ligne et sans mise en forme particulière.

5.1.2 Une première utilisation de `JTextField` et `JPasswordField`

Instanciation des composants

Constructeur sans paramètre

Pour créer le champ de saisie du *login*, on utilise le constructeur sans paramètre de `JTextField` :

```
JTextField tfLogin = new JTextField();
```

Constructeur avec un texte par défaut

Toutefois, sur tous ces composants texte, il est possible de préciser un texte affiché par défaut. Imaginons un formulaire de saisie quelconque. Il est fréquent d'avoir un champ « Date » qui permet de sauvegarder la date de création d'un nouvel objet métier quelconque. Ce champ « Date » peut être construit de telle sorte qu'il affiche par défaut la date du jour. Cela facilite la saisie de l'utilisateur, tout en lui laissant la possibilité de modifier cette date, si la date du jour n'est pas la valeur qui convient (figure 5.4).

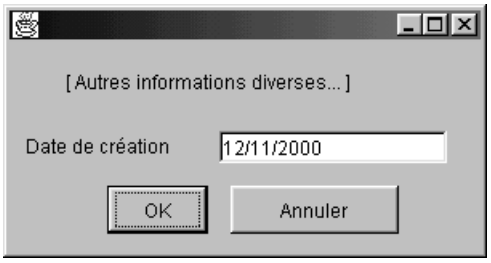


Figure 5.4 — `JTextField` avec un texte par défaut.

Pour obtenir la date du jour, il suffit d'utiliser le constructeur sans paramètre de la classe `java.util.Date`. L'affichage de la date peut se faire à l'aide d'un objet de formatage de date, par exemple une instance de `SimpleDateFormat`. Le texte à afficher par défaut est passé en paramètre du constructeur de `JTextField` :

```
Date dateAujourdhui = new Date();
SimpleDateFormat formatClassique = new
➥ SimpleDateFormat("dd/MM/yyyy");
JTextField tfDateCreation = new
➥ JTextField(formatClassique.format(dateAujourdhui));
```

Si on souhaite que la date de création ne soit pas modifiable par l'utilisateur, il suffit d'utiliser la méthode `setEditable` en passant `false` en paramètre pour rendre un champ de texte non saisissable. Par défaut, un composant texte est saisissable.

Pour créer le champ mot de passe de la fenêtre de login, on utilise le constructeur sans paramètre de `JPasswordField` :

```
JPasswordField pfMotDePasse = new JPasswordField();
```

Le caractère qui s'affiche par défaut lorsqu'on saisit un mot de passe est « * ». Il est possible de modifier ce caractère à l'aide de la méthode `setEchoChar(char c)` :

```
pfMotDePasse.setEchoChar('-');
```

Remarque : les actions Copier et Couper ne fonctionnent pas dans un champ de saisie d'un mot de passe pour des raisons de sécurité évidentes !

Si l'on souhaite malgré tout obtenir l'affichage « en clair » du mot de passe, il suffit de passer en paramètre de la méthode `setEchoChar` le chiffre 0, préalablement casté en `char`.

Constructeur avec une taille par défaut

Supposons que l'application qui va utiliser la fenêtre d'identification que nous venons d'écrire interdise les *login* de plus de 10 caractères et oblige tous les utilisateurs à avoir des mots de passe de 8 caractères exactement. Pour guider l'utilisateur dans sa saisie, nous pouvons faire en sorte que la largeur des deux composants s'adapte à cette contrainte.

Il est possible de passer en paramètre des constructeurs des composants texte un nombre de colonnes. Ce nombre de colonnes correspond au nombre maximal de caractères visibles dans le composant. La taille du champ, et plus précisément sa `preferredSize`, est positionnée à partir de cette indication.

Attention toutefois à cette correspondance entre une information concernant le contenu du composant texte et sa représentation graphique ! Certaines précautions sont nécessaires pour obtenir le résultat attendu :

- le layout utilisé pour disposer le composant dans le container doit tenir compte, au moins horizontalement, de la `preferredSize`, sinon le fait d'indiquer un nombre de colonnes n'aura aucun effet (voir chapitre 2 sur les `layouts` pour plus d'informations) ;
- la largeur d'une colonne est calculée à partir de la largeur de la lettre « m » dans la police courante. Pour que le nombre de colonnes spécifié corresponde exactement au nombre de caractères visibles à l'affichage, et ceci quel que soit le texte saisi, il faut utiliser une police non proportionnelle (figure 5.5).

Figure 5.5 — Fenêtre d'identification, avec initialisation du nombre de colonnes.

Définition : une police est dite « proportionnelle » lorsque l'espace nécessaire pour afficher chaque caractère varie. Ainsi un « m » sera plus large qu'un « i ». La police « Times new roman » est une police proportionnelle alors que la police « courrier new » est non proportionnelle.

Voici le code permettant cette initialisation :

```
Font police = new java.awt.Font("Monospaced", 0, 14);
tfLogin.setFont(police);
pfMotDePasse.setFont(police);
// ajout d'1 colonne supplémentaire pour une
// meilleure visibilité
JTextField tfLogin = new JTextField(11);
JPasswordField pfMotDePasse = new JPasswordField(9);
```

Gestion de l'apparence

Paramétrage de la police

Nous avons vu ci-dessus qu'il était possible de changer la police d'un composant texte. Nous avons utilisé pour cela la méthode `setFont` qui prend en paramètre un objet de la classe `Font`. Quelles sont les polices disponibles et comment crée-t-on sa propre police ?

Dans l'exemple qui suit, le but est d'obtenir la liste des polices disponibles, qui est affichée dans la partie gauche de l'écran. Le texte situé dans la partie droite change de police lorsqu'on sélectionne un item de la liste (figure 5.6).

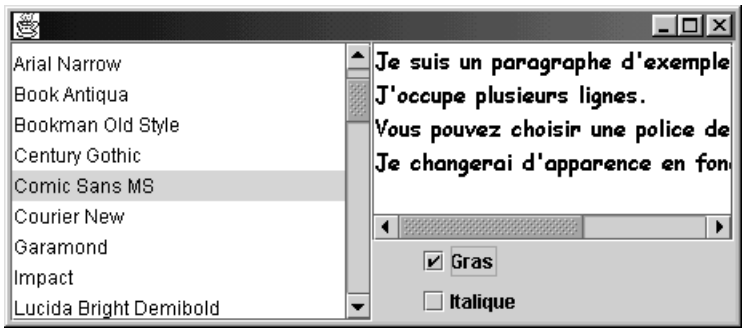


Figure 5.6 — Paramétrage de la police.

L'obtention de la liste des polices disponibles se fait de la façon suivante :

```
Font[] polices = GraphicsEnvironment
➥.getLocalGraphicsEnvironment().getAllFonts();
```

Cette méthode renvoie des objets `Font` pour lesquels la taille a été positionnée à 1. Pour que le texte soit visible dans le panneau de droite, il faut bien sûr augmenter cette taille. Regardons les méthodes proposées par la classe `Font`. Notez que cette classe se trouve dans le package `java.awt`. Elle existe depuis le JDK 1.0, mais de nombreux services ont été ajoutés à partir du JDK 1.2, puis 1.3.

Il est possible d'obtenir une nouvelle police à partir d'une police existante en modifiant sa taille, son style, ou tout autre attribut à l'aide de la méthode `deriveFont`.

```
float taille = 14;
taTexteExemple.setFont(f.deriveFont(taille));
```

Les polices sont regroupées par familles. Dans notre exemple, la liste des polices présentées n'est qu'un sous-ensemble des polices retournées par la méthode `getAllFonts`. Nous n'avons retenu qu'un exemplaire par famille de polices (élimination de toutes les combinaisons d'attributs : gras, italique…). La famille d'une police est une chaîne de caractères qui s'obtient par la méthode `getFamily`.

Les polices affichées dans la liste sont toutes de taille 1 et sans attributs. Nous avons vu comment la méthode `deriveFont(float taille)` de la classe `Font` nous permettait d'affecter au composant texte une police identique mais de taille différente de 1. Pour faire fonctionner les cases à cocher « Gras » et « Italique », nous allons procéder de la même manière : obtenir une nouvelle police à partir d'une police existante à l'aide de la méthode `deriveFont`. Cette fois, la signature paramétrique de la méthode est différente : `deriveFont(int style)`. Comme souvent, le paramètre entier `style` prend ses valeurs parmi des constantes attributs de la classe `Font`.

Soit *f* une police qui ne soit pas en gras. Voici une ligne de code qui remplacera la référence *f* par une nouvelle police identique à la précédente mais en gras :

```
f = f.deriveFont(Font.BOLD);
```

Maintenant, notre exemple fonctionne correctement. Nous souhaitons toutefois ajouter à la liste une police tout à fait personnalisée.

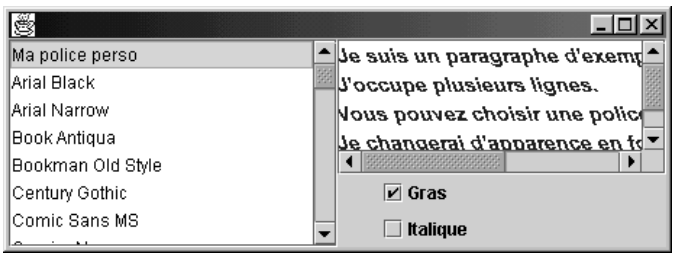


Figure 5.7 — Création d'une nouvelle police.

Pour la création de cette police personnalisée (figure 5.7), nous utilisons le constructeur de `Font`, qui prend en paramètre un dictionnaire d'attributs. Les clés du dictionnaire doivent être des constantes définies dans la classe `TextAttribute`. Cette classe se trouve dans le package `java.awt.font` qui est apparu avec la version 1.2 du JDK.

L'originalité de cette police personnelle réside dans l'inclinaison du texte vers la gauche. Pour obtenir cet effet, il faut appliquer une transformation affine aux caractères affichés.

Définition : une transformation affine est une transformation qui conserve le parallélisme. L'API Java 2D offre de nombreuses classes permettant de travailler sur le graphisme : création de transformations, compositions de formes, transformations d'images, etc.

Il existe plusieurs moyens d'obtenir une instance de `AffineTransform`.

Le premier et le plus générique est de passer par le constructeur. Celui-ci prend en paramètres des valeurs numériques qui constituent une matrice de transformation géométrique. Nous vous laissons consulter la documentation du JDK qui donne l'équation de la transformation.

Le deuxième moyen est plus simple : la classe `AffineTransform` constitue une fabrique de transformations ; elle propose des méthodes statiques qui créent des instances de `AffineTransform` pour certaines transformations courantes. Ainsi, la méthode `getShearInstance` permet de créer une transformation de type cisaillement. Les paramètres indiquent le sens du cisaillement et son importance.

La création de la police personnelle présentée ci-dessus se fait de la façon suivante :

```
Hashtable attributs = new Hashtable();
attributs.put(TextAttribute.FONT, new Font("Dialog", 0, 1));
attributs.put(TextAttribute.FOREGROUND, Color.blue);
TransformAttribute attributTransform = new
  TransformAttribute(AffineTransform.getShearInstance(
  ➥ 0.3, 0.0));
attributs.put(TextAttribute.TRANSFORM, attributTransform);
Font maPolice = new Font(attributs);
```

Paramétrage de l'alignement

Un autre paramétrage de l'apparence consiste à spécifier un alignement. Par défaut, l'alignement est de type `LEADING`. Cela signifie que le curseur est positionné à gauche dans le composant texte si l'application est écrite dans une langue qui s'écrit de gauche à droite, et *vice versa*. Pour déterminer la langue de l'application, le programme s'appuie sur la variable `Locale`.

Cependant, pour afficher des composants qui servent à saisir des nombres, certains préfèrent un alignement à droite. Imaginons la création d'une fenêtre de saisie pour un site de petites annonces (figure 5.8).

Nous pouvons reprendre le composant « Date de création » créé plus haut, et nous voulons ajouter un champ « Prix ».

Figure 5.8 — Composant texte avec un alignement à droite.

Voici le code nécessaire pour paramétrer l'alignement :

```
tfPrix.setHorizontalAlignment(SwingConstants.RIGHT);
```

Obtention des chaînes de caractères saisies

Dans notre exemple, nous souhaitons afficher un message d'erreur lorsque le couple *login*/mot de passe est différent de : « sesame/ouvretoi ». Pour cela, il faut obtenir les chaînes saisies par l'utilisateur dans les champs « Login » et « Mot de passe ». Dans le cas d'un composant `JTextField`, la méthode adéquate est `getText`. En revanche, pour le composant `JPasswordField`, la méthode `getText`, disponible par héritage, est indiquée `deprecated`. Il faut donc utiliser `getPassword`.

```
btOk.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      if (tfLogin.getText().equals("sesame") &&
         String.copyValueOf(pfMotDePasse.getPassword())
         .equals("ouvretoi")) {
         // Autoriser l'entrée dans une application…
         JOptionPane.showMessageDialog(Fenetre.this,
         "Identification acceptée",
         "Bienvenue", JOptionPane.INFORMATION_MESSAGE);
      }
      else {
         JOptionPane.showMessageDialog(Fenetre.this,
         "Le couple login / mot de passe est incorrect",
         "Erreur", JOptionPane.ERROR_MESSAGE);
      }
   }
});
```

Compatibilité AWT

Les classes `JTextField` et `JPasswordField` sont équivalentes aux classes `TextField` et `PasswordField` d'AWT. Si vous souhaitez faire évoluer une application AWT existante en Swing, il vous suffit de modifier les classes utilisées : remplacer `TextField` par `JTextField`, etc. Le code existant restera valide.

Remarque : avec un composant `javax.swing.JTextField`, comme avec `java.awt.TextField`, le fait d'appuyer sur la touche « Entrée » provoque donc un `ActionEvent`. Il suffit d'ajouter un `ActionListener` au composant texte pour traiter ces événements. Nous verrons dans la partie sur les raccourcis clavier qu'il est cependant possible de désactiver ce comportement si on le souhaite.

5.1.3 Une première utilisation de `JTextArea`

Reprenons notre exemple de formulaire de saisie d'une petite annonce. Nous souhaitons obtenir le résultat de la figure 5.9.

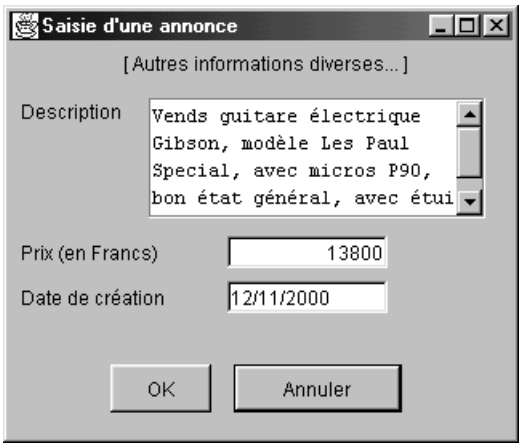


Figure 5.9 — Formulaire de saisie avec un `JtextArea`.

Pour cela, ajoutons un `JTextArea` à la fenêtre principale de cette application. Puisque nous ne souhaitons pas préciser de texte par défaut ou de nombre de colonnes, nous pouvons utiliser le constructeur sans paramètre. Pour que l'utilisateur puisse consulter la totalité du texte, il faut utiliser un panneau avec ascenseurs. Pour ce faire, créons une instance de `JScrollPane`. Il suffit de passer le composant à visualiser en paramètre du constructeur du `JscrollPane` :

```
JTextArea taDescription = new JTextArea();
JScrollPane jspDescription =
➥new JScrollPane(taDescription);
```

Si, pour une raison particulière, l'ajout du composant dans le `JScrollPane` ne peut pas se faire au moment de l'instanciation, il est possible de l'ajouter après. Ce cas peut se produire si le `JScrollPane` doit être affiché à l'écran dès le départ et le composant est ajouté seulement lorsque l'utilisateur effectue une action spécifique. Dans ce cas, il faut employer la méthode `getViewport` de la classe `JScrollPane` :

```
jspDescription.getViewport().add(taDescription, null);
```

Pour plus de détails sur l'utilisation du `JScrollPane`, reportez-vous chapitre 1 section 1.4.3 où est présentée l'interface `Scrollable`.

Le passage à la ligne

Si on ne paramètre pas davantage le composant texte, voilà figure 5.10 ce qu'on obtient.

Figure 5.10 — `JTextArea` sans passage à la ligne.

Il y a un problème de passage à la ligne. Nous souhaitons qu'un retour à la ligne soit fait automatiquement et donc qu'il n'y ait pas de barre de défilement horizontale. En revanche, une barre de défilement verticale peut apparaître si tout le texte n'est pas visible.

Pour obtenir cela, il faut demander au `JTextArea` de gérer le passage à la ligne :

```
taDescription.setLineWrap(true);
```

Avec cette instruction supplémentaire, voici figure 5.11 ce que nous obtenons.

Figure 5.11 — `JTextArea` avec coupure au niveau caractère.

Il faut indiquer que le passage à la ligne doit s'effectuer entre 2 mots pour obtenir le résultat escompté. La coupure des mots peut être refusée avec l'instruction suivante :

```
taDescription.setWrapStyleWord(true);
```

La touche Tabulation

Une particularité de la classe `JTextArea` réside dans le fait que la touche « Tabulation » permet d'insérer une tabulation dans le texte et non de passer le focus au composant suivant, comme c'est le cas pour les autres composants. Ce comportement est obtenu grâce à la redéfinition de la méthode `isManaging Focus`. Dans notre exemple, étant donné que le texte saisi doit être court, et avec une mise en forme minimale, nous souhaitons que la touche « Tab » garde son comportement habituel, c'est-à-dire passer le focus au composant suivant.

Il est alors nécessaire de définir une sous-classe de `JTextArea`. La méthode `isManagingFocus` doit renvoyer faux pour indiquer au gestionnaire de focus que le composant ne gère pas les touches Tab, Shift+Tab, Ctrl+Tab, Ctrl+Shift+Tab.

```
JTextArea taDescription = new JTextArea(){
  public boolean isManagingFocus() {
    return false;
  }
};
```

Compatibilité AWT

`JTextArea` peut facilement remplacer la classe `TextArea` du package `java.awt`, sauf qu'il existait 2 méthodes dans `TextArea` qui n'existent plus dans `JTextArea`. Ce sont les méthodes liées au défilement du texte grâce à un ascenseur. Les classes graphiques AWT géraient elles-mêmes le défilement. Ces méthodes n'ont plus lieu d'être avec Swing puisque le défilement est entièrement géré par le `JScrollPane`.

5.1.4 Une première utilisation de `JEditorPane`

Reprenons notre fameux exemple de la gestion de signets. Nous avons pour l'instant une fenêtre qui affiche une arborescence de signets dans la partie gauche de l'écran et la description plus précise d'un signet qui s'affiche dans la partie droite lorsqu'on sélectionne un nœud de l'arbre (figure 5.12).

Figure 5.12 — Fenêtre principale de la gestion de signets.

Le bouton « Voir URL » permet de visualiser la page référencée par l'URL. Pour cela, nous pourrions lancer un navigateur Internet et afficher le site. Mais nous pouvons également utiliser une instance de la classe `JEditorPane` de Swing. En effet, le `JEditorPane` permet d'afficher différents types de contenus texte formatés. Swing fournit l'implémentation pour afficher 3 types de format :

- texte non mis en forme ;
- HTML ;
- RTF (Rich Text Format).

Voici figure 5.13 le résultat que nous souhaitons obtenir. Ici, l'URL à afficher est l'index de la documentation de Java.

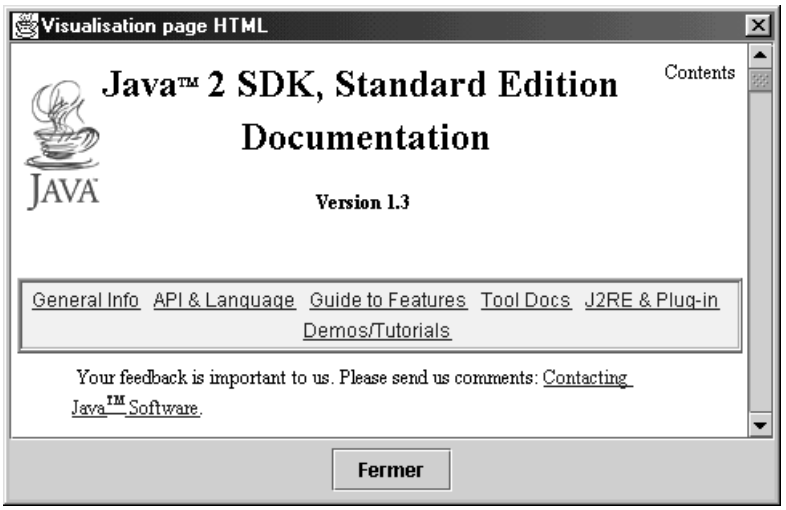


Figure 5.13 — Visualisation d'une page HTML dans un `JEditorPane`.

Pour afficher un contenu HTML dans un `JEditorPane`, il faut utiliser la méthode `setPage` en précisant l'URL en paramètre.

Voici le code complet de la classe `FenetreHtml` :

```
/**
 * Cette classe représente une fenêtre qui permet de
 * visualiser une page HTML.
 */
public class FenetreHtml extends JDialog {

 // Variables liées à l'IHM
 protected JFrame fenetrePpale;
 protected JScrollPane jspHtml;
 protected JPanel panBoutons;
 protected JButton btFermer;
 protected JEditorPane editPaneHtml;

 protected URL url;

 /**
  * Constructeur de la fenêtre de visualisation d'une
  * page HTML
  * @param fen la fenêtre principale à partir de laquelle
une
  * instance de FenetreHtml se lance
  * @param url l'URL à afficher dans l'EditorPane
  */
 public FenetreHtml(JFrame fen, URL url) {
   super(fen, "Visualisation page HTML", true);
   fenetrePpale = fen;
```

```
    this.url = url;
    initialiserIHM(url);
    afficherURL();
    gererEvenements();
  }

  /**
   * Initialisation de l'IHM
   */
  public void initialiserIHM(URL url) {
     setSize(new Dimension(500, 300));
     panBoutons = new JPanel();
     btFermer = new JButton();
     btFermer.setText("Fermer");
     editPaneHtml = new JEditorPane();
     jspHtml = new JScrollPane(editPaneHtml);
     panBoutons.add(btFermer, null);
     getContentPane().add(jspHtml, BorderLayout.CENTER);
     getContentPane().add(panBoutons, BorderLayout.SOUTH);
    }

    /**
     * Affichage de l'URL dans l'EditorPane
     */
  public void afficherURL() {
     editPaneHtml.setEditable(false);
     try {
        editPaneHtml.setPage(url);
     } catch(IOException e) {
        e.printStackTrace();
     }
    }

    /**
     * Gestion des événements : fermeture de la fenêtre et
     * clics sur liens hypertexte
     */
    public void gererEvenements() {
     // gestion de la fermeture de cette fenêtre
     this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
     btFermer.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
           dispose();
        }
     });
    }
  }
```

Notez que l'intérêt de la classe `JEditorPane` ne s'arrête pas à l'affichage de ces 3 formats : texte, HTML et RTF.

En effet, le mécanisme qui permet de supporter tel ou tel format ne se trouve pas dans la classe `JEditorPane` elle-même mais dans des classes nommées `EditorKit` dans des packages distincts :

- `javax.swing.text.html` pour le support du format HTML ;
- `javax.swing.text.rtf` pour le format RTF.

Ainsi, il est tout à fait possible de définir soi-même une implémentation d'un `EditorKit` pour supporter un autre format.

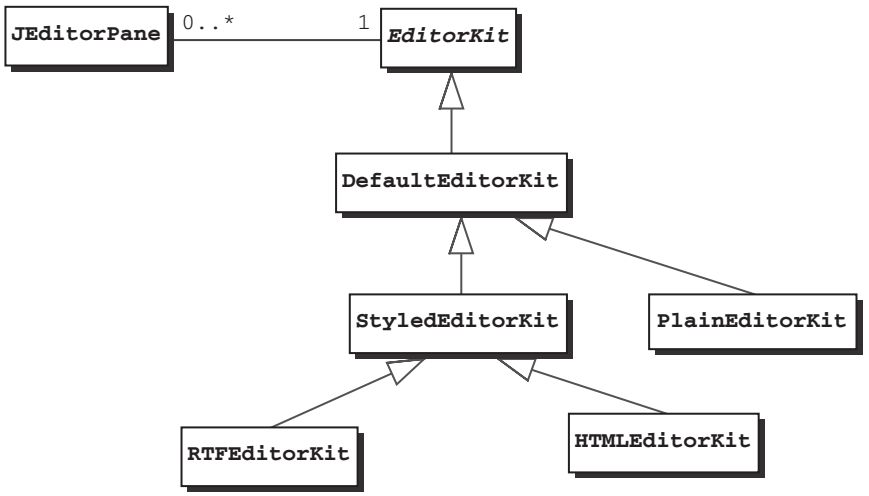


Figure 5.14 — Relation entre `JEditorPane` et `EditorKit`.

La classe `JEditorPane` possède une variable d'instance de type `EditorKit` (figure 5.14). Par défaut, cette variable est une instance de `PlainEditorKit`. La classe `PlainEditorKit` est une classe incluse de `JEditorPane` et elle est définie avec un accès *package*. Elle n'est donc pas utilisable à l'extérieur du package `javax.swing`.

Tous les composants texte utilisent un `EditorKit`, mais seuls `JEditorPane` et `JTextPane` permettent au programmeur d'accéder à cet objet.

Outre la méthode `setPage`, il existe deux autres façons d'ajouter du contenu à un `JEditorPane` :

- `setText`, qui prend en paramètre une chaîne de caractères ;
- `read`, qui prend en paramètre un flux de caractères.

Attention ! Quand on charge un contenu à partir d'une URL, le type de contenu de l'URL est utilisé comme format pour le `JEditorPane` (s'il est connu, bien sûr). En revanche, pour afficher du contenu à partir d'un flux de caractères, il faut positionner le format du `JEditorPane` pour qu'il corresponde à ce que l'on veut afficher.

Gestion des liens hypertexte

L'inconvénient de notre fenêtre pour visualiser une URL est de ne pas gérer les liens hypertexte. Certes, la visualisation d'un site Web sera toujours plus agréable dans un navigateur (barre de navigation, affichage de l'adresse…), mais il peut cependant être utile de permettre une navigation minimale. Pour cela, il faut gérer les événements de type « clic sur un lien hypertexte », autrement dit les `HyperlinkEvent`.

On ajoute un `HyperlinkListener` à notre `JEditorPane`. L'unique méthode à implémenter dans un `HyperlinkListener` est `hyperlinkUpdate`. Voici donc la portion de code à ajouter à la méthode `gererEvenements` de notre classe `FenetreHtml` :

```
// gestion des liens hypertexte
editPaneHtml.addHyperlinkListener( new HyperlinkListener()
{
   public void hyperlinkUpdate(HyperlinkEvent e) {
     if (e.getEventType() ==
        HyperlinkEvent.EventType.ACTIVATED) {
        try {
           editPaneHtml.setPage(e.getURL());
        } catch(IOException ex) {
           ex.printStackTrace();
        }
     }
   }
 });
```

Il est également possible de créer des composants `JEditorPane` éditables. Ils permettent de saisir du texte simple, du texte au format RTF, ou bien de l'HTML.

5.1.5 Une première utilisation de `JTextPane`

Ce composant apparaît comme le composant texte le plus riche. En effet, il hérite de `JEditorPane`, et offre des fonctionnalités supplémentaires.

Reprenons la copie d'écran présentée figure 5.2. Cette image présente tous les composants texte. Le `JTextPane` contient du texte avec différentes mises en forme ainsi qu'un composant `JButton` (figure 5.15). Comment cela a-t-il été fait ?

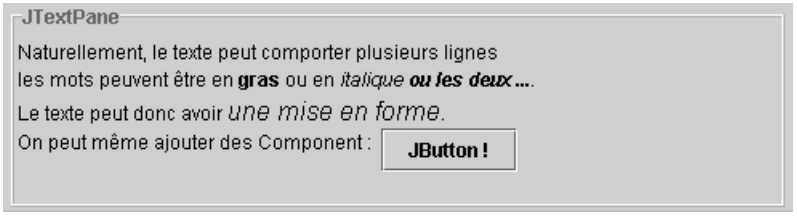


Figure 5.15 — `JTextPane` contenant un `JButton`.

Les apports de ce composant par rapport au précédent sont :

- chaque paragraphe peut être attaché à un style logique différent. Les styles sont définis en Java, et n'ont plus rien à voir avec des styles HTML ! Le texte affiché dans la copie d'écran ci-dessus n'est pas un texte HTML ;
- il est possible d'insérer des composants graphiques ou des images dans le texte grâce aux méthodes `insertComponent(Component c)` et `insertIcon(Icon i)`.

Pour réaliser le composant présenté figure 5.15, il faut tout d'abord définir différents styles.

Définir des styles

Un style est encapsulé dans un objet implémentant l'interface `Style`. Les styles forment une arborescence à parent unique et se définissent par rapport à un parent. Il faut donc partir d'une racine qui est le style par défaut :

```
// Obtention du style par défaut
Style def = StyleContext.getDefaultStyleContext()
➥.getStyle(StyleContext.DEFAULT_STYLE);
```

Ensuite, on modifie une caractéristique du style par défaut : on indique que la police doit être de la famille « SansSerif ». D'un point de vue typographique, les polices « SansSerif » sont plus lisibles car elles sont dépourvues de « barres » au pied de certaines lettres, comme le « m » par exemple.

```
// Modification du style par défaut
StyleConstants.setFontFamily(def, "SansSerif");
```

Les autres styles sont définis à partir de ce style par défaut. On les ajoute au composant `JTextPane`, qui est ici représenté par la variable `tp`. La création se fait en même temps que l'ajout au composant texte grâce à la méthode `addStyle`.

On crée tout d'abord un style nommé `regular`, qui est équivalent au style par défaut. Ensuite, on crée un style nommé `italic`, qui correspond au style `regular`, auquel on ajoute la caractéristique italique, et ainsi de suite.

Remarquez qu'on réutilise la variable `s` de type `Style` : on affecte une valeur à cette variable grâce à `addStyle`. Ensuite, on modifie une ou plusieurs caractéristiques.

```
// Création d'un nouveau style : regular
Style regular = tp.addStyle("regular", def);
// Création des styles dérivés de regular
Style s = tp.addStyle("italic", regular);
StyleConstants.setItalic(s, true);
s = tp.addStyle("bold", regular);
StyleConstants.setBold(s, true);
s = tp.addStyle("deux", regular);
StyleConstants.setBold(s, true);
StyleConstants.setItalic(s, true);
s = tp.addStyle("small", regular);
StyleConstants.setFontSize(s, 16);
StyleConstants.setItalic(s, true);
```

Pour l'ajout du bouton dans le composant texte, on crée un style spécifique, qu'on a choisi d'appeler `bouton`. Grâce à la méthode `setComponent`, il est possible d'ajouter n'importe quel type de composant :

```
Style bouton = tp.addStyle("bouton", def);
StyleConstants.setComponent(bouton, new JButton
➥("JButton !"));
```

Créer un contenu formaté

Pour ajouter des portions de texte avec un style particulier, il faut utiliser la méthode `insertString` de la classe `Document`. Les documents seront détaillés plus loin dans ce chapitre. Ici, nous avons juste besoin de créer une instance de `Document` pour pouvoir ajouter du texte avec des styles donnés. La classe à utiliser est `DefaultStyledDocument` :

```
DefaultStyledDocument doc = new DefaultStyledDocument();
```

Les insertions de texte se font ensuite sur cet objet `doc`. Le dernier paramètre de la méthode `insertString` correspond au style. Il est obtenu par son nom, grâce à la méthode `getStyle` sur l'objet `JTextPane`.

```
try {
 doc.insertString(doc.getLength(),
    "Naturellement, le texte peut comporter plusieurs
    + "lignes"
    + "\nles mots peuvent être en ",
    tp.getStyle("regular"));
```

```
            doc.insertString(doc.getLength(), "gras",
              tp.getStyle("bold"));
            doc.insertString(doc.getLength(), " ou en ",
              tp.getStyle("regular"));
            doc.insertString(doc.getLength(), "italique",
              tp.getStyle("italic"));
            doc.insertString(doc.getLength(), " ou les deux…",
              tp.getStyle("deux"));
            doc.insertString(doc.getLength(),
              ".\nLe texte peut donc avoir ",
              tp.getStyle("regular"));
            doc.insertString(doc.getLength(), "une mise en forme.",
              tp.getStyle("small"));
            doc.insertString(doc.getLength(),
              "\nOn peut même ajouter des Component :  ",
              tp.getStyle("regular"));

            // La petite " ruse " ici est de n'insérer aucun texte,
            // ou du moins un espace seul car le style associé
            // contient déjà un Component que l'on souhaite juste
            // afficher.
            doc.insertString(doc.getLength(), " ",
              tp.getStyle("bouton"));

        // Attention à l'exception que peut lever la méthode
        // insertString si la position d'insertion est incorrecte.
        } catch (BadLocationException e) { [ traitement de
        ➥l'exception ] }
```

Associer le contenu au composant texte

La dernière étape de cet exemple consiste à associer ce contenu porté par l'objet `doc` au composant graphique. Cela se fait grâce à la méthode `setDocument`.

```
tp.setDocument(doc);
```

Récapitulatif sur les `EditorKit`

Dans notre exemple, nous n'avons jamais eu à nous soucier de l'objet `EditorKit`. En effet, `JTextPane` utilise par défaut une instance de `StyledEditorKit`.

Toutefois, il faut être conscient que nous aurions pu faire exactement comme pour le `JEditorPane`, c'est-à-dire associer un `HTMLEditorKit` et afficher du texte au format HTML.

Tableau 5.2 — Les `EditorKit` possibles.

Classes graphiques	`EditorKit` par défaut	Autres `EditorKit` possibles
`JTextField`	`DefaultEditorKit`	Aucun
`JPasswordField`	`DefaultEditorKit`	Aucun
`JTextArea`	`DefaultEditorKit`	Aucun
`JEditorPane`	`PlainEditorKit`	`HTMLEditorKit` ou `RTFEditorKit`
`JTextPane`	`StyledEditorKit`	Sous-classes de `StyledEditorKit`

5.2 PERSONNALISER LES COMPOSANTS TEXTE AVEC LES DOCUMENTS

5.2.1 Redéfinir un document

Reprenons notre petit exemple de formulaire de saisie pour des petites annonces de vente de matériel d'occasion.

La description de l'objet à vendre doit être brève. Nous souhaitons limiter cette zone de texte à 200 caractères. Lorsque l'utilisateur essaiera de saisir un 201e caractère, l'application interdira la saisie et émettra un signal sonore (figure 5.16).

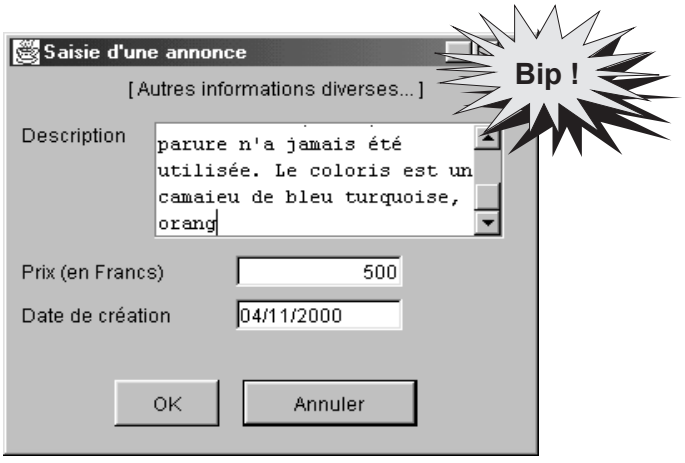


Figure 5.16 — Composant texte de taille limitée.

Nous devons donc surveiller le « contenu » du composant texte. Il est inutile de placer un listener sur le composant texte : il suffit de s'adresser à l'objet qui

gère le contenu. Dans la terminologie Swing, cet objet est appelé un *Document*. Quand l'utilisateur saisit un caractère au clavier, le document est mis à jour (figure 5.17).

Le document correspond au *modèle* dans la terminologie de *l'architecture* MVC.

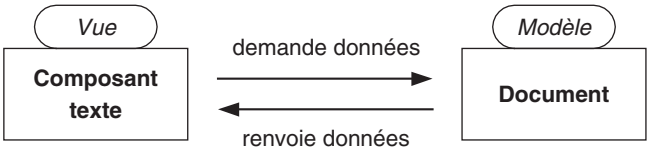


Figure 5.17 — Le rôle du document.

Pour notre exemple, nous devons associer un document particulier à la zone de saisie et lorsque le document sera informé que l'utilisateur insère du texte, on vérifiera que la taille maximale n'est pas atteinte.

L'interface `Document` nous renseigne sur les méthodes qui vont nous être utiles. Les méthodes `remove` et `insertString` permettent l'édition du texte.

Par ailleurs, un document propose des méthodes d'abonnement et de désabonnement pour des listeners qui seront avertis des modifications apportées au texte. Il permet également d'obtenir des informations sur le texte : taille, extraction d'une sous-chaîne, etc.

Regardons la hiérarchie des classes `Document` pour savoir de laquelle hériter (figure 5.18).

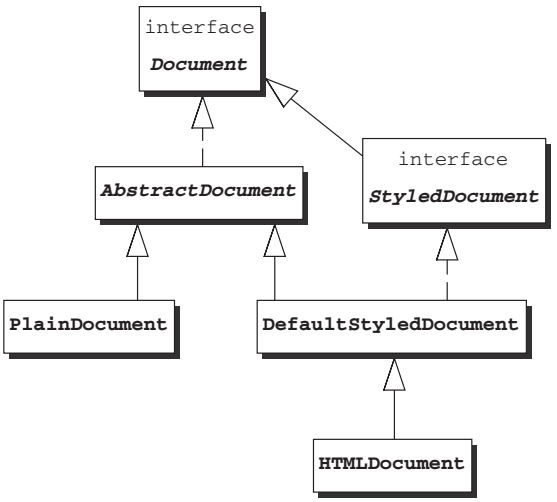


Figure 5.18 — Hiérarchie des documents.

Le document utilisé par défaut pour les `JTextField`, `JPasswordField`, `JTextArea` ainsi que pour `JEditorPane` est une instance de la classe `PlainDocument`. En revanche, un `JTextPane` utilise obligatoirement un `StyledDocument`.

Nous choisissons donc d'hériter de `PlainDocument` et de redéfinir la méthode `insertString` pour faire la vérification adéquate.

```
/**
 * Classe représentant un document de taille limitée
 */
public class DocumentTailleLimitee extends PlainDocument{

 int maxCarac;
 /**
  * Constructeur prenant en paramètre le nombre max de
  * caractères.
  */
 public DocumentTailleLimitee(int max) {
    maxCarac = max;
 }

 /**
  * Vérifie que la chaîne à insérer + le texte déjà présent
  * ne font pas plus de maxCarac caractères.
  */
 public void insertString(int offs, String str,
  ➥AttributeSet a)
  throws BadLocationException {
  if ((getLength() + str.length()) <= maxCarac) {
    super.insertString(offs, str, a);
  }
  else {
    Toolkit.getDefaultToolkit().beep();
  }
 }
}
```

Pour associer ce document au composant, on peut le passer en paramètre du constructeur, ou bien utiliser la méthode `setDocument` (qui est définie au niveau de `JTextComponent`) :

```
taDescription.setDocument(new DocumentTailleLimitee(200));
```

5.2.2 Être à l'écoute des changements

Il manque un détail dans notre saisie de petites annonces : les frais de livraison. Le vendeur doit les préciser dans un champ de saisie spécifique. Le prix du produit incluant la livraison s'affichera automatiquement.

Il faut donc créer un champ de texte non éditable qui affichera la somme des champs « Prix » et « Frais de livraison » (figure 5.19).

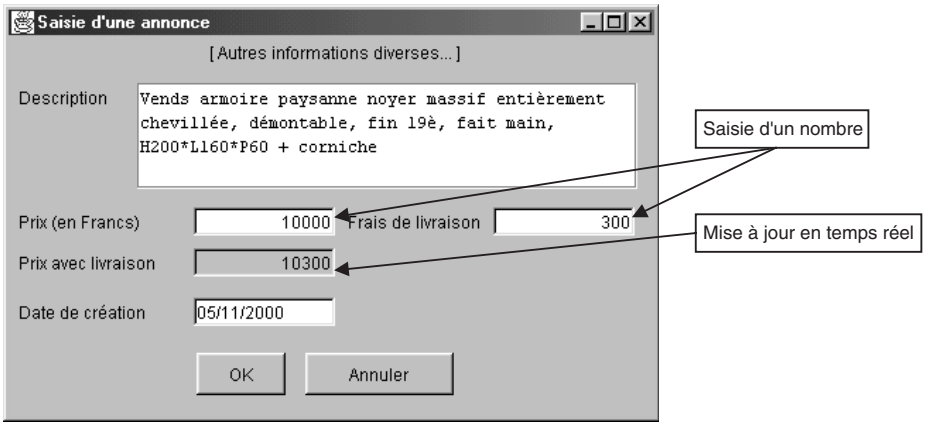


Figure 5.19 — Somme qui s'affiche en temps réel.

Pour calculer la somme des deux nombres saisis, il faut d'abord s'assurer qu'il s'agit bien de nombres. La méthode `parseInt` de la classe `Integer` permet d'obtenir un entier à partir d'une chaîne de caractères. Mais, dans notre exemple, lorsque l'utilisateur n'a rien saisi dans le champ « Frais de livraison », il faut que ce soit considéré comme le nombre 0 : le prix avec livraison est alors égal au prix du produit. Une solution satisfaisante pour gérer cette contrainte consiste à écrire une sous-classe de `JTextField`, qui possède deux méthodes supplémentaires : `getEntier()` et `setEntier(int x)`.

```
/**
 * Composant texte contenant des entiers
 */
public class TextFieldEntier extends JTextField {
 /**
  * Renvoie l'entier 0 si la chaîne est vide ou nulle,
  * sinon, renvoie l'entier représenté par la chaîne
  */
 public int getEntier() throws NumberFormatException {
   String s = getText();
   if (s==null || s.length()==0) {
     return 0;
   }
   else {
     return Integer.parseInt(s);
   }
 }
  /**
```

```
    * Remplit le champ de texte avec l'entier passé en
    * paramètre
    */
    public void setEntier(int x) {
       setText(String.valueOf(x));
       repaint();
    }
}
```

À l'inverse de l'exemple précédent, nous n'avons pas besoin de redéfinir un document, nous pouvons utiliser les instances de documents affectées par défaut aux composants texte. En effet, nous n'avons pas besoin de modifier le contenu des composants texte éditables, mais juste d'être averti lors des changements.

Nous souhaitons mettre à jour la zone « Prix avec livraison » à chaque fois que les deux champs « Prix » et « Frais de livraison » sont édités. Pour cela, créons un `DocumentListener` qui sera à l'écoute des changements sur les deux premiers `JTextField` et qui mettra à jour le troisième.

Attention ! Lorsque vous souhaitez « modifier » le contenu d'un composant texte en temps réel (empêcher l'insertion de caractères au-delà de 200 par exemple), vous devez travailler sur le document lui-même. Un listener n'est pas suffisant, car il reçoit une notification **après que** le changement a eu lieu. Il faudrait alors être capable de revenir en arrière si les derniers caractères saisis ne respectaient pas la contrainte (plus de 200 caractères par exemple). Or, nous verrons plus loin que la gestion de l'annulation est bien plus complexe…

Trois méthodes sont à implémenter pour définir un `DocumentListener` :

```
/**
 * Listener qui met à jour un champ de texte, avec la somme
 * des entiers contenus dans 2 autres champs de texte.
 */
public class ListenerSomme implements DocumentListener {
 // Variables d'instances
 TextFieldEntier t1;
 TextFieldEntier t2;
 TextFieldEntier tSomme;
 /**
  * Constructeur prenant en paramètres les
  * 3 TextFieldEntier concernés.
  */
 public ListenerSomme(TextFieldEntier t1,
 ➥TextFieldEntier t2,
  TextFieldEntier tSomme) {
  this.t1 = t1;
  this.t2 = t2;
  this.tSomme = tSomme;
 }
```

```
    // appelée lorsqu'un événement d'insertion a lieu
    public void insertUpdate(DocumentEvent e) {
      additionner(e);
    }
    // appelée lorsqu'un événement de suppression a lieu
    public void removeUpdate(DocumentEvent e) {
      additionner(e);
    }
    // appelée lorsqu'un événement de changement de style
    // a lieu
    public void changedUpdate(DocumentEvent e) {
    }
    /**
     * méthode qui effectue l'addition et la mise à jour
     * du champ
     */
    private void additionner(DocumentEvent e) {
     try {
       tSomme.setEntier(t1.getEntier()+t2.getEntier());
     } catch (NumberFormatException ex) {
       JOptionPane.showMessageDialog(Fenetre.this,
       "Format des prix incorrect", "Attention",
       JOptionPane.ERROR_MESSAGE);
     }
    }
}
```

Les méthodes `insertUpdate` et `removeUpdate` signalent des ajouts ou des retraits de caractères dans les champs. Elles appellent donc la méthode `additionner`, qui effectue la mise à jour. La méthode `changedUpdate` en revanche ne fait rien, car elle est appelée seulement lorsqu'un changement de style a lieu.

Remarque : observez le traitement de l'exception `NumberFormatException`. Nous souhaitons afficher une boîte de dialogue pour avertir l'utilisateur. Pour cela, on utilise la méthode `showMessageDialog`, qui prend comme premier paramètre la fenêtre propriétaire. Nous sommes dans une classe incluse de `Fenetre`. Pour désigner l'instance courante de la classe englobante, il faut utiliser la syntaxe `<Nom de la classe>.this`.

Pour ajouter le listener aux composants, on obtient le document par défaut à l'aide de `getDocument`, puis on effectue l'abonnement du listener :

```
// association du listener aux champs de saisie
listenerPrix = new ListenerSomme(tfPrix, tfLivraison,
➥tfPrixTotal);
tfPrix.getDocument().addDocumentListener(listenerPrix);
tfLivraison.getDocument().addDocumentListener
➥(listenerPrix);
```

Attention ! Ici, on ne s'est pas servi de l'objet `DocumentEvent`. Si vous avez besoin de le manipuler, il est intéressant de noter qu'il n'hérite pas de `EventObject`. La méthode `getSource` est remplacée par `getDocument`.

5.2.3 Le fonctionnement interne d'un document

Un document encapsule le contenu du champ de texte. Il utilise pour cela des objets de type `Element` (c'est-à-dire implémentant l'interface `Element`) organisés en arborescence : un élément racine représente tout le contenu, des éléments fils découpent le texte en sous-parties. À chaque élément, un ensemble d'attributs peut être appliqué : police, couleur, gras, souligné, etc. Cet ensemble d'attributs est encapsulé dans un objet implémentant l'interface `AttributeSet`.

Cette organisation avec une arborescence d'éléments permet de gérer le fait qu'une portion de texte est en gras, une autre de telle couleur, et ainsi de suite.

Un document manipule également des objets `Position`. Par défaut, deux références vers des objets `Position` sont accessibles, il s'agit du début et de la fin d'un document. On peut définir de nouvelles positions manuellement. Une position correspond à un point entre deux caractères précis. Si du texte est inséré avant cette position, la position est décalée pour être toujours entre les deux mêmes caractères.

La classe `AbstractDocument` spécifie un mécanisme qui sépare le stockage des données (le texte) et sa structuration (le style, la disposition en paragraphes…). Le texte proprement dit est stocké dans un objet qui implémente l'interface `Content`.

Ces notions sont importantes si vous souhaitez implémenter vous-même un document avec un mécanisme de stockage particulier.

5.3 UTILISATION DES ACTIONS

Dans ce paragraphe, nous allons voir comment utiliser ces objets si utiles que sont les actions, et nous verrons l'usage intensif qu'en font les composants texte.

5.3.1 Qu'est-ce qu'une action ?

Il vous est probablement déjà arrivé de vous dire : « Ce bouton et ce menu font exactement la même chose : je vais leur associer le même listener. » Allons plus loin, associons-leur directement la même action ! En plus d'avoir le même listener, le menu et le bouton auront le même texte, la même icône s'il y en a une. Leurs états (activable ou non) seront toujours en accord.

Une première action

Nous voulons créer un bouton et un item de menu qui partagent la même action. L'effet de cette action consiste uniquement à ouvrir une fenêtre de dialogue avec un message d'information (figure 5.20).

Figure 5.20 — Un bouton et un menu partageant la même action.

La création de l'action

Une action est en fait un `ActionListener` particulier, plus commode à utiliser, qui réduit le temps nécessaire à l'implémentation… Pour aboutir à l'interface décrite figure 5.20, nous pourrions écrire une classe qui implémente `ActionListener`. Il n'y a qu'une chose à changer pour en faire une action : hériter de `AbstractAction` au lieu d'implémenter `ActionListener` (figure 5.21).

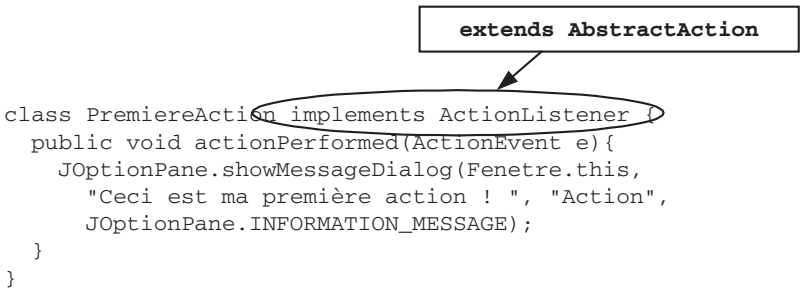


Figure 5.21 — Création d'une classe `Action`.

La classe `AbstractAction` implémente l'interface `Action`, qui elle-même hérite de `ActionListener`. La seule méthode qu'il est obligatoire de définir pour obtenir une classe instanciable est la méthode `actionPerformed`.

L'association aux composants graphiques

L'association de l'action aux composants graphiques ne se fait plus par la méthode `addActionListener`, comme pour les listeners habituels, mais par la méthode `setAction`.

Il est aussi possible de passer l'action en paramètre du constructeur :

```
btAction = new JButton();
btAction.setAction(action);
menuAction = new JMenuItem (action);
```

Attention ! Depuis le JDK1.3, il est recommandé de créer le composant graphique (bouton, item de menu), de lui associer l'action (par le biais du constructeur ou de la méthode `setAction`), puis d'ajouter le composant au menu ou à la barre d'outils. Les méthodes `add(Action a)` sur un menu ou une barre d'outils, qui créaient elles-mêmes le composant graphique adéquat, ne sont plus préconisées…

Les propriétés d'une action

Une action dispose d'un dictionnaire de propriétés qui sont identifiables grâce à des chaînes de caractères. Ces propriétés sont par exemple :

- le nom : chaîne de caractères utilisée comme texte à afficher sur un bouton ou un item de menu ;
- une icône : également utilisé pour paramétrer les composants graphiques représentant l'action ;
- une description courte : utilisée pour le texte du *tooltip* ;
- un raccourci clavier ;
- une description longue ;
- la possibilité d'ajouter d'autres caractéristiques…

Ces différentes propriétés ne sont pas stockées dans des variables d'instance pour une raison simple : `Action` est une interface, or une interface ne peut pas spécifier de variables d'instances. L'interface `Action` possède donc deux méthodes `putValue` et `getValue`, pour ajouter des propriétés dans un dictionnaire, ainsi que des constantes (variables de classe) correspondant aux clés des propriétés dans le dictionnaire.

Les composants graphiques reçoivent une notification lorsque les propriétés de l'action changent, et ils mettent à jour leur état.

Nous pouvons tester cela en complétant l'exemple ci-dessus. Ajoutons des boutons qui permettent de paramétrer l'action : le bouton et le menu sont mis à jour (figure 5.22).

Le bouton « Appliquer » met à jour les propriétés de l'action. L'état disponible ou non d'une action (`enabled` ou `disabled`) n'est pas une propriété du dictionnaire.

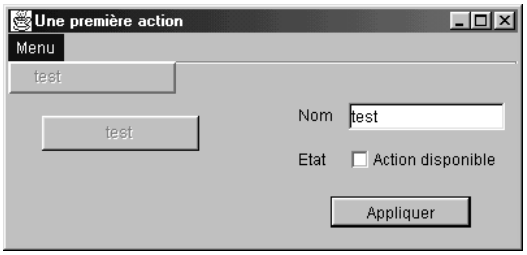


Figure 5.22 — Mise à jour des composants graphiques associés à l'action.

On la met à jour à l'aide de la méthode `setEnabled` :

```
btAppliquer.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e){
      action.putValue(Action.NAME, tfNom.getText());
      if (cocheSelectionnable.isSelected()!=action
      ➥.isEnabled()) {
      action.setEnabled(cocheSelectionnable.isSelected());
      }
      repaint();
   }
});
```

La mise à jour des composants graphiques se fait automatiquement parce que les propriétés de l'action sont ce qu'on appelle des propriétés liées : le changement d'une propriété déclenche un `PropertyChangeEvent` (voir chapitre 3). Or les composants graphiques sont eux aussi abonnés à l'action. Ils sont avertis lorsqu'une propriété liée change de valeur (figure 5.23).

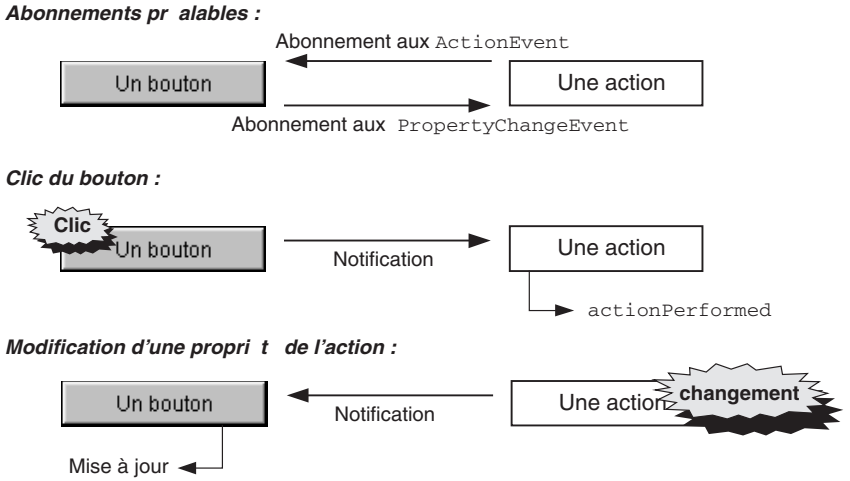


Figure 5.23 — Fonctionnement d'une action.

Les actions sont donc particulièrement utilisées pour gérer des fonctionnalités disponibles *via* différents composants graphiques : un bouton dans une barre d'outil, un item d'une barre de menus, un item d'un menu contextuel, un raccourci clavier, etc.

Le fait qu'on puisse ajouter toute sorte de propriétés à une action permet d'obtenir facilement des interfaces riches et agréables : on peut associer à une action des icônes de taille plus réduite pour avoir des raccourcis supplémentaires dans une barre d'état, ou bien un texte plus détaillé qui s'affiche dans une fenêtre d'aide contextuelle, etc.

Le design pattern Commande

Le principe des actions n'est pas quelque chose de nouveau dans la technologie Objet. En effet, un design pattern bien connu décrit à peu près ce fonctionnement : c'est le design pattern Commande.

Définition : un design pattern, ou *modèle de conception*, décrit une solution satisfaisante d'un point de vue objet (évolutive, réutilisable, intelligible), à un problème général de conception objet.

Le problème pour lequel le pattern Commande offre une solution est le suivant : certains objets doivent envoyer l'ordre d'exécuter une action à d'autres objets. Les objets qui déclenchent l'ordre ne savent pas en quoi consiste cette action, pas plus qu'ils ne connaissent les objets récepteurs de l'ordre.

Il s'agit typiquement des composants graphiques (boutons, items de menu) qui doivent déclencher une opération sur un autre objet qu'ils ne connaissent pas (par exemple mettre du texte en italique dans un composant texte).

L'ordre d'exécuter une opération est ici appelé Commande. Toutes les commandes héritent d'une classe abstraite qui spécifie une méthode `execute` (figure 5.24). C'est l'équivalent de l'`ActionListener` et sa méthode `actionPerformed`.

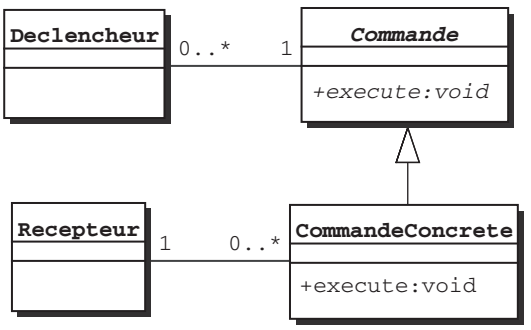


Figure 5.24 — Le design pattern Commande.

Dans la description du pattern Commande, il est indiqué que l'application qui crée une action positionne également son récepteur, par exemple en stockant le récepteur comme variable d'instance de l'action. Cette spécificité n'est pas vérifiée dans le modèle `Action` de Swing.

Pour savoir sur quel objet récepteur appliquer l'opération, la méthode `actionPerformed` de l'action procède de la façon suivante : elle recherche le composant source de l'événement (ceci est possible puisque l'événement est passé en paramètre de `actionPerformed`). S'il s'agit d'un composant texte, il est considéré comme l'objet récepteur. Sinon, l'objet récepteur est le dernier composant texte à avoir eu le focus.

Voici la méthode de `TextAction` qui effectue cette recherche :

```
protected final JTextComponent getTextComponent
➥(ActionEvent e) {
   if (e != null) {
      Object o = e.getSource();
      if (o instanceof JTextComponent) {
         return (JTextComponent) o;
      }
   }
   return getFocusedComponent();
}
```

Finalement, on peut considérer que le modèle `Action` constitue une variante d'implémentation du pattern Commande. Ce modèle présente plusieurs avantages : les objets `Action` sont complètement indépendants des objets déclencheurs, et plusieurs déclencheurs peuvent se partager la même instance d'une classe `Action`. Les actions peuvent être stockées dans des files d'attente, puis être exécutées ultérieurement. Elles peuvent être mémorisées pour être appliquées à nouveau, ou bien pour être exécutées au sein d'une transaction. Elles peuvent également être assemblées pour constituer des scénarios complets. Et enfin, elles permettent de gérer facilement l'annulation (*undo*) pour peu qu'elles possèdent une méthode de réversion.

5.3.2 `EditorKit` et les actions standard

Création des actions standard dans l'exemple de gestion des signets

Tous les composants texte supportent les commandes d'édition standard comme Couper, Copier, Coller, etc. Ces commandes sont implémentées par des objets `Action`.

Il est possible d'obtenir la liste des actions supportées par un composant :

```
Action[] tabActions = comp.getActions() ;
```

Reprenons notre exemple de gestion de signets. Nous souhaitons fournir une fenêtre d'édition pour saisir la description d'un signet. Cette fenêtre d'édition devra fournir toutes les fonctionnalités habituelles : Couper, Copier, Coller, ainsi que la possibilité de mettre du texte en gras, en italique ou souligné (figure 5.25).

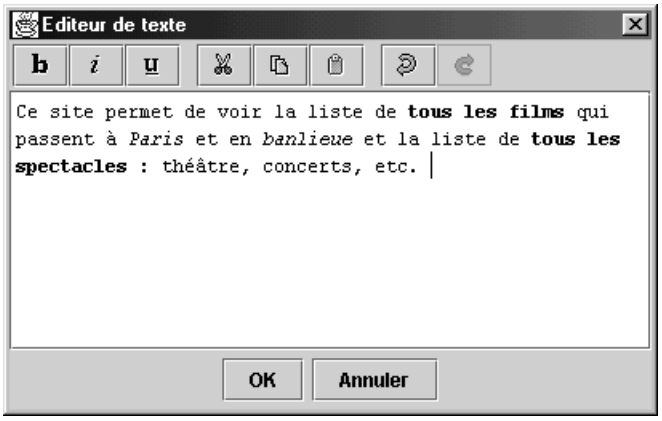


Figure 5.25 — Fenêtre d'édition pour la description d'un signet.

Pour obtenir ces actions standard, nous allons utiliser un objet `EditorKit`. L'`EditorKit` utilise des classes incluses qui sont des sous-classes de `TextAction` :

```
/**
 * Initialisation des actions
 */
public void initialiserActions() {
 // Gestion du style
 actionGras = new StyledEditorKit.BoldAction();
 actionGras.putValue(Action.NAME, GRAS_TEXT);
 URL imageUrl = ClassLoader.getSystemResource(
 ➥ICONE_TEXT_PATH+"Bold16.gif");
 actionGras.putValue(Action.SMALL_ICON,
 ➥new ImageIcon(imageUrl));

 actionItalique = new StyledEditorKit.ItalicAction();
 actionItalique.putValue(Action.NAME, ITAL_TEXT);
 imageUrl =  ClassLoader.getSystemResource(
 ➥ICONE_TEXT_PATH+"Italic16.gif");
 actionItalique.putValue(Action.SMALL_ICON, new
 ➥ImageIcon(imageUrl));
```

```
        actionSouligne = new StyledEditorKit.UnderlineAction();
        actionSouligne.putValue(Action.NAME, SOULIGNE_TEXT);
        imageUrl =  ClassLoader.getSystemResource(
        ➥ICONE_TEXT_PATH +"Underline16.gif");
        actionSouligne.putValue(Action.SMALL_ICON, new
         ImageIcon(imageUrl));

        // Gestion du "couper-copier-coller"
        actionCouper = new DefaultEditorKit.CutAction();
        actionCouper.putValue(Action.NAME, COUPER_TEXT);
        imageUrl = ClassLoader.getSystemResource(
        ➥ICONE_GENERAL_PATH+"Cut16.gif");
        actionCouper.putValue(Action.SMALL_ICON, new ImageIcon
        ➥(imageUrl));

        actionCopier = new DefaultEditorKit.CopyAction();
        actionCopier.putValue(Action.NAME, COPIER_TEXT);
        imageUrl = ClassLoader.getSystemResource(
        ➥ICONE_GENERAL_PATH+"Copy16.gif");
        actionCopier.putValue(Action.SMALL_ICON, new ImageIcon
        ➥(imageUrl));

        actionColler = new DefaultEditorKit.PasteAction();
        actionColler.putValue(Action.NAME, COLLER_TEXT);
        imageUrl = ClassLoader.getSystemResource(
        ➥ICONE_GENERAL_PATH+"Paste16.gif");
        actionColler.putValue(Action.SMALL_ICON, new
         ImageIcon(imageUrl));
    }
```

Association de ces actions aux composants graphiques

Nous avons choisi d'associer toutes ces actions à des boutons de la barre d'outils, et certaines d'entre elles à des items de menu contextuel. Les actions concernant la mise en forme du texte n'apparaissent que dans la barre d'outils.

Voici le code de création de la barre d'outils. Cette création se divise en plusieurs étapes : instanciation des boutons, paramétrage de ces boutons (le texte, la bordure, etc.), puis ajout à la barre d'outils :

```
    /**
     * Création de la barre d'outils
     */
    protected void creerBarreOutils() {
       // Création des boutons
       boutonGras = new JToggleButton(actionGras);
       boutonItalique = new JToggleButton(actionItalique);
       boutonSouligne = new JToggleButton(actionSouligne);
```

```
        boutonCouper = new JButton(actionCouper);
        boutonCopier = new JButton(actionCopier);
        boutonColler = new JButton(actionColler);
        boutonUndo = new JButton(actionUndo);
        boutonRedo = new JButton(actionRedo);

        // Pas de texte sur les boutons (seulement l'icône)
        boutonGras.setText("");
        boutonItalique.setText("");
        boutonSouligne.setText("");
        boutonCouper.setText("");
        boutonCopier.setText("");
        boutonColler.setText("");
        boutonUndo.setText("");
        boutonRedo.setText("");

        // Ajout d'un espace entre l'icône et le bord du bouton
        boutonGras.setMargin(new Insets(2,6,2,6));
        boutonItalique.setMargin(new Insets(2,6,2,6));
        boutonSouligne.setMargin(new Insets(2,6,2,6));
        boutonCouper.setMargin(new Insets(2,6,2,6));
        boutonCopier.setMargin(new Insets(2,6,2,6));
        boutonColler.setMargin(new Insets(2,6,2,6));
        boutonUndo.setMargin(new Insets(2,6,2,6));
        boutonRedo.setMargin(new Insets(2,6,2,6));

        // ajout des boutons dans la barre
        toolBar.add(boutonGras);
        toolBar.add(boutonItalique);
        toolBar.add(boutonSouligne);
        toolBar.addSeparator();
        toolBar.add(boutonCouper);
        toolBar.add(boutonCopier);
        toolBar.add(boutonColler);
        toolBar.addSeparator();
        toolBar.add(boutonUndo);
        toolBar.add(boutonRedo);
    }
```

La création du menu contextuel demande un peu plus de réflexion : la création du menu lui-même se fait à l'aide d'objets `JPopupMenu`, `JMenu` et `JMenuItem` emboîtés. Ensuite, nous devons gérer le fait que le menu apparaisse lors d'une action particulière de l'utilisateur. Cette action particulière pour afficher le menu contextuel est différente selon les plates-formes. Il s'agit par exemple d'un clic du bouton droit de la souris sous Windows. Dans la suite de ce paragraphe, nous parlerons de « clic droit » au sens large, pour désigner cette action qui est dépendante de la plate-forme.

Pour gérer l'apparition du menu, il est nécessaire d'écrire un `MouseListener`. Il n'existe pas de méthode spécifique de l'événement « clic droit ». Nous allons traiter les événements de type « bouton pressé » et « bouton relâché ». Lorsque l'un de ces événements survient, les méthodes `mousePressed` et `mouseReleased` du `MouseListener` sont respectivement appelées. L'objet `MouseEvent` est passé en paramètre de ces méthodes. Cet objet nous permet de déterminer s'il s'agit d'un clic ordinaire ou d'un clic droit. En effet, la classe `MouseEvent` propose une méthode `isPopupTrigger`, qui renvoie vrai lorsque le clic correspond au geste particulier qui permet l'affichage du menu contextuel.

```
/**
 * Création du menu contextuel
 */
protected void creerMenuContextuel() {
   mCouper = new JMenuItem(actionCouper);
   mCopier = new JMenuItem(actionCopier);
   mColler = new JMenuItem(actionColler);
   mUndo = new JMenuItem(actionUndo);
   mRedo = new JMenuItem(actionRedo);
   mEdition = new JMenu("Edition");
   mAnnulation = new JMenu("Annulation");
   popup = new JPopupMenu();
   mEdition.add(mCouper);
   mEdition.add(mCopier);
   mEdition.add(mColler);
   mAnnulation.add(mUndo);
   mAnnulation.add(mRedo);
   popup.add(mEdition);
   popup.add(mAnnulation);
   MouseListener popupListener = new PopupListener();
   textPane.addMouseListener(popupListener);
 }
/**
 * Classe incluse : mouse listener qui fait apparaître le
 * menu contextuel.
 */
class PopupListener extends MouseAdapter {
   public void mousePressed(MouseEvent e) {
      afficherPopup(e);
   }
 public void mouseReleased(MouseEvent e) {
      afficherPopup(e);
   }
 private void afficherPopup(MouseEvent e) {
      if (e.isPopupTrigger()) {
         popup.show(e.getComponent(), e.getX(), e.getY());
      }
 }
}
```

Récapitulatif des actions standard

La classe `DefaultEditorKit` définit une série de sous-classes de `TextAction`, qui sont identifiables par des chaînes de caractères constantes correspondant aux noms de ces actions. Par exemple, l'action *page down* (descendre d'une page) est identifiable par la chaîne constante :

```
public static final String pageDownAction = "page-down";
```

Parmi les quarante-six actions implémentées dans la classe `DefaultEditorKit`, huit sont des classes incluses publiques et peuvent être instanciées :

- `CopyAction`
- `CutAction`
- `PasteAction`
- `BeepAction`
- `DefaultKeyTypedAction`
- `InsertBreakAction`
- `InsertContentAction`
- `InsertTabAction`

La classe `StyledEditorKit` suit exactement le même fonctionnement. Elle propose des actions supplémentaires, spécifiques des composants avec du texte mis en forme, implémentées dans des classes incluses. Ce sont :

- `AlignmentAction`
- `BoldAction`
- `FontFamilyAction`
- `FontSizeAction`
- `ForegroundAction`
- `ItalicAction`
- `UnderLineAction`

Les autres actions implémentées dans des classes incluses privées peuvent être utilisées en demandant une instance à l'objet `EditorKit`. Une façon courante d'utiliser ces actions standard consiste à créer un dictionnaire des actions, chaque action pouvant être obtenue par son nom :

```
// construction d'un dictionnaire des actions standard
HashMap dico = new HashMap();
Action[] tabact =
➥tpDescription.getEditorKit().getActions();
```

```
for (int i=0; i<tabact.length; i++) {
dico.put(tabact[i].getValue(Action.NAME), tabact[i]);
}
```

Ensuite, l'obtention de l'action Couper par exemple se fait de la façon suivante :

```
Action couper = (Action) dico.get(
➥DefaultEditorKit.cutAction) ;
```

Voici figure 5.26 la hiérarchie de quelques-unes des classes `Action` standard.

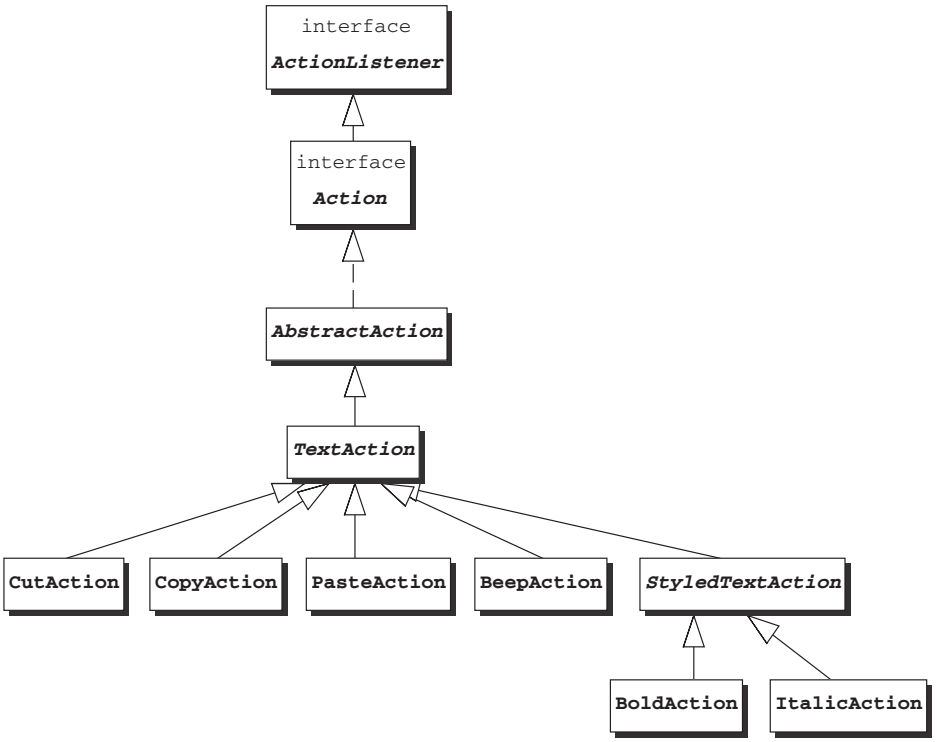


Figure 5.26 — Hiérarchie de quelques actions texte.

5.3.3 Les raccourcis clavier

Le fait de pouvoir invoquer une action à l'aide d'un raccourci clavier est une fonctionnalité très utile, qui permet un gain de temps lors de la saisie, en particulier pour les utilisateurs qui utilisent le clavier avec dextérité.

Ajout d'un raccourci

Reprenons notre exemple de formulaire de saisie pour une petite annonce. Nous avons constaté que l'expression « Bon état général » revenait très souvent dans

la description de l'objet à vendre. Proposons à l'utilisateur un raccourci (appui simultané sur les touches Control et B) pour afficher automatiquement ces trois mots.

Nous allons tout d'abord créer l'action que l'on souhaite déclencher par les touches Ctrl-B.

Pour cela, nous utilisons la méthode `getTextComponent` proposée par la classe `TextAction` qui permet d'identifier le composant auquel on souhaite appliquer un traitement. Ensuite, nous aurions pu accéder au document grâce à la méthode `getDocument`, récupérer la position du curseur, puis utiliser `insertString` pour insérer du texte. Cette solution est tout à fait acceptable. Toutefois, que se passe-t-il quand le raccourci est invoqué alors qu'une portion de texte est sélectionnée ? Pour avoir un comportement similaire au traditionnel «Coller», on souhaite remplacer l'éventuelle sélection par notre chaîne de caractères. La méthode `replaceSelection` permet justement de faire cela.

```
class AnnonceAction extends TextAction {
  public AnnonceAction(String nom) {
    super(nom);
  }
  public void actionPerformed(ActionEvent e) {
    JTextComponent compCible = getTextComponent(e);
    if (compCible != null && compCible.isEditable() &&
      compCible.isEnabled()) {
      compCible.replaceSelection("Bon état général");
    }
  }
}
```

Maintenant que la classe action est définie, nous devons créer un raccourci clavier pour pouvoir l'invoquer.

Pour gérer les raccourcis clavier, deux mécanismes sont offerts : l'un est utilisable pour tous les composants, l'autre est spécifique des composants texte. Puisque nous nous intéressons aux composants texte, utilisons le mécanisme spécifique.

Les raccourcis clavier sont supportés par un objet `Keymap`.

Un keymap est une collection d'associations entre une combinaison de touches et une action. Par exemple, Ctrl-C pour action copier, Ctrl-V pour action coller, etc.

Un keymap par défaut est associé à tous les composants texte. Il fournit tous les raccourcis « classiques » : les flèches (gauche, droite, haut et bas), la touche Entrée, les touches Fin et Début, et bien sûr les classiques Ctrl-C, Ctrl-V, etc.

Pour ajouter un nouveau raccourci clavier au composant, il est possible de modifier le keymap existant, ou bien d'ajouter un nouveau keymap. Modifier le keymap existant peut poser des problèmes, car le keymap par défaut est partagé par tous les composants texte de l'application. Si on souhaite que ce nouveau raccourci ne soit disponible que dans certains composants texte, il ne faut pas le modifier directement. La solution la plus sûre consiste à ajouter un nouveau keymap.

Les keymaps sont organisés en hiérarchie, et lorsqu'un événement de type `KeyEvent` est détecté, tous les keymaps sont explorés afin de rechercher un éventuel raccourci correspondant aux touches frappées dans l'ordre de la hiérarchie.

L'ajout d'un nouveau keymap se fait à l'aide de la méthode `addKeymap` de `JTextComponent`, en précisant en paramètre un nom et le keymap parent.

```
Keymap monKeymap =
   textPane.addKeymap("mesRaccourcis",
   ➥textPane.getKeymap());
```

Il est ensuite nécessaire de créer un `KeyStroke` (combinaison de touches) et de l'associer à notre action par le biais du keymap.

```
// Création de l'action "Bon état général"
AnnonceAction act = new AnnonceAction("BEG");
KeyStroke k = KeyStroke.getKeyStroke(KeyEvent.VK_B,
➥Event.CTRL_MASK);
monKeymap.addActionForKeyStroke(k,act);
```

Il suffit ensuite d'associer le keymap créé au composant texte :

```
textPane.setKeymap(monKeymap);
```

Désactivation d'un raccourci

Un problème inverse au précédent peut parfois se poser : nous souhaitons supprimer l'effet d'un raccourci clavier. Par exemple, sur un composant `JTextField`, le fait d'appuyer sur la touche « Entrée » génère un `ActionEvent` (même fonctionnement qu'un bouton), ceci a été fait afin d'assurer la compatibilité entre la classe `TextField` de AWT et `JTextField`.

Ce fonctionnement peut être nuisible, en particulier si l'on souhaite que l'appui sur la touche « Entrée » active le bouton par défaut de l'IHM. Pour désactiver ce comportement, il suffit d'obtenir le keymap du champ de texte, et de supprimer le lien correspondant au raccourci « clic sur Entrée ».

```
Keymap map = tfLogin.getKeymap() ;
// création d'un raccourci de type " clic sur Entrée "
Keystroke raccourci = KeyStroke.getKeyStroke(
➥KeyEvent.VK_ENTER, 0) ;
map.removeKeyStrokeBinding(entree) ;
```

Pour gérer des raccourcis disponibles sur l'ensemble d'une application, le procédé est assez similaire à ce qu'on vient de voir, à la différence près qu'on utilise des `InputMap` combinés à des `ActionMap`, au lieu de la classe `Keymap`.

5.4 IMPLÉMENTATION DU UNDO/REDO

Dans notre exemple de gestion de signets, nous souhaitons implémenter les actions annuler et rétablir dans la fenêtre d'édition. Ces actions seront disponibles *via* deux boutons de la barre d'outils et deux items de menu dans le menu contextuel (figure 5.27).

Figure 5.27 — Deux actions, annuler et rétablir.

5.4.1 Stocker les dernières actions effectuées

Pour supporter les actions d'annulation et de rétablissement, un composant texte doit être capable de mémoriser :

- les diverses éditions (insertion ou suppression de caractères, changements de mises en forme) en conservant l'ordre dans lequel elles se sont produites ;
- pour chaque édition, l'action d'annulation associée.

Pour gérer cette historisation, une instance de la classe `UndoManager` est nécessaire. Cette classe se trouve dans le package `javax.swing.undo`. Dans notre exemple, la classe `FenetreEditeur` possède donc une variable `undo` définie de la façon suivante :

```
UndoManager gestionnaireUndo = new UndoManager();
```

Nous allons utiliser cet objet `gestionnaireUndo` pour stocker toutes les éditions « annulables ». Pour cela, il faut être à l'écoute de ces éditions pour déclencher leur stockage dès qu'elles ont lieu.

5.4.2 Être à l'écoute des éditions

Les événements `UndoableEditEvent` sont lancés lorsqu'une opération pouvant être annulée est déclenchée sur un composant. Un événement `UndoableEditEvent` n'est pas généré par le composant lui-même, mais par son modèle ou document.

Les opérations capables de déclencher un tel événement sont :

- l'insertion de caractères ;
- la suppression de caractères ;
- la modification du style du texte.

L'implémentation de la gestion des annulations nécessite l'association d'un `listener` particulier au modèle du composant :

```
textPane.setDocument(document);
document.addUndoableEditListener
➥(new EvenementAnnulableListener());
```

La classe `EvenementAnnulableListener` implémente l'interface `UndoableEditListener`. Cette interface spécifie la méthode suivante :

```
public void undoableEditHappened(UndoableEditEvent e) ;
```

Nous allons implémenter cette méthode afin de stocker les informations nécessaires sur l'événement qui a eu lieu.

5.4.3 Ajouter les éditions au `UndoManager`

On n'ajoute pas les événements eux-mêmes au manager, mais des objets du type `UndoableEdit`. Ces objets encapsulent l'édition effectuée, et proposent divers services adaptés à la gestion des annulations :

- annuler cette édition ;
- rétablir cette édition ;
- indiquer si cette édition peut être annulée ou rétablie.

Ces objets `UndoableEdit` peuvent être obtenus à partir de l'événement grâce à la méthode `getEdit`.

Le listener se charge de :

- créer l'objet `UndoableEdit;`
- l'ajouter au manager ;

- mettre à jour l'état des actions `Undo` et `Redo` (ce qui provoquera la mise à jour de l'état — disponible ou non — des boutons et des menus).

```
/**
 * Classe incluse : listener sur les actions annulables
 */
class EvenementAnnulableListener implements
➥UndoableEditListener {
 public void undoableEditHappened(UndoableEditEvent e) {
    gestionnaireUndo.addEdit(e.getEdit());
    actionUndo.majEtat();
    actionRedo.majEtat();
 }
}
```

5.4.4 Création des actions `Undo` et `Redo`

Ces classes héritent de `AbstractAction`.

Le constructeur se charge d'initialiser la variable `UndoManager` de l'action, et de personnaliser l'action à l'aide d'un nom et d'une icône.

```
public class UndoAction extends AbstractAction {
  private static final String ICONE_GENERAL_PATH =
  ➥"toolbarButtonGraphics/general/";
  protected UndoManager gestionnaireUndo;

 /**
  * Construction de UndoAction, avec un nom et une icône
  */
  public UndoAction(UndoManager m) {
     super("Undo");
     gestionnaireUndo = m;
     setEnabled(false);
     putValue(Action.NAME, "Annuler");
     URL imageUrl =
        ClassLoader.getSystemResource(ICONE_GENERAL_PATH+
        ➥"Undo16.gif");
    putValue(Action.SMALL_ICON, new ImageIcon(imageUrl));
  }
 }
```

Lors de l'appel à cette action, on s'adresse au `gestionnaireUndo` pour annuler :

```
public void actionPerformed(ActionEvent e) {
  //annuler la dernière action
  try {
     gestionnaireUndo.undo();
  } catch (CannotUndoException ex) {
     System.out.println("Impossible d'annuler: " + ex);
     ex.printStackTrace();
  }
}
```

Par ailleurs, à chaque appel de l'action `Undo` ou `Redo`, il faut mettre à jour l'état des boutons et menus. En effet, imaginons le scénario suivant :

- on insère un caractère : cet événement annulable est stocké. L'action `Undo` devient disponible ;
- on annule cette insertion : l'action `Undo` n'est plus disponible (car il n'y a plus d'événement annulable stocké). En revanche, l'action `Redo` devient disponible.

Il faut donc mettre à jour l'état des deux actions (`Undo` et `Redo`) à chaque fois que l'une ou l'autre est appelée. Pour cela, écrivons un accesseur qui permet à l'action `Undo` de connaître l'action `Redo` et *vice versa*.

La méthode `actionPerformed` pourra ainsi mettre à jour l'état de l'action elle-même, ainsi que l'état de son action contraire. La mise à jour de l'état se fait grâce au `gestionnaireUndo` qui permet de savoir s'il existe en mémoire une édition à annuler ou une édition à rétablir.

Le code complet de la classe `UndoAction` est donc le suivant :

```
package applicatif;

import java.awt.event.ActionEvent;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.undo.*;
import java.net.URL;

/**
 * Classe implémentant l'action d'annulation
 */
public class UndoAction extends AbstractAction {

 private static final String ICONE_GENERAL_PATH =
 ➥"toolbarButtonGraphics/general/";
 protected UndoManager gestionnaireUndo;
 protected RedoAction redoAction;

/**
 * Construction de UndoAction, avec un nom et une icône
 */
 public UndoAction(UndoManager m) {
  super("Undo");
  gestionnaireUndo = m;
  setEnabled(false);
  putValue(Action.NAME, "Annuler");
  URL imageUrl = ClassLoader.getSystemResource(
  ➥ICONE_GENERAL_PATH+"Undo16.gif");
```

```
        putValue(Action.SMALL_ICON, new ImageIcon(imageUrl));
    }

    public void actionPerformed(ActionEvent e) {
        //annuler la dernière action
        try {
            gestionnaireUndo.undo();
        } catch (CannotUndoException ex) {
            System.out.println("Impossible d'annuler: " + ex);
            ex.printStackTrace();
        }
        //mise à jour de l'état des boutons (dispo ou non)
        majEtat();
        redoAction.majEtat();
    }

    /**
     * Met à jour l'état (disponible ou non) de l'action
     * et par extension de tous les composants graphiques
     * représentant cette action.
     */
    public void majEtat() {
        setEnabled(gestionnaireUndo.canUndo());
    }

    /**
     * Accesseur pour l'action Rétablir
     */
    public void setRedoAction(RedoAction act) {
        redoAction = act;
    }
}
```

Notez qu'il est possible d'améliorer l'ergonomie en affichant un texte plus précis que « Annuler » ou « Rétablir ». En effet, les `UndoableEdit` proposent également une méthode qui renvoie une chaîne de caractères caractérisant le type d'édition. Ainsi, on pourrait afficher « Annuler insertion » ou « Rétablir mise en forme » en fonction du type d'événement qui a eu lieu.

```
putValue(Action.NAME, "Annuler " +
                    gestionnaireUndo.getPresentationName());
```

Dans notre exemple de gestion de signets, nous n'avons pas utilisé cela, car les boutons de la barre d'outils ne portent aucun texte. Ils ont seulement une icône.

5.5 IMPLÉMENTATION D'UN CARETLISTENER

Nous souhaitons enrichir notre fenêtre d'édition avec une dernière fonctionnalité : les boutons de mise en forme (gras, italique et souligné) s'appliquent à la

sélection courante (ou à la position du curseur si la sélection est vide) grâce à un clic. Mais on souhaite également qu'ils reflètent l'état du texte à l'endroit où on positionne le curseur. Ainsi, si l'utilisateur positionne le curseur sur une portion de texte en gras et en italique, les deux boutons gras et italique doivent changer d'aspect pour indiquer un état ON (boutons enfoncés). Le bouton souligné doit rester à l'état OFF (figure 5.28).

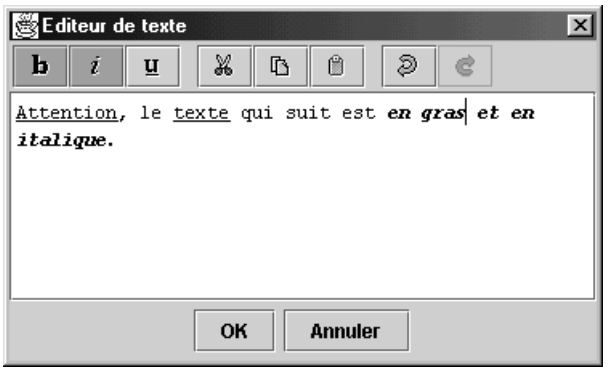


Figure 5.28 — Boutons reflétant l'état du texte à la position du curseur.

Pour implémenter ce fonctionnement, il faut être à l'écoute des événements « changements de position du curseur ». Cela se fait à l'aide d'un `CaretListener`. Lorsqu'un événement de ce type a lieu, il faut mettre à jour l'état des boutons (figure 5.29).

Mais allons-nous modifier explicitement ces trois boutons ? Si par la suite nous souhaitons ajouter d'autres composants graphiques représentant ces mêmes fonctionnalités (items de menu par exemple), nous ne souhaitons pas devoir modifier tout ce code. Il est donc judicieux de travailler sur les actions elles-mêmes plutôt que sur les composants graphiques. Pour gérer l'état sélectionné ou non, ajoutons une propriété supplémentaire sur les actions. Le changement de cette propriété doit entraîner le changement de l'aspect des boutons.

Choisissons d'appeler `SELECTIONNE` cette propriété que nous ajoutons sur les actions Gras, Italique et Souligné. Elle doit être une propriété dite liée (*BoundProperty*) selon la terminologie des Java Beans. Ainsi, lorsqu'elle subit une modification, elle émet des événements de type `PropertyChangeEvent`. Un listener doit donc être à l'écoute de ces événements et mettre à jour l'état des boutons.

Il existe deux manières de procéder.

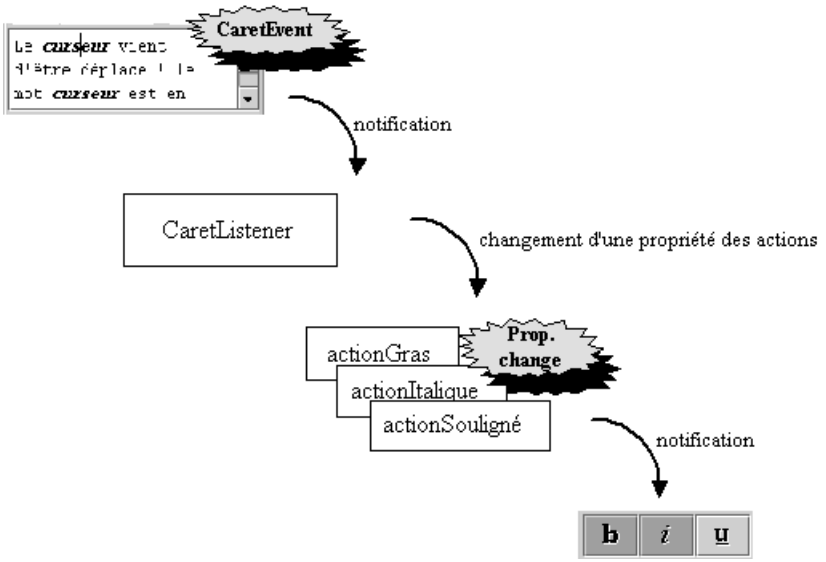


Figure 5.29 — Rôle du `CaretListener`.

La première solution consiste à sous-classer la classe `JToggleButton`, afin de redéfinir la méthode `createActionPropertyChangeListener`. Cette méthode crée un `PropertyChangeListener` qui met à jour le bouton lorsque les propriétés de l'action changent. Par défaut, cette méthode met à jour :

- le texte du bouton (changement de la propriété `NAME`) ;
- le tooltip (changement de la propriété `SHORT_DESCRIPTION`) ;
- l'icône (changement de la propriété `SMALL_ICON`) ;
- l'état activable ou non (changement de la variable `enabled`) ;
- la touche de raccourci (changement de la propriété `MNEMONIC_KEY`).

Pour gérer notre besoin, il suffirait d'ajouter quelques lignes de code pour que ce `PropertyChangeListener` mette à jour l'état sélectionné (avec la méthode `setSelected`) du bouton en fonction de la propriété `SELECTIONNE`.

Cette première solution respecte tout à fait la philosophie du code existant. Toutefois, elle oblige à créer des composants graphiques spécifiques, c'est-à-dire de la sous-classe de `JToggleButton`.

Si l'on ne souhaite pas définir une sous-classe de `JToggleButton`, il existe une deuxième solution. Elle consiste à écrire nous-mêmes un listener qui écoute les `PropertyChangeEvent` des trois actions en question. Ce listener se charge de mettre à jour l'état des boutons. Appelons cette classe `SelectionneListener` (figure 5.30).

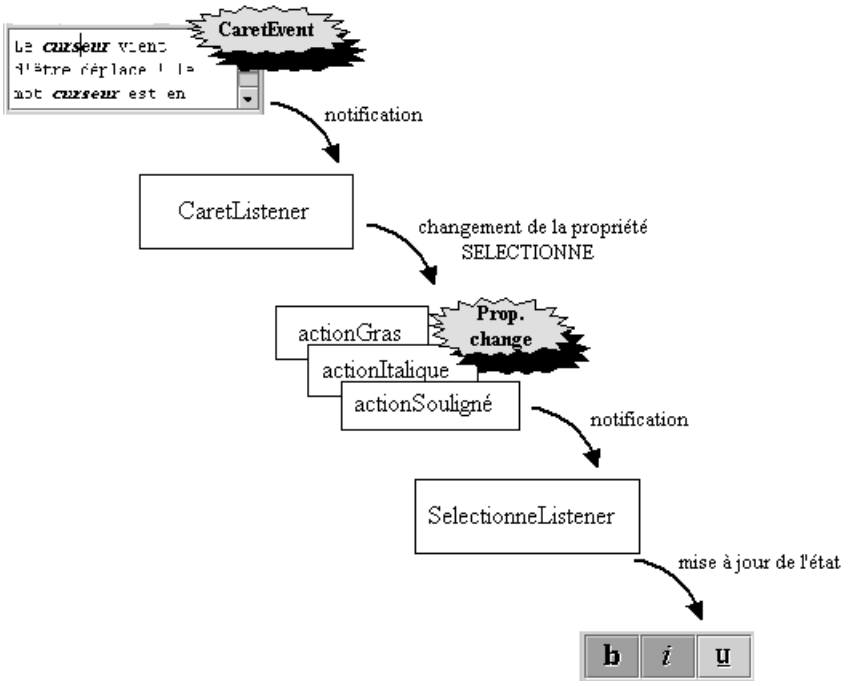


Figure 5.30 — Rôle du `SelectionneListener`.

Finalement, si nous choisissons la deuxième solution, voilà l'implémentation complète de cette fonctionnalité :

```
/**
 * Cette classe représente une fenêtre qui permet d'éditer
  * un texte.
  */
public class FenetreEditeur extends JDialog {
  [ . . . code présenté plus haut . . . ]
  /**
    * Méthode qui gère le fait que l'état des boutons
    * concernant le style (gras, italique et souligné)
    * soit mis à jour lorsque le curseur du texte est
    * positionné sur une zone étant en gras,
    * en italique ou en souligné.
    */
  public void gererMajBoutons() {
   // Lorsque le curseur est déplacé, les actions reçoivent
    // une notification de leur propriété "SELECTIONNE"
   textPane.addCaretListener(new CurseurTexteListener());
    // Il faut abonner les 3 actions à un listener qui
   // écoute les événements de type PropertyChangeEvent et
    // qui change l'état des boutons
    actionGras.addPropertyChangeListener(new
```

```
    ➥ SelectionneListener(boutonGras));
    actionItalique.addPropertyChangeListener(new
    ➥ SelectionneListener(boutonItalique));
    actionSouligne.addPropertyChangeListener(new
    ➥ SelectionneListener(boutonSouligne));
  }

  /**
   * Classe incluse : listener sur les mouvements du
   * curseur dans le texte. Les boutons gras, italique
   * et souligné sont mis à jour en fonction
   * du texte sur lequel se trouve le curseur.
   */
class CurseurTexteListener implements CaretListener {
    public void caretUpdate(CaretEvent e) {
      try {
      //On récupère le textPane à l'origine de l'événement
       JTextPane source = (JTextPane)e.getSource();
       // On récupère les propriétés des caractères au
       // point d'insertion courant
      AttributeSet attr = source.getCharacterAttributes();
       // On modifie les valeurs des actions en fonction
       // de l'état du texte au point d'insertion
       actionGras.putValue(SELECTIONNE,
       ➥new Boolean(StyleConstants.isBold(attr)));
       actionItalique.putValue(SELECTIONNE,
       ➥new Boolean(StyleConstants.isItalic(attr)));
       actionSouligne.putValue(SELECTIONNE,
       ➥new Boolean(StyleConstants.isUnderline(attr)));
      } catch (Exception ex) {
       // On peut avoir par exemple des ClassCastException
        ex.printStackTrace();
      }
    }
  }
  /**
   * Classe incluse : listener transformant la
   * modification de la propriété SELECTIONNE d'une action
   * en la sélection/désélection du ToggleButton associe
   */
  class SelectionneListener implements
    ➥PropertyChangeListener {
    protected AbstractButton bouton;
    public SelectionneListener(AbstractButton b) {
      bouton = b;
    }
    /**
     * Méthode appelée quand une propriété de l'objet
     * écouté (l'Action) change. Seul le cas de la propriété
     * SELECTIONNE nous intéresse.
     */
    public void propertyChange(PropertyChangeEvent e) {
```

```
            if (e.getPropertyName().equals(FenetreEditeur
            ➥ .SELECTIONNE)) {
              if (bouton != null) {
                bouton.setSelected(((Boolean)e.getNewValue())
                ➥.booleanValue());
              }
            }
          }
        }
      } // fin classe FenetreEditeur
```

6

Le drag and drop

Dans ce chapitre, nous allons voir comment implémenter une fonctionnalité très courante dans les applications actuelles et très appréciée des utilisateurs : le *drag and drop*. Ce terme désigne la possibilité de pouvoir déplacer graphiquement un élément quelconque d'un endroit vers un autre.

Ce déplacement s'effectue de la manière suivante : l'utilisateur sélectionne l'élément sur lequel il veut effectuer l'opération, il le déplace en maintenant le bouton adéquat de la souris appuyé (il s'agit du bouton gauche sous Windows), et il le dépose à l'endroit souhaité en relâchant le bouton.

Cette fonctionnalité permet d'effectuer en un seul geste une opération parfois complexe. Elle est incontournable si vous souhaitez créer une application ergonomique, ou si vous vous adressez à des utilisateurs ayant l'habitude d'utiliser ce genre de raccourcis, qui seront frustrés de ne pas l'avoir à leur disposition.

Le *drag and drop* est arrivé avec le JDK 1.2. Les classes nécessaires à une implémentation du *drag and drop* se trouvent dans les packages `java.awt.dnd` et `java.awt.datatransfer` (ce dernier existe depuis le JDK 1.1). Il est donc possible d'utiliser le *drag and drop* dans une application AWT ou Swing.

Intuitivement un *drag and drop* implique un composant source, un composant destination et un transfert d'information. Techniquement, c'est un peu plus complexe.

Ceci dit, on retrouve bien une source, `java.awt.dnd.DragSource`, une destination, `java.awt.dnd.DropTarget`, et une information transférée, `java.awt.datatransfer.Transferable`.

Depuis le JDK 1.4, tous les composants Swing sont potentiellement des sources et des cibles de *drag and drop*. De ce fait, le code pour ajouter un mécanisme de *drag and drop* entre deux composants Swing est devenu vraiment très simple.

Nous présentons ici deux mécanismes pour implémenter du *drag and drop* :

- Le mécanisme « historique » disponible depuis le JDK 1.2 et basé sur AWT.
- L'implémentation très simple qui est possible depuis le JDK 1.4.

6.1 UN EXEMPLE PRESQUE SIMPLE

Commençons tout de suite par un exemple ! Nous allons implémenter un *drag and drop* d'un `JLabel` vers un `JTextField`.

Préparons une interface graphique simple avec deux classes, une classe de test qui instanciera une `JFrame` et une sous-classe de `JPanel`. C'est dans cette dernière que nous coderons le *drag and drop*.

N'hésitez pas à demander le programme, le code complet de l'exemple se trouve au paragraphe 6.1.5.

6.1.1 Le décor

La sous-classe de `JPanel` est simple et classique, elle présente deux panneaux, l'un dans la partie `WEST` d'un `BorderLayout`, l'autre dans la partie `CENTER` du `BorderLayout`.

Ces deux panneaux sont associés à un `BoxLayout` pour assurer l'agencement vertical de leurs composants. Le panneau de la zone `WEST` contient deux labels. Le panneau de la zone `CENTER` contient un label et un `JtextField` (figure 6.1).

Nous utilisons la particularité du `BoxLayout` qui sait contenir les objets spéciaux produits par la classe Box. Une structure verticale de taille fixe vient ainsi s'intercaler entre les deux lignes, cette structure est créée par l'appel de méthode suivant :

```
Box.createVerticalStrut(20).
```

Figure 6.1 — L'interface graphique.

Le décor est posé, passons à l'intrigue.

6.1.2 Acte I, `DragSource`

Notre objectif est de permettre au `JLabel` de devenir un initiateur potentiel de *drag and drop*.

Un problème important lié au *drag and drop* est la portabilité. Pour initier un *drag and drop* l'utilisateur doit effectuer certaines actions comme presser une touche en même temps qu'un clic de souris. Or ces actions dépendent du système d'exploitation. Par exemple sous Windows, l'utilisateur doit cliquer sur le bouton gauche de la souris et maintenir la pression sur le bouton alors qu'avec MOTIF, l'utilisateur doit cliquer sur le bouton du milieu et presser la touche « Shift ». Heureusement, le JDK gère ces aspects pour nous. Il existe une classe dont le rôle est de reconnaître, selon la plate-forme, s'il faut initier un *drag and drop* en fonction des actions de l'utilisateur. La classe `DragGestureRecognizer` se charge de ce travail.

`DragSource` représente bien une source, mais pas nécessairement un composant source. Il existe plusieurs procédés d'utilisation de `DragSource` qui diffèrent sur le nombre d'instances de cette classe dans le système. L'un consiste à n'avoir qu'une seule instance de `DragSource` par machine virtuelle. Un autre consiste à instancier un `DragSource` pour chaque composant susceptible d'initier un *drag and drop*. En fait, le nombre d'instances de `DragSource` importe peu, il dépend de votre conception. Le plus simple est de n'avoir qu'une seule instance de `DragSource` par machine virtuelle. Dans ce cas, il est possible d'obtenir une instance de `DragSource`, qui devient un singleton, par une méthode statique : `DragSource.getDefaultDragSource()`.

Quel que soit le nombre d'instances de `DragSource`, il faut que le composant source soit prévenu qu'un *drag and drop* est initié et qu'il en est la source (rien n'interdit un composant d'être à la fois une source et une destination potentielle de *drag and drop*). C'est le `DragSourceRecognizer` qui va jouer ce rôle. Il doit connaître le composant source et un listener que nous devons implémenter : `DragGestureListener`.

Vous vous demandez peut-être pourquoi un listener est nécessaire. De fait, le *drag and drop* n'est pas si automatisé qu'on l'imagine. Lorsque l'implémentation du listener va nous « prévenir » qu'une action drag a débuté, c'est à nous de préparer la donnée à transférer.

Avant d'aller plus loin, voyons déjà comment implémenter tout ce que nous avons évoqué.

Acte I, scène 1 : obtenir l'instance par défaut de `DragSource`, partagée au sein de la machine virtuelle.

Ajoutons à cet effet un attribut à notre classe `Pan`.

```
protected DragSource dragSource = null;
```

L'attribut est initialisé dans le constructeur :

```
dragSource = DragSource.getDefaultDragSource();
```

Acte I, scène 2 : obtenir une instance de `DragGestureRecognizer` et lui associer le composant ainsi que l'implémentation du `DragGestureListener`.

La scène débute par l'arrivée du `DragGestureRecognizer` et de son inséparable `DragGestureEvent` :

```
protected class SrcDragGestureListener
➥implements DragGestureListener {
  public void dragGestureRecognized(DragGestureEvent event)  {
  }
}
```

Nous avons décidé de coder les listeners comme des classes incluses pour ne pas que les rôles de chacun se mélangent. Cette classe listener est donc définie à l'intérieur de la classe `Pan`, et l'instance de ce listener est une variable d'instance de `Pan`. Nous coderons la méthode `dragGestureRecognized` plus tard. Elle sera appelée dès que le `DragGestureRecognizer` détectera un *drag and drop* sur le `JLabel`.

La classe d'événement `DragGestureEvent` nous donnera toutes les informations utiles pour que nous déclenchions la suite des opérations car c'est bien au développeur de déclencher le début de l'opération drag. Cela peut sembler fastidieux et inutile, mais il est ainsi possible de paramétrer l'opération de *drag and drop* pour des situations variées. Par exemple, on peut décider de ne pas permettre l'opération drag à partir d'un composant « source » si les données fonctionnelles présentes dans le panneau ne sont pas dans un certain état.

Il faut ensuite instancier notre classe `SrcDragGestureListener`. Nous avons choisi de créer des variables d'instance pour chacun des listeners.

```
protected SrcDragGestureListener srcDragGestureListener =
new SrcDragGestureListener();
```

La scène se poursuit par l'instanciation du `DragGestureRecognizer` et sa liaison avec le composant et le listener. Cette instance s'obtient à l'aide de l'objet `DragSource`.

```
dragSource.createDefaultDragGestureRecognizer(src,
DnDConstants.ACTION_MOVE,
srcDragGestureListener);
```

Le deuxième paramètre fait appel à un figurant : `DnDConstants`. Ce paramètre permet de définir les types d'actions qui sont acceptés. Ces actions sont des attributs statiques de la classe `DnDConstants`. Il existe quatre variantes d'actions :

- `ACTION_COPY`
- `ACTION_MOVE`
- `ACTION_NONE`
- `ACTION_LINK`

Il est possible d'en permettre plusieurs en utilisant les constantes prévues pour cela, comme `DnDConstants.ACTION_COPY_OR_MOVE`. Ces actions définissent les critères de reconnaissance d'un début de *drag and drop* déclenché par l'utilisateur. Par exemple sous Windows, un *drag and drop* « classique » va déplacer l'information. Pour faire un *drag and drop* qui copiera l'information, il faut en plus presser la touche « Ctrl ». Dans ce cas, si l'action `ACTION_COPY` n'est pas autorisée, le `DragGestureRecognizer` ne permettra pas le *drag and drop*.

Passons à la scène 3, l'acte I touche à sa fin, il faut mettre du code dans le `DragGestureListener` que nous avions laissé en coulisses. Comme nous l'avions dit, c'est à nous de déclencher le *drag and drop*. Ici nous le déclencherons systématiquement. Notez l'entrée en scène du `DragSourceListener`.

```
protected class SrcDragGestureListener
➥implements DragGestureListener {
     public void dragGestureRecognized
     ➥(DragGestureEvent event) {
           // Ici il faut obtenir l'information
           // à véhiculer pendant l'opération de drag
           // and drop.
           // Nous verrons cela dans l'acte II.
              event.startDrag(DragSource.DefaultMoveDrop,
                        <l'information>,
                        srcDragSourceListener);
    }
}
```

La méthode importante est `startDrag`. C'est ici que le *drag and drop* débute réellement. Cette méthode prend en paramètre l'information à transférer, nous verrons comment préparer cette information lors de l'acte II.

Le premier paramètre est l'apparence que doit prendre le curseur. Ce paramètre attend une instance de `Cursor`, c'est une classe que nous avions vue au chapitre 1. Cela permet de donner une apparence personnalisée au curseur. Si vous entamiez un *drag and drop* et que vous imposiez un curseur de type sablier, le *drag and drop* aurait bien lieu. Techniquement, tout se passerait bien si ce n'est la surprise de l'utilisateur. C'est au développeur de gérer la cohérence entre l'apparence du curseur et l'action qui est codée.

La classe `DragSource` propose des attributs statiques pour les valeurs par défaut, comme `DefaultMoveDrop`.

Le dernier paramètre, enfin, est une instance de `DragSourceListener` que nous n'avions pas encore rencontré. Ce listener va nous permettre de suivre les différentes étapes du *drag and drop*. Comme d'habitude, nous en ferons une classe incluse et une variable d'instance référencera son instance.

```
protected class SrcDragSourceListener implements
➥DragSourceListener {
    public void dragDropEnd(DragSourceDropEvent event) {
          if (event.getDropSuccess()){
                // Ici il faudra coder une action
                // sur le composant source en cas
                // de succès de l'opération de drag
                // and drop.
          }
    }
    public void dragEnter(DragSourceDragEvent dsde) {
    }
    public void dragExit(DragSourceEvent dse) {
    }
    public void dragOver(DragSourceDragEvent dsde) {
    }
    public void dropActionChanged(DragSourceDragEvent dsde) {
    }
}
```

6.1.3 Acte II, `Transferable`

L'opération de *drag and drop* a bien commencé, mais nous ne lui avons pas encore adjoint une information à transporter.

Le transfert de l'information s'effectue *via* une interface : `Transferable`. Toute information transférable *via* un *drag and drop* doit implémenter cette interface. Il n'est pas possible de restreindre la nature des informations transférées à un ensemble de classes prédéfinies. De plus, un problème supplémentaire se pose. Pourquoi la nature de l'information contenue dans la source serait-elle strictement équivalente à celle attendue par la destination ? Il est par exemple possible de sélectionner un ensemble de fichiers dans un gestionnaire de fichiers et d'effectuer un *drag and drop* dans une application de gestion de texte. Le résultat attendu dans le gestionnaire de texte serait un paragraphe composé des noms ou du chemin complet des fichiers sélectionnés dans le gestionnaire de fichiers et non pas le contenu des fichiers. Ainsi, l'information n'a pas forcément la même nature dans la source et dans la destination. Il faut donc transférer la nature de l'information en plus de l'information elle-même, une méta-information. Ce concept est implémenté par la classe `DataFlavor`.

Une instance représentant une information, implémentant l'interface `Transferable`, doit être capable de fournir l'ensemble des `DataFlavor` qu'elle supporte. Notez que rien n'indique la nature réelle, interne de cette information. Les `DataFlavor` n'indiquent que ce qui est restituable à partir d'instances implémentant `Transferable`.

Nous allons revenir en arrière, souvenez-vous dans l'acte I, scène 3, nous avions laissé de côté une partie de l'implémentation du `DragGestureListener`. C'est dans ce listener que nous déclenchions le départ réel du *drag and drop*. Nous allons maintenant finaliser ce listener en constituant une instance qui implémente l'interface `Transferable` matérialisant l'information véhiculée par le *drag and drop*.

L'unique scène de l'acte II va donc voir l'entrée en scène de `StringSelection`. Cette classe implémente `Transferable` et permet de transporter des informations de type « chaîne de caractères ». Cette classe se trouve dans le package `java.awt.datatransfer`.

```
protected class SrcDragGestureListener
➥implements DragGestureListener {
    public void dragGestureRecognized
    ➥(DragGestureEvent event) {
        Object texte_a_transferer = src.getText();
        StringSelection texte = new StringSelection(
        ➥texte_a_transferer.toString());
        event.startDrag(DragSource.DefaultMoveDrop,
        ➥texte, srcDragSourceListener);
    }
}
```

La constitution de l'information se fait en deux étapes. La première étape est toute simple, il faut aller chercher l'information « brute » contenue dans le composant. Nous utilisons un `JLabel` comme composant source, aussi utilisons-nous la méthode `getText()`. Cet aspect est encore à la charge du développeur. De cette manière, nous pourrions aller chercher une information plus complexe que celle réellement contenue dans le composant. Par exemple, le composant aurait pu ne contenir qu'une clé technique permettant d'obtenir une information fonctionnelle complète. Ce serait le cas d'un champ qui ne contiendrait que le code postal d'une ville alors que le nom de la ville serait transféré.

La deuxième étape, c'est l'instanciation d'une classe qui implémente l'interface `Transferable`. Nous allons utiliser pour cet exemple une classe simple fournie par le JDK, mais rien ne nous empêcherait d'instancier une classe que nous aurions développée nous-mêmes.

6.1.4 Acte III, `DropTarget`

L'intrigue a atteint son paroxysme, et un suspense insoutenable nous assaille, comment le *drag and drop* va-t-il se terminer ? Comment allons-nous retrouver l'information transportée ?

La scène 1 voit l'arrivée de `DropTarget`. Cette classe est le pendant logique de `DragSource`. Il faut maintenant associer notre composant destinataire du

drag and drop avec un listener, le dernier, `DropTargetListener` qui nous « préviendra » qu'un *drag and drop* est en cours et plus précisément, que la « séquence drop » se réalise sur notre composant, le `JTextField`.

Comme toujours, notre listener est implémenté par une classe incluse :

```
protected class DestDropTargetListener
➥implements DropTargetListener {
    public void dragEnter(DropTargetDragEvent event) {
    }
    public void dragExit(DropTargetEvent dte) {
    }
    public void dragOver(DropTargetDragEvent dtde) {
    }
    public void drop(DropTargetDropEvent event) {
    }
    public void dropActionChanged(DropTargetDragEvent dtde) {
    }
}
```

Nous implémenterons ce listener dans la scène 2. Comme précédemment, une instance de ce listener est référencée par une variable d'instance :

```
protected DestDropTargetListener destDropTargetListener =
 new DestDropTargetListener();
```

Comment se fait l'association, au sein d'une instance de `DropTarget`, du composant destinataire du *drag and drop* et du listener ? Elle se fait très simplement à l'aide du constructeur :

```
dropTarget = new DropTarget(dest, destDropTargetListener);
```

C'est le développeur qui déclenche le *drag and drop* et construit l'information véhiculée, c'est donc encore le développeur qui doit déclencher l'acceptation du drop, extraire l'information puis la positionner dans le composant de destination.

La scène 2 voit l'intrigue se résoudre.

Notez qu'aux différentes étapes du *drag and drop*, le listener permet d'arrêter le processus, par exemple en utilisant la méthode `rejectDrag()` sur la classe `DropTargetDragEvent`.

Nous aurions donc pu refuser le drag en testant la valeur de certains critères, comme le fait que le `JTextField` soit désarmé ou non éditable…

Enfin, et c'est le cœur de l'action de l'acte III, nous allons maintenant récupérer l'information véhiculée par le *drag and drop* et l'affecter au composant `JTextField`. Tout cela se passe dans la méthode `drop` du listener.

La première étape consiste à obtenir à partir de l'événement la donnée qui a été transférée :

```
Transferable transferable = event.getTransferable();
```

Ensuite, il faut interroger cette instance afin de vérifier qu'elle peut nous restituer une information de type « chaîne de caractères », car c'est le seul type d'information que nous accepterons pour notre `JTextField`. Vous vous demandez peut-être à quoi peut bien servir une telle vérification, puisque nous sommes dans un *drag and drop* en provenance du `Jlabel`. En fait, nous n'en savons rien ici ! Nous savons seulement que nous sommes dans la partie « drop » d'un *drag and drop*. Peut-être celui-ci a-t-il été initié à partir d'une autre source ? Il est ainsi possible d'envisager un *drag and drop* entre deux applications.

Le principe est le suivant :

```
if (transferable.isDataFlavorSupported(
➥DataFlavor.stringFlavor)) {
     // poursuivre l'extraction de l'information
} else {
     // abandonner l'opération car nous ne savons pas
     // obtenir une information de type textuelle.

}
```

Pour poursuivre l'opération, il faut indiquer que nous acceptons le drop :

```
event.acceptDrop(DnDConstants.ACTION_MOVE);
```

L'extraction de l'information proprement dite est simple :

```
String s = (String)transferable.getTransferData
(DataFlavor.stringFlavor);
```

Cette extraction mérite quelques précautions, elle est susceptible de lever deux exceptions. La version finale de cette méthode du listener met en œuvre un bloc `try/catch` :

```
protected class DestDropTargetListener implements
➥DropTargetListener {
    public void dragEnter(DropTargetDragEvent event) {
         event.acceptDrag(DnDConstants.ACTION_MOVE);
    }
    public void dragExit(DropTargetEvent dte) {
    }
    public void dragOver(DropTargetDragEvent dtde) {
    }
    public void drop(DropTargetDropEvent event) {
         try {
            Transferable transferable =
            ➥event.getTransferable();
            // Nous n'acceptons que les String
```

```
                if(transferable.isDataFlavorSupported(
                ➥DataFlavor.stringFlavor)) {
                    event.acceptDrop(
                    ➥DnDConstants.ACTION_MOVE);
                    String s =
                    ➥(String)transferable.getTransferData(
                    ➥DataFlavor.stringFlavor);
                    dest.setText(s);
                    event.getDropTargetContext().
                    ➥dropComplete(true);
                } else {
                    System.out.println("Drop refusé !!!!!");
                    event.rejectDrop();
                }
            } catch (IOException exception) {
                exception.printStackTrace();
                System.err.println( "Exception"
                + exception.getMessage());
                event.rejectDrop();
            } catch (UnsupportedFlavorException
            ➥ufException) {
                ufException.printStackTrace();
                System.err.println( "Exception"
                + ufException.getMessage());
                event.rejectDrop();
            }
        }
        public void dropActionChanged(DropTargetDragEvent dtde) {
        }
}
```

6.1.5 Code complet de l'exemple

```
import java.awt.*;
import java.awt.dnd.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import javax.swing.*;
import java.io.*;

public class Pan extends JPanel {
     protected class SrcDragGestureListener
     ➥implements DragGestureListener {
          public void dragGestureRecognized
          ➥(DragGestureEvent event) {
            Object texte_a_transferer = src.getText();
            StringSelection texte = new StringSelection(
            ➥texte_a_transferer.toString());
            event.startDrag(DragSource.DefaultMoveDrop,
                            texte,
                            srcDragSourceListener);
          }
     }
```

```
protected class SrcDragSourceListener
➥implements DragSourceListener {
     public void dragDropEnd(
     ➥DragSourceDropEventevent) {
           if (event.getDropSuccess()){
           }
     }
     public void dragEnter(
     ➥DragSourceDragEvent dsde) {
     }
     public void dragExit(DragSourceEvent dse) {
     }
     public void dragOver(DragSourceDragEvent dsde) {
     }
     public void dropActionChanged
     ➥(DragSourceDragEvent dsde) {
     }
}

protected class DestDropTargetListener
➥implements DropTargetListener {
     public void dragEnter(DropTargetDragEvent
     ➥event) {
          event.acceptDrag(DnDConstants.ACTION_MOVE);
     }
     public void dragExit(DropTargetEvent dte) {
     }
     public void dragOver(DropTargetDragEvent dtde) {
     }
     public void drop(DropTargetDropEvent event) {
          try {
     Transferable transferable =
     ➥event.getTransferable();
     // Nous n'acceptons que les String
     if (transferable.isDataFlavorSupported(
     ➥DataFlavor.stringFlavor)) {
         event.acceptDrop(DnDConstants
         ➥.ACTION_MOVE);
         String s = (String)transferable.
         ➥getTransferData(DataFlavor
         ➥.stringFlavor);
         dest.setText(s);
         event.getDropTargetContext().
         ➥dropComplete(true);
          } else {
           System.out.println("Drop
           ➥refusé !!!!!");
           event.rejectDrop();
           }
       } catch (IOException exception) {
           exception.printStackTrace();
           System.err.println( "Exception"
           ➥+ exception.getMessage());
           event.rejectDrop();
```

```
            } catch (UnsupportedFlavorException
              ➥ufException) {
              ufException.printStackTrace();
              System.err.println( "Exception"
              ➥+ ufException.getMessage());
              event.rejectDrop();
            }
          }
          public void dropActionChanged
          ➥(DropTargetDragEvent dtde) {
          }
}

protected JPanel pGauche = new JPanel();
protected JPanel pDroite = new JPanel();
protected JLabel lbSrc = new JLabel("Source");
protected JLabel lbDest = new JLabel("Destination");
protected JLabel src = new JLabel("Source");
protected JTextField dest = new JTextField
➥("Destination");
protected BoxLayout layoutGauche =
➥new BoxLayout(pGauche, BoxLayout.Y_AXIS);
protected BoxLayout layoutDroite =
➥new BoxLayout(pDroite, BoxLayout.Y_AXIS);

protected DragSource dragSource = null;
protected DropTarget dropTarget = null;

protected SrcDragGestureListener
➥srcDragGestureListener =
➥new SrcDragGestureListener();
protected SrcDragSourceListener
➥srcDragSourceListener =
➥new SrcDragSourceListener();
protected DestDropTargetListener
➥destDropTargetListener =
➥new DestDropTargetListener();

public Pan() {
      setLayout(new BorderLayout(5, 5));
      pGauche.setLayout(layoutGauche);
      pDroite.setLayout(layoutDroite);
      pGauche.add(lbSrc);
      pGauche.add(Box.createVerticalStrut(20));
      pGauche.add(lbDest);
      pDroite.add(src);
      pDroite.add(Box.createVerticalStrut(20));
      pDroite.add(dest);
      add(pGauche, BorderLayout.WEST);
      add(pDroite, BorderLayout.CENTER);
      dest.setColumns(30);
```

```
                dragSource = DragSource.getDefaultDragSource();
                dragSource.createDefaultDragGestureRecognizer
                ➥(src, DnDConstants.ACTION_MOVE,
                ➥ srcDragGestureListener);
                dropTarget = new DropTarget
                ➥(dest, destDropTargetListener);
        }
    }
```

6.2 RÉSUMÉ DES ÉTAPES DE L'IMPLÉMENTATION DU DRAG AND DROP

Les principaux intervenants d'un *drag and drop* sont représentés figure 6.2.

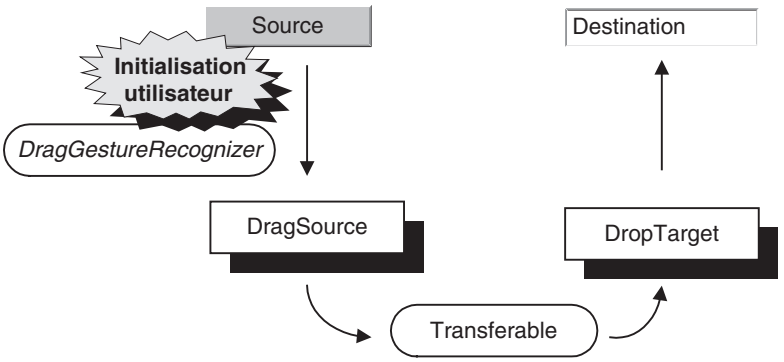


Figure 6.2 — Les principaux intervenants d'un *drag and drop*.

6.2.1 Création d'un objet `DragSource`

L'objet `DragSource` est responsable de l'initiation de l'opération de *drag and drop*. Il est associé au composant graphique à partir duquel on veut permettre un *drag*. Le composant doit bien sûr être capable de fournir des données à transférer. Il peut s'agir d'un champ de texte, d'une liste, etc.

Il existe deux façons d'avoir une instance de `DragSource` :

- on récupère l'instance par défaut : `DragSource.getDefaultDragSource()` ;
- on crée une nouvelle instance avec le constructeur sans paramètres.

La première méthode d'obtention permet de n'utiliser qu'une seule instance au sein d'une machine virtuelle.

6.2.2 Implémentation du `DragSourceListener`

Le `DragSourceListener` récupère les événements liés au *drag and drop* tout au long de l'opération, pas seulement quand on est dans le composant source.

L'événement `DragSourceEvent` permet d'obtenir le `DragSourceContext` qui rassemble les caractéristiques de la partie « drag » du *drag and drop*. Il est possible d'indiquer au `DragSourceContext` une forme de curseur.

Le listener est aussi prévenu lorsque le *drag and drop* se termine ce qui permet d'agir sur le composant source, par exemple en supprimant les données de ce composant ou encore en lui donnant une apparence particulière. On pourrait imaginer de changer la couleur du composant source, indiquant à l'utilisateur qu'il n'est pas possible de renouveler l'opération dans l'immédiat.

6.2.3 Création d'un objet `DragGestureRecognizer`

L'objet `DragGestureRecognizer` a pour rôle de reconnaître le geste qui initialise le *drag and drop* et ce, indépendamment de la plate-forme sur laquelle le programme s'exécute.

Quand on le crée, on indique le composant source, la liste des actions possibles sur ce composant, et l'instance du `DragGestureListener`.

Les actions possibles sont représentées par des constantes dans la classe `DnDConstants` : `ACTION_MOVE`, `ACTION_COPY_OR_MOVE`...

6.2.4 Implémentation d'un `DragGestureListener`

Le `DragGestureListener` est une classe de votre choix qui implémente l'interface `DragGestureListener`. L'abonnement auprès du `DragGestureRecognizer` se fera lors de l'instanciation de ce dernier.

L'unique méthode de ce listener est `DragGestureRecognized`. C'est dans cette méthode que le développeur doit démarrer le *drag and drop* par l'appel de la méthode `startDrag` proposée par l'événement `DragGestureEvent`.

Avant de démarrer le *drag and drop,* il faut encore récupérer l'élément à déplacer (objet qui implémente `Transferable`) depuis le composant graphique.

Notez qu'il est aussi possible d'appeler la méthode `startDrag` sur l'objet `DragSource`. Dans ce cas la méthode requiert plusieurs paramètres :

- l'événement `DragGestureEvent`, qui est à l'origine de l'initiation du drag and drop ;
- le curseur de souris à afficher au début de l'opération du drag and drop. Ce curseur peut être choisi par les constantes proposées par la classe

`DragSource`. Généralement, on choisira un curseur de type « NoDrop » pour indiquer à l'utilisateur qu'il ne se trouve pas encore sur un composant où il peut effectuer un drop ;

- éventuellement une image qui représente ce qui est transféré (par exemple une icône). Cette icône apparaîtra en dessous du curseur de la souris ;
- dans le cas où on a fourni une image à afficher, on peut ajouter les coordonnées auxquelles cette image s'affichera ;
- un objet de type `Transferable` qui représente les données à transférer ;
- une instance de `DragSourceListener`, qui a pour rôle de recevoir les notifications des événements qui décrivent l'opération de drag and drop. C'est donc ici que l'abonnement se fait auprès du `DragSourceListener`.

6.2.5 Implémentation d'un `DropTargetListener`

Ce listener est le symétrique du `DragSourceListener`, il permet de suivre les étapes du *drag and drop* du côté de la destination. Ce listener est une classe de votre choix implémentant l'interface `DropTargetListener`.

Aux différentes étapes du *drag and drop*, il est possible de refuser l'opération en fonction de vos propres critères. Pour cela, il faut utiliser la méthode `rejectDrag` portée par la classe d'événement `DropTargetDragEvent`. Un critère de rejet peut être le fait que le type d'information que l'on s'apprête à transférer sur la destination ne convient pas. Cela peut être détecté par la méthode :

```
boolean isDataFlavorSupported(DataFlavor df) ;
```

Enfin, la classe d'événement `DropTargetDropEvent` permet encore d'accepter ou de refuser l'opération de *drag and drop*.

La méthode `drop` est la plus importante de ce listener. C'est ici que le développeur récupère l'information sous la forme d'une instance implémentant l'interface `Transferable` et la décode en fonction des `DataFlavor` acceptables par la source.

C'est encore dans cette méthode enfin que le développeur décide d'arrêter l'opération de *drag and drop* par un appel à la méthode `getDropTargetContext().dropComplete(true)`.

6.2.6 Création d'un `DropTarget`

L'instance de `DropTarget` s'obtient simplement en utilisant le constructeur de la classe. C'est à ce moment que l'on abonne le `DropTargetListener` et que le composant de destination est lié au `DropTarget`.

6.3 L'APPLICATION DE GESTION DE SIGNETS

Nous avons implémenté la possibilité de réorganiser les nœuds dans l'arborescence des signets par *drag and drop*. Pour ce faire, nous avons choisi de faire une sous-classe de `JTree` que nous avons appelée `ArbreSignet` dans le package `ihm`. Nous reprendrons le même schéma d'implémentation du *drag and drop* que celui évoqué dans l'exemple de ce chapitre. Nous ne montrerons donc que les différences qui sont :

- nous implémentons le drag and drop dans une sous-classe d'un composant ;
- ce composant est à la fois source et destination du drag and drop.

Il y a une difficulté supplémentaire : nous ne transférons pas des chaînes, mais des `DefaultMutableTreeNode`. Nous avons donc codé une classe, `Noeud Transferable`, dans le package `ihm` qui implémente `Transferable`.

6.3.1 La classe `NoeudTransferable`

Cette classe doit être capable de transporter un `DefaultMutableTreeNode` dans une opération de *drag and drop*, elle implémente l'interface `Transferable`. Cette interface nous impose trois méthodes :

- `Object getTransferData(DataFlavor flavor)`. Cette méthode doit retourner l'objet transféré dans un type qui dépend de la `DataFlavor` demandée. Il faut donc que nous décidions de la(les) `DataFlavor` que nous allons supporter ;
- `DataFlavor[] getTransferDataFlavors()`. Cette méthode retourne la liste des `DataFlavor` supportées par la classe;
- `boolean isDataFlavorSupported(DataFlavor flavor)`. Cette méthode, enfin, teste si une `DataFlavor` donnée est supportée par cette classe.

Nous allons donc nous créer une `DataFlavor` dédiée au format `Default MutableTreeNode`. C'est le seul que nous accepterons. Au vu de cette `DataFlavor`, les utilisateurs de notre classe sauront qu'ils peuvent imposer le type `Default MutableTreeNode` à l'objet retourné par la méthode `getTransferData`.

Comment faire cette `DataFlavor` ? Inutile d'envisager une sous-classe de `DataFlavor`, le constructeur de cette classe permet d'obtenir des instances personnalisées.

Il nous faut un type `MIME` et une chaîne de description. La classe `DataFlavor` porte en attributs statiques un certain nombre de types `MIME` prédéfinis. Le type `javaJVMLocalObjectMimeType` caractérise précisément ce que nous souhaitons

faire. Notons que nous serons limités à des *drag and drop* internes à la machine virtuelle Java qui fait tourner l'application de gestion de signets.

Notre classe `NœudTransferable` est donc toute simple :

```
package ihm;

import java.awt.datatransfer.*;
import javax.swing.tree.DefaultMutableTreeNode;

public class NoeudTransferable implements Transferable {
     // Notre DataFlavor spécifique
     protected static final
     ➥DataFlavor noeudFlavor = new DataFlavor(
     ➥DataFlavor.javaJVMLocalObjectMimeType,
     ➥DataFlavor pour un
     ➥DefaultMutableTreeNode");

     // Le nœud que nous allons transporter.
     protected DefaultMutableTreeNode noeud;

     // Le constructeur impose de fournir un
     // DefaultMutableTreeNode.
     public NoeudTransferable(DefaultMutableTreeNode
     ➥info) {
          noeud = info;
     }

     public Object getTransferData(DataFlavor flavor) {
          if (isDataFlavorSupported(flavor))  {
               return noeud;
          } else {
               return null;
          }
     }
     public DataFlavor[] getTransferDataFlavors() {
          DataFlavor[] result = {noeudFlavor};
          return result;
     }
     public boolean isDataFlavorSupported(DataFlavor
     ➥flavor) {
          return flavor.equals(noeudFlavor);
     }

     public static DataFlavor getNoeudFlavor() {
          return noeudFlavor;
     }
}
```

6.3.2 La classe `ArbreSignet`

La constitution des éléments de base nécessaires au *drag and drop* ne change pas par rapport à l'exemple du début du chapitre. La seule petite différence est que notre sous-classe est à la fois source et destination du *drag and drop*. Voici comment s'initialisent la `DragSource` et la `DropTarget` :

```
dragSource = DragSource.getDefaultDragSource();
dragSource.createDefaultDragGestureRecognizer(this,
➥DnDConstants.ACTION_MOVE,
➥srcDragGestureListener);
dropTarget = new DropTarget(this, destDropTargetListener);
```

Voici maintenant le `DragGesturelistener` dont la seule différence par rapport à l'exemple est l'utilisation d'un `JTree` comme source de *drag and drop* en lieu et place du `JLabel`. C'est aussi dans cette classe que nous utilisons notre `NoeudTransferable`.

```
protected class SrcDragGestureListener
➥implements DragGestureListener {
     public void dragGestureRecognized (DragGestureEvent
     ➥event) {
           TreePath tp = ArbreSignet.this
           ➥.getSelectionPath();
           DefaultMutableTreeNode noeud_a_deplacer =
           ➥(DefaultMutableTreeNode)tp.getLastPathComponent();
           if (noeud_a_deplacer != null){
              NoeudTransferable transferable =
              ➥new NoeudTransferable(noeud_a_deplacer);
              event.startDrag(DragSource.DefaultMoveDrop,
              ➥transferable,
              ➥srcDragSourceListener);
           } else {
              System.out.println("Selectionnez un noeud !");
           }
     }
}
```

Voici enfin, le `DropTargetListener`. C'est lui qui subit le plus de changements. En effet, en plus d'obtenir l'instance de `DefaultMutableTreeNode` à partir de notre instance de `NoeudTransferable`, il est nécessaire d'effectuer une vérification fonctionnelle. Il est impossible de déposer le nœud à transférer ailleurs que dans une catégorie. Nous ne montrons ici que la méthode `drop` de ce listener :

```
public void drop(DropTargetDropEvent event) {
   try {
       Transferable transferable = event.getTransferable();
       // Nous n'acceptons que les DefaultMutableTreeNode
```

```
            if (transferable.isDataFlavorSupported(
            ➥NoeudTransferable.getNoeudFlavor())) {
               DefaultMutableTreeNode s =
               ➥(DefaultMutableTreeNode)
               ➥transferable.getTransferData(
                ➥NoeudTransferable
               ➥.getNoeudFlavor());
               Point p = event.getLocation();
               TreePath tp =
               ➥ArbreSignet.this.getPathForLocation
               ➥(p.x, p.y);
               DefaultMutableTreeNode parent =
               ➥(DefaultMutableTreeNode)tp
               ➥.getLastPathComponent();
               // Verification que le noeud, futur parent
               // est bien une catégorie.
               if (!verifierCategorie(parent))  {
                   System.out.println(
                   ➥Vous ne pouvez pas déposer ce noeud
                   ➥ici !");
                   event.rejectDrop();
               } else {
                   event.acceptDrop(DnDConstants.ACTION
                   ➥_MOVE);
                   DefaultTreeModel model =
                   ➥(DefaultTreeModel)
                   ➥ArbreSignet.this.getModel();
                   parent.add(s);
                   model.reload();
                   event.getDropTargetContext().
                   ➥dropComplete(true);
                }
            } else {
                System.out.println("Drop refusé !!!!!");
                event.rejectDrop();
            }
        } catch (IOException exception) {
           exception.printStackTrace();
           System.err.println( "Exception" +
                   ➥exception.getMessage());
                   event.rejectDrop();
                } catch (UnsupportedFlavorException
                   ➥ufException) {
                   ufException.printStackTrace();
                   System.err.println( "Exception" +
                   ➥ufException.getMessage());
                   event.rejectDrop();
                }
             }
```

6.4 SIMPLIFICATION DEPUIS LE JDK 1.4

L'arrivée du *drag and drop* s'est effectuée en deux étapes. La première étape consistait tout simplement à rendre le *drag and drop* possible, cette implémentation que nous venons de présenter était basée sur AWT. Elle est assez complexe à mettre en œuvre car elle se place à un niveau d'abstraction assez bas. En contrepartie, il est possible de tout contrôler. Le JDK 1.4 apporte un niveau d'abstraction beaucoup plus élevé pour les opérations de *drag and drop*. Cet ajout se situe au niveau Swing et non plus AWT. Comme à l'accoutumée, il ne remet pas en cause les développements précédents utilisant AWT : la compatibilité ascendante est respectée. Il est donc toujours possible d'utiliser l'implémentation présentée ci-dessus, qui permet de personnaliser chaque étape du *drag and drop* et de tout contrôler, au prix d'un code plus complexe.

Depuis le JDK 1.4, il est donc beaucoup plus facile d'implémenter un mécanisme de transfert d'information entre deux composants. Le même mécanisme est utilisé dans le cadre d'une opération de *drag and drop*, ou bien dans le cadre d'un copier-coller. Le copier-coller à l'aide des habituels raccourcis clavier (Ctrl-C et Ctrl-V) est disponible par défaut dans les composants qui affichent du texte. En revanche, la possibilité d'effectuer une opération de *drag and drop* doit être activée à l'aide de la méthode suivante : `setDragEnabled(true)`.

Voici un exemple très simple de *drag and drop* entre une liste et un champ texte, instance de `JTextField`. Par défaut, le composant `JList` est déjà programmé pour pouvoir être la source d'une opération de *drag and drop* en mode « copie ». Il sera donc très facile d'implémenter le *drag and drop* permettant de recopier les éléments sélectionnés de la liste vers le champ texte.

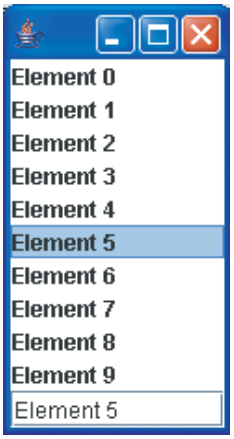


Figure 6.3 – Drag and drop simple entre une liste et un champ texte

Voici le code de cet exemple :

```
public class Liste extends JList {
     public static final int taille = 10;

     public Liste() {
          setDragEnabled(true);
         DefaultListModel model = new DefaultListModel();
          for (int i = 0; i < taille; i++) {
               model.addElement("Element " + i);
          }
          setModel(model);
     }
}
```

Il n'y a rien d'autre à ajouter. Le *drag and drop* en mode « copie » est dans ce cas automatique.

Si nous voulons ajouter à cette liste un drag en mode « déplacement » — l'équivalent d'un « couper-coller » —, ou toute autre logique supplémentaire, alors il faut ajouter du code et gérer l'opération sans l'aide des automatismes. Cela reste tout de même plus simple qu'en utilisant l'implémentation AWT que nous décrivions dans le paragraphe précédent.

La classe `TransferHandler` est responsable du transfert de données entre deux composants. Tout composant Swing dispose d'un `TransferHandler`, et peut donc être potentiellement la source ou la cible d'une opération de *drag and drop*.

Dans notre première version de l'exemple, nous avons utilisé implicitement le `TransferHandler` disponible sur le composant `JList`. Mais ce `TransferHandler` ne correspond pas tout à fait à nos besoins, car nous souhaitons effectuer un *drag and drop* en mode « déplacement ».

Nous allons spécifier un `TransferHandler` particulier, à l'aide d'une sous-classe.

Les objets que l'on souhaite transférer doivent implémenter l'interface `Transferable` comme avec l'implémentation AWT. La classe `StringSelection` implémente cette interface et correspond à une implémentation simple pour transférer une chaîne de caractère.

Voici donc le code de la liste permettant un *drag and drop* en mode copie et/ou déplacement. Nous utilisons la méthode `setTransferHandler()` qui prend en paramètre une instance de notre propre `TransferHandler`. Nous utilisons ici le mécanisme des classes anonymes qui est bien pratique.

```
public class Liste extends JList {
  public static final int taille = 10;
  private DefaultListModel modele;

  public Liste() {
    setDragEnabled(true);
    modele = new DefaultListModel();
    for (int i = 0; i < taille; i++) {
      modele.addElement("Element " + i);
    }
    setModel(modele);
    setTransferHandler(new TransferHandler() {
      protected Transferable createTransferable
      ➥(JComponent c) {
        return new StringSelection(
        ➥getSelectedValue().toString());
      }

      public int getSourceActions(JComponent c) {
        return COPY_OR_MOVE;
      }

      protected void exportDone(JComponent source,
                                Transferable data,
                                ➥int action) {
        if (action == DnDConstants.ACTION_MOVE) {
          try {
            String str = (String)data.getTransferData(
                         ➥DataFlavor.stringFlavor);
            modele.removeElement(str);
          } catch (UnsupportedFlavorException e) {
            e.printStackTrace();
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
    });
  }
}
```

La première méthode que nous redéfinissons est `createTransferable(…)` dont le rôle est de retourner une instance implémentant l'interface `Transferable` et contenant la donnée à transférer. Du fait que nous sommes à l'intérieur même du composant la tache se trouve simplifiée.

Ensuite, la méthode `getSourceActions(…)` spécifie la ou les actions de *drag and drop* qui seront autorisées. Ici, nous utilisons la constante `COPY_OR_MOVE` qui est un raccourci vers `DnDConstants.ACTION_COPY_OR_MOVE`.

Enfin, la méthode `exportDone(…)` est déclenchée lorsque le transfert est terminé. Dans le cas d'un drag en mode déplacement, il nous faut encore supprimer

la donnée de la liste de départ. La première étape consiste donc à se procurer cette donnée et la deuxième étape à s'adresser au modèle pour sa suppression. Notez que l'utilisation des `DataFlavor` est tout à fait semblable à l'implémentation du *drag and drop* AWT que nous décrivions précédemment.

Il n'est pas utile de redéfinir la méthode `importData(…)` car la liste ne reçoit pas de drop.

Testons notre code en utilisant un *drag and drop* en mode déplacement, qui est le mode par défaut. Ensuite, en maintenant la touche *control* enfoncée, si l'on est sous Windows, effectuons un *drag and drop* en mode copie. Notre code autorise bien les deux modes.

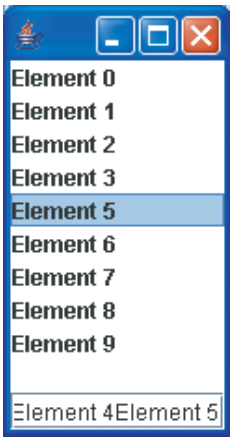


Figure 6.4 – Drag and drop en mode copie ou déplacement

Le tableau 6.1 donne un aperçu des capacités de *drag and drop* intégrées par défaut aux composants.

Tableau 6.1 – *Drag and drop*

Composant	Drag copie	Drag déplacement	Drop	Couper	Copier	Coller
JEditorPane	X	X	X	X	X	X
JTextField	X	X	X	X	X	X
JTextArea	X	X	X	X	X	X
JTable	X				X	
JTree	X				X	
JList	X				X	

Il faudra adopter le même type de code pour adjoindre des comportements différents en terme de *drag and drop* à d'autres composant Swing. Par exemple, `JTable` ne propose pas de comportement automatique en cas de Drop. En effet, quelles sont les cellules concernées par ce drop ? Quels types de données sont valides ? C'est au développeur de le préciser dans une sous-classe de `TransferHandler` qui lui est propre.

Notez une facilité proposée par la classe `TransferHandler` qui peut être utile : si l'information que vous souhaitez véhiculer par *drag and drop* correspond à une propriété, au sens propriété d'un JavaBean, c'est-à-dire avec un getter et un setter disponibles, il suffit d'utiliser le constructeur de `TransferHandler` prenant le nom de cette propriété en paramètre. Cela vous épargne l'écriture de la méthode `createTransferable`.

7

Concepts avancés

Nous allons maintenant aborder des concepts avancés tels que parallélisme, tests et multifenêtrage.

Pour ce faire nous allons partir de l'exemple principal et le rendre multifenêtres avec le `JDesktopPane`, le bureau multifenêtres de Swing.

En quoi est-ce important ?

Nous entendons souvent des réflexions sur la lenteur de Swing. Et pourtant, tout est là pour qu'une application Swing soit plus rapide et conviviale qu'une application de type Web :

- pas de contrainte réseau pour la partie purement client graphique.
- pas de texte à parcourir et interpréter.
- pas de navigateur Web.

Alors d'où vient cette impression de lenteur ? Commençons par faire des tests simples et classiques comme l'affichage d'une table avec 10 000 instances différentes, une par ligne. Naturellement, c'est plus lent qu'avec une seule ligne, mais c'est tout de même très rapide.

Pourtant, l'impression globale de lenteur ressentie par beaucoup est bien réelle. Pour chaque situation, il faut essayer de déterminer plus précisément ce qui est lent. Le résultat de ces investigations sera une aide précieuse pour le choix d'une architecture. Il arrive parfois que les choix se portent vers une architecture Web (interface graphique HTML) au détriment d'une application Swing uniquement pour cette raison, ressentie et non évaluée, de lenteur de Swing. S'il n'est pas question de changer la logique métier ni sa persistance dans une base de

données relationnelle, alors il ne reste plus qu'à gagner du temps sur le poste client. Swing peut se montrer véloce si l'on sait en titrer la substantifique moelle avec les threads. Ce n'est pas toujours un problème technique mais souvent un problème conceptuel : on ne pense pas à être asynchrone. Voici quelques pistes :

- Faire au plus tôt ce qui devra être fait. Préinstancier le maximum d'objets lors du chargement de l'application ou pendant l'étape de login en utilisant les threads.
- Ne pas faire ce qui n'est pas tenu d'être fait. Attendre que l'utilisateur le demande et penser au « *lazy loading* » dans ce cas.
- Penser à utiliser `setVisible()` plutôt que de laisser se détruire une boîte de dialogue qui pourra être réutilisée par la suite.
- Envisager des actions systématiquement asynchrones plutôt que de compliquer la solution avec des timers et des principes tels que « si une action est plus longue qu'un certain seuil, alors et alors seulement, l'exécution s'effectue en parallèle ».
  Tous les traitements peuvent se dérouler en arrière-plan et interagir avec l'interface graphique.

Nous verrons ensuite comment il est possible de tester une interface graphique. Les tests deviennent de plus en plus importants. Il en existe de nombreux types.

7.1 LES THREADS

Les threads impressionnent beaucoup, ils ont la mauvaise réputation d'être difficile d'appréhension. Essayons de prouver le contraire. Tout d'abord, qu'est-ce qu'un thread ?

Un thread est en Java un élément logiciel de parallélisme. Il est souvent nécessaire d'exécuter des blocs de code « en même temps » et les threads permettent de définir ces blocs de code. Un mécanisme présent dans la machine virtuelle Java utilise ces threads comme des descriptions et assure l'exécution « simultanée » des différents threads rencontrés.

Il faut être assez prudent lorsqu'on parle de « simultanéité ». En premier lieu, que signifie exécution simultanée sur une machine dotée d'un unique processeur ? Depuis longtemps déjà, le microprocesseur est utilisé en « temps partagé » par le système d'exploitation pour simuler une exécution simultanée des processus dont il a la charge. De fait, s'il y a deux processus à gérer, il suffit d'allouer 50 % du temps du microprocesseur à chacun d'eux pour obtenir l'apparence de la simultanéité. Naturellement, les choses sont bien plus complexes que cette introduction.

Le parallélisme mis en œuvre avec les threads est différent. La forme de parallélisme que nous évoquions se nomme multi processus alors que les threads permettent un parallélisme multi thread. Les threads ne se situent pas au même niveau : ils vivent dans un processus système. Quand un processus bénéficie du temps du processeur, chacun des threads qui le compose se partage à nouveau ce temps de sorte que les différents threads s'exécutent avec l'apparence de la simultanéité.

Cela étant posé, il apparaît évident que le programmeur ne doit pas faire d'hypothèse sur l'ordre d'exécution de différents threads. En d'autres termes, une fois deux threads définis, rien n'indique lequel s'exécutera en premier ou lequel se terminera en premier. De plus, si l'on observe qu'un thread se termine avant un autre au cours d'une exécution, rien n'indique que cela se reproduise lors de l'exécution suivante. À moins, bien sûr, qu'il ne s'agisse d'un rendez-vous explicitement programmé, ou synchronisation entre threads.

Les processeurs les plus récents dits *Hyper Threaded* sont particulièrement optimisés pour la gestion des threads. Aujourd'hui les processeurs *multi core* arrivent, augmentant encore leur capacité à gérer le parallélisme. Mais, naturellement, ces optimisations matérielles ne seront prises en compte que si le logiciel utilise à son tour le parallélisme.

7.1.1 Un premier exemple

En quoi les threads nécessitent-ils une attention particulière ?

Un cas intéressant et classique est celui dit du « producteur – consommateur ».

Prenons l'exemple d'un compte en banque utilisé dans deux threads différents. Imaginons une version simplissime de cette classe Compte :

```
public class Compte {

   private float solde;

   public float getSolde() {
      return solde;
   }

   public void setSolde(float solde) {
      this.solde = solde;
   }

   public void ajouter(float montant) {
      setSolde(getSolde() + montant);
   }
}
```

Imaginons maintenant que deux threads différents cherchent à ajouter respectivement 100 et 1000 euros. Le montant initial est de zéro. Que peut-il se passer ?

Voici le premier thread :

```
public class Thread1 extends Thread {

   private Compte compte = ...;

   public void run() {
      compte.ajouter(100);
   }
}
```

Une façon de définir un thread en Java est d'hériter de la classe `Thread`. Il est aussi possible d'utiliser directement une instance de la classe `Thread` et de lui passer en paramètre un objet implémentant l'interface `Runnable`. Après instanciation d'un objet `Thread1`, il sera possible d'exécuter le code défini dans la méthode `run()` de manière asynchrone.

Si nous appelons directement la méthode `run()` sur cette instance, l'exécution ne sera pas asynchrone mais synchrone. Pour obtenir le parallélisme, il faut utiliser la méthode `start()` qui est définie dans la super classe `Thread`. L'appel à la méthode `start()` déclenche le mécanisme du parallélisme et appelle ensuite notre méthode `run()` :

```
Thread t1 = new Thread1(); // ligne 1
// Cela va ensuite appeler notre methode run()
t1.start(); // ligne 2
System.out.println(…) // ligne 3
```

Ce qu'il faut bien comprendre, c'est que la ligne 3 va s'exécuter sans attendre la fin du traitement de notre méthode `run()`. L'action de créditer le compte en banque s'effectue dans un autre thread, en même temps. Les lignes 1 et 3 s'exécutent dans le thread par défaut, tandis que la ligne 2 lance un nouveau thread. Vous constatez maintenant — et ce constat n'est pas sans rappeler un certain monsieur Jourdain — qu'il y a toujours eu des threads, même lorsque le développeur ne les manipule pas explicitement. De la même façon, le *garbage collector* tourne dans un thread particulier, afin que notre application et la libération mémoire, s'exécutent en parallèle.

Pour pouvoir ajouter n'importe quelle somme au compte en banque, nous allons améliorer notre classe Thread1 afin qu'elle prenne en paramètre du constructeur le compte bancaire et la somme à créditer. Ainsi, chaque instance de cette classe représentera une opération bancaire particulière. Le code devient donc :

```
public class ThreadExemple extends Thread {

   private Compte compte;
   private float montant;
```

```
   public ThreadExemple(Compte compte, float montant) {
      this.compte = compte;
      this.montant = montant;
   }

   public void run() {
      System.out.println("Ajout de " + montant);
      compte.ajouter(montant);
   }
}
```

Pour utiliser notre thread exemple, il faut fournir un compte et le montant qui sera ensuite ajouté :

```
public class Main {

   public static void main(String[] args) {
      Compte c = new Compte();
      Thread t1 = new ThreadExemple(c, 100);
      Thread t2 = new ThreadExemple(c, 1000);
      System.out.println("Debut de main, le solde est
      ➥de "+c.getSolde());
      t1.start();
      t2.start();
      System.out.println("Fin de main, le solde est
      ➥de "+c.getSolde());
   }
}
```

Quel est le résultat de l'exécution ?

```
Debut de main, le solde est de 0.0
Fin de main, le solde est de 0.0
Ajout de 100.0
Ajout de 1000.0
```

Ainsi, le solde résultant est de 0 au lieu de 1100 escompté. Que s'est-il passé ?

7.1.2 L'attente active

Le thread principal (ou thread par défaut) se termine trop tôt, sans attendre que les opérations bancaires aient été effectuées. Après avoir lancé les deux threads, l'exécution se poursuit dans le thread principal, et nous arrivons à la fin de la méthode main : le programme se termine. Pour remédier à cela, nous devons donc attendre que les threads 1 et 2 soient terminés. Le plus simple est de leur demander s'ils ont fini leur exécution :

```
public class ThreadExemple extends Thread {

   private Compte compte;
   private float montant;
   private boolean fini = false;
```

```
    public ThreadExemple(Compte compte, float montant) {
       this.compte = compte;
       this.montant = montant;
    }

    public void run() {
       fini = false;
       System.out.println("Ajout de " + montant);
       compte.ajouter(montant);
       fini = true;
    }

    public boolean isFini() {
       return fini;
    }
}
```

Avec cette version, il devient possible de savoir si le traitement est terminé. Chaque instance de `ThreadExemple` positionne un booléen quand le traitement est terminé. Voici une nouvelle version du programme principale qui exploite ce booléen :

```
Compte c = new Compte();
ThreadExemple t1 = new ThreadExemple(c, 100);
ThreadExemple t2 = new ThreadExemple(c, 1000);
System.out.println("Debut de main, le solde est de "
➥+ c.getSolde());
t1.start();
t2.start();

while (!t1.isFini() || !t2.isFini()) {
}
System.out.println("Fin de main, le solde est de "
➥+ c.getSolde());
```

Une boucle `while` est ajoutée afin d'attendre que les deux threads soient terminés avant d'afficher le résultat. Voici l'affichage après une exécution :

```
Debut de main, le solde est de 0.0
Ajout de 100.0
Ajout de 1000.0
Fin de main, le solde est de 1100.0
```

Que devons-nous améliorer ? L'attente des deux threads est réelle mais inefficace. Rappelons la nuance entre ne rien faire et faire « rien ». Au lien de boucler à l'infini, ce qui n'est pas rien faire, jusqu'à ce que les deux threads soient terminés, essayons de ne réellement rien faire. Pour un thread, rien faire équivaut à une suspension pendant un laps de temps donné. Pour ce faire utilisons la méthode statique suivante : `Thread.sleep(attente en millisecondes)`, le résultat est une suspension du thread courant pendant un temps donné. Le thread ainsi

suspendu ne sollicitera pas le microprocesseur, ce qui est bien plus efficace. En effet, le temps ainsi libéré par cette suspension devient disponible pour les autres threads. Dans notre cas, nous laissons donc du temps processeur disponible pour les que les autres threads se terminent au plus vite. La boucle d'attente devient :

```
while (!t1.isFini() || !t2.isFini()) {
   try {
      // le thread qui execute cette instruction est
      ➥suspendu
      Thread.sleep(50);
   } catch (InterruptedException e) {
   }
}
```

Il faut bien sûr doser convenablement la durée de cette attente. Un temps d'attente trop long risque d'augmenter inutilement la durée totale d'exécution, puisque les deux threads peuvent avoir terminé bien avant que le thread principal sorte de son attente. Un temps d'attente trop court se rapproche de la situation avec la boucle à vide, que l'on appelle « attente active », puisque le test des booléens `isFini` s'effectue à intervalles très rapprochés.

7.1.3 Les sections critiques

Tout semble se dérouler au mieux mais nous devons prévoir le pire des scénarios pour que notre programme soit robuste. Du fait du temps partagé, chaque thread se voit allouer un certain temps processeur. Il faut donc prendre en compte la possibilité que le traitement d'un thread soit interrompu, que le processeur traite un autre thread, puis reprenne le thread interrompu. Tout cela pourrait être transparent, mais notre exemple simple présente en fait un problème face à ses coupures. Il se trouve que le temps d'exécution de `compte.ajouter()` est si faible que la situation n'arrive pas. En d'autres termes, le premier intervalle de temps alloué est déjà suffisant pour que chaque thread se termine et aucun problème ne se pose. Imaginons un instant qu'une coupure se produise malgré tout lors de l'exécution du premier thread. Comment se déroulent ses coupures ? Elles peuvent avoir lieu entre deux instructions de code compilé (Byte Code). Voyons à quoi ressemblent les méthodes `setSolde` et `ajouter` en Byte Code (le Byte Code est en commentaire après le code Java correspondant) :

```
public void setSolde(float solde)
{
     this.solde = solde;
//     0     0:aload_0
//     1     1:fload_1
//     2     2:putfield  #2 <Field float solde>
//     3     5:return
}
```

```
public void ajouter(float montant)
{
     setSolde(getSolde() + montant);
//     0     0:aload_0
//     1     1:aload_0
//     2     2:invokevirtual #3 <Method float getSolde()>
//     3     5:fload_1
//     4     6:fadd
//     5     7:invokevirtual #4 <Method void setSolde
              ➥(float)>
//     6    10:return
}
```

La méthode `ajouter` que nous écrivons en une ligne, peut aussi être écrite en deux instructions Java, pour se rapprocher davantage de la décomposition en instructions Byte Code :

```
public void ajouter(float montant) {
   float tmp = getSolde() + montant;
   // Coupure possible ici.
   setSolde(tmp);
}
```

Que se passe t-il si le traitement est coupé entre les lignes 2 et 5 de la méthode ajouter ? Ces deux lignes correspondant à un appel de méthode : `invokevirtual`. Le problème de concurrence survient du fait que la même instance de Compte est partagée entre les deux threads. Prenons un exemple :

Solde	Thread numéro 1	Thread numéro 2	Commentaire
0	`aload_0`		Le thread numéro 1 est alloué au processeur, il faut ajouter 100
0	`aload_0`		
0	`Invokevirtual #3 <Method float getSolde()>`		Appel à la méthode getSolde qui renvoie 0
0	`fload_1`		Création d'une variable temporaire de type float
0	`fadd`		`tmp = getSolde()+100`, `tmp` vaut donc 100
0		`aload_0`	le thread numéro 2 est alloué au processeur et le thread numéro 1 est suspendu
0		`aload_0`	

Solde	Thread numéro 1	Thread numéro 2	Commentaire
0		`Invokevirtual #3 <Method float getSolde()>`	getSolde renvoie toujours 0
0		`fload_1`	Création d'une nouvelle variable temporaire tmp
0		`fadd`	`tmp = getSolde()+1000`, tmp vaut donc 1000
1000		`Invokevirtual #4 <Method void setSolde(float)>`	`setSolde(1000)`
1000		`return`	
100	`Invokevirtual #4 <Method void setSolde(float)>`		Le thread numéro 1 prend à nouveau la main et se termine par `setSolde(100)`
100	`return`		

La valeur initiale du solde du compte est 0. Le thread numéro 1 s'exécute jusqu'à `tmp = getSolde()+100`. Quel est le solde du compte à cet instant précis ? 100 ? Non, 0 car le thread numéro 1 a été coupé avant de pouvoir affecter le solde. Le thread numéro 2 prend le relais mais lui n'est pas interrompu. Le solde du compte est maintenant fixé à 1000 dans l'instance du compte. Or cette instance est partagée par les deux threads. Lorsque le thread numéro 1 reprend son exécution, il ignore que le montant du compte a changé entre-temps. En d'autres termes, un thread ne « sait » pas qu'il a été interrompu. Le code se termine par l'exécution de `setSolde(tmp)`, soit `setSolde(100)`.

Il n'est donc pas possible de modifier impunément une unique ressource partagée par des traitements concurrents. De fait, quand de telles situations de produisent dans la vie de tous les jours, il faut faire la queue. Si nous considérons un distributeur automatique de billets comme une ressource, les clients font la queue, il n'est pas possible d'utiliser à plusieurs le même clavier !

Comment organiser une queue d'attente pour des threads ? Il faut mettre en place des sections critiques protégées par des sémaphores, en Java cela correspond au mot clé `synchronized`.

Pour illustrer ce qui se passe et comprendre pourquoi ce genre de verrou se nomme sémaphore, nous pouvons prendre l'exemple de trains. Deux voies permettent à deux trains de se croiser. Malheureusement, à cause d'une montagne ou d'un pont par exemple, il faut creuser un tunnel ne permettant le passage

que d'un train à la fois. Il y a donc une section à voie unique : le tunnel. Il faut empêcher des trains d'entrer en collision sur la voie unique du tunnel par des sémaphores. Dans cet exemple, le tunnel, la ressource partagée, est protégé par les sémaphores.

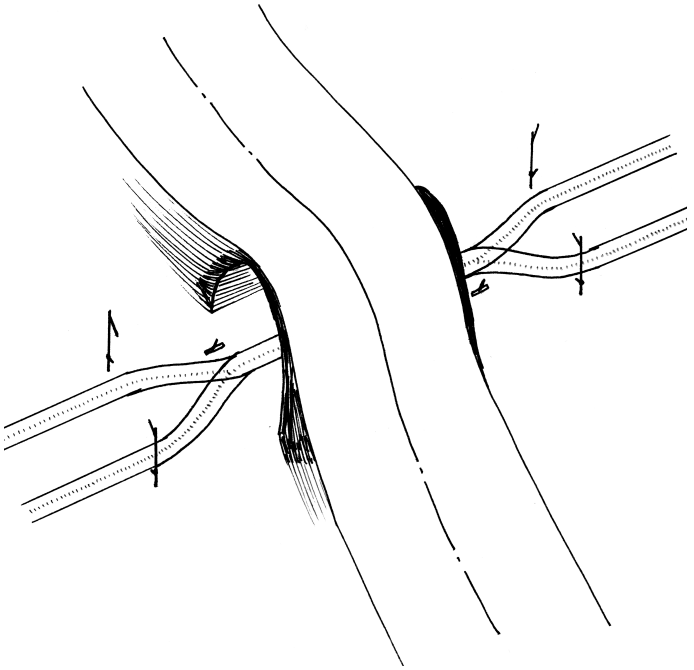


Figure 7.1 – Illustration ferroviaire des sémaphores

Dans le code, où devons nous mettre un sémaphore ? La ressource est l'instance de `Compte` et plus précisément son solde. Trois méthodes en manipulent la valeur : `getSolde()`, `setSolde(…)` et `ajouter(…)`. La méthode `getSolde()` ne pose pas de problème car il est possible d'accéder sans protection à une valeur en lecture seule. C'est l'exemple du panneau annonçant le départ des trains dans une gare. Tout le monde peut consulter en même temps les informations, il n'y a pas de queue.

La méthode `setSolde(…)` modifie la valeur du solde mais n'est appelée que par la méthode `ajouter(…)`. De plus puisque nous souhaitons renforcer la fiabilité de notre code, nous devrions envisager que la méthode `setSolde(…)` soit `private`. De cette façon, il n'est plus possible d'y accéder de l'extérieur.

Nous devons donc protéger la méthode `ajouter(…)`. Afin de bien comprendre en quoi elle sera protégée, il nous faut comprendre le mécanisme d'action des sémaphores.

Lorsqu'une méthode est synchronisée, tout appel à cette méthode subit un test. Le sémaphore est-il ouvert ? Si le sémaphore n'est pas ouvert, le thread effectuant l'appel est mis en attente jusqu'à ce que le sémaphore soit ouvert. En fait, cela signifie qu'un autre thread est actuellement en train d'exécuter cette méthode.

```
public class Compte {

   private float solde;

   public float getSolde() {
      return solde;
   }

   private void setSolde(float solde) {
      this.solde = solde;
   }

   public synchronized void ajouter(float montant) {
      setSolde(getSolde() + montant);
   }
}
```

Ainsi, lorsque le thread numéro 1 commence son exécution, il acquiert le jeton lorsque la méthode `ajouter` est appelée. Même si le thread numéro 2 prend la main avant l'affectation `setSolde(100)`, il ne peut pas démarrer l'exécution de la méthode `ajouter` car le jeton n'est pas disponible. Il « rend » donc la main au processeur très rapidement (il est mis en attente). Il doit attendre que l'exécution de la méthode `ajouter` par le thread numéro 1 soit terminée, et donc que le jeton soit libéré. De cette façon, même en cas de coupure au « mauvais endroit », le solde sera correct.

Il est donc possible d'envisager du parallélisme, y compris sur des ressources partagées, mais il faut les protéger par des sémaphores.

7.1.4 Optimisation de l'attente active

Est-il encore possible d'optimiser notre attente ? Nous pourrions envisager un tout autre principe de fonctionnement. Au lieu d'attendre un temps inconnu qu'un événement se produise, essayons d'intervenir quand cet événement se produit. Pourquoi ne pas chercher à signaler au thread principal que le traitement est terminé ? Si nous y parvenions, nous n'aurions plus besoin de demander incessamment à chaque thread si son traitement est terminé. Pour cela, utilisons les méthodes `wait()` et `notify()`. La méthode `wait()` va mettre en attente le thread qui exécute cette instruction jusqu'à ce que la méthode `notify()` soit appelée. Les méthodes `wait` et `notify` ne sont pas des méthodes du thread

lui-même, elles sont définies sur la classe `Object` : un objet, pouvant être partagé entre différents threads, est utilisé comme verrou et c'est sur lui que sont appelées ces méthodes. N'importe quel type d'objet peut être utilisé pour jouer le rôle de ce verrou. Ici, afin de clarifier l'exemple, un objet dédié à cet usage est utilisé : `verrou`.

Voici la méthode principale :

```
Object verrou = new Object();
Compte c = new Compte();
ThreadExemple t1 = new ThreadExemple(verrou, c, 100);
ThreadExemple t2 = new ThreadExemple(verrou, c, 1000);
System.out.println("Debut de main, le solde est de "
➥+ c.getSolde());
t1.start();
t2.start();

while (!t1.isFini() || !t2.isFini()) {
   synchronized (verrou) {
      try {
         verrou.wait();
      } catch (InterruptedException e) {
      }
   }
}
```

Cet objet `verrou` est passé à chaque thread lors de leur création. Après cela, à la suite du lancement des deux threads, le thread principal est mis en attente par l'appel `verrou.wait()`.

Pour pouvoir utiliser une des trois méthodes `wait()`, `notify()` et `notifyAll()`, il faut avoir acquis un jeton sur l'objet. Pour faire cela, on définit une section critique ou section synchronisée. Ces sections fonctionnent sur le même principe qu'une méthode synchronisée mais à l'échelle d'un bloc de code. Une autre différence importante est que nous avons ici le choix de l'objet sur lequel le jeton est acquis : `verrou`. Dans le cas d'une méthode synchronisée, cet objet est l'instance courante, `this`. Un thread ne peut exécuter une section critique que s'il a acquis un jeton sur l'objet en question. Dans le cas contraire, il est mis en attente jusqu'à obtention du jeton.

Différents scénarios peuvent avoir lieu :

1. Les deux threads s'exécutent si vite qu'ils sont déjà terminés, quand le thread principal arrive à la boucle `while`. Dans ce cas, le test d'entrée dans la boucle ne se vérifie pas et le thread principal ne se met pas en attente.
2. L'un des deux seulement est terminé à cette étape.
3. Aucun n'est terminé et ils s'exécutent toujours quand nous arrivons à cette étape d'attente.

Le deuxième cas suppose que l'une au moins des deux conditions de la boucle `while` soit vérifiée. Nous entrons donc dans la boucle et le thread courant est mis en attente. Pendant ce temps le thread qui n'est pas terminé poursuit son exécution. Il faut maintenant trouver un moyen de réveiller le thread principal lorsque `t1` ou `t2` se termine. Voici donc la nouvelle version de la classe `ThreadExemple`.

```
public class ThreadExemple extends Thread {

   private Object verrou;
   private Compte compte;
   private float montant;
   private boolean fini = false;

   public ThreadExemple(Object verrou, Compte compte,
   ➥float montant) {
      this.verrou = verrou;
      this.compte = compte;
      this.montant = montant;
   }

   public void run() {
      fini = false;
      System.out.println("Ajout de " + montant);
      compte.ajouter(montant);
      fini = true;
      synchronized (verrou) {
        verrou.notify();
      }
   }

   public boolean isFini() {
      return fini;
   }
}
```

Notons tout d'abord le constructeur qui prend maintenant un paramètre supplémentaire : le verrou. Observons maintenant la fin de la méthode `run()`. Le verrou est acquis à l'aide d'un bloc `synchronized`, et un des threads éventuellement en attente sur ce verrou sera réveillé par l'instruction `notify`.

Vous vous demandez sans doute comment on peut enter dans ce bloc `synchronized`, puisque le verrou était possédé par le thread principal ? La mise en attente par un `wait` a la particularité de libérer le verrou, qui pourra ainsi être utilisé par un autre thread qui, lui, pourra alors tenter de réveiller sa Belle au bois dormant : le thread principal. Attention, une fois réveillé, le thread principal doit de nouveau acquérir le jeton pour pouvoir continuer son exécution. Il faut toujours garder à l'esprit qu'un troisième thread (ni la belle, ni le prince, une fée donc…) puisse intervenir dans la compétition qui a lieu pour acquérir le jeton sur le verrou.

Le troisième cas n'est pas tellement plus complexe au premier abord. C'est lui qui justifie la boucle `while`. Le début du programme peut se dérouler de la façon suivante :

- Thread principal : démarrage de la méthode main, puis mise en attente avec `wait`
- Thread t1 : démarrage de la méthode `run`
- Thread t2 : démarrage de la méthode `run`
- Thread t1 : fin de la méthode `run`, réveil du thread principal avec `notify`

Ici, deux scénarios sont possibles : le jeton étant libéré par t1, il devient disponible pour l'un des deux threads qui sont encore en train de tourner. Le fait que le thread principal vienne juste d'être réveillé ne lui procure aucune prédisposition pour prendre le jeton en premier : les deux scénarios sont équiprobables.

Examinons les cas possibles :

- Le thread principal prend le jeton et exécute la condition de la boucle `while`, pendant que t2 est toujours en train de tourner. Dans ce cas, les choses sont simples : le thread principal est mis en attente car la condition du `while` est toujours vraie, et sera réveillé par le `notify` à la fin de l'exécution de t2.
- Le thread t2 devance le thread principal en prenant le jeton à ce moment précis et finit son exécution avant que le thread principal n'ait pu faire quoi que ce soit. Lorsque le thread principal reprend la main, il teste à nouveau la condition du `while` qui est maintenant fausse et il termine normalement son exécution.
- Un dernier cas de figure n'est pas géré correctement par notre programme : en effet, imaginez que le thread principal reprenne la main le temps d'exécuter la condition du `while` (qui est vraie à ce moment-là), puis il est interrompu et c'est t2 qui reprend la main. Il acquiert le jeton qui est toujours libre et termine son exécution. L'instruction `notify` ne réveille personne puisque le thread principal n'est pas encore entré dans le `wait`. Lorsque le thread principal reprend la main, il est mis en attente et il n'y a plus personne pour le réveiller. Nous aboutissons à une situation de blocage, et c'est ce type de piège qu'il faut apprendre à détecter lorsqu'on fait de la programmation multithread.

Afin de corriger le problème, nous devons inclure la boucle `while` et son test dans la section critique. Ainsi corrigé, le scénario catastrophe ne peut plus se produire.

```
t1.start();
t2.start();

synchronized (verrou) {
  while (!t1.isFini() || !t2.isFini()) {
    try {
      verrou.wait();
    } catch (InterruptedException e) {
    }
  }
}
```

Cet exemple a été construit et volontairement présenté progressivement pour bien montrer le côté subtil et parfois déroutant des threads. Le développeur est donc encouragé à être particulièrement paranoïaque et soupçonneux : il faut toujours envisager tous les scénarios possibles, et pas seulement celui du prince qui réveille la Belle au bois dormant.

Notez aussi que la position dans le code de la ligne `fini=true` dans `Thread Exemple` est très importante.

```
<du code>
fini = true;
synchronized (verrou) {
  verrou.notify();
}
```

Nous avons choisi de positionner cette affectation avant la section critique, de sorte que le thread principal fraîchement réveillé puisse utiliser la méthode `isFini()` sans danger. Si nous l'avions placée après la section critique, une situation de blocage était possible : t2 interrompu juste après la section critique, le thread principal reprend la main et se remet en attente puisque le booléen fini n'a pas encore été positionné à `true`.

Une dernière remarque concernant le parallélisme : ce type de code est particulièrement difficile à debugger. Plusieurs exécutions consécutives sont susceptibles de produire des résultats différents selon que les jetons sont acquis par tel ou tel thread plus rapidement. Il est de ce fait parfois très délicat de reproduire certains problèmes. L'usage du debugger de votre outil de développement peut éclipser les problèmes par une gestion différente du parallélisme. La machine virtuelle Java est plus lente en mode *debug* et les jetons ne seront probablement pas acquis avec la même dynamique que sans le debugger. Enfin, tout ajout de traces de type `System.out.println` aura tendance à synchroniser le code sur le flux de sortie `out`. Ici encore le comportement sera différent avec et sans les traces. Plus il y aura de traces, plus le parallélisme s'éteindra car tous les threads en cours devront attendre d'avoir accès au flux out pour y écrire. Cette réflexion s'étend au système de *logs* qui est utilisé. Le problème du parallélisme et de son extinction par le système de log choisi est presque toujours occulté. Il est vrai que concevoir

un simple système de log peut devenir une tâche complexe si l'on y ajoute la contrainte du parallélisme car il faut rendre les appels type `log.warn(…)` asynchrones.

7.2 PARALLÉLISME ET MULTIFENÊTRAGE

Nous en savons maintenant assez sur les threads pour nous attacher à leur utilisation dans une application avec une interface Swing. Nous évoquions en introduction des raisons matérielles (processeur *Hyper Threaded*) pour démontrer l'intérêt du multithreading, mais il n'est pas nécessaire d'être équipé de tels processeurs pour en justifier l'usage. Dès lors que l'utilisateur demande une action au système au travers d'un menu ou d'un bouton pourquoi ne pas rendre la main à l'interface graphique ? Si nous ne le faisons pas, nous utilisons le thread principal, le thread graphique, pour exécuter notre code applicatif. Pendant ce temps d'exécution, aussi court soit il, le thread graphique ne peut plus gérer l'écran, et en conséquence l'interface n'est plus rafraîchie. C'est dans ce cas de figure que les utilisateurs peuvent se plaindre d'une « lenteur » de réaction de l'interface graphique. Il faut donc paralléliser les tâches : laisser le thread graphique s'occuper de l'interface graphique (rafraîchissement, détection des actions de l'utilisateur par exemple) et utiliser un thread applicatif pour nos traitements.

L'utilisation du multifenêtrage est une raison supplémentaire qui vient s'ajouter en faveur du parallélisme. En effet, les interfaces graphiques modernes comportent plusieurs fenêtres indépendantes. Par exemple un éditeur de documents gère plusieurs documents au sein d'une fenêtre principale. Cela permet à l'utilisateur de lire ou modifier plusieurs documents sans avoir à fermer puis ouvrir l'application. En d'autres termes, l'application n'est plus dédiée à un seul et unique document à un moment donné, même si l'utilisateur ne peut modifier qu'un seul document à un instant précis car il ne dispose que d'un seul clavier. Il y a donc toujours une seule sous-fenêtre active au sein de la fenêtre principale. Prenons l'exemple de la sauvegarde d'un document : ce traitement peut avoir une durée non négligeable. Faut-il pour autant bloquer toute l'application lorsqu'une sauvegarde d'un seul document est demandée ? Pourquoi ne pas paralléliser cette sauvegarde et laisser l'application libre d'entamer d'autres actions sur d'autres documents ? Dans ce cas, une unique fenêtre active ne signifie pas une seule action en cours.

Avant d'implémenter le parallélisme, voyons tout d'abord comment sont gérées ces différentes fenêtres au sein de la fenêtre principale ? Il existe différentes options ergonomiques. Par exemple, on peut avoir plusieurs fenêtres visibles en même temps sur le bureau : elles se partagent l'espace disponible sans se chevaucher.

L'option dite MDI (Multiple Document Interface) permet à l'utilisateur de retailler les fenêtres comme il le souhaite, la fenêtre active n'occupant pas forcément tout l'espace disponible. C'est le mode du `JDesktopPane`.

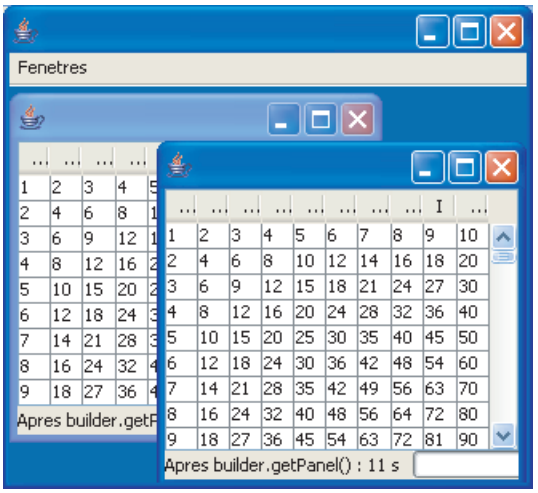


Figure 7.2 – Un exemple d'application multi fenêtres utilisant le `JDesktopPane`.

L'option « fenêtre ancrable » (*docking window*) permet de déplacer par *drag and drop* une fenêtre d'une zone à une autre. Une zone peut être vue comme un point d'ancrage de la fenêtre.

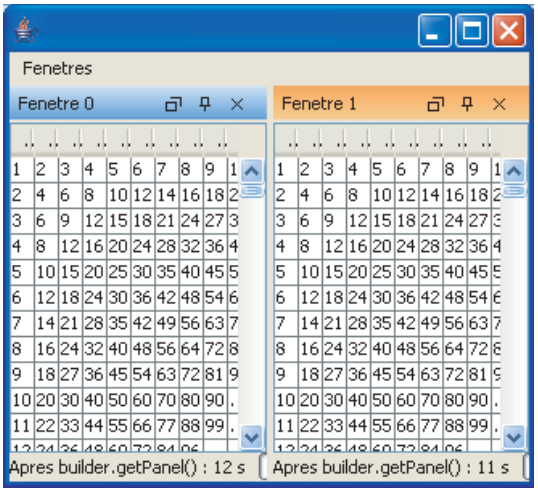


Figure 7.3 – Exemple du mode « fenêtre ancrable ».
Les fenêtres occupent des zones verticales.

On peut changer l'organisation des fenêtres par drag and drop. Ainsi, pour passer d'un agencement vertical à un agencement horizontal, il faut agir sur l'onglet et le déplacer vers la future zone verticale.

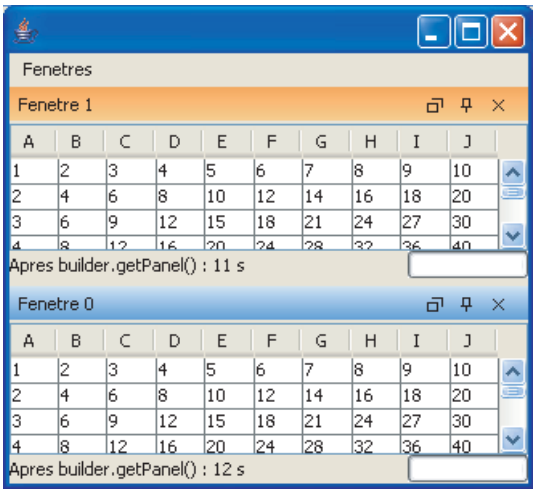


Figure 7.4 – Exemple du mode « fenêtre ancrable ». Les fenêtres occupent des zones horizontales.

Dans chaque zone, on peut avoir plusieurs fenêtres ouvertes : la fenêtre active est affichée au premier plan, les autres sont comme empilées « derrière » la fenêtre active et sont accessibles au moyen d'onglets.

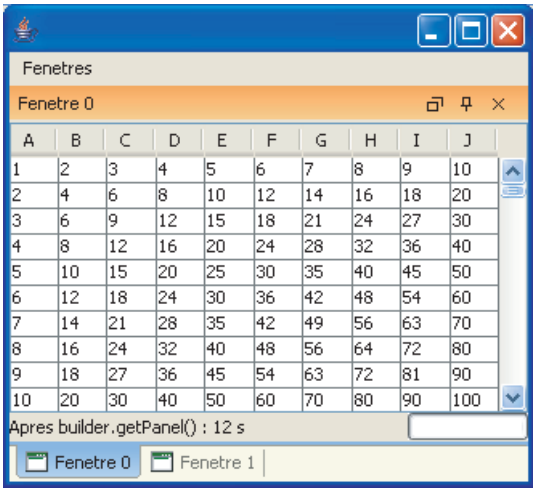


Figure 7.5 – Plusieurs fenêtres occupent la même zone.

Cette ergonomie est assez courante, elle est utilisée par exemple dans Eclipse. Nous utilisons ici la librairie de Jide software.

Dès lors qu'il est possible d'ouvrir plusieurs fenêtres dans une application, l'utilisateur est susceptible de lancer une action dans une première fenêtre puis très rapidement ensuite, une autre action dans une seconde fenêtre. Pour ne pas créer de temps d'attente, il faut que l'interface graphique soit immédiatement disponible après la première action, et que l'utilisateur puisse reprendre la main et déplacer le curseur pour effectuer la seconde action. Cette seconde action doit démarrer immédiatement. Pour cela, il est nécessaire d'utiliser le parallélisme, et d'allouer un thread pour chaque action déclenchée par l'utilisateur. Le thread gérant l'interface graphique sera toujours disponible et les différentes tâches applicatives pourront s'exécuter en parallèle.

JInternalFrame et JDesktopPane

Avant d'implémenter ce parallélisme, commençons par bâtir une interface graphique multifenêtres. Les fenêtres internes, `JInternalFrame`, évoluent au sein d'un bureau, le `JDesktopPane` qui peut être vu dans un premier temps comme un simple container. Notre application peut donc être composée d'une fenêtre principale classique au sein de laquelle le bureau, `JDesktopPane`, est en place. Une instance de `JFrame` et un `BorderLayout` feront l'affaire. La fenêtre principale contient également une barre d'état pour afficher diverses informations sur le statut de l'application.

```
public class FenetrePrincipale extends JFrame {
    private JDesktopPane desktopPane;
    private JLabel status;

    public FenetrePrincipale() throws HeadlessException {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.out.println("Au revoir !");
                System.exit(0);
            }
        });
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON
        ➥_CLOSE);
        desktopPane = new JDesktopPane();
        setLayout(new BorderLayout());
        add(desktopPane, BorderLayout.CENTER);
        status = new JLabel();
        add(status, BorderLayout.SOUTH);
        setSize(640, 480);
        addMenus();
    }
}
```

Comment ajouter une fenêtre interne ? Au lieu de prévoir du code au cas par cas selon les fenêtres que notre application devra gérer, pourquoi ne pas immédiatement concevoir un système plus générique ? Prévoyons donc tout de suite une sous-classe de JInternalFrame qui nous est propre.

```
public class FenetreInterne extends JInternalFrame {
     private JLabel status;
     private JProgressBar progress;

     public FenetreInterne() {
          setResizable(true);
          setClosable(true);
          setMaximizable(true);
          setIconifiable(true);
          setLayout(new BorderLayout());
          status = new JLabel("init");
          JPanel statusBar = new JPanel();
          statusBar.setLayout(
            new FormLayout("left:pref:grow, 5dlu, 50dlu",
            ➥"pref"));
          CellConstraints cc = new CellConstraints();
          statusBar.add(status, cc.xy(1, 1));
          progress = new JProgressBar(JProgressBar.
          ➥HORIZONTAL, 0, 100);
          statusBar.add(progress, cc.xy(3, 1));
          add(statusBar, BorderLayout.SOUTH);
          progress.setIndeterminate(true);
     }

     public void init() {
          <du code>
     }
}
```

Cette conception présente plusieurs avantages. Notons dès maintenant que toutes les fenêtres internes auront la même apparence dès lors qu'elles hériteront de `FenetreInterne`. Nous avons, ici encore, ajouté une barre d'état ainsi qu'une barre de progression. Les fenêtres pouvant évoluer indépendamment les unes des autres, chacune d'elles doit posséder sa propre barre d'état et de progression.

Afin que toute fenêtre interne soit ajoutée au bureau de la même manière, ajoutons une nouvelle méthode à `FenetrePrincipale`.

```
     public void nouvelleFenetre() {
          FenetreInterne fenetre = new FenetreInterne();

          fenetre.setSize(desktopPane.getSize());
          desktopPane.add(fenetre);
          fenetre.setVisible(true);
          fenetre.moveToFront();
          fenetre.init();
     }
```

La fenêtre interne est donc successivement instanciée, taillée puis ajoutée au bureau. Le bureau, `desktopPane`, n'est donc qu'un container de fenêtres internes. Notez que le fait d'ajouter une fenêtre interne ne la rend pas implicitement visible : il faut utiliser la méthode `setVisible`. On s'assure ensuite que la nouvelle fenêtre interne prend le focus en appelant `fenetre.moveToFront()`. Enfin, la méthode `init()` est utilisée pour que la fenêtre construise son contenu.

La fenêtre principale étant unique, il n'y a pas de risque à placer en son sein du code valable pour toute l'application. Cette classe joue le rôle de singleton graphique pour l'application.

Pour ajouter une nouvelle fenêtre interne il faudra donc :

- Écrire une sous-classe de `FenetreInterne`,
- Se procurer une référence sur la fenêtre principale,
- Utiliser la méthode `nouvelleFenetre()`.

Le problème est que cette méthode `nouvelleFenetre` n'est pas tellement générique, car l'instruction `new FenetreInterne()` est codée en dur. Si nous décidions d'utiliser un autre type de fenêtre interne, il faudrait changer cette méthode `nouvelleFenetre` et donc éventuellement ne plus être compatible avec du code existant. Notre code ne doit pas contraindre le développeur à utiliser la classe `FenetreInterne`. Nous pouvons même aller plus loin et rendre notre code indépendant de l'utilisation de `JInternalFrame`. En effet, il existe sur le marché d'autres librairies de classes graphiques et il est toujours préférable de laisser le code « ouvert », c'est-à-dire que de futures versions de l'application pourront utiliser d'autres composants graphiques, sans devoir reprendre le développement de zéro.

Comment améliorer notre code ? Introduisons une *fabrique*. Une fabrique est un *design pattern* très utilisé : il permet d'isoler l'instanciation d'objets dans une classe à part et facilite l'indépendance des différentes couches logicielles. Notre fabrique a la responsabilité de produire un container simple (`JPanel`) prêt à être ajouté dans une fenêtre. Si nous avons un `JPanel`, contenant un formulaire ou tout autre chose dont notre application a besoin, nous serons capables de l'ajouter à n'importe quel container racine, `JInternalFrame` notamment, mais aussi `JFrame`, `JDialog`, ...

Voici donc la super-classe de toutes les fabriques.

```
public abstract class Fabrique {
     public final JComponent getPanel() {
          JComponent resultat = doGetPanel();
          return resultat;
     }

     public abstract JComponent doGetPanel();
}
```

Pourquoi utiliser les méthodes `getPanel()` et `doGetPanel()` ?
Il aurait été plus simple de proposer une méthode `public abstract JPanel getPanel()`.Cette conception à base d'une méthode publique appelant une méthode abstraite *protected* présente de nombreux avantages.
Le premier est que la méthode qui est publique, ici `getPanel()`, peut-être *final* et est ainsi protégée contre des redéfinitions intempestives.
Le principal avantage est une meilleure extensibilité. Imaginons que cette version très simple soit rapidement complétée, il sera probablement nécessaire d'effectuer des actions avant et/ou après l'appel à la méthode `doGetPanel()` de la sous-classe. Si nous avions opté pour la version simple, nous devrions maintenant supposer sur les sous-classes appellent toutes `super.getPanel()`. Sans cet appel, que nous ne pouvons pas garantir, nos actions définies au niveau `Fabrique` ne seront pas exécutées. Avec le système de la méthode `doGetPanel`, l'encapsulation est renforcée car il n'est pas nécessaire de connaître la logique interne de `Fabrique` pour l'utiliser. Lors de la création d'une sous-classe, le compilateur rappellerait à l'ordre le développeur qui aurait oublié de coder la méthode `doGetPanel`. Enfin, il lui est impossible de redéfinir la méthode `getPanel`, l'appel à sa méthode `doGetPanel` est donc garanti.

Construire une fenêtre dans notre application consistera donc en une sous-classe de `Fabrique`. Voici un diagramme UML afin d'éclaircir cette conception.

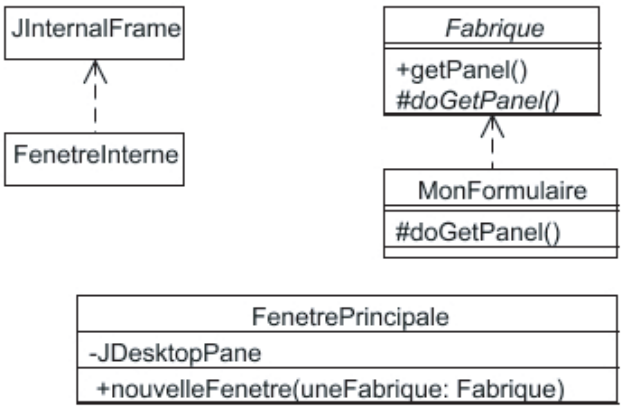


Figure 7.6 – Schéma UML de notre conception pour le multifenêtrage

L'avantage de cette conception est que nos containers principaux peuvent être génériques et ne dépendent plus du contenu spécifique qu'ils doivent gérer. Un nouveau contenu, un formulaire dans une fenêtre interne par exemple,

nécessitera une sous-classe de `Fabrique` dans laquelle on codera le formulaire proprement dit.

Voici maintenant le code de la méthode `nouvelleFenetre` qui implémente ce mécanisme.

```
public void nouvelleFenetre(Fabrique fabrique) {
    FenetreInterne fenetre = new FenetreInterne(fabrique);

    fenetre.setSize(desktopPane.getSize());
    desktopPane.add(fenetre);
    fenetre.setVisible(true);
    fenetre.moveToFront();
    fenetre.init();
}
```

Finalement, le constructeur de `FenetreInterne` gère une instance de Fabrique particulière comme un nouvel attribut. Voici le code de ce constructeur.

```
public FenetreInterne(Fabrique fabrique) {
     <…>
     this.fabrique = fabrique;
     <…>
}
```

Tout se passe donc dans la méthode `init()` de `FenetreInterne`.

```
public void init() {
     JComponent panel = fabrique.getPanel();
     add(panel, BorderLayout.CENTER);
}
```

En résumé, voici les différentes étapes :

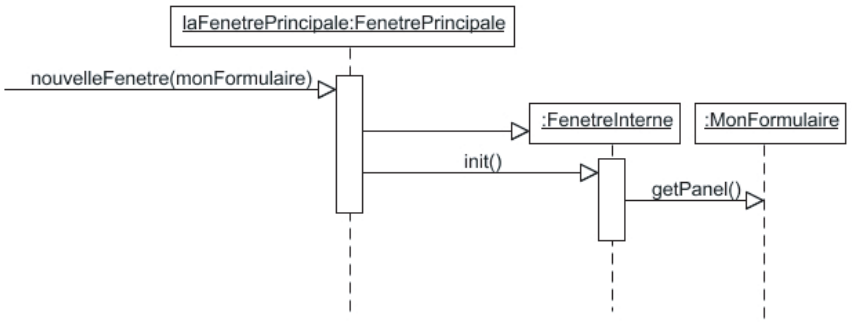


Figure 7.7 – Schéma UML de la séquence de création d'une fenêtre

- Une instance de `MonFormulaire`, sous-classe de `Fabrique`, est créée.
- Cette instance est passée à la méthode `nouvelleFenetre(Fabrique fabrique)` de l'instance unique de `FenetrePrincipale` (c'est un singleton).
- Cette méthode instancie une `FenetreInterne`, lui passe la `Fabrique` (en fait l'instance de `MonFormulaire`) et appelle la méthode `init()` de la `FenetreInterne` qui, enfin, utilise la méthode `JPanel getPanel()` définie sur la `Fabrique`.

Nous pourrions coder cette séquence de code dans une action qui aurait la charge d'afficher une fenêtre contenant le formulaire.

Ici le code devient très simple et tient en une ligne : `getFenetrePrincipale().nouvelleFenetre(new MonFormulaire())`. Un seul point reste dans l'ombre, comment obtenons-nous l'instance de la fenêtre principale ? Nous avons ici une action abstraite qui se charge de ce problème via la méthode `getFenetrePrincipale()`.

```
public class MonFormulaireAction extends ActionAbstraite {
     public MonFormulaireAction () { }

     protected void doActionPerformed(ActionEvent e) {
          getFenetrePrincipale().nouvelleFenetre(new
          ➥MonFormulaire ());
     }
}
```

De fait, l'implémentation au niveau abstrait est des plus simples :

```
protected FenetrePrincipale getFenetrePrincipale() {
     return Main.getFenetrePrincipale();
}
```

L'instanciation de la fenêtre principale se fait dans la classe Main.

```
public class Main {
    private static FenetrePrincipale fenetrePrincipale;

    public static void main(String[] args) {
        try {
              UIManager.setLookAndFeel(
              ➥UIManager.getSystemLookAndFeelClassName());
        } catch (Exception e) {
             e.printStackTrace();
        }
        fenetrePrincipale = new FenetrePrincipale();
        fenetrePrincipale.setVisible(true);
    }
```

```
    public static FenetrePrincipale getFenetre
    ➥Principale() {
        return fenetrePrincipale;
    }
}
```

Ici encore, le code est simple. Notons le positionnement du *look and feel* système pour que l'utilisateur trouve une interface qui ne le dépayse pas trop.

La fenêtre principale est donc un singleton qui est référencé par un attribut statique dans la classe Main.

Nous voici finalement dans la description d'un véritable *framework* graphique. Nous avons à notre disposition :

- Une structure de démarrage dans la classe Main,
- Une fenêtre principale, singleton, accessible de n'importe où par `Main.getFenetrePrincipale()`,
- Un système abstrait de fabrique de panel,
- Une fenêtre interne générique capable de se construire à partir d'une instance d'une sous-classe de fabrique.

Le code source est disponible intégralement sur le site de l'éditeur : www.dunod.fr

Voici un résumé des modifications nécessaires pour utiliser les fenêtres ancrables de la librairie de Jide software. Ces modifications permettront de valider l'ouverture de notre framework.

La classe `FenetrePrincipale` doit sous-classer `DefaultDockingHandler` et non plus `JFrame`. Voici la nouvelle version de la méthode `nouvelleFenetre`.

```
public void nouvelleFenetre(Fabrique fabrique) {
    final FenetreAncrable fenetre = new FenetreAncrable
    ➥(fabrique);
    fenetre.setKey("Fenetre " + numFen);
    numFen++;
    getDockingManager().addFrame(fenetre);
    fenetre.init();
}
```

La nouvelle classe `FenetreAncrable` se définit comme suit :

```
public class FenetreAncrable extends DockableFrame
➥implements FenetreAbstraite {
}
```

Les différences entre cette classe et `FenetreInterne` sont infimes.

Nous pouvons ici mesurer le bénéfice de notre framework : nous réutilisons entièrement les fabriques qui constituent le cœur de l'applicatif. Les classes que nous avons dû modifier sont purement techniques.

Ajoutons le parallélisme

Codons une nouvelle fenêtre via une sous-classe de `Fabrique` et arrangeons-nous pour que l'exécution soit volontairement très longue, cela afin d'illustrer la nécessité du parallélisme. Pour faire simple, remplissons une `JTable` et utilisons l'instruction `Thread.sleep` pour ralentir encore l'exécution.

```
public class ListeNombre extends Fabrique {
     private final int COL = 10;
     private final int LIG = 10 000;

     public JComponent doGetPanel() {
          JPanel panel = new JPanel();
          panel.setLayout(new BorderLayout());
          final JTable table = new JTable(LIG, COL);
          final TableModel model = table.getModel();

          for (int i = 1; i <= LIG; i++) {
               for (int j = 1; j <= COL; j++) {
                    model.setValueAt(new Integer(i * j),
                    ➥i - 1, j - 1);
               }
               try {
                    Thread.sleep(1);
               } catch (InterruptedException e) {
               }
          }

          final JScrollPane comp = new JScrollPane
          ➥(table);
          panel.add(comp, BorderLayout.CENTER);

          return panel;
     }
}
```

Afin d'être capable de lancer notre nouvelle fenêtre, ajoutons l'action qui convient, conformément au framework.

```
public class ListeNombreAction extends ActionAbstraite {
     public ListeNombreAction() {
     }

     protected void doActionPerformed(ActionEvent e) {
     getFenetrePrincipale().nouvelleFenetre(new
     ➥ListeNombre());
     }
}
```

Enfin, pour déclencher notre action, ajoutons un menu à la fenêtre principale.

```
private void addMenus() {
    JMenuBar menuBar = new JMenuBar();
    JMenu menu = new JMenu("Fenetres");
```

```
        JMenuItem menuItem =
        ➥new JMenuItem(new ListeNombreAction());
        menu.add(menuItem);
        menuBar.add(menu);
        setJMenuBar(menuBar);
    }
```

Cette méthode est appelée par le constructeur de `FenetrePrincipale`.

Plusieurs points se dégagent de l'exécution.

- L'exécution est assez longue comme prévu.
- L'interface graphique est figée et le menu ne se ferme même pas. L'utilisateur est pris en otage par le temps d'exécution de l'action.

Figure 7.8 – La fenêtre est figée, le menu ne se ferme pas

- Le système fonctionne tout de même convenablement et notre table de nombres s'affiche finalement.

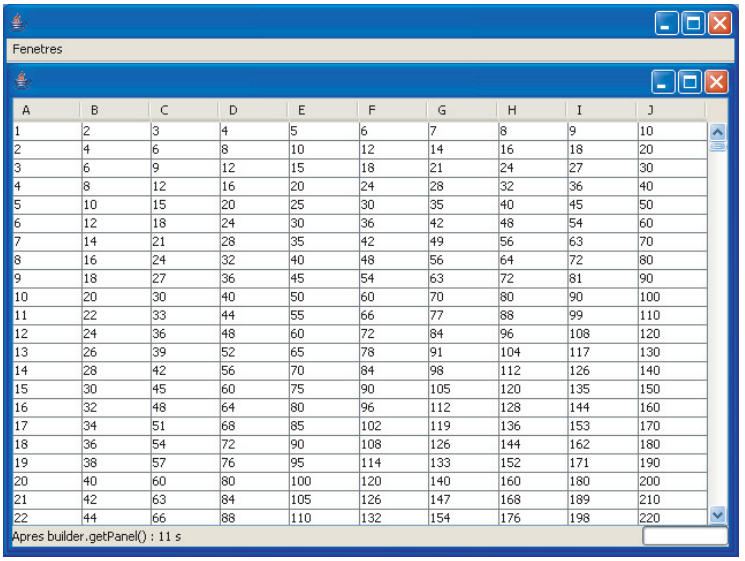


Figure 7.9 – Affichage de la table de nombres

Pour remédier à ce problème d'attente, l'action doit se dérouler en arrière-plan dans un thread séparé. Nous allons donc lancer un nouveau thread à chaque fois que l'action est déclenchée. Une fois encore, au lieu d'écrire du code spécifique pour chaque action, factorisons le code en rendant notre classe `ActionAbstraite` asynchrone. Le code est plus simple qu'il n'y parait.

```
public final void actionPerformed(final ActionEvent e) {
    if (ASYNCHRONE) {
        new Thread() {
             public void run() {
                  doActionPerformed(e);
             }
        }.start();
    } else {
        doActionPerformed(e);
    }
 }
 protected abstract void doActionPerformed(
 ➥final ActionEvent e);
```

Notons la présence d'une constante permettant les deux modes de fonctionnements, synchrone ou asynchrone. Cela est en particulier utile pour debugger. Grâce à notre framework, tout le code existant bénéficie immédiatement de notre ajout.

Premier constat, le menu se referme et le traitement se déroule correctement en arrière-plan, mais la fenêtre ne se rafraîchit pas.

Nous touchons ici au cœur de la problématique Swing et parallélisme. À partir d'un thread nous essayons d'effectuer des actions graphiques (affichage de la fenêtre interne dans le bureau). Or les actions graphiques ne doivent se dérouler que dans le thread graphique, responsable de l'affichage, celui que nous avons justement quitté en créant notre propre thread. Comment allons-nous pouvoir le rejoindre ?

Un mécanisme est prévu pour cela. Il est fourni par Swing grâce à une méthode de la classe `SwingUtilities`. C'est le seul point auquel il faut faire attention. Une règle est donc clairement établie.

> Il ne faut pas déclencher d'action graphique en dehors du thread graphique (aussi appelé thread d'affichage).

Le cœur de ce système asynchrone est en fait dans la classe `FenetreInterne`. La méthode `init()` est indirectement appelée par l'action, maintenant asynchrone. Il faut donc bien faire attention au fait que cette méthode est dorénavant appelée dans un thread qui n'est pas le thread graphique.

```
public void init() {
    // première partie
    JComponent panneauAttente = creerPaneauAttente();
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            add(panneauAttente, BorderLayout.CENTER);
            status.setText("Avant builder.getPanel()");
        }
    });

    // deuxième partie
    long debut = System.currentTimeMillis();
    JComponent panel = fabrique.getPanel();
    long fin = System.currentTimeMillis();

    // troisième partie
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            remove(panneauAttente);
            status.setText("Apres builder.getPanel() : "
                  + ((fin - debut) / 1000) + " s");
            add(panel, BorderLayout.CENTER);
            progress.setIndeterminate(false);
        }
    });
}
```

L'idée est de conserver en arrière-plan la création du contenu de la fenêtre. En effet, il faut essayer de faire le maximum de choses en dehors du thread graphique, sinon nous synchronisons le code. Utiliser des classes graphiques en dehors du thread d'affichage est tout à fait possible, dès lors qu'aucune instruction d'affichage n'est effectuée. Ainsi, dans notre exemple, la création du JPanel, futur contenu de la fenêtre, se fait dans le thread séparé. C'est au dernier moment, quand il faut ajouter ce container à la fenêtre qu'il faut accéder au thread graphique et lui demander d'exécuter les instructions concernant l'affichage.

Pendant le temps de l'attente, pourquoi ne pas affecter à la fenêtre un contenu temporaire d'attente que nous retirerons le moment venu ?

C'est ce panneau temporaire qui est créé par la méthode `creerPaneau Attente()`. Voici son code.

```
private JComponent creerPaneauAttente() {
     return new JLabel("Patientez...");
}
```

La création du panneau d'attente se déroule en arrière-plan, en dehors du thread graphique. Pour insérer ce composant comme contenu de la fenêtre, il faut rejoindre le thread d'affichage. Cela se fait de la façon suivante :

```
JComponent panneauAttente = creerPaneauAttente();
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        add(panneauAttente, BorderLayout.CENTER);
        status.setText("Avant builder.getPanel()");
    }
});
```

Pour rejoindre le thread graphique, nous utilisons la méthode statique `SwingUtilities.invokeLater()` qui prend en paramètre une instance implémentant `Runnable`. Rappelons que l'interface `Runnable` permet de définir un traitement asynchrone, et que la classe Thread elle-même implémente cette interface.

Pour être précis, nous ne rejoignons pas le thread d'affichage. Nous ajoutons un événement dans une queue d'événements. Le thread graphique et notre thread mettent en œuvre un système type *producteur consommateur* dont nous avons parlé précédemment. Notre thread ajoute des événements dans la queue, qui peut être vue comme une file d'attente. Dans le même temps, le thread graphique consomme les événements.

Puisque le code source de Java est disponible, allons voir de plus près cette méthode `invokeLater`.

```
public static void invokeLater(Runnable runnable) {
    Toolkit.getEventQueue().postEvent(
        new InvocationEvent(Toolkit.getDefaultToolkit(),
        ➥runnable));
}
```

Nous voyons ici l'obtention de la queue d'événements : `Toolkit.getEventQueue()`. La méthode `postEvent()` prend en paramètre un `AWTEvent`. Cela explique pourquoi cette queue d'événements est parfois appelée queue d'événements AWT, et le thread graphique est appelé en anglais *AWT event dispatching thread*.

`InvocationEvent` est un type d'événement possible, c'est une sous-classe de AWTEvent. Cet événement est capable de stocker des traitements à exécuter car il implémente l'interface `Runnable`. Finalement, le thread graphique, en tant que consommateur, obtient de la queue une instance de `InvocationEvent` et appelle en synchrone la méthode `run()`.

L'affichage du panneau d'attente et la mise à jour de la barre d'état sont définis dans la méthode run :

```
add(panneauAttente, BorderLayout.CENTER);
status.setText("Avant builder.getPanel()");
```

C'est ainsi que notre traitement se retrouve défini dans un thread séparé mais est récupéré puis exécuté dans le thread graphique.

Une autre méthode statique de la classe `SwingUtilities` permet de définir du code devant s'exécuter dans le thread d'affichage de façon synchrone. Il s'agit de la méthode `invokeAndWait`. Lorsqu'un thread rencontre cette instruction, le traitement à exécuter est placé dans la queue des événements afin d'être exécuté par le thread d'affichage, mais le thread qui a appelé l'instruction ne continue pas tant que le traitement n'est pas terminé.

La méthode `invokeLater()` ayant pour rôle d'ajouter l'instance de `InvocationEvent` dans la queue, il apparaît clairement que nous ne savons pas quand ce traitement d'affichage va réellement s'exécuter. C'est le propre du parallélisme. La méthode `invokeLater()` se termine donc rapidement et nous passons à la section centrale de notre méthode `init()`. Cette section, non graphique, s'exécute pendant que le thread graphique gère sa queue d'événements et va donc incessamment sous peu ajouter le panneau d'attente à la fenêtre.

```
long debut = System.currentTimeMillis();
JComponent panel = fabrique.getPanel();
long fin = System.currentTimeMillis();
```

Cette section contient le code qui est long à s'exécuter : l'appel à la fabrique. Ce code s'exécute bien dans un thread séparé du thread d'affichage. Même si le code prend un peu de temps, il ne bloque personne et n'encombre pas le thread graphique. L'interface reste active, elle affiche le panneau d'attente et elle peut toujours réagir aux actions de l'utilisateur s'il y en a.

La dernière partie de la méthode `init()` va maintenant ajouter le panneau qui est enfin prêt, à la fenêtre principale. N'oublions pas d'enlever notre panneau d'attente. Nous pouvons même en complément afficher le temps d'exécution dans la barre d'état. Ces actions nécessitent de s'exécuter dans le thread graphique.

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        remove(panneauAttente);
        status.setText("Apres builder.getPanel() : "
              + ((fin - debut) / 1000) + " s");
        add(panel, BorderLayout.CENTER);
        progress.setIndeterminate(false);
    }
});
```

Souvenons-nous que cette méthode `init()` est appelée par la méthode `ajouterFenetre()` comme suit :

```
public void nouvelleFenetre(Fabrique fabrique) {
    FenetreInterne fenetre = new FenetreInterne(fabrique);

    fenetre.setSize(desktopPane.getSize());
    desktopPane.add(fenetre);
```

```
        fenetre.setVisible(true);
        fenetre.moveToFront();
        fenetre.init();
    }
```

La méthode `init()` est appelée à la fin du processus de création d'une nouvelle fenêtre. Ainsi, lors de l'attente, la fenêtre est déjà visible et enregistrée par le `desktopPane`.

En résumé, il est possible d'optimiser les performances et de fluidifier une interface graphique à l'aide des threads. Pour ce faire, il faut respecter une règle d'or : tout appel orienté graphique doit se faire dans le thread graphique. Au besoin, il faut le rejoindre en utilisant SwingUtilities.

Un récent article de Sun sur le sujet précise que la manipulation d'un modèle, que nous serions tenté de considérer comme non graphique, doit être traité dans le thread graphique.

On peut manipuler le modèle en dehors du thread graphique, dès lors qu'il n'est pas encore associé à une vue. C'est, de plus, une optimisation notable en terme de performance car dans ce cas, le modèle ne cherche pas à signaler les nouvelles données à la vue (MVC). Il est donc intéressant d'ajouter des données volumineuses au Modèle avant de l'associer à sa Vue.

Notre architecture logicielle pourrait donc être améliorée afin de respecter à la lettre les préconisations de Sun. En effet, notre exemple utilise un modèle qui est associé à une vue lors de son remplissage en arrière-plan.

7.3 LES TESTS

Swing étant réputé fonctionner correctement, il n'est pas utile d'envisager des tests unitaires graphiques au sens strict. Par exemple, il est inutile de tester qu'un composant est bien placé au sud d'un container si vous l'avez programmé comme cela. Ce point particulier est en fait testé visuellement chaque fois que l'application est lancée et il n'y a pas lieu de l'automatiser dès lors que l'interface graphique est programmée à la main et non pas générée dynamiquement.

La logique métier de l'application doit être testée unitairement. Du fait des principes de séparation des couches, ces tests ne concernent pas la couche cliente ou graphique. Sinon, il y a un problème d'architecture logicielle.

Il est cependant nécessaire de tester une interface graphique. Elle contient en elle-même une certaine logique, une logique d'interaction, qu'il faut tester. De plus par son intermédiaire, c'est toute l'application qui sera testée car une action sur la couche graphique traverse ensuite toutes les couches logicielles. Nous

évoquerons donc deux types de tests concernant la couche cliente en général. Les tests unitaires qui nous permettront de tester unitairement les classes non graphiques de la couche cliente, typiquement des classes de la partie *Modèle* du modèle MVC. Les tests fonctionnels qui nous permettront de tester tout l'applicatif au travers de la couche graphique. Ces deux sortes de tests diffèrent aussi par leur niveau d'abstraction. Les tests unitaires sont plus microscopiques, se placent au niveau d'une seule classe alors que les tests fonctionnels de placent au niveau de l'application et sont donc macroscopiques.

7.3.1 Les tests unitaires

En ce qui concerne les tests unitaires, nous pouvons utiliser *JUnit*, une librairie Open Source bien connue. Un test unitaire utilisant *JUnit* est une sous-classe de la classe `TestCase` en respectant quelques règles de nommage particulières :

- Toujours préfixer le nom de la sous-classe par Test. Pour tester une classe `Client`, il faut coder une classe `TestClient` héritant de `TestCase`.
- Toujours préfixer le nom des méthodes par test. Cette classe `TestClient` codera le test du « client mécontent », un cas de figure pris en compte par le système, dans une méthode `public void testClientMecontent()` par exemple.

Ce faisant, nous bénéficions de l'aide de nombreux environnements de développement qui prennent en compte cette norme. Il est ainsi souvent possible de lancer un ou plusieurs tests par un simple clic droit sur une classe de test. En effet, de tels tests unitaires ne se lancent pas comme un programme classique. Il n'y a pas de méthode `main(…)`, nous n'avons pas à instancier les classes de tests (sous-classes de `TestCase`) nous-même. Pour lancer un ou plusieurs tests unitaires il faut donc un environnement de développement ou un autre outil, comme par exemple un outil spécifique pour les tests. Il en existe de nombreux sur internet.

Imaginons par exemple les tests unitaires d'une classe non graphique mais faisant partie de la couche cliente, une classe *Modèle* ou *Contrôleur* dans le modèle MVC. Si nous devons tester une méthode `int ajoute2(int i)` qui ajouterait 2 à son paramètre et retournerait le résultat. Comment tester une telle méthode ?

Écrivons notre premier test unitaire avec *JUnit* et vérifions qu'en entrant 6 le résultat est 8. Voici donc notre code à tester :

```
public class ATester {

   public int ajouter2(int i) {
      return i + 2;
   }
}
```

Voici ensuite notre classe de tests unitaires pour la classe `ATester`.

```
public class TestATester extends TestCase {

   private ATester aTester = null;

   protected void setUp() throws Exception {
      aTester = new ATester();
   }

   public void testAjouter2() {
      int resultat = aTester.ajouter2(6);
      assertEquals(8, resultat);
   }
}
```

N'oubliez pas d'inclure la librairie `junit.jar` au *classpath*. Observons la méthode `testAjouter2()`. Cette méthode utilise d'entrée de jeu une instance de la classe `ATester` au travers de l'attribut `aTester`. Cet attribut est initialisé dans la méthode `setup()`. Il faut savoir que cette méthode est déclenchée par *JUnit* avant tout appel à une méthode de test.

Lorsque notre méthode de test `testAjouter2()` est exécutée, nous savons donc que l'attribut `aTester` est initialisé correctement. Ce mode de fonctionnement permet de réinitialiser au sein d'une unique méthode, `setup()`, toutes les valeurs entre deux tests.

Un test classique se compose d'une utilisation de la classe à tester puis d'une série d'assertion qui doivent toutes se vérifier. Ces assertions constituant des règles exprimées en utilisant les méthodes `assert(…)` fournies par *JUnit* et définies dans `TestCase`.

Ici, nous vérifions que la variable `resultat` est bien égale à 8 après avoir entré 6.

Il est bon aussi de tester les cas limites. Nous pourrions donc enrichir le test unitaire en testant le résultat pour une valeur négative et pour la valeur 0. Lorsqu'on écrit des tests unitaires, il faut toujours penser à l'ensemble des valeurs autorisées en paramètres et il faut s'efforcer de tester des valeurs susceptibles d'entraîner des différences de comportement.

```
public void testAjouter2() {
   assertEquals(8, aTester.ajouter2(6));
   assertEquals(2, aTester.ajouter2(0));
   assertEquals(-8, aTester.ajouter2(-10));
}
```

La librairie *JUnit* nous propose un vaste éventail de méthodes `assert(…)` qui permettent de tester tous les cas pouvant se présenter. Par exemple, `assertNot Null(Object o)` permet de s'assurer qu'une référence n'est pas nulle.

S'il est nécessaire de coder un test très complexe, il est toujours possible d'utiliser la méthode `assertTrue(boolean b)`. Par exemple, pour tester qu'une chaîne de caractères contient au moins la sous-chaîne « test » :

```
public void testChaine() {
   String resultat = ...
   assertTrue(resultat.indexOf("test") > 0);
}
```

Attention cependant à ne pas tester un calcul en le faisant deux fois. Par exemple :

```
public void mauvaisTest() {
   int entree = 2;
   int resultat = aTester.ajouter2(entree);
   assertTrue(resultat == entree + 2);
}
```

Un tel test repose sur l'idée qu'en codant deux fois une fonction, une fois dans le test et une fois dans le code, il n'y aura pas deux fois une erreur. Ce postulat est très contestable.

Au lieu de coder une nouvelle fois l'algorithme, il est préférable de tester quelques valeurs en dur, sans refaire de calcul, puis de tester le contrat de service que la classe à tester définit implicitement.

Enfin, n'hésitez pas à fractionner les tests en autant de méthodes de test que nécessaire. En cas de problème, il sera plus facile de se concentrer sur quelques lignes de code qui ne donnent plus satisfaction plutôt que sur un bloc de code devenu incompréhensible pas l'ajout de strates de tests successives.

Depuis le JDK 1.4, il est possible d'utiliser une instruction `assert` pour vérifier certaines « certitudes » du développeur. Il ne s'agit pas ici d'effectuer des tests unitaires mais seulement d'adopter une habitude de programmation peu coûteuse en temps mais qui peut permettre de détecter des bugs potentiels assez tôt. Lorsqu'une assertion n'est pas vérifiée, une exception de type `AssertionError` est levée. La syntaxe est la suivante :

```
assert var>0 ;
```

Si la variable var n'est pas positive, une exception sera levée, et le bug immédiatement détecté.

Cet ajout du JDK 1.4 a le mérite d'être très simple à utiliser. Toutefois, les assertions font partie intégrante du code source, ce qui n'est pas toujours idéal. En revanche, cela facilite le maintien à jour des tests au fur et à mesure de l'évolution du code.

Notez qu'il est possible de désactiver les assertions afin d'éviter d'exécuter les tests sur une application en production.

Ce court paragraphe ne prétend pas faire le tour du thème des tests unitaires, mais cette introduction nous semble nécessaire vu l'importance croissante des tests dans les projets d'aujourd'hui.

Enfin, ce paragraphe donne un point accès aux tests fonctionnels, dont la présentation suit et qui sont les seuls à véritablement tester l'interface graphique et toute l'application, car ceux-ci utilisent indirectement la librairie *JUnit*. Les tests fonctionnels peuvent donc être considérés comme une spécialisation des tests unitaires.

7.3.2 Les tests fonctionnels

Insistons sur le fait que de tels tests sont très importants car ils mettent en œuvre toute l'application. Pour cette raison ces tests se nomment en anglais « *acceptance tests* », appelons les « tests fonctionnels ».

La meilleure solution consiste à utiliser directement l'application et à interagir par programmation avec l'interface graphique. En d'autres termes, piloter l'interface graphique par programme mais de l'extérieur, comme le ferait un utilisateur. À la première exception, le test est considéré comme ayant échoué. Comment mettre en œuvres ces tests fonctionnels ? Nous proposons un exemple utilisant la librairie *Exactor*, un projet Open Source hébergé sur le site source forge (`http://sourceforge.net/projects/exactor`). L'idée simple de cette librairie est de considérer qu'un test est un script de pilotage de l'applicatif. Chaque mot clé du script doit correspondre à une classe (une sous-classe de `Command` qui est fournie par la librairie). Cette librairie peut être appliquée à Swing en rendant systématique l'usage de la méthode `public void setName(String name)` disponible sur la classe Component. Le principe est que chaque composant qui doit être accessible par un script de test devra avoir un nom. À vous de trouver une notation comprenant éventuellement des points à la manière des noms de packages pour structurer l'espace de nommage et pour pouvoir retrouver facilement de quel composant il s'agit.

Une des premières choses à apporter à cet outillage est une méthode permettant de retrouver un composant en fonction de son nom. Voici un exemple de méthode récursive qui, à partir d'un container, retourne le premier composant qui porte le nom demandé.

```
protected Component chercheComposant(Container container,
➥String nom) {
   Component comp = null;
   Component[] composantFils = container.getComponents();
   for (int i = 0; i < composantFils.length; i++) {
      if (nom.equalsIgnoreCase(composantFils[i]
      ➥.getName())) {
```

```
            comp = composantFils[i];
            break;
         } else if (composantFils[i] instanceof Container) {
            comp = chercheComposant((Container)
            ➥composantFils[i], nom);
            if (comp != null) {
               break;
            }
         }
      }
      return comp;
   }
```

Reprenons l'exemple d'un formulaire de saisie du nom, prénom et de l'adresse afin de construire un exemple de test fonctionnel. Naturellement, cet exemple est extrêmement limité car il n'est pas connecté à une quelconque logique métier, serveur d'application ou base de données. Cet exemple permettra toutefois de construire notre premier test fonctionnel.

```
public class Fenetre extends JFrame {

   protected JLabel lbNom = new JLabel();
   protected JLabel lbPrenom = new JLabel();
   protected JTextField tNom = new JTextField();
   protected JTextField tPrenom = new JTextField();
   protected JLabel lbAdresse = new JLabel();
   protected JTextArea taAdresse = new JTextArea();
   protected JButton ok = new JButton("Ok");
   protected JButton annule = new JButton("Annule");

   public Fenetre() {
      lbPrenom.setText("Prénom");
      tPrenom.setColumns(20);
      tPrenom.setName("prenom");
      lbNom.setText("Nom");
      tNom.setColumns(20);
      tNom.setName("nom");
      lbAdresse.setText("Adresse");
      taAdresse.setColumns(30);
      taAdresse.setRows(5);
      taAdresse.setName("adresse");
      FormLayout layout = new FormLayout("pref, 5dlu,
      ➥pref:grow",
              "pref, pref, pref, fill:pref:grow, pref");
      CellConstraints cc = new CellConstraints();
      JPanel panelFormulaire = new JPanel();
      panelFormulaire.setLayout(layout);
      int ligne = 1;
      panelFormulaire.add(lbNom, cc.xy(1, ligne));
      panelFormulaire.add(tNom, cc.xy(3, ligne));
      ligne++;
      panelFormulaire.add(lbPrenom, cc.xy(1, ligne));
```

```
        panelFormulaire.add(tPrenom, cc.xy(3, ligne));
        ligne++;
        panelFormulaire.add(lbAdresse, cc.xy(1, ligne));
        panelFormulaire.add(taAdresse, cc.xyw(1, 4, ligne));

        ok.setName("ok");
        annule.setName("annule");
        JPanel panelBoutons = new JPanel();
        panelBoutons.setLayout(new FlowLayout(FlowLayout
        ➥.RIGHT));
        panelBoutons.add(annule);
        panelBoutons.add(ok);

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(panelFormulaire,
        ➥BorderLayout.CENTER);
        getContentPane().add(panelBoutons,
        ➥BorderLayout.SOUTH);

        setResizable(true);
        pack();
    }

    public static void main(String[] args) {
        Fenetre f = new Fenetre();
        f.pack();
        f.setVisible(true);
    }
}
```

Nous avons donc ajouté deux boutons que nous avons nommés, ainsi que tous les composants sauf les labels, grâce à la méthode `setName(…)`. Un script de test simple serait :

- Démarrage de l'application.
- Entrée de valeurs dans les champs.
- Clic sur le bouton.
- Fin du test.

Comment construire cela ? Chaque mot clé doit être une commande, sous-classe de `Command`. Presque toutes nos actions vont consister en la recherche de composant par leur nom puis une action sur ces composants ainsi récupérés. Il faut donc envisager d'écrire notre propre sous-classe de `Command` que toutes les autres commandes utiliseront. En effet, nous avons déjà une idée des commandes dont nous aimerions disposer : clic sur un bouton, entrée d'une valeur dans un champ, démarrage et fin de l'application. Toutes ces commandes nécessiteront la recherche d'un composant à partir du container racine : la fenêtre principale.

Cela nous permet d'envisager la classe suivante :

```
abstract class MaCommande extends Command {

   protected static final String CLE_APPLICATION =
   ➥"fenetre principale";

   protected Component chercheComposant(String nom) {
     return chercheComposant(getFenetrePrincipale(), nom);
   }

   protected Component chercheComposant(Container container,
   ➥String nom) {
      Component comp = null;
      Component[] composantFils = container
      ➥.getComponents();

      for (int i = 0; i < composantFils.length; i++) {
         if (nom.equalsIgnoreCase(composantFils[i].
         ➥getName())) {
            comp = composantFils[i];
            break;
         } else if (composantFils[i] instanceof
            ➥Container) {
            comp = chercheComposant((Container)
            ➥composantFils[i], nom);
            if (comp != null) {
               break;
            }
         }
      }
      return comp;
   }

   protected Fenetre getFenetrePrincipale() {
      return (Fenetre)
getScript().getScriptSet().getContext().get
➥(CLE_APPLICATION);
   }
}
```

La librairie *Exactor* propose une notion de contexte pour un script. Ce contexte est une Hashtable (ensemble de couples clé-valeur) et est accessible en appelant `getScript().getScriptSet().getContext()`. À nous de l'utiliser pour stocker la fenêtre principale par exemple. Cette classe, `MaCommande`, est l'occasion de redéfinir la méthode `chercheComposant()` en utilisant comme container par défaut la fenêtre principale.

Finalement, qui positionne la fenêtre principale dans le contexte ? Voici un élément de réponse avec notre première commande : `Demarrer`.

```
public class Demarrer extends MaCommande {

   public void execute() throws Exception {
      Fenetre f = new Fenetre();
      f.pack();
      f.setVisible(true);
      getScript().getScriptSet().getContext().put
      ➥(CLE_APPLICATION, f);
   }
}
```

Nous avons choisi pour plus de simplicité de dupliquer le code de la méthode `Main(String[] args)`. Nous pouvons donc rédiger la première ligne de notre script de test : le démarrage de l'application. Nommons le fichier `test.act`, `act` étant l'extension par défaut pour un script *Exactor*.

```
Demarrer
```

Ce script lancera la fenêtre. Il nous reste donc encore à piloter l'application en entrant des valeurs dans les champs texte puis à valider le formulaire. Cependant, une question se pose maintenant, comment lancer le script *Exactor* ?

De fait, la librairie est fournie avec un lanceur qui va interpréter le fichier de script, vérifier que chaque mot clé (ici `Demarrer`) correspond bien à une classe se trouvant dans le *classpath* du lanceur, exécuter chaque mot clé dans l'ordre du script. La classe du lanceur est `com.exoftware.exactor.Runner`. Lancer un script *Exactor* revient donc à exécuter cette classe qui prend en paramètre de sa méthode `main` un répertoire ou un fichier à exécuter.

La commande, dans une fenêtre DOS, est donc par exemple :

```
java com.exoftware.exactor.Runner c:\test\test.act
```

N'oublions pas de positionner le *classpath*, par exemple en utilisant `java -cp`, incluant `junit.jar`, `exactor.jar` ainsi que nos propres classes `Fenetre`, `MaCommande` et `Demarrer`.

Enrichissons maintenant notre script de test :

```
Demarrer
EntrerValeur nom Dupont
EntrerValeur prenom Jean
EntrerValeur adresse "2bis rue des fleurs jaunes"
ClicBouton ok
Stop
```

Exactor permet pour chaque mot clé d'avoir des paramètres séparés par au moins un espace. Ainsi la ligne du script `EntrerValeur nom Dupont` peut se comprendre par exécute la commande `EntrerValeur` en tenant compte de deux paramètres, `nom` et `Dupont`.

Si un paramètre contient lui-même des espaces, il est nécessaire d'utiliser des guillemets pour délimiter cette valeur. C'est le cas de la valeur de l'adresse dans le script. Développons la classe `EntrerValeur`.

```
public class EntrerValeur extends MaCommande {

   public void execute() throws Exception {
      String nomChamp = getParameter(0).stringValue();
      String texte = getParameter(1).stringValue();

      Component comp = chercheComposant(nomChamp);

      if (comp instanceof JTextComponent) {
         JTextComponent textComponent = (JTextComponent)
         ➥comp;
         textComponent.setText(texte);
      }
   }
}
```

La valeur des paramètres du script est obtenue dans le code de la commande à l'aide de la méthode `getParameter`. Pour accéder au premier paramètre qui correspond au nom du composant, on utilise : `getParameter(0).stringValue()`. L'instance de ce composant est récupérée en utilisant la méthode `cherche Composant` que nous avons développée dans la classe abstraite `MaCommande` dont héritent toutes nos commandes.

Ensuite, afin de rendre notre commande plus générique, nous utilisons la classe intermédiaire `JTextComponent` car nous n'avons besoin que de positionner le texte dans un champ, peu importe que l'instance soit de type `JTextField` ou `JTextArea`.

La commande `ClicBouton` suit les mêmes principes :

```
public class ClicBouton extends MaCommande {

   public void execute() throws Exception {
      String buttonName = getParameter(0).stringValue();
      Component comp = chercheComposant(buttonName);

      JButton boutton = (JButton) comp;
      boutton.doClick();
   }
}
```

Si on lance le script, le formulaire reste ouvert et le script signale une erreur : la commande Stop est absente.

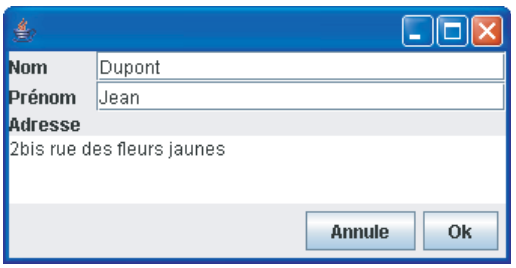


Figure 7.10 – Les champs textes remplis par le test fonctionnel de la fenêtre principale.

Ajouter la commande Stop qui est très simple :

```
public class Stop extends MaCommande {

   public void execute() throws Exception {
      Fenetre f = getFenetrePrincipale();

      f.setVisible(false);
      f.dispose();
   }
}
```

Il reste maintenant un point important, en quoi cela est-il un test fonctionnel ?

Ce type de script est déjà en soi un test car il déclenche les fonctionnalités de l'application. Remarquez que la signature paramétrique de la méthode `execute` intègre la levée d'Exception. De ce fait, la moindre erreur est ainsi captée et le test ne passe pas. Pour nous en convaincre, ajouter un listener au bouton « Ok » et levons une `NullPointerException`.

Voici le code du listener dans la classe Fenetre :

```
ok.setName("ok");
ok.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      throw new NullPointerException("Pour tester");
   }
});
```

Une fois lancée, la fenêtre disparaît, mais voici la trace laissée par l'exécution du script :

```
ScriptSet started
     started: test.act
           OK: Demarrer
           OK: EntrerValeur nom Dupont
           OK: EntrerValeur prenom Jean
```

```
            OK: EntrerValeur adresse 2bis rue des fleurs
            ➥jaunes
            Error: ClicBouton ok Line: 5
                  Pour tester
java.lang.NullPointerException: Pour tester
     at formlayout.Fenetre$1.actionPerformed
     ➥(Fenetre.java:57)
[Suite de l'exception coupée pour gagner de la place]

            OK: Stop
     ended: test.act

Scripts run: 1
Failures: 0
Errors: 1

Duration: 0s
```

La librairie *Exactor* produit un résumé de l'exécution globale du script à la fin. Même dans le cas fort peu probable où nous n'aurions pas vu la trace d'exception, le résumé est formel : le test ne passe pas. Nous pouvons donc considérer ce script comme un test fonctionnel simple. Aucune fonctionnalité que nous lançons de la sorte ne pourra désormais générer une erreur sans que nous en soyons avertis, c'est un point important car nous contrôlons ainsi une partie des régressions possibles pour l'applicatif. À charge ensuite pour l'équipe de développement de lancer ces scripts régulièrement. Il faudra aussi maintenir à jour le script de test en ajoutant le déclenchement des nouvelles fonctionnalités.

De tels tests peuvent devenir plus complets et tester des valeurs au lieu de simplement déclencher des fonctionnalités. C'est à ce point qu'intervient à nouveau *JUnit* dont nous parlions dans le paragraphe précédant. En effet, la classe Command d'*Exactor* hérite elle-même de `TestCase` que nous connaissons. Il est aussi aisé de tester une valeur dans une commande.

Ajoutons par exemple du code à notre fenêtre pour y créer une barre de messages toute simple, comprenant uniquement un label. Ce label permet de contenir un message pour l'utilisateur. Changeons aussi le listener sur le bouton « Ok » pour qu'il affiche un message à l'utilisateur.

```
ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         try {
            <appel au code métier>
            message.setText("Ok");
         } catch (Exception e) {
            message.setText(e.getMessage();
         }
      }
   });
```

La classe `Fenetre` se voit donc dotée d'un nouvel attribut : `protected JLabel message = new JLabel()`. Ajoutons ce label pour en faire une barre de messages :

```
JPanel panelBoutons = new JPanel();
panelBoutons.setLayout(new FlowLayout(FlowLayout.RIGHT));
panelBoutons.add(annule);
panelBoutons.add(ok);
message.setName("message");
message.setText("Pret");

JPanel panelSud = new JPanel();
panelSud.setLayout(new BorderLayout());
panelSud.add(panelBoutons, BorderLayout.NORTH);
Border border = BorderFactory.createLineBorder(Color.black);
message.setBorder(border);
panelSud.add(message, BorderLayout.SOUTH);

getContentPane().setLayout(new BorderLayout());
getContentPane().add(panelFormulaire, BorderLayout.CENTER);
getContentPane().add(panelSud, BorderLayout.SOUTH);
```

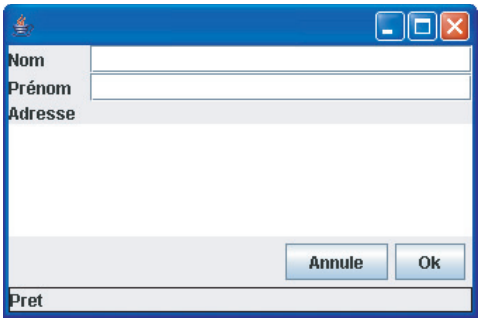


Figure 7.11 – Ajout d'une barre d'état au formulaire.

Nous pouvons maintenant envisager de tester la valeur du message après avoir cliqué sur le bouton. Le script de test devient donc :

```
Demarrer
EntrerValeur nom Dupont
EntrerValeur prenom Jean
EntrerValeur adresse "2bis rue des fleurs jaunes"
ClicBouton ok
TestValeur message Ok
Stop
```

Voici le code de la commande `TestValeur`.

```
public class TestValeur extends MaCommande {

   public void execute() throws Exception {
      String nomChamp = getParameter(0).stringValue();
      String texte = getParameter(1).stringValue();
      Component comp = chercheComposant(nomChamp);

      if (comp instanceof JTextComponent) {
         JTextComponent textComponent = (JTextComponent)
         ➥comp;
         assertEquals(texte, textComponent.getText());
      } else if (comp instanceof JLabel) {
         JLabel label = (JLabel) comp;
         assertEquals(texte, label.getText());
      } else {
         fail(nomChamp + " n'est ni un JLabel ni un
         ➥composant texte");
      }
   }
}
```

C'est ici que l'association *JUnit + Exactor* prend toute sa puissance. Il est donc possible d'entrer des assertions qui devront se vérifier pour que le test passe. Toutes les assertions *JUnit* sont disponibles car la classe `Command` hérite de `TestCase`.

Dès lors que l'on commence à tester des valeurs dans des champs ou des labels, il est nécessaire d'avoir des données de référence. Dans le cas précédent, notre donnée de référence était codée en dur dans le code du listener. Ce cas simple présente le mérite d'illustrer simplement notre propos et de montrer un test fonctionnel. Dans le cadre d'un projet, vous aurez certainement besoin de données de référence persistantes. Si vous utilisez une base de données, il sera nécessaire d'en dédier une aux tests. Les tests fonctionnels ne devront être lancés qu'avec cette base. L'inconvénient d'une telle démarche est qu'il faut sans cesse adapter la base, tant son contenu que sa structure, et les tests. C'est au prix du maintien de cette cohérence que les tests fonctionnels rendent l'immense service de signaler les régressions.

8

Les apports des dernières versions de Java

Début août 2005, la version la plus récente de Java est la version 5.0. Les noms des produits et les numéros de versions utilisés par Sun peuvent être assez déroutants, un petit rappel historique s'impose :

- Le JDK (pour *Java Development Kit*) 1.1 date de 1997.
- À la fin de l'année 1998, le JDK 1.2 est disponible. Pour marquer les profonds changements qui caractérisent ce JDK, le nom « Java 2 » est lancé par Sun pour indiquer qu'il s'agit d'une seconde génération de la plate-forme Java. Le sigle J2SE pour Java 2 Standard Edition, s'impose pour désigner l'ensemble de la plate-forme Java (JDK, JRE et outils fournis en standard).
- En 2000, le JDK 1.3 (ou plus généralement le J2SE 1.3) voit le jour.
- En 2002, c'est au tour du JDK 1.4, dont les nouveautés concernant Swing sont rassemblées ci-dessous.
- En septembre 2004, une nouvelle version de Java apparaît, mais cette fois, pour refléter plus fidèlement le degré de maturité du langage Java, Sun décide de changer le numéro de version de 1.5 en 5.0. Toutefois, le numéro de version « interne » reste 1.5. Ainsi, lorsque vous exécutez la commande `java -version`, vous obtiendrez 1.5.0.
- Le JDK 6.0, en cours de développement à l'heure actuelle, est annoncé pour le premier semestre 2006. Des articles concernant les nouveautés de cette version sont d'ores et déjà disponibles sur Internet, et une version intermédiaire du code peut aussi être téléchargée sur www.java.net.

Les projets de développement des nouvelles versions de Java ont toujours des noms de code « internes » avant d'être rendu publics sous la marque J2SE. Ainsi, le JDK 6.0 s'appelle Mustang, alors que le 5.0 s'appelait Tiger et le 1.4 Merlin. Pas de surprise donc, si vous entendez parler de Mustang en lieu et place de JDK 6.0.

Certains connaisseurs vous diront qu'il y a quelques années, on parlait de J2SDK. Ce terme de Standard Development Kit a finalement été abandonné par Sun, étant donné que la plus grande partie de la communauté Java, imperturbable, continuait d'utiliser l'abréviation JDK, tellement plus simple.

Nous allons présenter ci-dessous les nouveautés des JDK 1.4, 5.0 et les nouveautés annoncées du 6.0.

8.1 LES APPORTS DU JDK 1.4

8.1.1 Un champ de saisie contrôlé

Un problème classique de programmation d'interfaces graphiques réside dans le contrôle des données saisies par l'utilisateur. En particulier, lors de la saisie dans un champ de texte, on est très souvent obligé de réécrire notre propre modèle de document pour pouvoir contrôler en temps réel la validité des caractères entrés par l'utilisateur.

Là encore, face à une difficulté extrêmement courante, le JDK 1.4 apporte une solution en ajoutant un nouveau composant texte exactement adapté à ces problèmes de contrôles.

Principe de la classe JFormattedTextField

La classe `JFormattedTextField` est une sous-classe de `JTextField`. Cette classe ne gère pas elle-même le contrôle des données saisies. Elle délègue cette tâche à un objet du type `AbstractFormatter`. Cet objet est obtenu à l'aide d'une classe *Factory*, dont le rôle est de fournir des instances (design pattern *Factory*).

Pour indiquer le format de ce qui doit être saisi par l'utilisateur, on peut utiliser :

- une instance d'une sous-classe concrète de `AbstractFormatter`,
- une instance d'une sous-classe concrète de la classe `java.text.Format`, que nous détaillons plus bas,
- directement un objet particulier (une date, un nombre…). Dans ce cas, Java construit un formateur associé.

Rappels sur la classe java.text.Format

Le package `java.text` existe depuis le JDK1.1. Rappelons l'utilisation de trois classes courantes de ce package.

SimpleDateFormat

`SimpleDateFormat` est une classe instanciable, qui permet de définir un format d'affichage d'une date. À partir de ce « masque » décrivant le format, un objet `SimpleDateFormat` est capable de convertir une chaîne de caractères au bon format en un objet `Date` et inversement.

Ainsi, pour afficher une date de la façon suivante : « 01/04/2005 », on utilise un `SimpleDateFormat` avec le masque « dd/MM/yyyy ».

```
SimpleDateFormat unFormat = new SimpleDateFormat
➥("dd/MM/yyyy");
Date dateCourante = new Date();
System.out.println("La date courante formatée : " +
   unFormat.format(dateCourante));
```

L'opération dans le sens inverse utilise la méthode `Date parse(String source)` qui reconstruit une date à partir d'une chaîne, conformément au format de date précisé par le constructeur. Il est aussi possible d'affecter un format de date à une instance par la méthode `void applyPattern(String pattern)`.

NumberFormat

`NumberFormat` est une classe abstraite, qui permet de convertir une chaîne de caractères représentant un nombre, formatée selon les paramètres locaux, en un objet `Number`. De la même façon, à partir d'un nombre, on peut obtenir la chaîne le représentant.

La classe `NumberFormat` s'appuie sur la `Locale` par défaut, mais on peut également spécifier une `Locale` particulière. Par exemple, avec la Locale « France », on obtient une virgule pour séparer la partie entière de la partie décimale, et un espace pour séparer les milliers.

```
String ch = NumberFormat.getInstance().format(589546.42);
System.out.println("Le nombre formaté : "+ch);
```

Ce code donne l'affichage suivant : `Le nombre formaté : 589 546,42`

DecimalFormat

`DecimalFormat` est une sous-classe concrète de `NumberFormat`. On indique le format souhaité à l'aide d'un `pattern` : un modèle pour l'affichage de nombres décimaux.

À l'aide de la syntaxe proposée pour définir un pattern, il est possible de définir une multitude de patterns différents.

Ainsi, il est possible de définir deux sous-patterns : l'un pour les nombres positifs, l'autre pour les nombres négatifs. Si le pattern pour les nombres négatifs est omis, c'est le premier pattern qui est utilisé, précédé du signe « - ».

On indique le format d'un nombre à l'aide des caractères '0' ou '#', '0' représentant une position obligatoire, '#' une position facultative.

Ainsi, pour indiquer que 2 chiffres après la virgule sont obligatoires, qu'un troisième est autorisé, mais que davantage de chiffres dans la partie décimale sont ignorés, on utilise le pattern suivant :

```
DecimalFormat format1 = new DecimalFormat("###.00#");
Regardez ce petit exemple :
String ch1 = format1.format(55.1);
System.out.println("ch1 formatée : "+ch1);
String ch2 = format1.format(55.123);
System.out.println("ch2 formatée : "+ch2);
String ch3 = format1.format(55.1234);
System.out.println("ch3 formatée : "+ch3);
Avec le format défini, les nombres « 55,1 », « 55,123 »
et « 55,1234 » sont respectivement formatés comme suit :
ch1 formatée : 55,10
ch2 formatée : 55,123
ch3 formatée : 55,123
```

On peut indiquer aussi l'endroit où sera inséré le séparateur des milliers, car dans certains pays, il s'agit en fait d'un séparateur des « dix milliers », qui est donc inséré toutes les 4 positions.

On peut préciser un préfixe et un suffixe, par exemple pour afficher une unité, ou une devise.

```
DecimalFormat euros = new DecimalFormat("###,###,##0.00_");
DecimalFormat dollars = new DecimalFormat("$###,###,##0.##");

String p1 = euros.format(9876.5);
System.out.println("Un prix en euros : "+p1);
String p2 = dollars.format(9876);
System.out.println("Un prix en dollars : "+p2);
```

On obtient alors l'affichage suivant :

```
Un prix en euros : 9 876,50€
Un prix en dollars : $9 876
```

On peut aussi utiliser la notation scientifique avec la lettre E pour « exposant ». On peut en outre choisir chaque symbole particulier (le séparateur de la partie entière et la partie décimale, le séparateur de milliers, etc.) si on ne souhaite pas s'appuyer sur ceux de la Locale.

L'API de JFormattedTextField

Ce composant graphique délègue donc la tâche de contrôle à un objet `AbstractFormatter`, qui lui-même s'appuie sur un objet `Format` pour connaître le format autorisé.

Lorsque l'on crée un `JFormattedTextField`, on peut donc procéder de différentes façons.

Pour créer un champ de saisie de dates, en s'appuyant sur les paramètres de la locale et en positionnant le champ à la date courante par défaut, on peut écrire tout simplement :

```
new JFormattedTextField(new Date());
```

Pour spécifier un format particulier, on peut utiliser n'importe quelle sous-classe de `Format` :

```
new JFormattedTextField(new DecimalFormat("#0.###"));
```

Enfin, on peut également utiliser une instance d'une sous-classe de `AbstractFormatter`, par exemple la classe `MaskFormatter`, qui permet de définir un masque pour tout type de caractères (nombres, lettres, etc.) Pour saisir un numéro de téléphone français, on peut créer le champ suivant :

```
new JFormattedTextField(new MaskFormatter("##-##-##-##-##"));
```

Les méthodes `getValue` et `setValue` servent respectivement à obtenir la valeur saisie par l'utilisateur, et à indiquer un objet à afficher. `getValue` parse la chaîne du champ de texte et renvoie l'objet obtenu. `setValue` au contraire formate un objet et affiche la chaîne obtenue dans le champ.

Il est possible de configurer l'action à faire lors de la perte du focus. La méthode `setFocusLostBehavior` permet de choisir un comportement : contrôler la valeur et lever une exception en cas d'erreur, contrôler et revenir à l'ancienne valeur en cas d'erreur, accepter la nouvelle valeur, … Il est aussi possible d'empêcher la perte du focus tant que la valeur n'est pas valide. Attention cependant à de telles règles qui ont coutume d'agacer les utilisateurs.

On peut aussi paramétrer le formateur pour qu'il filtre en cours de saisie les caractères non autorisés.

8.1.2 JSpinner

Vous vous souvenez probablement du composant que nous avons créé dans le chapitre sur les événements pour illustrer un composant Java Bean. Nous avions besoin d'un composant de saisie d'un nombre, muni de deux boutons pour incrémenter ou décrémenter la valeur courante.

Ce type de composant est très courant dans les applications actuelles. Son apparition en tant que composant Swing standard était vivement attendue par la communauté Java. Il a été ajouté avec le JDK 1.4.

Imaginez une interface graphique pour saisir des demandes de congés. L'utilisateur devra saisir la durée de ses congés et la date de début. Ces deux données se

prêtent tout à fait à l'utilisation d'un `JSpinner`. En effet, le `JSpinner` est particulièrement adéquat lorsque l'utilisateur doit choisir parmi un ensemble de valeurs qui se suivent, par exemple, un entier entre 1 et n, ou bien une date à partir de la date du jour.

Le rôle du `JSpinner` est assez proche de celui d'une liste déroulante, puisqu'il permet un choix parmi une liste de valeurs possibles. Cependant, la liste déroulante ne peut être utilisée lorsque le nombre de valeurs possibles est très important, car il est très désagréable pour l'utilisateur de devoir faire défiler le contenu de la liste avant de trouver la valeur souhaitée. De plus, la liste « envahit » l'écran lorsque l'utilisateur effectue son choix et peut lui cacher le reste de l'interface. Le `JSpinner` est souvent préféré dans le cas d'un formulaire de saisie « chargé », mais bien évidemment, il ne peut être utilisé qu'en cas d'une séquence de valeurs intuitive, puisque l'utilisateur ne peut voir que la valeur courante.

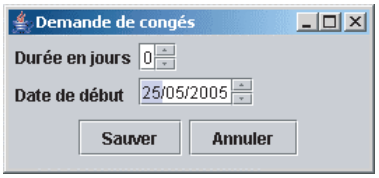


Figure 8.1– Des champs de type « Spinner »

Notez que les touches du clavier « flèche vers le haut » et « flèche vers le bas » ont la même action que les boutons du `JSpinner`, et que l'utilisateur peut également saisir directement dans le champ d'édition.

Dans notre exemple, nous souhaitons paramétrer les deux `JSpinner` : le premier ne peut contenir que des nombres entiers et nous souhaitons ajouter une contrainte sur l'ensemble des nombres autorisés : il ne peut contenir que des nombres entiers entre 0 et 50. Cela se fait de la façon suivante, en utilisant un objet correspondant au modèle de données, tout comme pour tous les autres composants Swing.

```
JSpinner spinDuree = new JSpinner();
SpinnerNumberModel dureeModel = new SpinnerNumberModel
➥(0, 0, 50, 1);
spinDuree.setModel(dureeModel);
```

Le second champ permet de sélectionner une date. Il est initialisé à la date du jour et les flèches permettent d'augmenter ou de diminuer d'un jour. Cette fois, le modèle que nous allons utiliser est une instance de la classe `SpinnerDateModel` :

```
JSpinner spinJour = new JSpinner();
SpinnerDateModel jourModel = new SpinnerDateModel();
jourModel.setCalendarField(Calendar.DATE);
```

```
spinJour.setModel(jourModel);
JSpinner.DateEditor editeur = new JSpinner.DateEditor
➥(spinJour,
"dd/MM/yyyy");
spinJour.setEditor(editeur);
```

Nous avons aussi paramétré l'éditeur utilisé par le champ afin d'avoir un masque de saisie correspondant à dd/MM/yyyy.

Les deux modèles que nous avons utilisés, `SpinnerNumberModel` et `SpinnerDateModel` sont deux classes qui implémentent l'interface `SpinnerModel`. Un peu à la manière du `ListModel`, cette interface `SpinnerModel` permet d'obtenir la valeur courante, elle offre des méthodes d'abonnement et de désabonnement pour des listeners sur les changements du modèle. De plus, elle offre la possibilité de modifier la valeur courante, et d'accéder aux valeurs précédente et suivante.

Plusieurs implémentations de l'interface `SpinnerModel` sont disponibles dans l'API Swing :

- La classe `SpinnerListModel` est adaptée à une séquence de données quelconque contenue dans une collection de type `java.util.List` ou un tableau (`Array`).
- La classe `SpinnerDateModel` encapsule un modèle de données pour la saisie d'une date. Le programmeur doit indiquer l'unité de temps correspondant à ce qui est incrémenté à l'aide des boutons « flèches ». Des constantes de la classe `Calendar` permettent de désigner les unités suivantes : année, mois, jour dans l'année, jour dans le mois, heure, etc. Il peut aussi indiquer deux dates limites.
- La classe `SpinnerNumberModel` encapsule un modèle de données pour la saisie d'un nombre. Il suffit d'indiquer en paramètre du constructeur la valeur par défaut (qui peut être de n'importe quel type numérique de Byte à Double), le pas (la valeur d'incrément), et éventuellement des nombres limites.

8.1.3 Une barre d'attente

Le composant `JProgressBar` de Swing ne proposait jusqu'au JDK 1.3 un seul mode de fonctionnement : il s'agit d'une barre qui avance proportionnellement à l'avancement d'une tâche. Dans de nombreuses applications, on trouve un autre composant relativement proche, qui a aussi pour rôle d'indiquer à l'utilisateur qu'une action se déroule en arrière-plan. Mais à la différence de la barre de progression de Sun, ce composant n'indique pas un degré d'avancement. Il s'agit d'une barre contenant un rectangle plus ou moins large qui effectue des allers-retours indéfiniment. On appelle parfois ce composant une barre d'attente, dans

la mesure où son unique but est de faire patienter l'utilisateur, sans qu'il n'y ait de notion de progression.

Figure 8.2 – Une barre d'attente à des stades différents

Le JDK 1.4 apporte de nouvelles méthodes à la classe `JProgressBar` afin qu'elle puisse être utilisée comme une barre d'attente. La méthode suivante permet de passer d'un mode de fonctionnement à un autre. En passant `true` en paramètre, on obtient une barre d'attente :

```
public void setIndeterminate (boolean b) ;
```

Il est également possible de paramétrer la vitesse de déplacement du rectangle dans la barre, ainsi que la fréquence du rafraîchissement. Ces deux notions sont représentées par les chaînes de caractères suivantes : `ProgressBar.cycleTime` et `ProgressBar.repaintInterval`. Naturellement, la durée d'un cycle (d'un aller-retour) doit être en accord avec la fréquence de rafraîchissement, sinon l'impression « d'aller-retour » n'apparaîtra pas clairement. La moitié de `cycleTime` doit être un multiple de `repaintInterval`. Ces deux valeurs sont exprimées en millisecondes.

Pour les modifier, il faut utiliser la méthode put de la classe `UIManager` :

```
UIManager.put("ProgressBar.cycleTime", new Integer(1000));
UIManager.put("ProgressBar.repaintInterval",
➥new Integer(100));
```

Il est également possible grâce à de nouvelles API d'implémenter un look and feel particulier pour les barres de progression. On peut ainsi paramétrer l'apparence du rectangle qui se déplace dans la barre.

8.1.4 Les composants contextuels

Certains composants particuliers, comme les menus contextuels ou les tooltips apparaissent au-dessus des autres composants, indépendamment de la hiérarchie de containers : ils n'appartiennent pas à un container particulier.

Ces composants sont souvent appelés des composants contextuels, ou *popup* en anglais, car leur apparition et leur contenu dépendent de l'endroit où se trouve le curseur de la souris à un instant donné : ils dépendent donc du contexte.

Ces objets contextuels peuvent être obtenus depuis n'importe quel autre composant graphique : le comportement est toujours similaire. Pour cette raison, leur gestion est déléguée à un objet de la classe `Popup`, obtenu grâce à une `PopupFactory`.

Actuellement, les classes `Popup` et `PopupFactory` sont d'accès package : elles ne sont donc pas connues à l'extérieur du package. Apple a demandé que ces classes soient rendues publiques, afin de pouvoir en hériter pour personnaliser l'aspect et le comportement de ces *popup* en fonction du look and feel.

Les classes deviennent donc publiques à partir du JDK1.4 et leur interface a été légèrement modifiée pour rendre plus facile la personnalisation.

8.1.5 Sérialisation

Depuis le JDK 1.4, il est possible de sérialiser des composants Swing dans un flux XML.

Pour cela il faut utiliser `l'XMLEncoder`.

Voici un exemple simple illustrant cette sérialisation.

```
FileOutputStream fichier = new FileOutputStream
➥("Test.xml");
BufferedOutputStream flux = new BufferedOutputStream
➥(fichier);
XMLEncoder e = new XMLEncoder(flux);
JFrame fenetre = new JFrame();
fenetre.setLayout(new BorderLayout());
fenetre.add(new JButton("Ok"), BorderLayout.SOUTH);
e.writeObject(fenetre);
e.close();
```

Et voici le fichier XML résultant

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_04" class="java.beans.XMLDecoder">
 <object class="javax.swing.JFrame">
  <void property="contentPane">
   <void method="add">
    <object id="JButton0" class="javax.swing.JButton">
     <string>Ok</string>
    </object>
```

```
    </void>
    <void property="layout">
     <object class="java.awt.BorderLayout">
      <void method="addLayoutComponent">
       <object idref="JButton0"/>
       <string>South</string>
      </void>
     </object>
    </void>
   </void>
   <void property="name">
    <string>frame0</string>
   </void>
  </object>
</java>
```

8.2 LES APPORTS DU JDK 5.0

Les apports du JDK 1.5 sont particulièrement importants en ce qui concerne l'évolution du langage lui-même. En revanche, ils sont plus modestes pour l'API Swing.

On note cependant l'arrivée de deux nouveaux look and feel : Ocean et Synth.

Ocean est un nouvel habillage du look and feel Java, aussi appelé look and feel Metal. Bien sûr, pour respecter le principe de compatibilité ascendante, il est possible de conserver l'aspect obtenu avec ce look and feel en JDK 1.4 en modifiant une propriété système :

```
System.setProperty("swing.metalTheme", "steel");
```

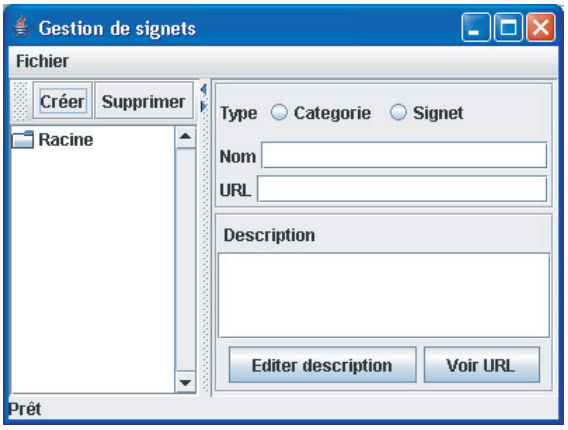


Figure 8.3 – L'application gestion de signets avec le look and feel Ocean

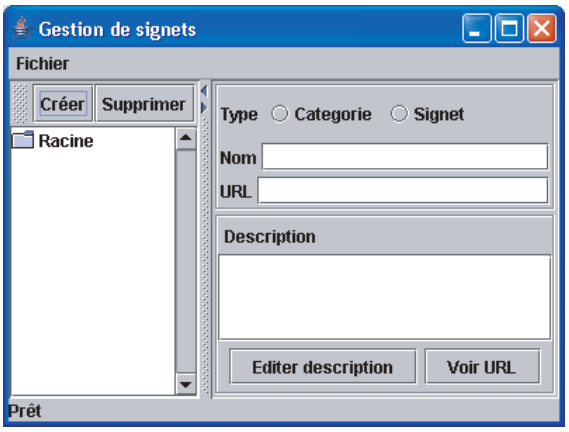


Figure 8.4 – La même application avec la propriété *steel* positionnée.

Synth est un look and feel personnalisable. De très nombreuses propriétés d'affichage peuvent être spécifiées dans un fichier au format XML. Grâce à ce mécanisme, un spécialiste du graphisme peut définir ces propriétés, sans avoir besoin d'écrire une ligne de code Java. Avant Synth, il était fastidieux de définir son propre look and feel, cela demandait une certaine expertise et prenait beaucoup de temps.

Il est possible de configurer le look and feel Synth via un fichier XML. Voici le code permettant de charger le fichier xml s'il se trouve au même niveau que `MyClass` dans l'arborescence (chargement d'un fichier avec le `classpath` et non pas le chemin).

```
SynthLookAndFeel laf = new SynthLookAndFeel();
  laf.load(MyClass.class.getResourceAsStream("laf.xml",
MyClass.class));
  UIManager.setLookAndFeel(laf);
```

Voici un extrait d'un tel fichier XML. Celui-ci configure l'apparence d'un bouton. La DTD complète du fichier est disponible sur Internet.

```
<style id="button">
<opaque value="true"/>
<insets top="4" left="4" right="4" bottom="4"/>
<font name="Dialog" size="12"/>
</style>
```

Une autre modification du JDK qui est tout à fait triviale, et qui rend service à absolument tous les développeurs d'interfaces graphiques : à partir du JDK 5.0, il est possible d'utiliser la méthode `add` et la méthode `setLayout` directement sur un objet `JFrame` sans passer par le `contentPane`.

L'ajout d'un panneau se faisait avec le code suivant :

```
fenetre.getContentPane().add(panneau, BorderLayout.CENTER);
```

Cela peut maintenant s'écrire de la façon suivante :

```
fenetre.add(panneau, BorderLayout.CENTER);
```

On peut citer une troisième amélioration avec la version 5.0, il s'agit de la possibilité d'imprimer facilement un tableau (composant JTable).

Il suffit désormais d'appeler la méthode `print()` sur une instance de `JTable` pour obtenir la boîte de dialogue système de choix de l'imprimante puis de déclencher l'impression.

8.3 LES APPORTS ENVISAGÉS POUR LE JDK 6.0

Une fonctionnalité très répandue dans les interfaces graphiques actuelles est la possibilité de trier les lignes d'un tableau en cliquant sur l'en-tête d'une colonne. Cela sera dorénavant facile à implémenter grâce à la notion de `TableRowSorter`, sorte de « modèle de tri » qui est positionné sur un composant `JTable`. Il est également possible d'ajouter un filtre pour ne visualiser qu'une sous-partie des lignes du tableau. Les interfaces `RowSorter` et `RowFilter` spécifient les méthodes à implémenter et pourront être utilisées dans d'autres contextes que le `JTable`.

Le JDK 6.0 marque l'arrivée de la classe `SwingWorker`. Cette classe utilitaire est une aide précieuse pour le parallélisme. Elle est typiquement prévue pour implémenter des actions devant se dérouler en arrière-plan, mais ayant besoin d'interagir avec l'interface graphique. Par exemple, dans le cas où un traitement assez long est en train de s'exécuter, on souhaite souvent afficher une boîte de dialogue avec l'étape en cours et le degré d'avancement. Il faut donc mettre à jour l'affichage depuis un thread différent du thread graphique. Cela devient très facile à écrire avec le `SwingWorker`. Ce type d'utilitaire circule déjà depuis quelques années sur Internet sous diverses formes. Son intégration dans le JDK le standardise.

Il sera dorénavant possible de gérer les onglets comme des containers. Cette fonctionnalité permettra de disposer dans les onglets n'importe quel composant ou ensemble de composants. La classe `JtabbedPane` s'enrichit de la méthode `setTabComponent(un composant)`.

N'importe quel `JTextComponent` sera imprimable par un simple appel à la méthode `print()`.

A

Le formalisme UML

Ce court chapitre a pour but d'éclaircir certaines notations UML que nous utilisons dans cet ouvrage. Nous supposons que vous manipulez déjà les concepts objet.

UML est l'acronyme de *Unified Modeling Language*. UML est donc un langage, sa syntaxe est graphique et sa sémantique est celle des concepts objet. Parmi l'ensemble du formalisme UML, nous utiliserons dans ce livre le diagramme de classe et le diagramme de séquence, que nous allons présenter ci-dessous.

Tous les diagrammes UML de cet ouvrage ont été réalisés avec l'outil *Together* de *TogetherSoft*.

A.1 DIAGRAMME DE CLASSES

Un diagramme de classes est un formalisme graphique qui permet de montrer un ensemble de classes et leurs relations, ainsi que des informations sur ces classes et ces relations.

A.1.1 Classe

Une classe est décrite par (figure A.1) :

- son nom ;
- le nom et le type des attributs ;
- la signature paramétrique des méthodes et le type de retour ;

- la visibilité des membres (*private*, *protected* ou *public*) ;
- la nature des membres : abstrait ou concret ;
- la portée des membres : membre d'instance ou membre de classe.

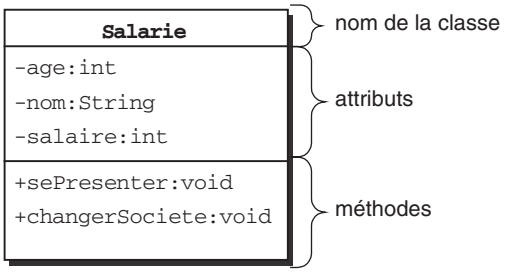


Figure A.1 — Représentation UML d'une classe.

Notez qu'en UML, la syntaxe est inversée par rapport à Java : <nom de l'attribut> : <type de l'attribut>.

A.1.2 Instance

Il est parfois utile de présenter sur un diagramme de classes certaines instances pour illustrer une partie difficile du modèle ou pour éviter toute ambiguïté (figure A.2).

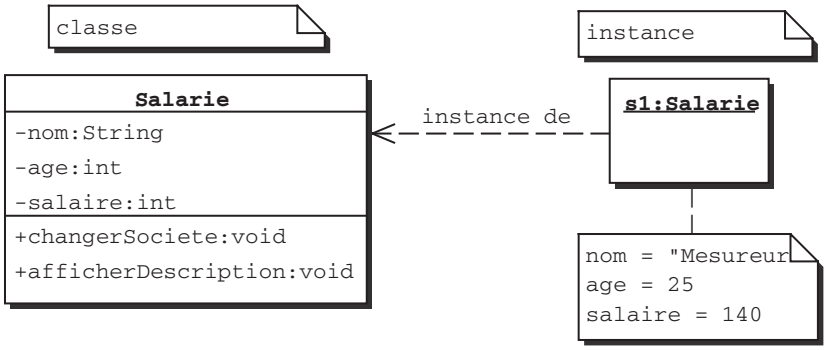


Figure A.2 — Représentation UML d'une instance.

Notez que les termes « instance » et « objet » sont équivalents. Le terme « instance » est cependant plus précis et il a notre préférence. On mentionne par exemple la « programmation orientée objet », mais derrière le terme « objet » on évoque le concept de classe autant que celui d'instance.

Il peut sembler redondant de nommer les instances, mais il n'en est rien. Ici s1 est le nom donné à une instance de la classe Salarie. s1 est assimilable à un OID, *Object Identifier*, des bases de données orientées objet.

Pour une instance on ne répète pas le type des attributs ni, bien sûr, les méthodes. Seules les valeurs des attributs sont mentionnées.

A.1.3 Visibilité

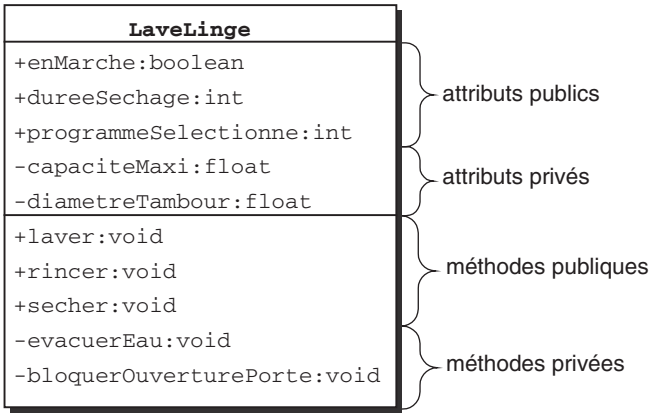


Figure A.3 — Représentation UML de la visibilité des membres d'une classe.

Les notations UML pour identifier un membre *private*, *public* et *protected* sont respectivement « – », « + » et « # ».

A.1.4 Relations entre les classes

Héritage

La figure A.4 montre la représentation UML de l'héritage.

Notez que le formalisme UML permet l'héritage multiple, mais Java ne le permet pas (figure A.5).

Le nom d'une classe abstraite est représenté en italique.

De même, les membres abstraits d'une classe sont représentés en italique.

Association

Les associations sont la forme la plus simple de relations entre classes. Elles peuvent correspondre à des attributs ou des références dans le code (figure A.6).

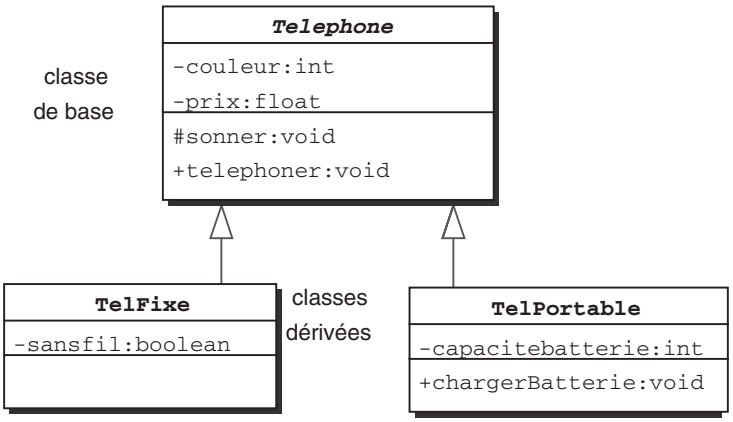


Figure A.4 — Représentation UML de l'héritage.

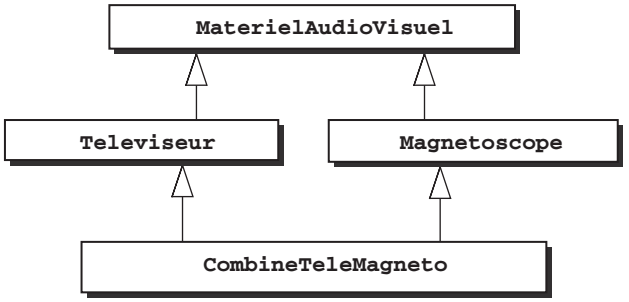


Figure A.5 — Héritage multiple.

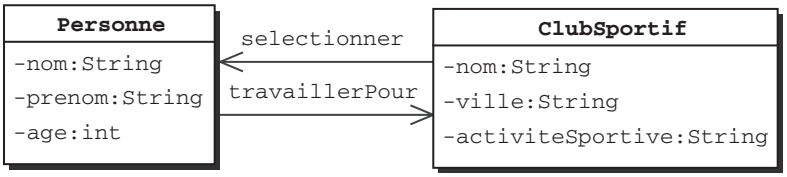


Figure A.6 — Représentation UML des associations.

Les associations sont qualifiées par un texte porté par le trait de l'association. Dans le diagramme (figure A.6), cela permet par exemple de distinguer la nature des deux relations qui unissent ces deux classes.

Ce diagramme pourrait se traduire par le code suivant, bien qu'il manque des informations pour que le code soit complet :

```
public class Personne {
     visibilité type nom;
     visibilité type prenom;
     visibilité type age;
     visibilité ClubSportif club;
}

public class ClubSportif {
     visibilité type nom;
     visibilité type ville;
     visibilité type activitesSportives;
     visibilité Personne membreDuClub;
}
```

Il manque encore des informations concernant la cardinalité, de sorte qu'il est difficile de déduire tout le code que représenterait ce diagramme UML.

Cardinalité des associations

Cardinalité 1-1

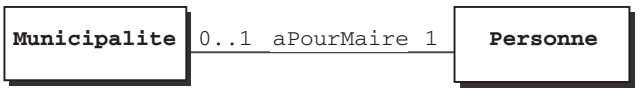


Figure A.7 — Cardinalité 1-1 sur une association (obligatoire).

Cardinalité 1-n

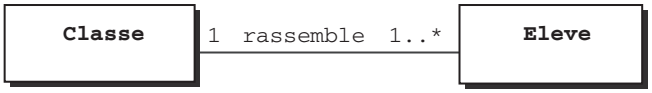


Figure A.8 — Cardinalité 1-n sur une association.

Cardinalité 0 ou 1, indique que c'est optionnel.

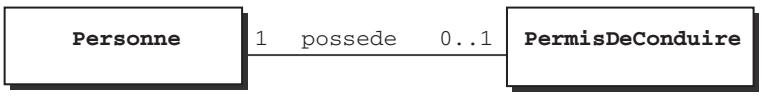


Figure A.9 — Cardinalité 1-0..1 sur une association (optionnel).

Cardinalité n-n

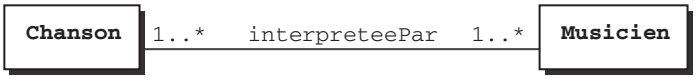


Figure A.10 — Cardinalité n-n sur une association.

Autres expressions valides de la cardinalité

- 1 (exactement une)
- 1..* (un ou plus)
- 3..5 (trois à cinq)
- 2,4 (deux ou quatre)

Attention à la syntaxe qui est souvent une source de confusion.

Figure A.11 — Comment lire la cardinalité sur une association.

Le schéma figure A.11 se lit de la façon suivante :

- une instance de *Classe1* est liée à *n* instance(s) de *Classe2* ;
- une instance de *Classe2* est liée à *m* instance(s) de *Classe1*.

Agrégation

L'agrégation est une forme de relation entre classes plus forte que l'association. Elle peut se traduire par la phrase « est composé de ». Dans l'exemple figure A.12, il est clair qu'une chaîne hi-fi est composée de deux enceintes, d'un ampli et d'un lecteur CD.

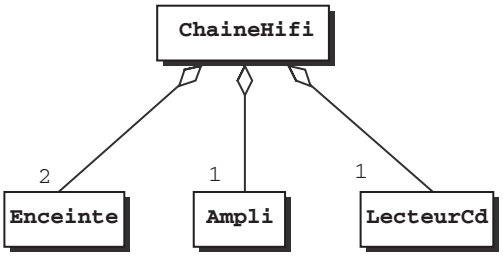


Figure A.12 — Représentation UML de l'agrégation.

En revanche, dans l'exemple présenté un peu plus haut, on avait une association entre la classe Chanson et la classe Musicien. Il s'agit bien d'une simple association car une chanson n'est pas composée de musiciens, ni réciproquement.

L'agrégation possède une variante : la composition. La composition se représente avec un losange noirci. Elle est encore plus forte que l'agrégation. Sémantiquement, elle se traduit toujours par « est composé de », mais les classes sont plus fortement liées. Par exemple, une chaîne hi-fi est composée de deux enceintes, c'est une agrégation. Un être humain est composé de deux bras, c'est une composition. Pourquoi ? Parce que les bras sont liés à l'homme plus intimement que les enceintes ne sont liées à la chaîne hi-fi. Une autre manière de voir les choses consiste à considérer la probabilité, si A est composé de B, de trouver des instances de B sans A. Il est possible de trouver une enceinte sans chaîne hi-fi, dans un magasin de pièces détachées par exemple. Un homme sans tête est si rare, que nous n'hésiterions pas à utiliser une composition si nous devions modéliser la relation entre la classe Homme et le classe Tête. Dans l'implémentation, cela pourrait se traduire par les notions suivantes :

- la classe `Tête` n'est pas absolument indispensable : ses membres pourraient être directement rattachés à la classe `Homme`;
- la classe `Tête` pourrait être une classe incluse de la classe `Homme` ;
- l'attribut `Tête` de la classe `Homme` ne devrait, dans ce cas, jamais être égal à `null`.

Utilité des diagrammes d'instances

Voici figure A.13 un diagramme de classes.

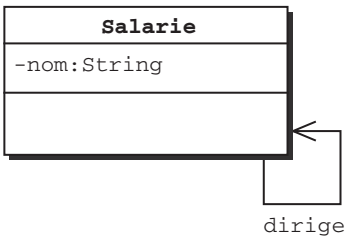


Figure A.13 — Une association réflexive.

Ce diagramme montre une relation réflexive qui n'est pas toujours simple à comprendre. Voici figure A.14 un diagramme d'instances qui l'illustre.

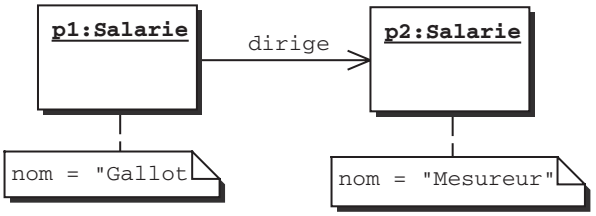


Figure A.14 — Diagramme d'instance illustrant un diagramme de classe.

Nous pourrions en écrire encore long sur le diagramme de classe en UML, mais cette annexe a pour unique but de vous résumer les principaux éléments de syntaxe. Voyons maintenant la syntaxe graphique du diagramme de séquence.

A.2 DIAGRAMME DE SÉQUENCE

L'objectif du diagramme de séquence est de décrire l'enchaînement dans le temps des messages que s'échangent les instances. Le mot « message » est un terme générique qui désigne des appels de méthode ou des transmissions d'événements. Le temps s'écoule du haut vers le bas (figure A.15).

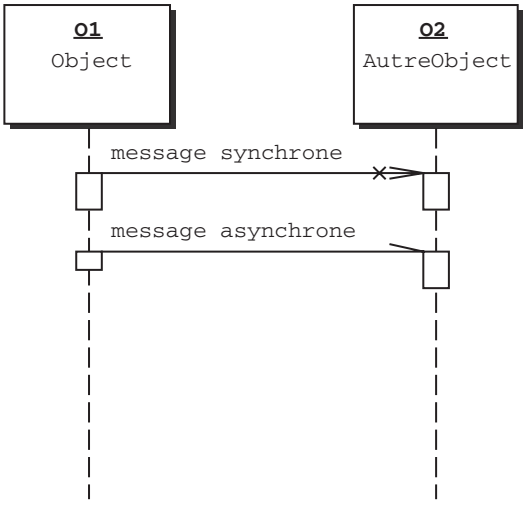


Figure A.15 — Diagramme de séquence.

Un message est représenté par une flèche. Si le message désigne un appel de méthode, alors la méthode se trouve sur l'objet que désigne la pointe. Si « message asynchrone » est un appel de méthode, alors la classe `AutreObjet`, dont `o2` est une instance, porte la méthode `message asynchrone`. En d'autres

termes, `O1`, instance de la classe `Objet` possède une référence de type `AutreObjet` sur l'instance `O2` et utilise la méthode `message asynchrone` de `AutreObjet`.

Un message est asynchrone quand l'appelant n'attend pas de réponse, le message n'est pas bloquant. Dans la terminologie Java, cela correspond à déclencher un *thread*.

Voici comment représenter une instanciation, une destruction d'une instance et un appel de méthode réflexif.

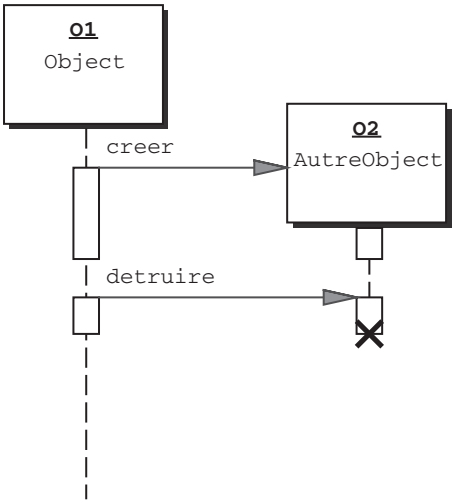


Figure A.16 — Des messages particuliers, le constructeur et le destructeur.

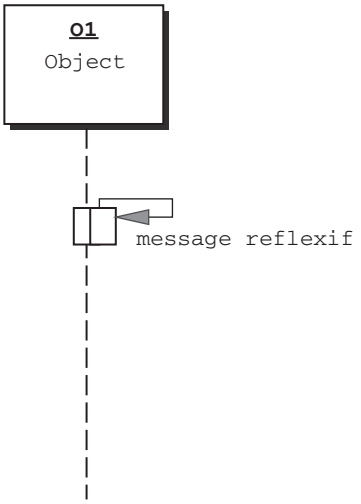


Figure A.17 — Un message réflexif.

A.3 UML : BIEN PLUS QU'UNE REPRÉSENTATION DU CODE

Attention toutefois à ne pas trop prendre les modèles UML comme une sorte de programmation graphique. De fait, dans le cadre de cet ouvrage, nous avons beaucoup utilisé UML pour décrire schématiquement des extraits de code, des patterns. Il ne faut pas réduire le formalisme UML à cet usage. Une première raison à cela est que ce formalisme est également prévu pour représenter des cas d'utilisation, *use cases*. Les diagrammes de *use cases* ont pour but de définir le système d'un point de vue fonctionnel, on ne se préoccupe pas du tout du code à cette étape-là.

Dans le cadre d'un projet orienté objet, nous rencontrons les mêmes concepts objet, que l'on se place à une étape très générale de spécifications fonctionnelles ou bien dans un module de code très bas niveau : les concepts orientés objet sont invariants quel que soit le niveau de précision auquel on se trouve. Ces concepts sont donc fractals.

C'est ainsi que l'on peut utiliser UML pour découvrir la structure des classes métier dès la phase de spécifications. Mais on peut également utiliser le même type de diagramme — diagramme de classes — en cours de réalisation pour représenter une partie du code à l'aide d'un outil de *reverse engineering*.

Index