Introduction aux Interfaces Homme-Machine

Année Universitaire 2017-2018

Auteurs : Bernard GIRAU, Samuel CRUZ-LARA, Slim OUNI, Vincent THOMAS, Isabelle DEBLED-RENNESSON

1 Eclipse

1.1 Présentation générale d'Eclipse

Pré-requis

Avant d'aborder cette partie, vous devez savoir :
— compiler des classes à partir d'une console.
— exécuter des classes à partir d'une console.
— lancer des programmes avec des bibliothèques (`classpath`).

Contenu

Cette section présentera :
— le fonctionnement d'Eclipse.
— la notion de vue et de perspective.
— la manière d'utiliser au mieux Eclipse .

1.2 Présentation

Eclipse est un **environnement de développement intégré** (`IDE`), c'est à dire un programme qui regroupe :
— un éditeur de texte pour écrire et modifier du code source.
— un compilateur pour produire les applications.
— des outils de fabrication automatique (structure pré-remplie pour définir une classe, des *getters*, des *setters*, ...).
— et souvent un debugger pour pouvoir suivre l'exécution du programme pas à pas et détecter des bugs.

De plus, Eclipse est une plate forme modulaire à laquelle il est possible de rajouter de nombreux plug-ins proposant de nouvelles fonctionnalités (modélisation UML, accès aux bases de données, XML, ...). Ainsi, Eclipse a été au départ conçu pour du développement JAVA mais de nombreux plug-ins permettent d'utiliser Eclipse comme environnement de développement pour de nombreux autres langages (PHP, C++, Python, ...).

1.3 Installation

Eclipse est installé sur les différentes machines de l'IUT.

Eclipse est un IDE gratuit et "open source". La page officielle `http://www.eclipse.org/` regroupe toute l'information liée à Eclipse (plateforme et plugins) ainsi qu'une section de téléchargement où vous pourrez obtenir la dernière version.

1.4 Philosophie d'Eclipse

Eclipse fonctionne par **projet**. Un projet correspond à une application à part entière constituée
— d'un ensemble de packages
— d'un ensemble de classes/interfaces dans ces packages
— d'un ensemble de ressources (fichiers sonores, images, fichiers textes, ...)

Les fichiers associés aux différents projets sont stockés dans un espace de travail (*workspace*). A l'IUT, sous Windows, ce workspace se trouve par défaut dans le répertoire `u:\workspace`, mais il est possible de changer la localisation du workspace au lancement d'Eclipse.

Chaque projet correspond à un sous-répertoire de l'espace de travail.

1.5 Interface d'Eclipse

Eclipse est basé sur des **perspectives**. Une perspective correspond à une utilisation particulière d'Eclipse et présente le code selon certains aspects.

Une perspective contient plusieurs **vues**, chaque vue présentant un aspect particulier de l'application.

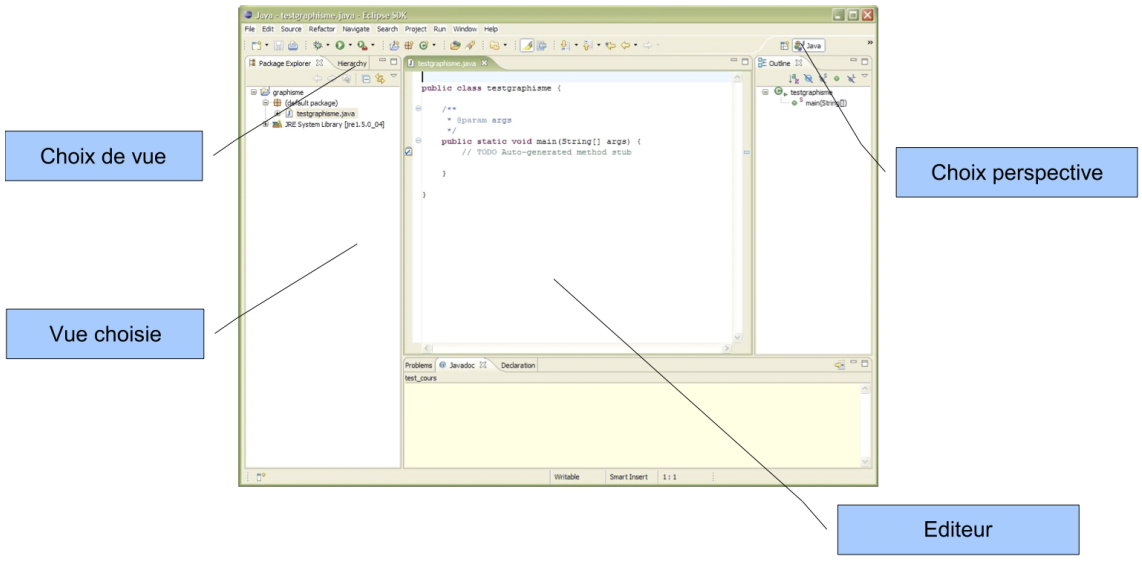


FIGURE 1 – Interface générale d'Eclipse

1.5.1 Perspectives

Eclipse propose les perspectives suivantes :

— **Java** une perspective utilisée pour développer du code JAVA, c'est celle que vous serez amenés à utiliser le plus fréquemment.
— **Navigation JAVA** une perspective utilisée pour parcourir rapidement les différentes classes d'un projet.
— **Hiérarchie JAVA** une perspective utilisée pour parcourir les classes en fonction de l'arbre de hiérarchie entre les classes (relation d'héritage).
— **Debuggage** une perspective utilisée pour suivre les variables en cours d'exécution de l'application et localiser des bugs.
— **Ressource** une perspective utilisée pour afficher les ressources associées au projet (fichiers).

Les autres perspectives ne seront pas abordées, Il reste néanmoins

— **Développement de Plug-ins** une perspective utilisée pour développer des plugins d'Eclipse
— **Synchronisation** une perspective utilisée pour travailler à plusieurs
— **CVS** une perspective utilisée pour le travail collaboratif.

1.5.2 Vues

Eclipse propose de nombreuses vues associées aux différentes perspectives. Parmi les vues possibles, les suivantes peuvent se révéler intéressantes pour vos projets

— **Package explorer** Cette vue permet d'afficher l'ensemble des classes, méthodes, attributs d'un projet et de pouvoir accéder directement au code de la classe ou de la méthode en cliquant sur le nom
— **Hierarchy** Cette vue permet d'afficher la hiérarchie d'une classe et de naviguer dans les sous classes et les super classes
— **Javadoc** Cette vue affiche la javadoc de la méthode surlignée dans l'éditeur
— **Déclaration** Cette vue affiche la déclaration de la méthode surlignée dans l'éditeur

Il est possible d'afficher une vue spécifique en utilisant le menu `'Window --> Show view'`.

1.6 Utilisation d'Eclipse

Développer une application sous Eclipse se fait en plusieurs étapes

1. Créer un projet
2. Remplir le projet avec des entités (packages, classes, interfaces, ...)
3. Compiler l'application
4. Exécuter l'application

1.6.1 Créer un projet

Pour créer un projet, il suffit de sélectionner le menu `'File --> New --> Project'` ou de sélectionner l'icône `'new project'`.

Il suffit ensuite de sélectionner projet JAVA parmi tous les types de projets proposés. Une fenêtre présentant les différentes caractéristiques du projet s'ouvre. Il est possible de spécifier les bibliothèques à inclure dans le projet.

1.6.2 Créer des entités

Une fois qu'un projet est défini, il suffit de créer des classes, des interfaces et des packages qui sont automatiquement ajoutés au projet. Pour créer une entité, il suffit de sélectionner le menu `'File --> New --> Entité_a_créer'`.

Au moment de la création d'une classe, il est possible de lui spécifier différentes caractéristiques (comme le fait que la classe puisse être exécutable ou non).

Une fois l'entité validée, Eclipse crée le fichier et écrit le squelette de la classe (avec éventuellement un `main` si la classe a été spécifiée comme exécutable).

1.6.3 Compilation

Eclipse effectue une compilation dès qu'une modification du code source est faite. Vous pouvez donc suivre à tout moment les erreurs générées à la compilation.

Ces erreurs sont surlignées en rouge dans l'éditeur. Toutes les erreurs associées aux projets en cours sont répertoriées dans la vue `'Problems'`. En cliquant sur l'erreur, on accède directement dans l'éditeur à l'endroit où l'erreur a été détectée.

Si aucune erreur n'a été détectée, le code est correctement compilé et est prêt à être exécuté.

1.6.4 Exécution

Pour exécuter un projet, le plus simple consiste à effectuer un clic droit sur la classe constituant le point d'entrée du projet et choisir le menu `'Run as --> JAVA application'`.

Une fenêtre s'ouvre et permet de choisir diverses options (y compris les options transmises à la JVM)
Au lancement de l'application, la vue `'Console'` permet de suivre ce qui se passe à l'écran.

1.6.5 Fermeture des projets

Dés que vous ne travaillez plus sur un projet, pensez à le fermer. Dans le cas contraire, la vue `'Problems'` présentera toutes les erreurs liées à tous les projets ouverts et peut vite devenir illisible.

1.7 Éditeur de code

L'éditeur de code permet de modifier du code source. Il est constitué de deux éléments
— le code source lui même
— une ligne d'icônes sur la gauche répertoriant les différentes erreurs et avertissements pour y accéder plus rapidement
— une ligne sur la droite présentant une vue d'ensemble du fichier.
L'éditeur offre de nombreuses fonctionnalités. Ce document en aborde un certain nombre.

1.7.1 Complétion automatique

Il est possible de demander à Eclipse des suggestions pour compléter un début de code. Il suffit pour cela de commencer à écrire et d'appuyer sur les touches **CTRL + espace**. Eclipse propose alors plusieurs possibilités pour compléter le mot.

> **ATTENTION** Il s'agit d'un des outils les plus pratiques d'Eclipse qui font qu'il est très agréable de l'utiliser. Mais une utilisation abusive de cet outil peut être très dangereuse. Évitez à tout prix de sélectionner les suggestions d'Eclipse sans être sûr de ce que vous faites.

La complétion automatique fonctionne
— sur les méthodes : il suffit d'écrire le début de la méthode et Eclipse complète en fournissant les noms des méthodes compatibles ;
— sur les classes anonymes : il suffit d'écrire le début de la classe Anonyme et Eclipse complète en ajoutant les méthodes à redéfinir ;

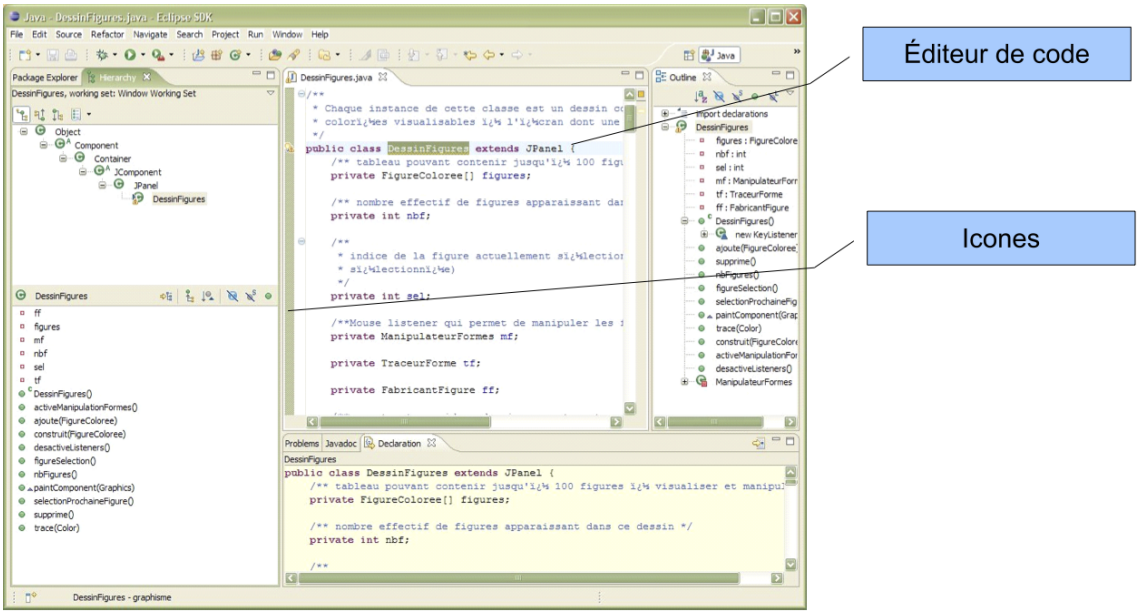


FIGURE 2 – Éditeur de code

— sur certaines structures : il suffit d'écrire `'if'` et Eclipse construit un squelette correspondant à une condition ;
— sur certaines macro prédéfinies : il suffit d'écrire `'sysout'` et d'appuyer sur control+espace et Eclipse remplace la chaîne par `System.out.println`.

Certaines macros (`'sysout'` par exemple) et certaines structures (`'if'`, ...) sont définies et modifiables dans les options de l'éditeur. Ces options sont accessibles par le menu `'Window --> Preferences --> JAVA --> Editor'`.

1.7.2 Correction automatique des erreurs

Lorsqu'Eclipse détecte une erreur de compilation, la colonne à gauche de l'éditeur présente un panneau rouge. Lorsqu'on clique sur ce panneau, Eclipse propose certaines réponses classiques pour résoudre le problème.

> **ATTENTION (bis)** Il s'agit à nouveau d'un outil très pratique, mais soyez bien sûr de comprendre ce qu'Eclipse vous propose avant de sélectionner une réponse.

1.7.3 Génération de code

Eclipse peut aussi générer automatiquement des morceaux de code comme
— les importations : sélectionner la classe à ajouter et appuyer sur **control+M**.
— les constructeurs : dans le menu `'Source --> Generate constructor using fields'`
— les *getter* et *setter* : dans le menu `'Source --> Generate getters et setters'`

1.7.4 Modification de code

Eclipse propose aussi comme fonctionnalité

— le formatage de code : sélectionner le texte à formater et sélectionner le menu `'Source --> format'`. Les règles de formatage peuvent être modifiées dans le menu `'Window --> preferences --> JAVA'`
— la mise en commentaire d'une partie de code : sélectionner la portion de code et sélectionner le menu `'Source --> Commentaires'`
— les blocs try/catch : sélectionner le bloc à protéger et choisir le menu `'Source --> surround with --> bloc try/catch'`

1.7.5 Le *refactoring*

Eclipse permet aussi de renommer automatiquement une classe, un attribut ou une méthode. Dans ce cas, ce n'est pas seulement la déclaration de la classe ou de la méthode qui est modifiée mais aussi toutes les références contenues dans tous les fichiers du projet.

On accède à cette option par le menu `'Refactor --> Rename'`.

1.7.6 Mise en garde

L'utilisation d'Eclipse permet d'automatiser un certain nombre d'opérations et cela constitue un de ses intérêts. Méfiez vous cependant de l'utilisation abusive de ces fonctionnalités qui peuvent conduire à de nombreuses erreurs.

Par exemple, la complétion automatique des noms de méthodes est très pratique, mais il faut que vous connaissiez à l'avance la méthode à rechercher. En choisir une en fonction de son nom sans connaître l'API peut conduire à des désastres.

1.8 Références externes

Si vous souhaitez plus d'informations sur Eclipse, voici quelques références
— le site web d'Eclipse
`http://www.eclipse.org/`
— le document très complet "développons en JAVA avec Eclipse" de JM Doudoux
`http://www.jmdoudoux.fr/accueil_java.htm`

2 Debugger avec Eclipse

Contenu

Dans cette partie, on expliquera
— ce qu'est un debugger et à quoi il sert
— comment fonctionne le debugger sous Eclipse
— comment mettre des points d'arrêts (avec conditions éventuelles)
— comment faire du suivi de variable
— comment parcourir le contenu de la mémoire

2.1 Définition

2.1.1 Debugger

Un debugger est un programme qui permet de suivre l'exécution d'un autre programme, de l'arrêter à certains endroits, de vérifier la valeur de certaines expressions et de sonder l'état de la mémoire.

Il s'agit d'un programme très pratique pour permettre au concepteur d'une application de détecter des erreurs et de les corriger.

2.1.2 Utilisation

Le programme à debugger s'exécute par l'intermédiaire du debugger. Le debugger permet alors de contrôler le programme

- il est possible de mettre en pause le programme à tout instant
- il est possible de suivre le contenu de certaines variables et l'état complet de la mémoire
- lorsque le programme à debugger est mis en pause, le debugger spécifie l'endroit du code source où le programme a été interrompu.

2.2 Debugger sous Eclipse

2.2.1 Accéder au debugger

Eclipse intègre un debugger. Pour accéder au débugger, il suffit simplement de sélectionner la perspective 'debugger' représentée par l'icône Debug

FIGURE 3 – Icone debug

2.2.2 Description de la perspective debugger

La perspective debugger propose par défaut plusieurs vues

- le cadre en haut à gauche présente la Vue `'Debug'` qui affiche la liste des processus JAVA en cours d'exécution. Il est possible de sélectionner un processus, de le mettre en pause et d'effectuer une exécution pas à pas.
- le cadre en haut à droite présente différentes vues : la vue `'Breakpoints'` pour la gestion des points d'arrêts (cf partie 2.3.1), la vue `'Variable'` pour le suivi du contenu de la mémoire et la vue `'Expressions'` pour le suivi des expressions.
- le cadre au milieu présente l'éditeur. Les points d'arrêts sont visibles sur le coté gauche (dans la ligne d'icône)
- le cadre en bas présente la vue `'Console'`.

2.2.3 Lancer un programme avec le debugger

Pour lancer un programme java avec le debugger, il suffit de cliquer sur l'icône `'Debug'`. C'est alors le dernier programme JAVA qui a été lancé qui est exécuté.

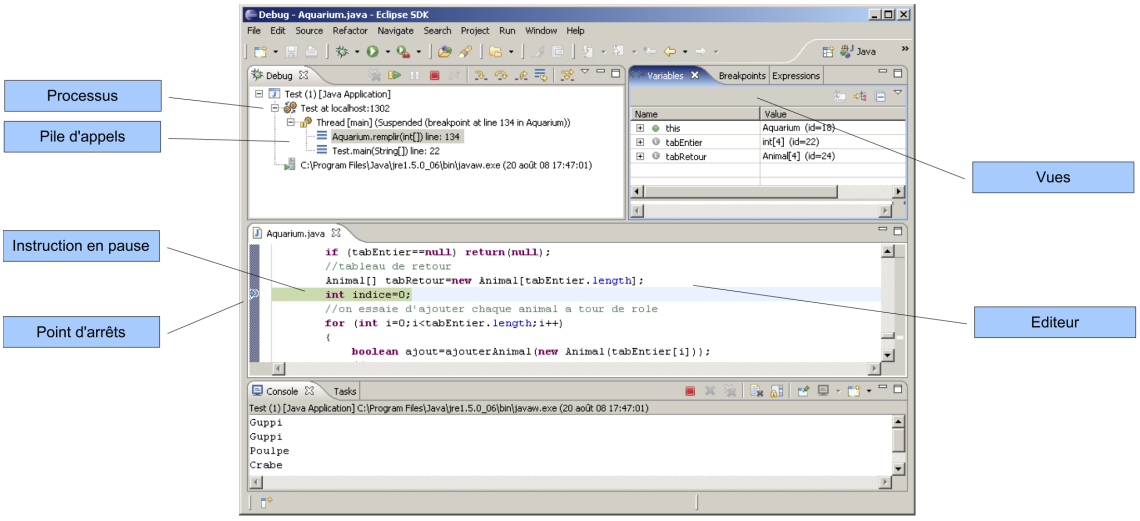


FIGURE 4 – Perspective Debug

Pour plus d'options, il faut sélectionner une classe et choisir le menu `'debug as --> debug'` après un clic droit.

2.3 Outils

Le debugger d'Eclipse propose plusieurs outils classiques pour debugger un programme.

2.3.1 Les points d'arrêt

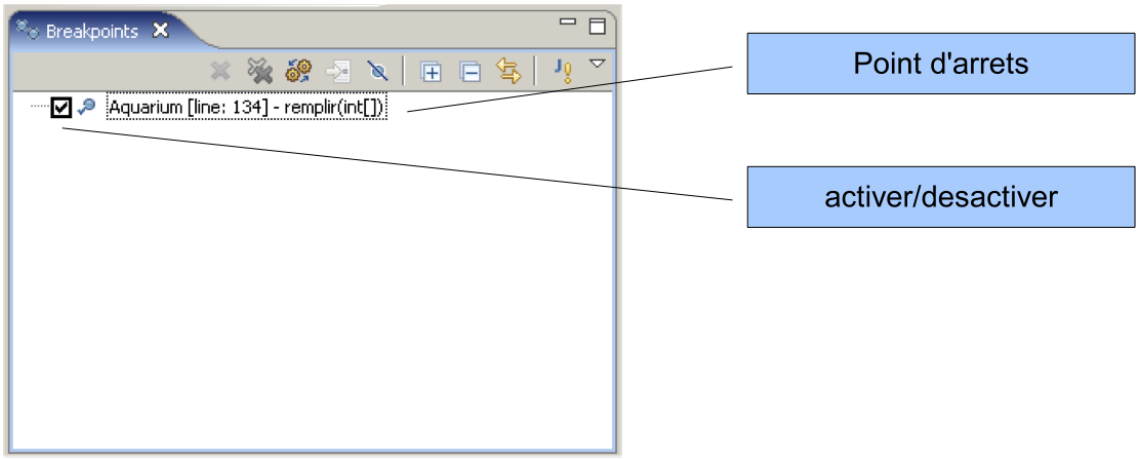


FIGURE 5 – Vue Breakpoints : La liste des points d'arrêt

Il est possible d'ajouter des points d'arrêt dans le code. Dés que le debugger arrive sur un point d'arrêt, il met automatiquement le programme en pause.

Cela permet au concepteur du programme d'arriver sur l'endroit du code qui parait problématique pour analyser ce qui s'y passe en détail.

Par défaut Il suffit de double-cliquer sur la ligne d'icône de l'éditeur pour poser un point d'arrêt (représenté par une puce bleue) à la ligne correspondante.

Sous condition Il est possible de rajouter des conditions d'arrêt sur le point d'arrêt (bouton droit sur un point d'arrêt et sélectionner le menu `propriétés`)

— soit par rapport à une condition booléenne

— soit par rapport à un nombre de passages sur le point d'arrêt - ce qui peut être très utile pour accéder à une certaine itération d'une boucle

2.3.2 Exécution pas à pas

Quand l'application est en pause (soit à cause d'un point d'arrêt soit manuellement) il est possible d'exécuter l'application pas à pas en utilisant les icônes de la vue `'Debug'`.

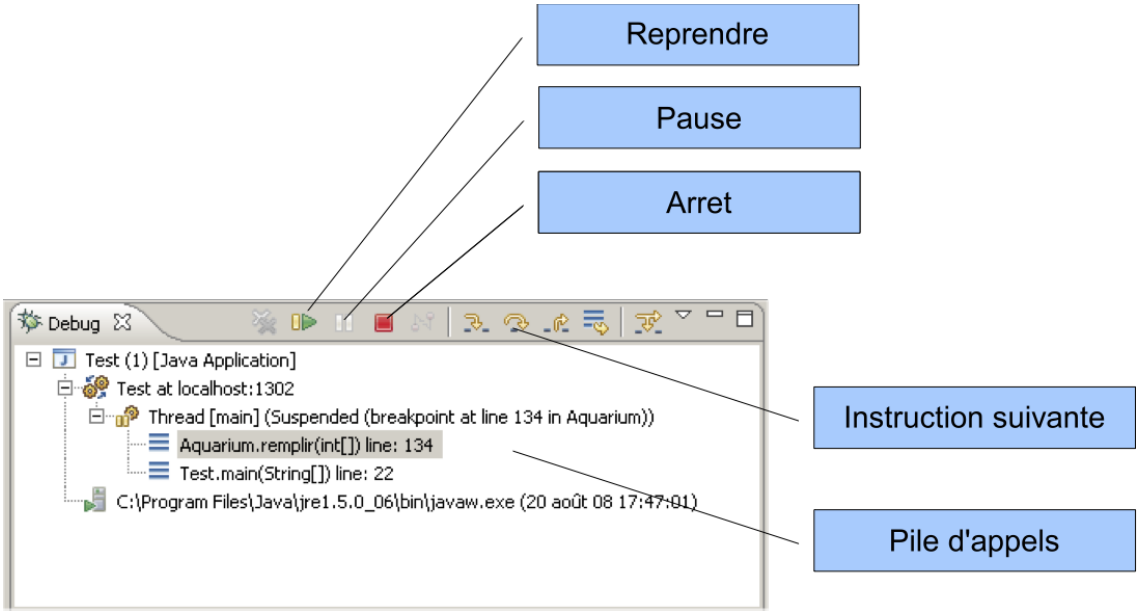


FIGURE 6 – Vue Debug : Contrôle de l'exécution

2.3.3 Le suivi d'expressions

La vue `'Expressions'` permet de suivre l'évolution d'expressions préalablement écrites. Une expression peut être une variable ou une opération impliquant des variables et des constantes.

Suivi d'expression Le suivi d'expression est automatique. A chaque pause de l'application java à debugger, la valeur des expressions est mise à jour

Ajouter une expression Il suffit de faire un clic droit sur la vue des expressions et de suivre les menus.

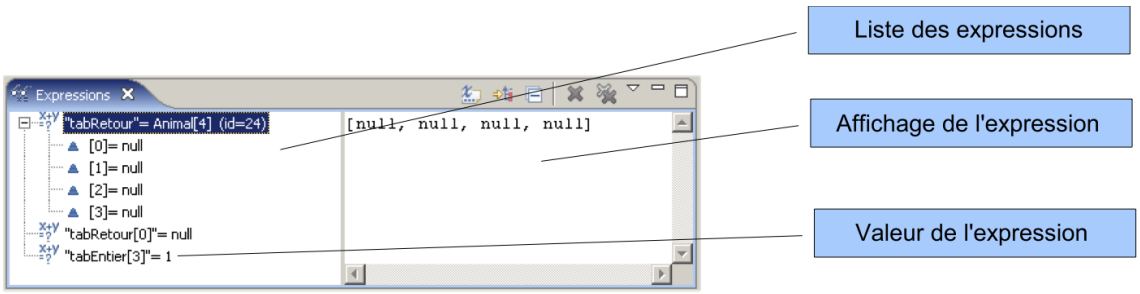


FIGURE 7 – Vue Expression : Suivi d'expressions à l'exécution

2.3.4 L'affichage du contenu de la mémoire

La vue `'Variable'` permet de vérifier le contenu de la mémoire. On accède à la pile et il est possible de suivre les références en cliquant sur le + pour connaître le contenu de la variable.

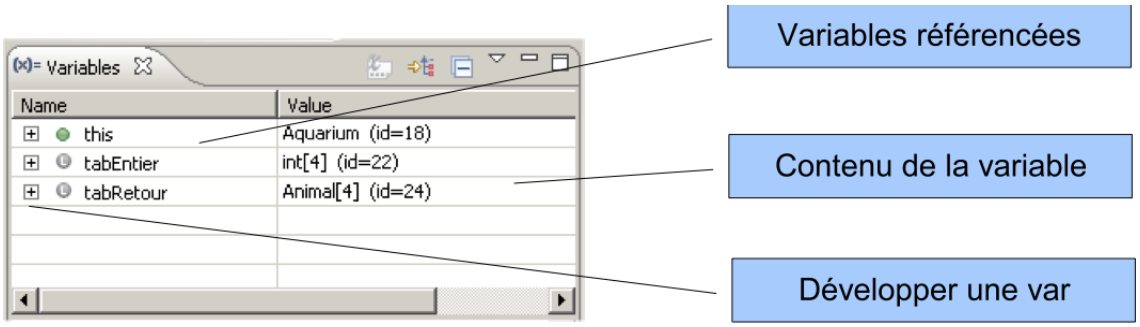


FIGURE 8 – Vue Variable : contenu de la mémoire

Il est par exemple possible d'accéder au contenu d'un tableau ou d'une `ArrayList`.
Le contenu de la variable s'exprime de la forme suivante :
— si c'est un **type primitif**, la valeur est affichée directement sous la forme : `valeur` .
— si c'est un **type objet**, le type objet suivi de la référence est affiché sous la forme : `type(id=ref)`
  — où `type` désigne le type de l'objet
  — et `ref` est un nombre désignant la référence de l'objet

2.3.5 Bilan

Le debugger est un outil très puissant qui permet d'éviter d'insérer des `println` dans un code (pratique désastreuse).

Il permet de suivre l'évolution d'un programme, de ses variables et du contenu de la mémoire.

Il est à noter que seule l'utilisation fréquente d'un debugger permet d'en tirer convenablement parti. Il est nécessaire d'expérimenter et de se familiariser avec ces applications pour pouvoir réellement en profiter. Mais cela constitue ensuite un réel plus lorsque vous serez confrontés à un problème.

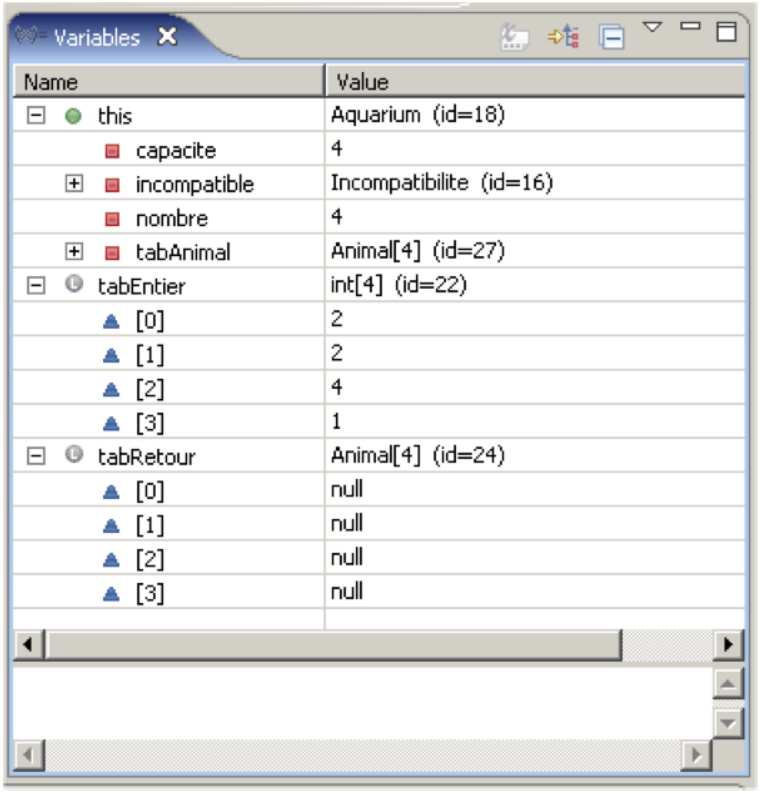


FIGURE 9 – Vue Variable : contenu plus développé

2.4 Objectif pédagogique

A l'issue de cette section vous devrez savoir
— comment exécuter un projet en mode debug sous Eclipse
— comment poser des points d'arrêts
— comment vérifier le contenu de la mémoire et des différentes variables

2.5 References

— le document très complet "développons en JAVA avec Eclipse" de JM Doudoux chapitre 8 `http://www.jmdoudoux.fr/accueil_java.htm`

3 Introduction au graphisme

Introduction

La réalisation d'une application munie d'une bonne *interface graphique* (fenêtres, menus, etc.) est une tâche assez délicate. Une interface moderne comporte des dizaines de menus, boites de dialogue, fenêtres, etc., ce qui aboutit à des centaines de lignes de code. De plus, ces lignes de code introduisent une complexité qui n'est pas liée aux algorithmes et aux traitements que l'interface graphique est chargée d'illustrer.

Dans la pratique, il est donc indispensable de bien séparer la partie interface graphique du reste du programme. Pour ce faire, l'emploi d'une approche objet est particulièrement appropriée.

3.1 Principes fondamentaux

3.1.1 Deux systèmes graphiques

Le langage `Java` contient deux systèmes distincts destinés à la création d'interfaces graphiques. L'ancien système (développé à partir de la version `1.1` de `Java`) porte le nom de *Abstract Window Toolkit* (AWT), ce qui signifie (dans ce contexte) "bibliothèque abstraite de fenêtrage".

Le nouveau système porte le nom de *Swing*. Il a été développé pour être compatible avec la version `1.1` de `Java` et est directement inclus dans le langage depuis la version `1.2`. *Swing* est une amélioration notable de *AWT*.

3.1.2 Composants graphiques *Swing*

Techniquement, *Swing* est un ensemble de classes, permettant de créer des objets. Chaque objet représente un **composant graphique**. Il existe de nombreux composants graphiques comme par exemple :

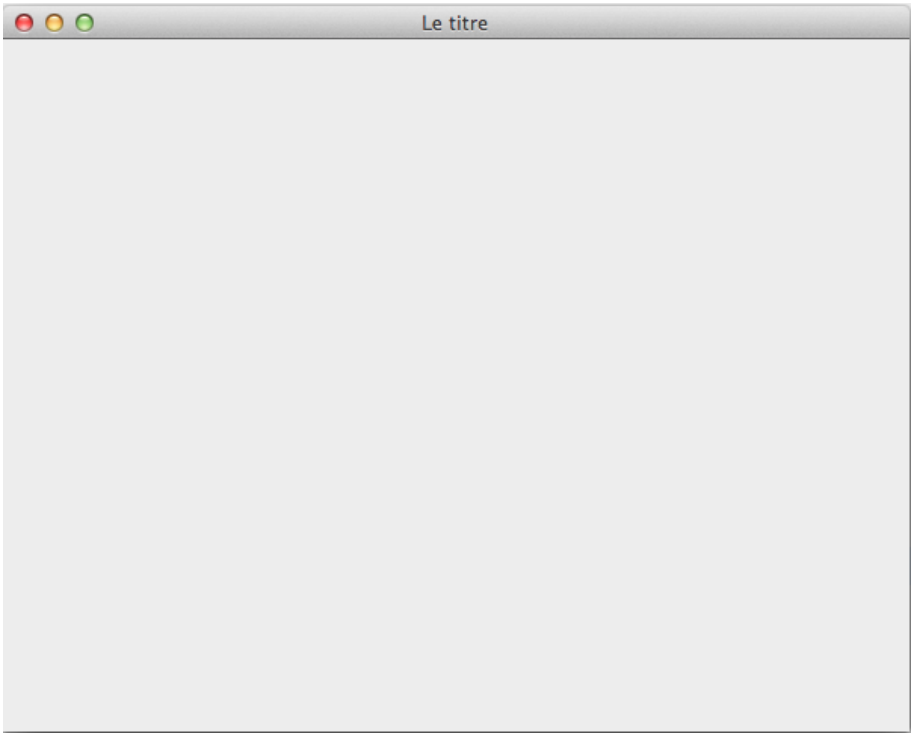
- `JFrame` : il s'agit de la fenêtre principale d'une application, celle qui possède un titre et une bordure.
- `JButton` : il s'agit d'un bouton, une zone rectangulaire sur laquelle on peut cliquer pour déclencher une action d'un programme.
- `JScrollBar` : il s'agit d'une barre de défilement, utilisée lorsque la totalité du `JPanel` ne peut être visualisée dans le `JFrame`.
- `JPanel` : il s'agit d'une zone rectangulaire vide dans laquelle un programme peut dessiner. Un `JPanel` peut aussi contenir d'autres composants graphiques. N.B. : un `JPanel` doit s'inclure dans un objet `JFrame` pour pouvoir être affiché.
- `JMenuBar` : il s'agit de la barre de menus qui dans un `JFrame` contient les différents menus de l'application.
- `JMenu` : chaque objet `JMenu` représente un menu de l'application et s'insère dans un `JMenuBar`.
- ...

Il existe dans la pratique de nombreux autres composants graphiques possibles, comme par exemple les boites de dialogue, les zones de saisie, ... etc.

3.1.3 Création d'une fenêtre principale

L'exemple suivant se contente de fabriquer et de visualiser une fenêtre vide.

```
1         import javax.swing.*;
2         public class CreationFenetreComplete {
3           public static void main(String[] args) {
4             JFrame fenetre=new JFrame("Le titre");
5             fenetre.setSize(600,480);
6             fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7             fenetre.setVisible(true);
8           }
9         }
```



La ligne 1 correspond à l'utilisation d'une partie des classes qui constituent le package *Swing*. La ligne 4 crée l'objet `JFrame` muni d'un titre (qui aparaîtra dans le haut du cadre de la fenêtre), la ligne 5 précise sa taille (en pixels), la ligne 6 permet de terminer l'application dès que la fenêtre est fermée, et la ligne 7 demande l'affichage de la fenêtre à l'écran.

Il est en général inutile de préciser la taille de la fenêtre principale, car la méthode `pack` (qui sera détaillée ultérieurement) permet d'adapter automatiquement la taille d'une fenêtre à son contenu.

Ce programme utilise donc les propriétés suivantes de la classe `JFrame` :

— `JFrame(String title)`
— `void setSize(int width,int height)`
— `void setDefaultCloseOperation(int code)` : le code est à choisir parmi les attributs de classe constants de la classe `JFrame`
— `void setVisible(boolean visible)`

3.1.4 Dessiner dans un `JPanel`

Une fenêtre `JFrame` est munie de toutes les propriétés de manipulation des fenêtres graphiques. Lorsqu'on veut préciser ce qui se passe à l'intérieur de la fenêtre, on passe par le "contenant principal" de la fenêtre, qui est représenté par un `JPanel`. La classe `JFrame` propose deux méthodes pour manipuler son "contenant principal", sachant que la classe `JPanel` dérive de la classe `Container` qui définit tous les composants graphiques capables de contenir d'autres composants graphiques :

— `Container getContentPane()` : retourne le panneau (contenant) principal de la fenêtre.

— `void setContentPane(Container c)` : donne un nouveau panneau(contenant) principal à la fenêtre.

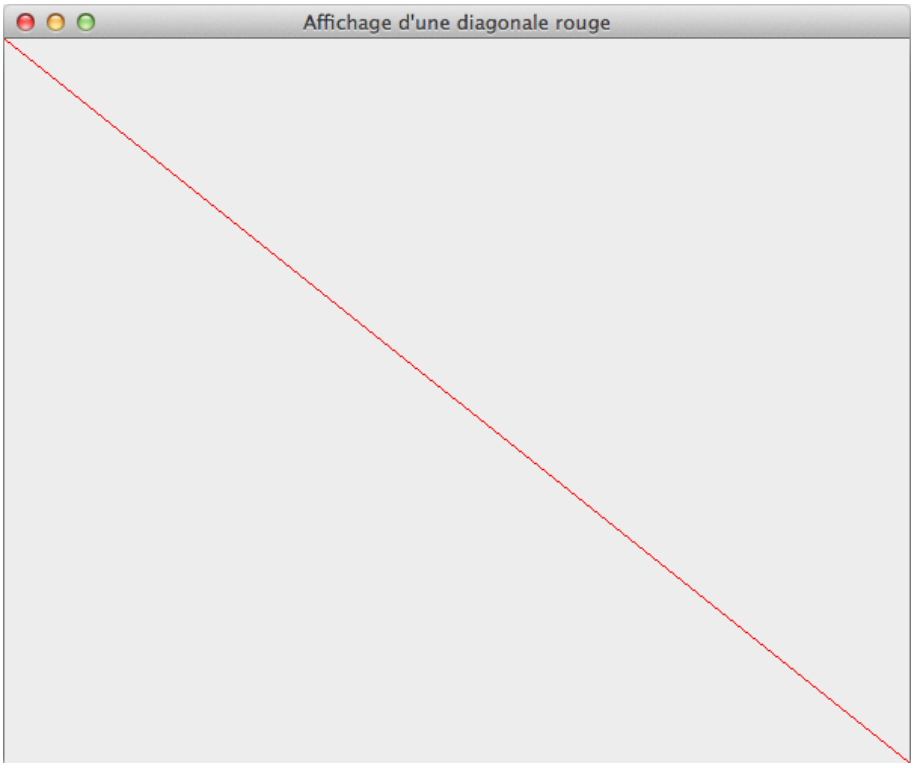
Un composant `JPanel` est muni de toutes les propriétés nécessaires pour définir son aspect et y insérer des sous-composants graphiques. Néanmoins, le modèle par défaut que définit la classe `JPanel` correspond à un "contenant vide".

Pour dessiner dans un `JPanel`, il faut personnaliser sa méthode d'affichage à l'écran, c'est à dire redéfinir la méthode `void paintComponent(Graphics g)` dans une classe qui dérive de `JPanel`.

Cette méthode est **automatiquement appelée** à chaque fois que la fenêtre contenant le `JPanel` est rafraîchie, déplacée, rouverte, ...etc. Le paramètre alors fourni est l'environnement graphique de dessin du `JPanel` : c'est cet objet qui propose les différentes méthodes de dessin.

L'exemple suivant trace une ligne rouge dans la diagonale d'une fenêtre.

```
1       import javax.swing.*;
2       import java.awt.*;
3       public class DessinDiagonale extends JPanel {
4          public void paintComponent(Graphics g) {
5            super.paintComponent(g);
6            int h=getHeight();
7            int w=getWidth();
8            g.setColor(Color.red);
9            g.drawLine(0,0,w-1,h-1);
10         }
11       }
12       public class AffichageDiagonale {
13         public static void main(String[] args) {
14           JFrame fenetre=new JFrame("Affichage␣d'une␣diagonale␣rouge");
15           fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16           DessinDiagonale dessin=new DessinDiagonale();
17           dessin.setPreferredSize(new Dimension(600,480));
18           fenetre.setContentPane(dessin);
19           fenetre.pack();
20           fenetre.setVisible(true);
21         }
22       }
```



Analyse du programme ligne par ligne :

— ligne 2 : cet `import` est indispensable pour pouvoir utiliser `Graphics`.
— classe `DessinDiagonale` : définition du `JPanel` personnalisé
  — ligne 4 : redéfinition de la méthode `paintComponent`
  — ligne 5 : permet d'éviter certains bugs d'affichage, en recommençant proprement l'affichage du composant
  — lignes 6 et 7 : on récupère la hauteur et la largeur du `JPanel` `this`
  — ligne 8 : changement de couleur de tracé dans l'environnement de dessin
  — ligne 9 : traçage d'une ligne dans le `JPanel`, du coin supérieur gauche `(0,0)` vers le coin inférieur droit `(w-1,h-1)`.
— classe `AffichageDiagonale` : programme principal faisant apparaître la fenêtre
— lignes 14,15,20 : voir l'exemple précédent
— ligne 16 : création du `JPanel` personnalisé
— ligne 17 : on indique la dimension souhaitée pour ce `JPanel` (en passant par la classe `Dimension`, dont le constructeur précise la largeur et la hauteur)
— ligne 18 : on déclare ce `JPanel` personnalisé comme "contenant principal" de la fenêtre
— ligne 19 : indique à la fenêtre de determiner sa taille en fonction des préférences des composants graphiques qu'elle contient.

Ce programme utilise donc une nouvelle propriété de la classe `JFrame` :

— `void pack()`

ainsi que diverses propriétés de la classe `JPanel` dont dérive `DessinDiagonale`.

— `int getHeight()`
— `int getWidth()`

— `void setPreferredSize(Dimension dim)`

Les instructions de dessin sont confiées à l'objet de classe `Graphics` :

— `void setColor(Color c)`

— `void drawLine(int x_from,int y_from,int x_to,int y_to)`

3.2 Méthodes de dessin

3.2.1 Modèle employé

Le `JPanel` représente une zone rectangulaire de l'écran de l'ordinateur sur laquelle on peut dessiner, via son environnement graphique. Il faut tenir compte du modèle de cet environnement.

La zone rectangulaire est constituée de points élémentaires, appelés *pixels*. Un pixel possède une surface, le `JPanel` est donc constitué d'une mosaïque de points.

Les points sont repérés par deux coordonnées **entières** (des `int`s) positives. L'origine du repère est le **coin supérieur gauche** : le pixel le plus en haut à gauche possède donc les coordonnées $(0, 0)$. L'axe des x (la première coordonnée) est orienté vers la droite. L'axe des y (la seconde coordonnée) est orienté **vers le bas** de l'écran. Attention : il s'agit donc d'un repère indirect.

Comme toujours en `Java`, un `JPanel` à `n` lignes (`getHeight()`) et `p` colonnes (`getWidth()`) correspond à des pixels dont la coordonnée en x est comprise au sens large entre `0` et `p-1` et dont la coordonnée en y est comprise au sens large entre `0` et `n-1`.

Il est important de noter que la taille de la fenêtre est *dynamique*. En effet, l'utilisateur peut à tout moment changer les dimensions de la fenêtre principale, ce qui entraîne un appel automatique à `paintComponent`. On doit donc considérer que chaque appel de `paintComponent` se fait dans un `JPanel` avec de nouvelles dimensions.

3.2.2 Tracés

Pour dessiner dans la zone rectangulaire disponible, il faut redéfinir la méthode `paintComponent`. Pour réaliser le tracé effectif, on utilise l'objet `Graphics` transmis à cette méthode.

Les méthodes proposées par la classe `Graphics` sont très nombreuses.

Outre `setColor` et `drawLine`, voici quelques méthodes utiles :

— `void drawRect(int x,int y,int width,int height)`
Trace le contour du rectangle décrit par les 4 paramètres numériques de la méthode : `x` et `y` sont les coordonnées du coin supérieur gauche du rectangle, `x+width` et `y+height` sont les coordonnées du coin inférieur droit du rectangle.

— `void fillRect(int x,int y,int width,int height)`
Même chose que la méthode précédente, mais en remplissant le rectangle avec la couleur du tracé.

— `void drawOval(int x,int y,int width,int height)`
Trace une ellipse tenant exactement dans le rectangle décrit par les 4 paramètres numériques de la méthode : `x` et `y` sont les coordonnées du coin supérieur gauche du rectangle, `x+width` et `y+height` sont les coordonnées du coin inférieur droit du rectangle.

— `void fillOval(int x,int y,int width,int height)`
Même chose que la méthode précédente, mais en remplissant l'ellipse avec la couleur du tracé.

— `void drawPolygon(int[] x_pts,int[] y_pts,int nb_pts)`

Trace le polygone dont les `nb_pts` sommets ont leurs abscisses dans le tableau `x_pts` et leurs ordonnées dans le tableau `y_pts`.

— `void fillPolygon(int[] x_pts,int[] y_pts,int nb_pts)`
Même chose que la méthode précédente, mais en remplissant le polygone avec la couleur du tracé.

— `void drawString(String s,int x,int y)`
Affiche (avec la couleur du tracé) le texte contenu dans `s` en partant du point de coordonnées (x,y).

3.2.3 Modifications dynamiques d'un dessin

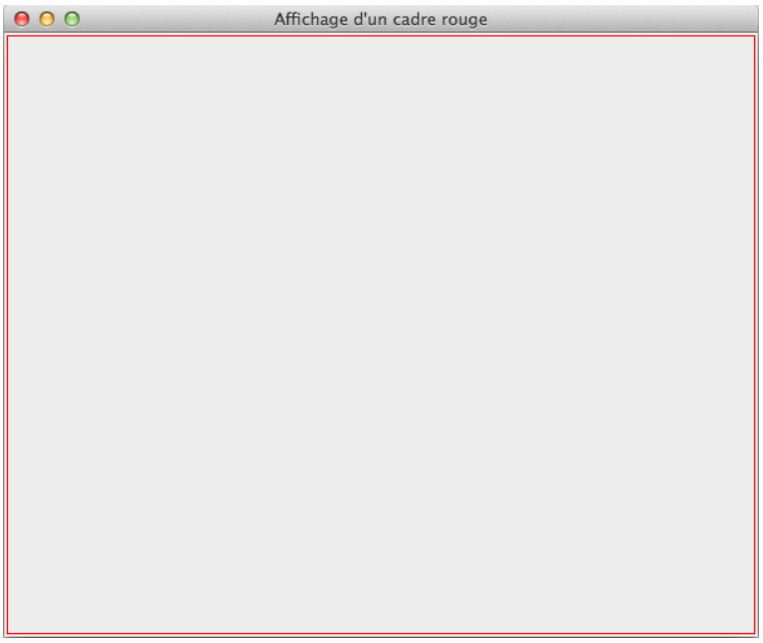
Dans l'exemple précédent, la fenêtre `JFrame` est affichée grâce à l'utilisation de la méthode `setVisible`. Comme le `DessinDiagonale` est inclus dans la fenêtre principale, il est aussi affiché, ce qui se traduit par l'appel automatique de la méthode `paintComponent`. Quand l'utilisateur modifie la taille de la fenêtre, l'ordinateur appelle encore automatiquement `paintComponent`, pour obtenir le réaffichage du dessin, dans la fenêtre retaillée. Lors de ce nouvel appel de `paintComponent`, le dessin a changé de taille, et l'utilisation des méthodes `getHeight` et `getWidth` permet d'obtenir les nouvelles dimensions, et donc d'adapter si besoin est la forme des différents tracés au nouveau cadre de la fenêtre.

Si certaines parties du programme effectuent des modifications de paramètres du tracé, sans qu'il y ait manipulation de la fenêtre, il est nécessaire de provoquer **explicitement** un appel aux méthodes d'affichage du dessin. Pour cela, il ne faut pas appeler la méthode `paintComponent`, mais une autre méthode de la classe `JPanel` : `void repaint()` qui se chargera d'appeler correctement les méthodes d'affichage des différents composants graphiques concernés.

Dans l'exemple suivant, on affiche un cadre rouge proche du bord de la fenêtre. Les modifications de forme de la fenêtre sont immédiatement répercutées sur la forme du cadre. En revanche, ce cadre a une certaine épaisseur (obtenue par rectangles superposés) que l'utilisateur peut modifier : un appel à `repaint()` permet alors de réafficher immédiatement le cadre.

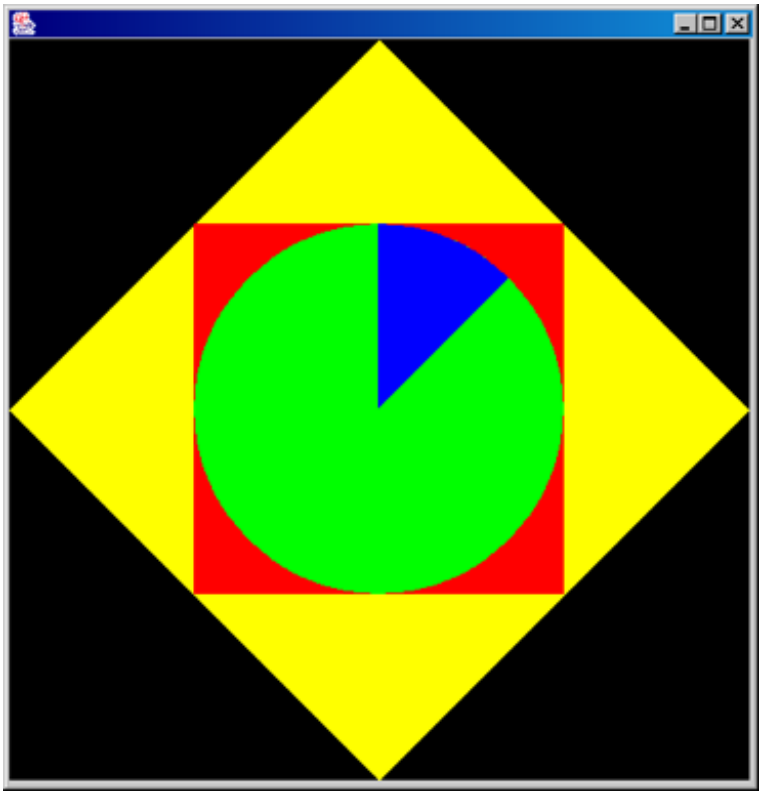
```
import javax.swing.*;
import java.awt.*;
import java.util.*;

public class DessinCadreRouge extends JPanel {
  private int epaisseur;
  public DessinCadreRouge(int e) {
    epaisseur=e;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.red);
    int h=getHeight();
    int w=getWidth();
    for(int i=2;i<2+epaisseur;i++)
      g.drawRect(i,i,w-2*i-1,h-2*i-1);
  }
```

```
  public void setEpaisseur(int e) {
    epaisseur=e;
    repaint();
  }
}
public class AffichageCadreRouge {
  public static void main(String[] args) {
    int e=1;
    Scanner scan;
    scan = new Scanner(System.in);
    JFrame fenetre=new JFrame("Affichage d'un cadre rouge");
    fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    DessinCadreRouge dessin=new DessinCadreRouge(e);
    dessin.setPreferredSize(new Dimension(600,480));
    fenetre.setContentPane(dessin);
    fenetre.pack();
    fenetre.setVisible(true);
    while (true) {
      System.out.print("Epaisseur : ");
      e = scan.nextInt() ;
      if(e>0) {
        dessin.setEpaisseur(e);
      }
    }
  }
}
```



Une boucle demande à l'utilisateur de saisir une nouvelle épaisseur, puis modifie l'objet auquel `dessin` fait référence (un `DessinCadreRouge`). Comme la méthode `setEpaisseur` de sa classe contient un appel à `repaint` (méthode héritée de `JPanel`), chaque modification de l'épaisseur du cadre va se traduire par la modification de l'affichage (l'objet auquel `dessin` fait référence est celui que la `JFrame` utilise pour produire l'affichage) : le cadre aura la nouvelle épaisseur choisie.

Exercice : Créer le dessin ci-dessous dans une fenêtre de dimension 490x490.



4 Gestion des événements

4.1 Classes internes et classes anonymes

Complément préalable : droit d'accès `protected`

Il existe quatre niveaux de droits d'accès aux propriétés d'une classe, parmi lesquels `public` et `private` ont été plus largement vus jusque-là. Dans l'ordre croissant des restrictions, les droits d'accès sont les suivants :

public (catégorie `public`) : accès autorisé à n'importe quelle autre classe.

protected (catégorie `protected`) : accès réservé aux classes dérivées et aux classes du même package (en l'absence de déclaration de package, on assimilera le package d'une classe à toutes les classes du même répertoire).

friendly (par défaut, c.à.d. catégorie non précisée) : accès réservé aux classes du même package (en l'absence de déclaration de package, on assimilera le package d'une classe à toutes les classes du même répertoire).

private (catégorie `private`) : accès interdit à toute autre classe.

La catégorie `protected` permet donc de déclarer par exemple des attributs accessibles depuis les classes dérivées, tout en interdisant l'accès aux autres packages.

Néanmoins, l'utilisation de ce droit d'accès ne doit se faire que s'il est véritablement motivé, et non pas juste pour éviter les éventuels problèmes liés aux attributs `private` hérités.

Définition et utilisation des classes internes

Nous n'avons vu jusqu'à présent que des programmes constitués de classes distinctes déclarant des attributs, des constructeurs et des méthodes.

Il est également possible de déclarer des classes internes ou interfaces internes à l'intérieur d'une classe, dite alors *classe externe*. Il s'agit là de propriétés de la classe externe, au même titre que les attributs ou les méthodes : une classe interne définit en général un ensemble d'objets qui n'ont de sens que dans le contexte défini par la classe externe, donc un ensemble d'objets dont l'existence même est une "propriété" de la classe externe.

Bien qu'il ne soit pas obligatoire de s'en servir, les classes internes apportent un plus pour l'organisation et la programmation des classes d'un programme :

- L'existence de certaines classes n'a de sens que dans le contexte d'une autre classe. Dans ce cas, il peut être intéressant de les déclarer comme classes internes pour montrer aux utilisateurs de ces classes dans quel contexte elles s'utilisent. C'est notamment le cas lorsqu'on crée certaines classes de composants graphiques personnalisés à l'intérieur d'une interface graphique.
- Toutes les classes d'un package donné sont accessibles à toutes les autres classes de ce package (même sans déclarer la classe `public`), ce qui n'a pas toujours l'effet désiré. Une classe interne dont le contrôle d'accès est `private` n'est accessible que par la classe externe dans laquelle elle est déclarée.
- Le nommage des classes est simplifié : certaines classes utilitaires de même genre utilisées dans différents contextes peuvent être déclarées comme classes internes avec le même nom sans interférer entre elles.

- ❑ Comme une classe interne peut être déclarée n'importe où dans une classe, il est permis de la rapprocher de l'endroit où elle est le plus utilisée.
- ❑ Une classe interne est une propriété de classe. Donc elle peut être déclarée `static` ou pas. La possibilité d'utiliser directement tous les attributs d'instance et les méthodes d'instance de la classe externe dans une classe interne non `static` simplifie dans de nombreux cas la programmation, notamment pour la gestion des événements graphiques.

Déclaration

La syntaxe de déclaration d'une classe interne est semblable à celle des classes externes vues jusqu'à présent : on se contente de placer cette déclaration à l'intérieur de la définition d'une classe externe.

```
class NomClasseExterne{

  // declararation d'une classe interne
  categorie class NomClasseInterne {
    // Corps de ClasseInterne :
    // declaration des champs, des methodes, des constructeurs,...
  }

  // declaration d'une classe interne derivant d'une classe parente
  // et implantant une interface
  categorie class NomClasseInterne2 extends NomClasseParente
                                    implements NomInterface {
    // ...
  }

  // declaration d'une interface interne
  categorie interface NomInterfaceInterne {
    // ...
  }

  // Autres declarations
}
```

La catégorie indiquée pour une classe interne suit les règles des autres déclarations de propriétés. Elle est optionnelle et comporte un ou plusieurs mots parmi les suivants :

- — `public`, `private`, `protected` : indiquent le contrôle d'accès de la classe interne.
- — `final` : comme pour une classe externe, une classe interne déclarée `final` ne peut être dérivée.
- — `abstract` : comme pour une classe externe, il est impossible de créer une instance d'une classe interne déclarée `abstract`, et toute classe interne comportant au moins une méthode abstraite doit être déclarée abstraite.
- — `static` : comme pour les autres propriétés `static`, une classe interne `static` ne dépend d'aucune instance de la classe externe dans laquelle elle est définie. Dans ce cas, il est possible de

créer une instance d'une classe interne simplement par `new ClasseExterne.ClasseInterne(...)` par exemple.
En revanche, l'instance d'une classe interne non `static` stocke automatiquement une référence vers l'instance de la classe externe `ClasseExterne` dont elle dépend, ce qui permet **d'utiliser directement toutes les méthodes et les attributs de cette instance**. Cette référence doit être précisée à la création d'un objet de cette classe interne comme par exemple dans l'instruction `objetClasseExterne.new ClasseInterne(...)` pour créer un objet de classe `ClasseInterne` dépendant de l'objet `objetClasseExterne`. Une classe interne non `static` ne peut pas déclarer des attributs ou des méthodes `static`.

La catégorie indiquée pour une interface interne suit les mêmes règles (mais sans les catégories `final` et `abstract`).

Une classe interne peut déclarer elle-même d'autres classes internes.

Pour chacune des classes internes déclarées, un fichier `NomClassExterne$NomClasseInterne.class` est généré à la compilation.

Les classes anonymes

Il est également possible de définir des classes internes localement dans un bloc, à condition qu'elle soient `final` ou `abstract`. Cette utilisation des classes internes est d'un usage très réduit.

Par extension de ces classes internes *locales*, il est possible de déclarer localement des classes internes dont une seule instance est créée (dès la déclaration). L'unicité de cette instance rend non nécessaire le fait de nommer une telle classe interne puisqu'elle ne sera pas utilisée ultérieurement. On parle ainsi de classe *anonyme*.

Déclaration d'une classe anonyme

C'est un ajout à la syntaxe de l'opérateur `new` : après l'instruction `new NomClasse(...)`, il est possible d'ajouter un bloc permettant de modifier localement le comportement de `NomClasse`, en redéfinissant une ou plusieurs méthodes de `NomClasse`. Un objet d'une classe *anonyme* dérivée de `NomClasse` est alors créé, et le sous-typage permet immédiatement de stocker sa référence dans une variable de type `NomClasse`.

Comme toute classe interne, une classe anonyme peut également déclarer un ensemble d'attributs et de méthodes d'instance, mais l'usage principal des classes anonymes concerne la redéfinition locale de méthodes.

```
NomClasse obj = new NomClasse (/* argument1, argument2, ...*/) {
    // redefinition de certaines methodes de NomClasse
    // pour modifier le comportement de NomClasse
    // uniquement en ce qui concerne l'objet cree ici
    //
    // declaration eventuelle de nouveaux attributs et/ou methodes
};
```

Dans la même logique, il est possible de créer une instance d'une classe anonyme implantant une interface `NomInterface`, grâce à l'instruction :

```
NomInterface obj2 = new NomInterface() {
    // Implantation de TOUTES les methodes de NomInterface
};
```

Dans ce cas, le bloc qui suit `new NomInterface()` doit implanter toutes les méthodes de `NomInterface` pour qu'il soit possible de créer une instance d'une telle classe.

Pour chacune des classes anonymes déclarées, un fichier `.class` est généré à la compilation : ce nom est constitué du nom de la classe externe (celle où se trouve la création de l'objet de classe anonyme) suivi du symbole $ et d'un identifiant numérique généré par le compilateur, comme par exemple `NomClasseExterne$1.class`.

Une utilisation abusive des classes anonymes rend un programme moins lisible. Néanmoins, il existe un ensemble de circonstances où il est intéressant de les utiliser : pour créer une instance d'un objet d'une classe dont on veut redéfinir juste une ou deux méthodes, à condition que cette redéfinition n'ait qu'une utilité purement locale. C'est notamment le cas lorsqu'on associe à certains composants graphiques des gestionnaires d'événements graphiques simples.

Conseil : pour éviter toute confusion avec le reste des instructions, utilisez des règles d'écriture et une indentation claires pour l'écriture des classes anonymes.

Tout paramètre ou toute variable locale (de la méthode de la classe `NomClasseExterne` dans laquelle l'objet de classe anonyme est créé) utilisé(e) dans une classe anonyme doit être déclaré(e) `final`. Ceci permet à cette classe anonyme d'utiliser ces variables temporaires sans risque qu'elles soient modifiées ultérieurement.

En revanche, tous les attributs d'instance ou de classe de `NomClasseExterne` existent tant que l'objet est en mémoire et peuvent donc être utilisés dans une classe anonyme.

```
public class NomClasse {
    private int valeur;
    public NomClasse(int v) {
        valeur=v;
    }
    public void methode() {
        System.out.println(valeur);
    }
}
public class NomClasseExterne {
    private int attribut;

    public NomClasseExterne(int a) {
        attribut=a;
    }
    public void action() {
        final String s="bonjour";
        NomClasse nc=new NomClasse(12) {
                public void methode() {
                    System.out.println(s);
                    System.out.println(attribut);
```

```
                    super.methode();
                }
            };
        nc.methode();
    }
    public static void main(String[] args) {
        new NomClasseExterne(15).action();
        // affiche sur 3 lignes : bonjour  15  12
    }
}
```

4.2 Interaction avec l'utilisateur

4.2.1 Généralités

L'utilisation d'un dialogue entre le programme et l'utilisateur via une fenêtre de commande permet, comme dans l'exemple précédent, des modifications dynamiques de l'interface graphique. Néanmoins, ce type d'interaction reste limité par rapport à ce que proposent des programmes évolués. Il est possible de proposer une interactivité plus évoluée, grâce à la notion d'**événement** graphique.

Un événement graphique est la traduction informatique d'une action de l'utilisateur d'un programme muni d'une interface graphique. Les actions possibles sont de nature très diverse : un mouvement de la souris, la pression d'un des boutons de la souris, la pression d'une touche du clavier, la fermeture de la fenêtre du programme, ... etc.

La notion d'événement recouvre d'autres aspects que les interfaces graphiques. L'étude qui va en être faite se limitera néanmoins aux événements graphiques.

4.2.2 Principes

De manière générale, en Java, un événement est représenté par un objet d'une classe dérivée de `java.util.EventObject`. Il est lié, d'une part, à un "objet source" (composant graphique dans cette section) et, d'autre part, à un "objet auditeur".

- L'objet source de l'événement est défini comme étant l'objet qui génère l'événement, ou encore comme l'objet sur lequel un événement se produit. Par exemple, dans le cas d'un clic sur un bouton `JButton`, un événement de classe `ActionEvent` est généré, et sa source est l'objet bouton.
- Un objet auditeur (ou encore dit écouteur) de l'événement est défini comme étant un objet chargé de traiter cet événement. La classe à laquelle un objet auditeur appartient implante des méthodes dont le paramètre est l'objet événement à traiter. Par exemple, les objets auditeurs des événements de classe `ActionEvent` doivent implanter l'interface `ActionListener`, c'est à dire redéfinir la méthode `void actionPerformed(ActionEvent e)`.

Pour qu'un événement soit traité, **il faut relier** son objet source à un (ou plusieurs) objet auditeur. Il est possible de relier un objet source d'un événement à plusieurs auditeurs de cet événement. De plus un même objet peut être la source de différents types d'événements, et donc relié, pour chacun d'eux, à plusieurs auditeurs.

Exemple : pour qu'un clic sur un bouton `JButton` soit traité, il faut que ce bouton exécute la méthode `void addActionListener(ActionListener a)`. Dans ce cas, à chaque clic sur ce bouton,

l'événement de classe `ActionEvent` généré sera transmis en paramètre à un appel **automatique** de la méthode `actionPerformed` de l'objet auditeur relié au bouton.

On a donc entre les classes impliquées dans la gestion de ces événements les relations représentées par la figure 10.

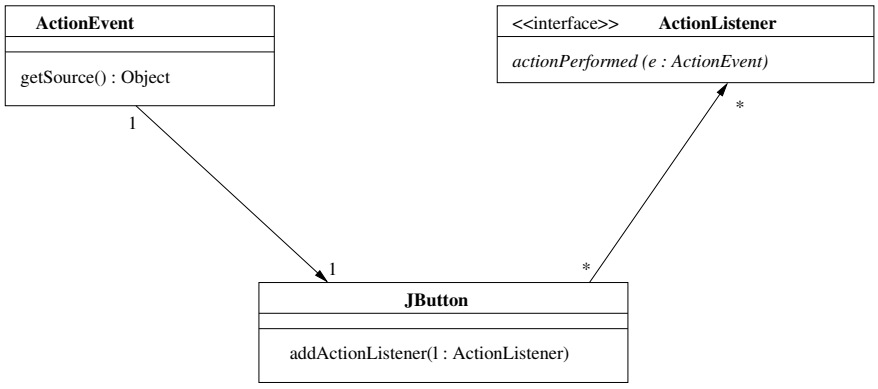


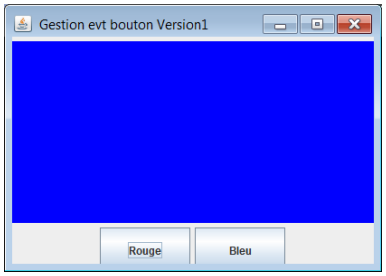
FIGURE 10 – Relations entre événements, composants, et auditeurs

Pour résumer, différentes étapes doivent être prises en compte pour assurer la gestion d'événements :

1. Identifier clairement les objets source et auditeur.
2. Redéfinir les méthodes automatiquement appelées des objets auditeurs de manière à personnaliser le traitement des événements.
3. Relier les objets auditeurs aux objets sources des événements qu'ils doivent traiter.

La seconde étape exige d'adopter une nouvelle démarche de programmation, parfois appelée "programmation événementielle" : il ne s'agit plus de décrire un algorithme qui résoud entièrement une tâche donnée, mais de donner un sens aux enchaînements d'actions de l'utilisateur, par le traitement **séparé** de chacune de ces actions. Une des difficultés de la programmation événementielle est l'imprévisibilité du comportement de l'utilisateur : il peut enchaîner ses différentes actions d'une manière "illogique" par rapport à la résolution de la tâche telle qu'elle a été imaginée par le programmeur. Celui-ci doit donc faire en sorte que le traitement de tout événement ait un sens à tout instant de l'évolution du programme.

Un exemple avec 3 versions : Dans l'interface ci-dessous, si l'utilisateur appuie sur le bouton avec la chaine de caractères "Rouge", le fond de l'interface devient rouge et s'il appuie sur le bouton "Bleu", le fond devient bleu.



Le programme principal est le suivant :

```
public class FenetreEvtJButton {
 public static void main(String[] args) {
  JFrame fenetre = new JFrame ("␣Gestion␣evt␣bouton");
  fenetre.setDefaultCloseOperation ( JFrame.EXIT_ON_CLOSE );
  EvtJButton pan =new EvtJButton();
  pan.setPreferredSize (new Dimension (400 ,250));
  fenetre.setContentPane ( pan );
  fenetre.pack ();
  fenetre.setVisible (true);
 }
}
```

Et la classe EvtJButton utilisée peut être définie de plusieurs manières, trois sont présentées ci-après.
Version 1 : Utilisation d'une classe interne non anonyme d'auditeurs

```
public class EvtJButton extends JPanel {
 private JButton br, bb;
 private JPanel dess ;

 public EvtJButton(){
  dess= new JPanel();
  dess.setPreferredSize(new Dimension(400,200));
  dess.setBackground(Color.blue);
  EcouteurBouton ecouteur = new EcouteurBouton();
  br=new JButton("Rouge");
  //Mise en relation entre  br et un auditeur, instance de EcouteurBouton
  br.addActionListener(ecouteur);
  br.setPreferredSize(new Dimension(100,50));
  bb= new JButton("Bleu");
  //Mise en relation entre bb et un auditeur, instance de EcouteurBouton
  bb.addActionListener(ecouteur);
  bb.setPreferredSize(new Dimension(100,50));
  //Ajout des composants dans le JPanel
  this.add(dess);
  this.add(br);
  this.add(bb);
  }
 //Classe interne auditeur
 private class EcouteurBouton implements ActionListener{
  //Methode declenchee apres appui sur les boutons br et bb
  public void actionPerformed(ActionEvent e){
   Object source = e.getSource(); // Recuperation de la source de l'evt
   if(source.equals(br))  dess.setBackground(Color.red);
   if(source.equals(bb)) dess.setBackground(Color.blue);
```

```
  }
 }
}
```

Version 2 : Utilisation d'auditeurs anonymes

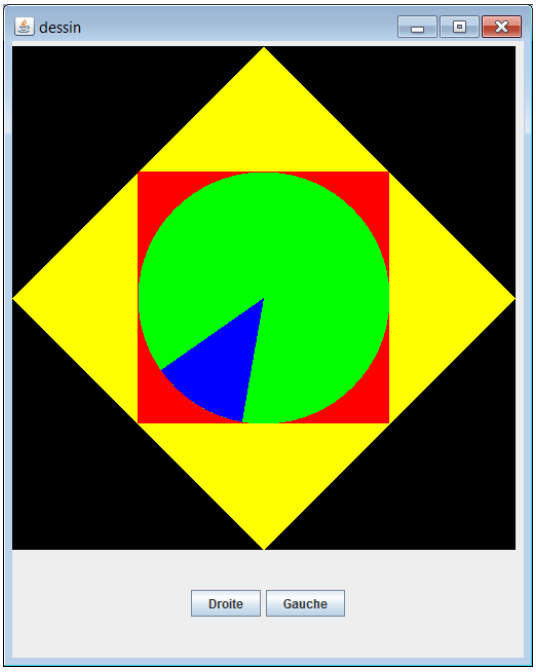
```
public class EvtJButton extends JPanel{
 private JButton br,bb;
 private JPanel dess ;

 public EvtJButton (){
  dess= new JPanel();
  dess.setPreferredSize(new Dimension(400,200));
  dess.setBackground(Color.blue);
  br=new JButton("Rouge");
  //Creation d'un auditeur anonyme sur br
  br.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
     dess.setBackground(Color.red);
    }
  } );
  br.setPreferredSize(new Dimension(100,50));
  bb= new JButton("Bleu");
  //Creation d'un auditeur anonyme sur bb
  bb.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e){
    dess.setBackground(Color.blue);
   }
  } );
  bb.setPreferredSize(new Dimension(100,50));
  //Ajout des composants dans le JPanel
  this.add(dess);
  this.add(br);
  this.add(bb);
 }
}
```

Version 3 : Utilisation d'objets sources et auditeurs à la fois

```
public class EvtJButton extends JPanel  implements ActionListener{
 private JButton br,bb;
 private JPanel dess ;

 public EvtJButton(){
  dess= new JPanel();
  dess.setPreferredSize(new Dimension(400,200));
```

```
  dess.setBackground(Color.blue);
  br=new JButton("Rouge");
  //Le JPanel est auditeur des evts sur le bouton br
  br.addActionListener(this);
  br.setPreferredSize(new Dimension(100,50));
  bb= new JButton("Bleu");
  //Le JPanel est auditeur des evts sur le bouton bb
  bb.addActionListener(this);
  bb.setPreferredSize(new Dimension(100,50));
  //Ajout des composants dans le JPanel
  this.add(dess);
  this.add(br);
  this.add(bb);
  }

 public void actionPerformed(ActionEvent e){
  Object source = e.getSource(); // Recuperation de la source de l'evt
  if(source.equals(br))  dess.setBackground(Color.red);
  if(source.equals(bb)) dess.setBackground(Color.blue);
 }
}
```

Exercice : Ajouter 2 boutons à l'interface réalisée dans la section précédente, l'appui sur un de ces boutons permet de déplacer l'arc de cercle bleu de dix degrés vers la droite ou vers la gauche.



4.2.3 Exemple : gestion du clavier

Il est possible de gérer relativement simplement le clavier de l'ordinateur, afin qu'une simple pression sur une touche provoque une modification de l'affichage proposé. Dans l'exemple de 3.2.3, on souhaite faire en sorte qu'une pression sur la touche `+` provoque immédiatement une augmentation de l'épaisseur du cadre, alors qu'une pression sur la touche `-` provoquera au contraire une diminution de l'épaisseur. Le programme ne prévoit pas à l'avance l'enchaînement des pressions sur les touches `+` et `-`.

Ce programme va utiliser des relations entre les classes impliquées dans la gestion des événements clavier.

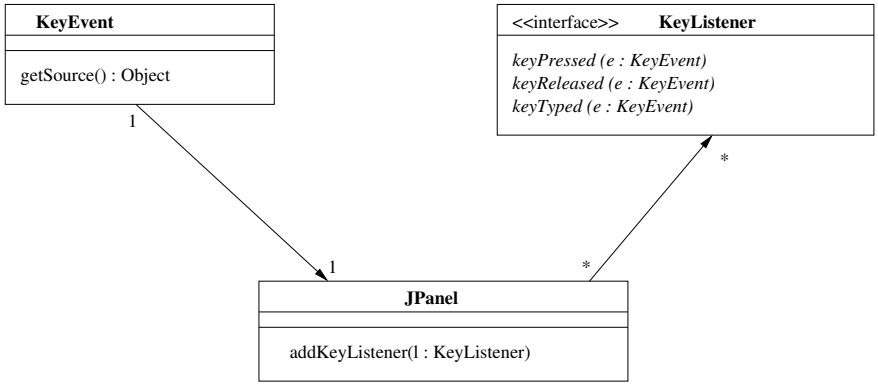


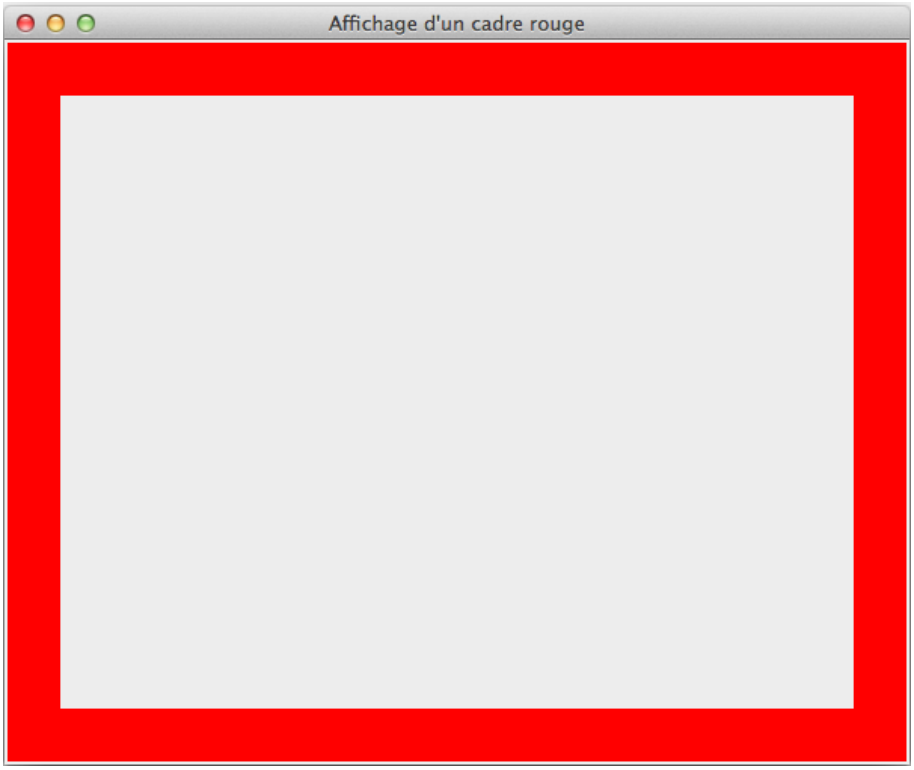
FIGURE 11 – Relations entre événements, composants, et auditeurs : cas du clavier

— Les événement générés par des frappes sur le clavier lors du déroulement d'un programme avec interface graphique sont de classe `KeyEvent`.
— La source de ces événements sera ici la zone de dessin `JPanel`. Tout auditeur de classe `KeyListener` relié à cette source exécutera automatiquement différentes méthodes :
  — `void keyPressed(KeyEvent e)` : dès qu'un événement est généré par la pression sur une touche quelconque du clavier (N.B. : suivant la configuration de votre environnement, tant que cette touche reste enfoncée, de nouveaux événements sont générés).
  — `void keyReleased(KeyEvent e)` : dès qu'un événement est généré par le relachement d'une touche quelconque du clavier.
  — `void keyTyped(KeyEvent e)` : dès qu'un événement est généré par la pression puis le relachement d'une touche du clavier qui correspond à un caractère (pas les touches spéciales). Il est possible que la pression d'une combinaison de touche ne provoque qu'un seul appel à cette méthode : par exemple majuscule suivi de `a` provoque un seul appel à `keyTyped`, avec comme paramètre un `KeyEvent` correspond au caractère `'A'`.
— Un seul auditeur d'événements `KeyEvent` va être relié à cette source : il devra traiter les événements grâce à des appels aux différentes méthodes d'instance de `DessinCadreRouge`. Pour cela, il est plus aisé de faire en sorte que cet auditeur ait un accès direct aux propriétés d'instance de l'objet de classe `DessinCadreRouge`, ce que l'utilisation de classes internes (et plus précisément ici d'une classe anonyme) rend possible.

```
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
public class DessinCadreRouge extends JPanel {
  private int epaisseur;
  public DessinCadreRouge(int ep) {
    epaisseur=ep;
    KeyListener kl=new KeyListener() {
            public void keyTyped(KeyEvent e) {
            }
            public void keyReleased(KeyEvent e) {
            }
            public void keyPressed(KeyEvent e) {
                char c=e.getKeyChar();
                if(c=='+') {
                    setEpaisseur(epaisseur+1);
                } else if(c=='-') {
                    if(epaisseur>1) {
                        setEpaisseur(epaisseur-1);
                    }
                }
            }
        };
    this.addKeyListener(kl);
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.red);
    int h=getHeight();
    int w=getWidth();
    for(int i=2;i<2+epaisseur;i++)
      g.drawRect(i,i,w-2*i-1,h-2*i-1);
  }
  public void setEpaisseur(int e) {
    epaisseur=e;
    repaint();
  }
}
public class ControleCadreRouge {
  public static void main(String[] args) {
    JFrame fenetre=new JFrame("Affichage␣d'un␣cadre␣rouge");
    fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    DessinCadreRouge dessin=new DessinCadreRouge(1);
    dessin.setPreferredSize(new Dimension(600,480));
    fenetre.setContentPane(dessin);
    fenetre.pack();
```

```
    fenetre.setVisible(true);
    dessin.requestFocusInWindow();
  }
}
```



Explications :

— La plupart des classes d'événements graphiques et de leurs auditeurs sont définies dans le package `java.awt.event`.

— L'implantation de l'interface `KeyListener` exige la définition de 3 méthodes, même si on se contente d'effectuer un véritable traitement d'événement dans la méthode `keyPressed`.

— La dernière instruction du constructeur de `DessinCadreRouge` associe l'objet source (`this`, qui est un `JPanel`) à l'auditeur de classe anonyme (ou "auditeur anonyme") créé à l'instruction précédente.

— La méthode `keyPressed` de cet auditeur est donc automatiquement appelée à chaque enfoncement de touche du clavier, et elle reçoit alors en paramètre un objet de type `KeyEvent`, qui décrit ici la touche qui a été enfoncée. On peut obtenir le caractère inscrit sur la touche appuyée en appelant la méthode d'instance `getKeyChar` de la classe `KeyEvent`.

— Dans une programmation non événementielle, on compterait le nombre de fois où la touche est enfoncée, puis on en déduirait la nouvelle épaisseur du cadre. Ici, chaque événement est traité séparément des autres : on se contente d'incrémenter ou de décrémenter de 1 pixel l'épaisseur du cadre à chaque nouvel événement.

— L'instruction `dessin.requestFocusInWindow()` indique que l'objet graphique (de type `DessinCadreRouge`) demande que les pressions des touches lui soient transmises. Si on oublie cette ligne, les

pressions des touches sont indiquées à d'autres composants graphiques. Notons que seule l'interaction basée sur les événements clavier demande l'utilisation de cette méthode.

Quand on lance l'application ainsi construite, et après avoir placé le pointeur de la souris dans la fenêtre (et éventuellement cliqué dans cette fenêtre pour l'activer, suivant la configuration de votre environnement), dès qu'on appuie sur la touche `+`, l'épaisseur du cadre augmente. Si on appuie sur la touche `-`, l'épaisseur diminue immédiatement.

Remarque sur la classe `KeyEvent`

Cette classe définit diverses méthodes, parmi lesquelles :

— `char getKeyChar()` : cette méthode, utilisée dans l'exemple précédent, renvoie sous forme d'un `char` la touche qui a été enfoncée.

— `int getKeyCode()` : cette méthode renvoie sous forme d'un code spécifique à `Java` la touche qui a été enfoncée.

La différence principale entre les deux méthodes est que la deuxième permet d'étudier les touches spéciales (comme par exemple les touches de déplacement du curseur) qui ne correspondent pas à un caractère. Les codes utilisés par `Java` pour représenter les touches spéciales (par exemple 16 pour la touche *majuscule*) ne doivent pas être devinés ou utilisés comme des entiers. La bonne façon d'utiliser `getKeyCode` passe par l'emploi de constantes de la classe `KeyEvent`. Par exemple la constante `KeyEvent.VK_SHIFT` désigne la touche *shift* (c'est-à-dire majuscule), `KeyEvent.VK_UP` la touche "flèche vers le haut", ... etc.

Exercice : Ajouter, à l'interface réalisée dans la section précédente, la gestion de l'appui sur les touches flèches "droite" et "gauche" du clavier, l'appui sur ces touches permet de déplacer l'arc de cercle bleu de dix degrés vers la droite ou vers la gauche.

4.2.4 Gestion de la souris

Évènement souris

La notion d'évènement graphique recouvre également les actions effectuées à la souris par l'utilisateur. Pour représenter ces actions, l'ordinateur utilise la classe `MouseEvent` qui comporte notamment les méthodes suivantes :

— `int getX()` : renvoie la coordonnée horizontale de la souris au moment de l'évènement.

— `int getY()` : renvoie la coordonnée verticale de la souris au moment de l'évènement.

Les événements souris sont générés par diverses actions de l'utilisateur : clic sur un bouton de la souris, déplacement simple de la souris, déplacement de la souris avec un bouton enfoncé, pointeur de la souris sortant de la fenêtre, ... etc.

N.B. : Pour déterminer le bouton concerné par la pression (bouton de gauche, du centre ou de droite), on peut utiliser la classe `SwingUtilities` qui définit notamment les méthodes **de classe** suivantes :

— `static boolean isLeftMouseButton(MouseEvent e)` : renvoie `true` si et seulement si `e` correspond à une pression du bouton gauche de la souris.

— `static boolean isMiddleMouseButton(MouseEvent e)` : renvoie `true` si et seulement si `e` correspond à une pression du bouton du milieu de la souris.

— `static boolean isRightMouseButton(MouseEvent e)` : renvoie `true` si et seulement si `e` correspond à une pression du bouton droit de la souris.

Auditeur d'événement souris

Comme pour le clavier, les composants graphiques ne réagissent pas naturellement à la souris. Pour que les événements soient traités, il faut utiliser des relations similaires à celles des événemens clavier, c'est à dire relier les objets sources d'événements souris à des auditeurs de ces événements, selon l'organisation indiquée par la figure 12. On remarque que deux catégories d'auditeurs sont prévues pour la souris : `MouseListener` pour les évènements ponctuels, c'est-à-dire la réaction à la pression sur un bouton ou à l'entrée du pointeur de la souris dans un composant, `MouseMotionListener` pour les évènements correspondant aux mouvements du pointeur de la souris dans un composant donné. La classe de l'objet source est ici `JComponent`, ce qui recouvre par dérivation tous les composants `swing`, y compris `JPanel`.

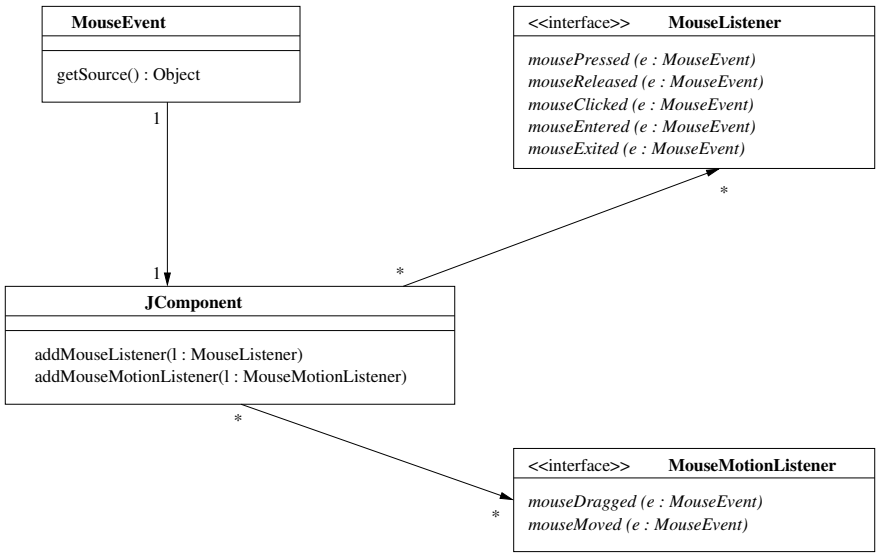


FIGURE 12 – Relations entre événements, composants, et auditeurs : cas de la souris

— `void mouseClicked(MouseEvent e)` : méthode appelée en cas de clic, c'est-à-dire de bouton enfoncé puis relâché (sans **aucun** mouvement de la souris entre les deux actions).
— `void mouseEntered(MouseEvent e)` : méthode appelée quand le pointeur de la souris entre dans le composant.
— `void mouseExited(MouseEvent e)` : méthode appelée quand le pointeur de la souris sort du composant.
— `void mousePressed(MouseEvent e)` : méthode appelée quand un bouton de la souris est enfoncé.
— `void mouseReleased(MouseEvent e)` : méthode appelée quand un bouton de la souris préalablement enfoncé est relâché.
— `void mouseDragged(MouseEvent e)` : méthode appelée quand la souris est déplacée avec un bouton maintenu enfoncé.
— `void mouseMoved(MouseEvent e)` : méthode appelée quand la souris est déplacée sans bouton enfoncé.

A nouveau, ces différentes méthodes ne sont appelées (automatiquement) que si les auditeurs ont été reliés aux composants sources des événements :

— `void addMouseListener(MouseListener l)` : ajoute au composant appelant l'objet `l` pour la gestion des évènements ponctuels de souris.

— `void addMouseMotionListener(MouseMotionListener l)` : ajoute au composant appelant l'objet `l` pour la gestion des mouvements de souris.

Exemple simple

On se contente ici d'afficher un message pour chaque évènement, indiquant la position et le type d'événement souris :

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SuiviSouris {
  public static void afficheEvenement(String pos,MouseEvent e) {
    System.out.print(pos+" : ("+e.getX()+","+e.getY()+")");
    if(SwingUtilities.isLeftMouseButton(e)) {
      System.out.println(" bouton de gauche");
    } else if(SwingUtilities.isMiddleMouseButton(e)) {
      System.out.println(" bouton du milieu");
    } else if(SwingUtilities.isRightMouseButton(e)) {
      System.out.println(" bouton de droite");
    } else System.out.println();
  }
  public static void main(String[] args) {
    JFrame fen=new JFrame("Exemple simple de traitement des evenements souris")
    fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JPanel contenant=(JPanel)(fen.getContentPane());
    contenant.setPreferredSize(new Dimension(200,250));
    fen.pack();
    fen.setVisible(true);
    // creation de l'auditeur anonyme
    MouseListener ml=new MouseListener() {
            public void mouseClicked(MouseEvent e) {
                afficheEvenement("clicked",e);
            }
            public void mouseEntered(MouseEvent e) {
                afficheEvenement("entered",e);
            }
            public void mouseExited(MouseEvent e) {
                afficheEvenement("exited",e);
            }
            public void mousePressed(MouseEvent e) {
                afficheEvenement("pressed",e);
```

```
            }
            public void mouseReleased(MouseEvent e) {
                afficheEvenement("released",e);
            }
        };
    MouseMotionListener mml=new MouseMotionListener() {
            public void mouseMoved(MouseEvent e) {
                afficheEvenement("moved",e);
            }
            public void mouseDragged(MouseEvent e) {
                afficheEvenement("dragged",e);
            }
        };
    // association source-auditeurs
    contenant.addMouseListener(ml);
    contenant.addMouseMotionListener(mml);
  }
}
```

Quand on lance ce programme, une fenêtre vide s'affiche. Quand on place le pointeur de la souris dans la fenêtre, un événement souris est généré et un appel à `mouseEntered` se produit. Cela se traduit par un affichage et on obtient par exemple :

```
entered : (147,242)
```

En déplaçant la souris, des appels successifs à `mouseMoved` se produisent. Quand le déplacement de la souris est rapide, on obtient des évènements correspondant à des points d'origine espacés :

```
moved : (60,272)
moved : (62,256)
moved : (70,242)
moved : (86,226)
```

Si par contre le mouvement est lent, on obtient un évènement pour chaque déplacement d'un pixel :

```
moved : (160,269)
moved : (161,269)
moved : (162,269)
moved : (162,268)
```

En cliquant sur le bouton de gauche, on obtient par exemple l'affichage suivant :

```
pressed : (137,107) bouton de gauche
released : (137,107) bouton de gauche
clicked : (137,107) bouton de gauche
```

Notons qu'un appel à `mouseClicked` ne se produit que si on appuie et relâche le bouton de la souris *sans bouger celle-ci*. Si on bouge la souris tout en maintenant le bouton enfoncé, on obtient des appels à `mouseDragged`, de façon assez rapide, comme pour les `mouseMoved`. Par exemple, avec le bouton droit :

```
pressed : (114,252) bouton de droite
dragged : (116,256) bouton de droite
dragged : (118,255) bouton de droite
dragged : (130,251) bouton de droite
dragged : (148,245) bouton de droite
dragged : (170,239) bouton de droite
released : (174,242) bouton de droite
```

Si on place ensuite le pointeur de la souris en dehors de la fenêtre, on obtient un appel à (`mouseExited`), ce qui provoque par exemple l'affichage suivant :

```
exited : (128,248)
```

4.2.5 Programmation événementielle

Dans les exemples précédents, on a essentiellement cherché à réagir de manière instantanée à des actions de l'utilisateur. A chaque action, `java` génère un objet événement qui est transmis à un auditeur (qu'on a préalablement relié à l'objet source des événements). L'auditeur traite de manière isolée et instantanée cet événement. Il s'agit déjà de programmation événementielle, mais dans un cas simple : il n'est pas question encore de fabriquer au fur et à mesure des actions de l'utilisateur un résultat qui dépend justement de plusieurs actions.

La programmation événementielle demande davantage d'attention lorsqu'elle sert à résoudre une tâche dépendant de plusieurs actions, alors que chaque action génère un événement qui est nécessairement traité seul.

Pour illustrer cette difficulté, on va ici faire un parallèle entre deux tâches similaires, la première étant traitée par un algorithme classique alors que la seconde est programmée de manière événementielle.

Exemple "séquentiel" : compter les caractères tappés par un utilisateur

Le programme suivant sert à compter le nombre de caractères tappés au clavier par un utilisateur (en plusieurs lignes), jusqu'à ce que cet utilisateur indique la fin de la tâche en tappant le mot "fin". L'algorithme utilisé passe par une boucle.

```
public class CompteCaracteres {
    public static void main(String[] args) {
        Scanner scan;
        scan = new Scanner(System.in);
        System.out.println("Tappez␣des␣caracteres␣...");
        String s = scan.next() ;
        int n=0;
        while (s.compareTo("fin")!=0) {
            n+=s.length();
            s = scan.next() ;
        }
        System.out.println("Vous␣avez␣tappe␣"+n+"␣caracteres");
    }
}
```

Exemple "événementiel" : compter les pressions sur le bouton gauche de la souris

Le programme suivant sert à compter le nombre de fois où un utilisateur clique sur le bouton gauche de la souris, jusqu'à ce que cet utilisateur indique la fin de la tâche en cliquant sur le bouton droit. Il n'y a plus de boucle, chaque pression permet d'incrémenter un compteur, sauf les pressions sur le bouton droit.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
// utilisation d'une classe externe d'auditeurs
public class CompteurClics implements MouseListener {
    private int nb;
    public CompteurClics() {
        nb=0;
    }
    public void mouseClicked(MouseEvent e) {
    }
    public void mouseEntered(MouseEvent e) {
    }
    public void mouseExited(MouseEvent e) {
    }
    public void mouseReleased(MouseEvent e) {
    }
    public void mousePressed(MouseEvent e) {
        if (SwingUtilities.isLeftMouseButton(e)) nb++;
        else if (SwingUtilities.isRightMouseButton(e)) {
            System.out.println(nb+" clics");
            nb=0;
        }
    }
}
public class CompteClics {
    public static void main(String[] args) {
        JFrame fen=new JFrame("Exemple simple de traitement des evenements souris")
        fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel contenant=(JPanel)(fen.getContentPane());
        contenant.setPreferredSize(new Dimension(200,250));
        fen.pack();
        fen.setVisible(true);
        // auditeur
        contenant.addMouseListener(new CompteurClics());
    }
}
```

4.2.6 Classes d'auditeurs, auditeurs anonymes, ...

Il existe différentes manières de créer des auditeurs, à choisir selon les cas de façon à simplifier la programmation, tout en maintenant un code aussi bien structuré que possible. Pour illustrer différentes méthodes de programmation de la gestion des événements, on choisit ici une application très simple : afficher le mot "clic" à chaque clic de souris.

Le programme principal est le suivant :

```
public class Clic {
    public static void main(String[] args) {
        JFrame fen=new JFrame("Clic");
        fen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fen.setContentPane(new PanneauClics());
        fen.pack();
        fen.setVisible(true);
    }
}
```

Reste à définir la classe `PanneauClics`.

Utilisation d'un auditeur anonyme

Cette solution a pour avantage d'autoriser un accès direct aux attributs de l'objet source des événements dans le traitement de ces événements, puisque ce traitement est réalisé par un objet d'une classe anonyme créée par l'objet source lui-même.

Bien qu'une telle approche utilise un objet source différent de l'objet auditeur, elle peut manquer de clarté, puisque le code source de l'auditeur se trouve dans celui de l'objet source. A éviter donc si des traitements complexes sont requis sans nécessiter d'accès aux attributs de l'objet source.

```
public class PanneauClics extends JPanel {
    public PanneauClics() {
        setPreferredSize(new Dimension(300,250));
        addMouseListener(new MouseListener() {
                public void mouseClicked(MouseEvent e) {}
                public void mouseEntered(MouseEvent e) {}
                public void mouseExited(MouseEvent e) {}
                public void mouseReleased(MouseEvent e) {}
                public void mousePressed(MouseEvent e) {
                    System.out.println("clic");
                }
            });
    }
}
```

Utilisation d'une classe d'auditeurs

Cette solution, très lisible, est parfois peu pratique lorsque l'auditeur cherche à manipuler les attributs de l'objet source : il faut alors stocker une référence à l'objet source dans l'objet auditeur,

et effectuer des appels de méthodes sur cet attribut.

```
public class PanneauClics extends JPanel {
    public PanneauClics() {
        setPreferredSize(new Dimension(300,250));
        addMouseListener(new Cliqueur());
    }
}
public class Cliqueur implements MouseListener {
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {
        System.out.println("clic");
    }
}
```

Utilisation d'objets source et auditeur à la fois

Cette solution permet de minimiser le nombre de classes, et d'avoir un accès évident aux attributs de l'objet source des événements dans le traitement de ces événements, puisque ce traitement est réalisé par l'objet source lui-même (il est en même temps auditeur).

Néanmoins, pour des traitements complexes, une telle solution manque de lisibilité puisqu'elle mélange les fonctionnalités de composant graphique et celles d'auditeur d'événements.

```
public class PanneauClics extends JPanel implements MouseListener {
    public PanneauClics() {
        setPreferredSize(new Dimension(300,250));
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {
        System.out.println("clic");
    }
}
```

Utilisation d'une classe interne non anonyme d'auditeurs

Cette solution, bien qu'un peu plus lourde que les auditeurs anonymes, offre les mêmes avantages et est davantage compatible avec des traitements complexes.

```
public class PanneauClics extends JPanel {
    private class Cliqueur implements MouseListener {
```

```
        public void mouseClicked(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
        public void mouseReleased(MouseEvent e) {}
        public void mousePressed(MouseEvent e) {
            System.out.println("clic");
        }
    }
    public PanneauClics() {
        setPreferredSize(new Dimension(300,250));
        addMouseListener(new Cliqueur());
    }
}
```

Utilisation d'adaptateur

Certaines interfaces d'auditeurs définissent de nombreuses méthodes, alors que seulement une ou deux d'entre elles vont contenir un véritable traitement. Plutôt que de redéfinir toutes les méthodes dont la plupart avec un corps vide, `java` propose de dériver de classes d'adaptateurs qui elles-mêmes ont déjà redéfini toutes les méthodes des interfaces (sans effectuer aucun véritable traitement). Ainsi on peut par exemple réécrire la version précédente de manière un peu plus compacte.

```
public class PanneauClics extends JPanel {
    private class Cliqueur extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            System.out.println("clic");
        }
    }
    public PanneauClics() {
        setPreferredSize(new Dimension(300,250));
        addMouseListener(new Cliqueur());
    }
}
```

5 Utilisation de composants graphiques

La classe `JPanel` a pour l'instant été utilisée afin de réaliser divers dessins ou tracés. Une autre propriété essentielle de cette classe est de pouvoir y insérer différents composants graphiques prédéfinis, tels que des boutons, des boîtes à cocher, des boutons radio, des menus déroulants, des barres de défilement, ... etc.

En effet, la classe `JPanel` dérive de `Container`. Cette classe définit les composants graphiques pouvant contenir d'autres composants, de n'importe quel type. N.B. : la classe `JFrame` dérive également indirectement de `Container` (elle contient notamment son panneau principal), mais la classe `JPanel` ne désigne qu'une zone rectangulaire dans une fenêtre, tandis que la classe `JFrame` désigne la fenêtre elle-même.

5.1 Les composants graphiques

S'il est possible de définir ses propres composants graphiques ou de personnaliser ceux déjà proposés par le package `swing`, dans la plupart des cas, l'utilisation directe des composants contenus dans ce package est largement suffisante. Ces composants ont des comportements prédéfinis en réaction aux actions que l'utilisateur leur applique à la souris, et différentes sortes d'événements sont alors générés et éventuellement traités par les classes correspondantes d'auditeurs : le cas d'un bouton enfoncé (événement `ActionEvent` traités par un auditeur `ActionListener` via l'appel automatique à `actionPerformed(ActionEvent)`) a déjà été cité ; dans le cas d'un menu déroulant dans lequel l'utilisateur effectue un choix à la souris, ce sont les classes `ItemEvent` et `ItemListener` qui sont impliquées, ... etc.

Tous les composants `swing` dérivent directement ou indirectement de la classe `JComponent`.

Exemple

L'exemple suivant reprend le programme `DessinDiagonale`, en y ajoutant un bouton permettant d'effacer la diagonale, ainsi qu'un menu déroulant permettant de modifier la couleur de la diagonale. Pour chacun de ces composants, trois étapes sont nécessaires :

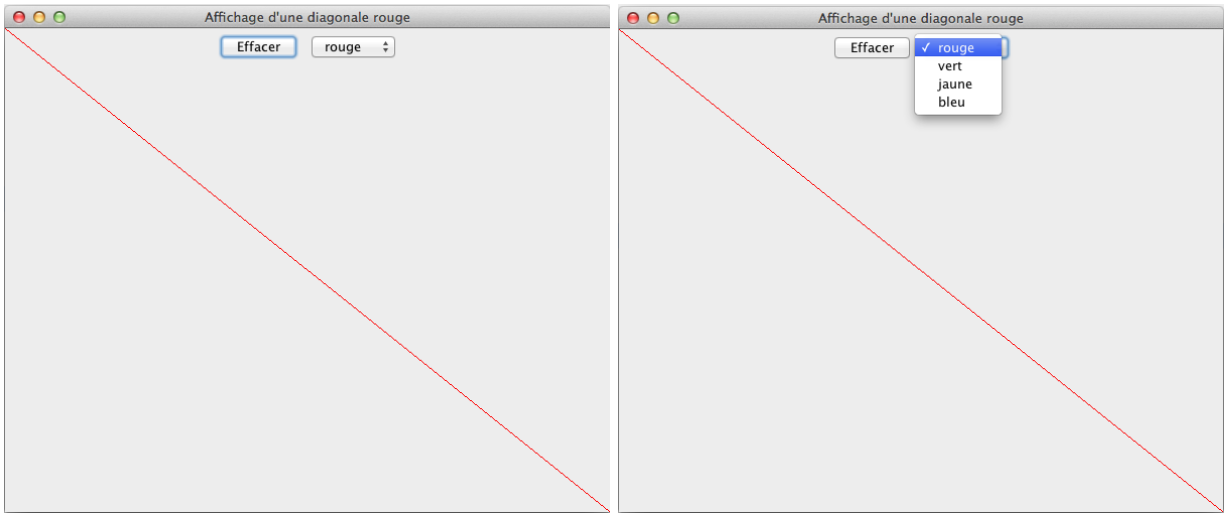
1. création du composant
2. ajout au composant d'un auditeur adapté
3. ajout du composant au `JPanel`

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class DessinDiagonaleComposants extends JPanel {
    private Color couleur;
    public DessinDiagonaleComposants() {
        couleur=Color.red;
        // bouton "Effacer"
        JButton b=new JButton("Effacer");
        b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    getGraphics().clearRect(0,0,getWidth(),getHeight());
```

```
                }
            });
        this.add(b);
        // choix de la couleur de la diagonale
        final JComboBox c=new JComboBox(new String[] {"rouge","vert","jaune","b
        c.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    switch (c.getSelectedIndex()) {
                    case 0 :
                        couleur=Color.red;
                        break;
                    case 1 :
                        couleur=Color.green;
                        break;
                    case 2 :
                        couleur=Color.yellow;
                        break;
                    case 3 :
                        couleur=Color.blue;
                        break;
                    default :
                        couleur=Color.black;
                    }
                    repaint();
                }
            });
        this.add(c);
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int h=getHeight();
        int w=getWidth();
        g.setColor(couleur);
        g.drawLine(0,0,w-1,h-1);
    }
}
public class AffichageDiagonaleComposants {
    public static void main(String[] args) {
        JFrame fenetre=new JFrame("Affichage␣d'une␣diagonale␣rouge");
        fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        DessinDiagonaleComposants dessin=new DessinDiagonaleComposants();
        dessin.setPreferredSize(new Dimension(600,480));
        fenetre.setContentPane(dessin);
        fenetre.pack();
        fenetre.setVisible(true);
```

```
    }
}
```

Ce programme produit l'interface suivante (à droite, l'utilisateur est en cours de sélection d'un champ du menu déroulant) :



Quelques explications :

— A la création du `JPanel` de classe `DessinDiagonaleComposants`, un bouton d'étiquette `''Effacer''` est créé. Toute pression effectuée à la souris sur ce bouton génère un événement de classe `ActionEvent`. Ces événements vont alors être traités par un auditeur anonyme qui implante l'interface `ActionListener` (une seule méthode : `actionPerformed(ActionEvent)`). L'instruction

```
b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            getGraphics().clearRect(0,0,getWidth(),getHeight());
        }
    });
```

a donc pour conséquence d'associer au bouton un auditeur qui traitera chaque clic sur ce bouton par un effacement de la zone de dessin.
N.B. : la source de l'événement est le bouton, c'est donc lui qui doit exécuter la méthode `addActionListener`.

— La méthode `getGraphics()` de la classe `JPanel` renvoie l'environnement graphique (objet de classe `Graphics` associé au `JPanel`).

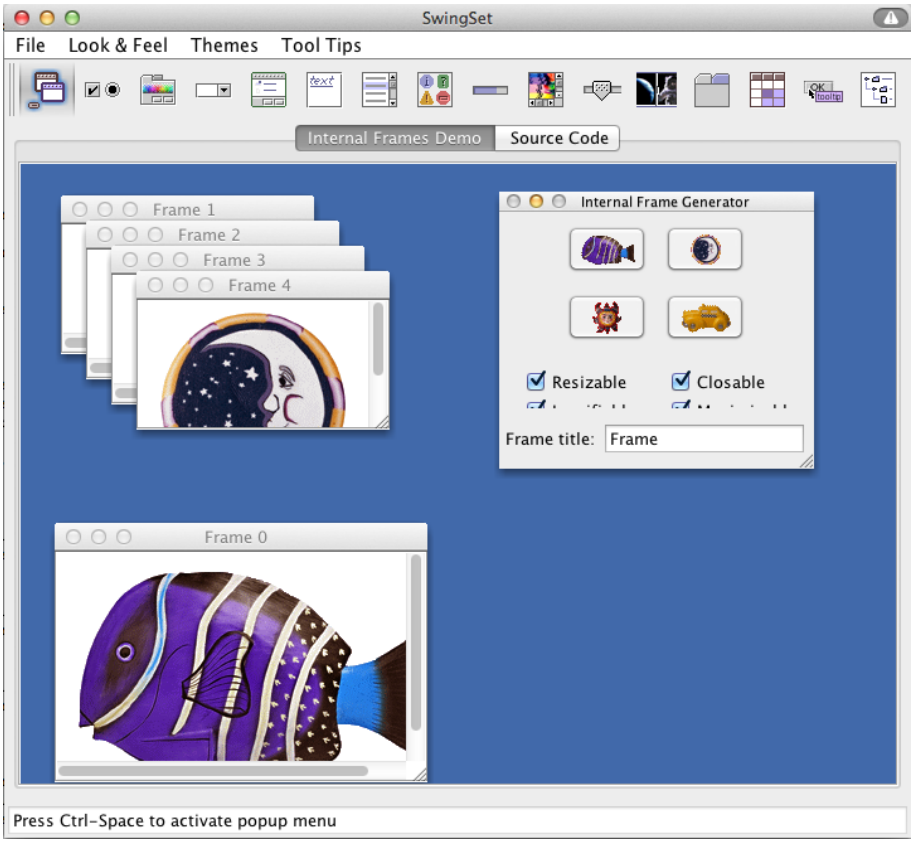
— Les méthodes `getGraphics()`, `getWidth()` et `getHeight()` sont des méthodes d'instance de `JPanel`, et peuvent donc être appelées par l'objet de classe interne anonyme implantant `ActionListener`.

— De la même manière, un composant `JComboBox` (menu déroulant) est créé, avec quatre champs possibles, grâce au constructeur `JComboBox(Object[])`. Lorsque le champ sélectionné est modifié, un événement de classe `ItemEvent` est généré. N.B. : un événement `ActionEvent` est également généré lorsque l'utilisateur achève de choisir un champ, quel qu'il soit (différent ou non du précédent). Un auditeur anonyme implantant l'interface `ItemListener` est as-

socié au `JComboBox`, et redéfinit la méthode `itemStateChanged(ItemEvent)` de manière à modifier la couleur de la diagonale si nécessaire.
— La méthode `getSelectedIndex()` de la classe `JComboBox` retourne l'indice du champ en cours de sélection.
— La variable `c` de type `JComboBox` est déclarée `final` pour pouvoir être utilisée par l'objet de classe interne anonyme implantant `ItemListener`.
— **La méthode `add(Component)` de la classe `JPanel` permet l'ajout des composants graphiques** (ici le bouton puis le menu déroulant).

5.2 Autres composants

En consultant le document suivant :
`https://java.net/projects/ojt-accessibility/downloads/download/Demos/SwingSet2.jar`

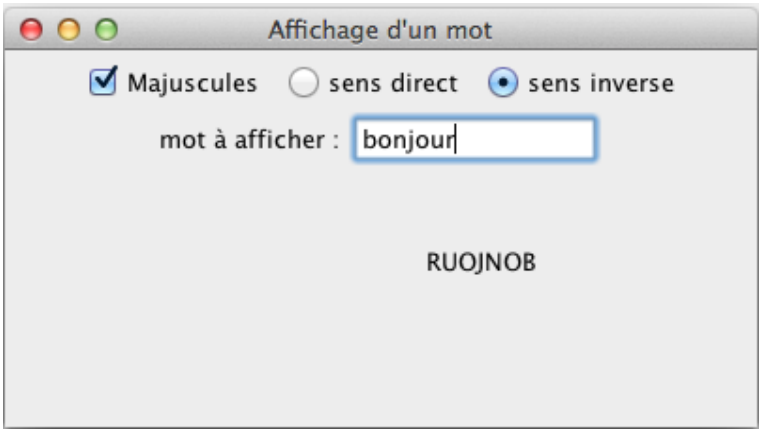


il est possible d'avoir un aperçu de tous les composants `swing` et de leur utilisation. Ces composants sont nombreux, et leur utilisation est très flexible (le cas des composants `JButton` et `JComboBox` ne se limite pas à l'exemple précédent). Il n'est pas question de décrire ici l'ensemble de ces composants. Pour la plupart, ils correspondent à des comportements prédéfinis qui génèrent des événements de diverses classes. Ces événements sont traitables par différentes sortes d'auditeurs qui doivent être associés à la source des événements. De plus, ces composants proposent diverses méthodes permettant de connaître et manipuler leur état courant.

Parmi les composants courants, on peut citer :

— Les cases à cocher, de classe `JCheckBox`, qui ont des étiquettes et peuvent être sélectionnées ou pas.
— Les boutons radio, de classe `JRadioButton`, sont également munis d'étiquettes et peuvent être sélectionnés ou pas. Il est possible de les regrouper dans des objets de classe `ButtonGroup`, ce qui impose alors qu'un seul bouton radio par groupe peut être sélectionné à chaque instant : on utilise ces boutons pour représenter des choix qui s'excluent.
— Les champs textuels, de classe `JTextField`, permettent notamment à l'utilisateur de saisir des chaînes de caractères directement sur l'interface graphique. On peut expliciter le rôle d'un `JTextField` en plaçant une étiquette de classe `JLabel` juste à côté.

L'exemple qui suit illustre l'utilisation de ces trois nouveaux composants, et génère l'interface suivante :

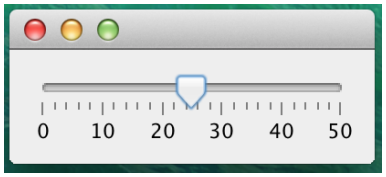


Dans cet exemple, les événements générés lors des actions de l'utilisateur sur les `JCheckBox` (`ItemEvent` permettant de savoir s'il y a eu sélection ou déselection via la méthode `getStateChange()`) et `JRadioButton` (`ActionEvent`) ne sont pas traités. On se contente d'observer l'état de ces composants à chaque pression sur un bouton de la souris : la chaîne de caractères du `JTextField` est récupérée, mise en majuscule si la `JCheckBox` est sélectionnée, renversée si le bouton radio d'étiquette ``sens inverse'' est sélectionné (auquel cas l'autre bouton radio est nécessairement désélectionné) et affichée à l'endroit où on a cliqué.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class AfficheurMot extends JPanel {
    private int pos_x,pos_y;
    private String mot;
    public AfficheurMot() {
        mot="";
        pos_x=0;
        pos_y=0;
        // case a cocher indiquant si on affiche le mot en majuscules ou pas
        final JCheckBox maj=new JCheckBox("Majuscules");
        this.add(maj);
        // bouton radio pour afficher le mot a l'endroit
```

```
        JRadioButton b1=new JRadioButton("sens direct");
        // bouton radio pour afficher le mot a l'envers
        final JRadioButton b2=new JRadioButton("sens inverse");
        // groupement des boutons radio
        ButtonGroup bg=new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        b1.setSelected(true);
        this.add(b1);
        this.add(b2);
        // champ de texte : mot a afficher
        this.add(new JLabel("mot a afficher :"));
        final JTextField tf=new JTextField("");
        tf.setColumns(10);
        this.add(tf);
        // gestion des clics souris
        this.addMouseListener(new MouseAdapter() {
                public void mousePressed(MouseEvent e) {
                    pos_x=e.getX();
                    pos_y=e.getY();
                    mot=tf.getText();
                    if (maj.isSelected()) mot=mot.toUpperCase();
                    if (b2.isSelected()) mot=new StringBuffer(mot).reverse().to
                    repaint();
                }
            });
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawString(mot,pos_x,pos_y);
    }
}
public class TroisComposants {
    public static void main(String[] args) {
        JFrame fenetre=new JFrame("Affichage d'un mot");
        fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        AfficheurMot dessin=new AfficheurMot();
        dessin.setPreferredSize(new Dimension(400,200));
        fenetre.setContentPane(dessin);
        fenetre.pack();
        fenetre.setVisible(true);
    }
}
```

Finalement, on pourra citer également la classe `JSlider`. Cette classe permet de créer et de manipuler des éléments d'interface utilisateur similaires à celui montré dans la figure ci-dessous :



Voici le code correspondant à cet exemple :

```
import javax.swing.*;

public class SliderExample extends JFrame{

        public SliderExample() {

                JSlider slider = new JSlider(JSlider.HORIZONTAL, 0, 50, 25);
                slider.setMinorTickSpacing(2);
                slider.setMajorTickSpacing(10);

                slider.setPaintTicks(true);
                slider.setPaintLabels(true);

                JPanel panel=new JPanel();
                panel.add(slider);
                add(panel);
        }

        public static void main(String s[]) {
                SliderExample frame=new SliderExample();
                frame.pack();
                frame.setVisible(true);

        }
}
```

Un exemple plus complet de l'utilisation de la classe `JSlider` sera présenté dans la section "Approche Modèle - Vue - Contrôleur".

5.3 Composants graphiques et enchaînement des actions de l'utilisateur

En programmation événementielle faisant intervenir des composants graphiques tels que des boutons, des menus déroulants, ..., il n'est pas toujours facile de prévoir tous les enchaînements possibles d'actions de l'utilisateur, et donc d'assurer la cohérence du comportement obtenu. Il est alors parfois utile d'interdire momentanément toute action sur un certain nombre de composants, avant de les autoriser à nouveau, grâce à la méthode `setEnabled(boolean)`. L'exemple partiel suivant crée deux boutons sur lesquels les pressions doivent se faire en alternance.

```
public class ExempleEnable extends JPanel {
    public ExempleEnable() {
```

```
        final JButton b1=new JButton("bouton 1");
        final JButton b2=new JButton("bouton 2");
        b2.setEnabled(false);
        ActionListener l=new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    b1.setEnabled(true);
                    b2.setEnabled(true);
                    ((JButton)e.getSource()).setEnabled(false);
                }
            };
        b1.addActionListener(l);
        b2.addActionListener(l);

    }
}
```

5.4 Gestionnaires de mise en page

Dans les exemples précédents, il n'est a priori pas possible de contrôler l'emplacement des composants graphiques rajoutés au `JPanel`. Java gère le plus souvent automatiquement le choix de la taille et de la position des composants peu à peu ajoutés. Néanmoins, Java propose divers types de *gestionnaires de mise en page* (layout managers) qui permettent d'influer plus ou moins sur la disposition des composants.

L'apparence réelle des composants graphiques à l'écran dépend de deux choses :

— l'ordre dans lequel ils sont ajoutés au panneau,

— le gestionnaire utilisé par le panneau pour leur mise en page : il détermine comment ce panneau va être "découpé" et comment les composants seront placés dans le panneau.

Chaque panneau peut avoir son propre gestionnaire de mise en page. "Arranger" l'interface utilisateur consiste à imbriquer les panneaux et à utiliser les gestionnaires appropriés de mise en page. Seuls les gestionnaires `FlowLayout` et `BorderLayout` seront étudiés ici. D'autres types de gestionnaires de mise en page existent : voir les classes implantant les interfaces `LayoutManager` et `LayoutManager2`. N.B. : tous les gestionnaires de mise en page implantent l'interface `LayoutManager`.

5.4.1 Gestionnaire `FlowLayout`

Jusque-là, la mise en page automatique est celle utilisée par défaut dans la classe `JPanel` : Java place les composants au fur et à mesure de leur ajout, ligne par ligne en partant du haut, en plaçant le plus possible de composants par ligne, et en les centrant. La taille des composants graphiques est en général calculée automatiquement par rapport à leurs caractéristiques (taille préférée). Pour bien observer cette mise en page, il suffit de modifier à la souris la taille des fenêtres dans les exemples précédents : la position des éléments de l'interface est recalculée en conséquence.

Cette gestion correspond aux gestionnaires de classe `FlowLayout`. A la création d'un `JPanel`, un objet de classe `FlowLayout` est donc créé, et il gère la mise en page des composants ajoutés dans ce `JPanel`.

5.4.2 Gestionnaire `BorderLayout`

La méthode setLayout() associe un gestionnaire à un panneau donné. Si un `JPanel` doit appliquer un autre gestionnaire que `FlowLayout`, il suffit de lui faire exécuter la méthode `setLayout(LayoutManager)`. Par exemple :

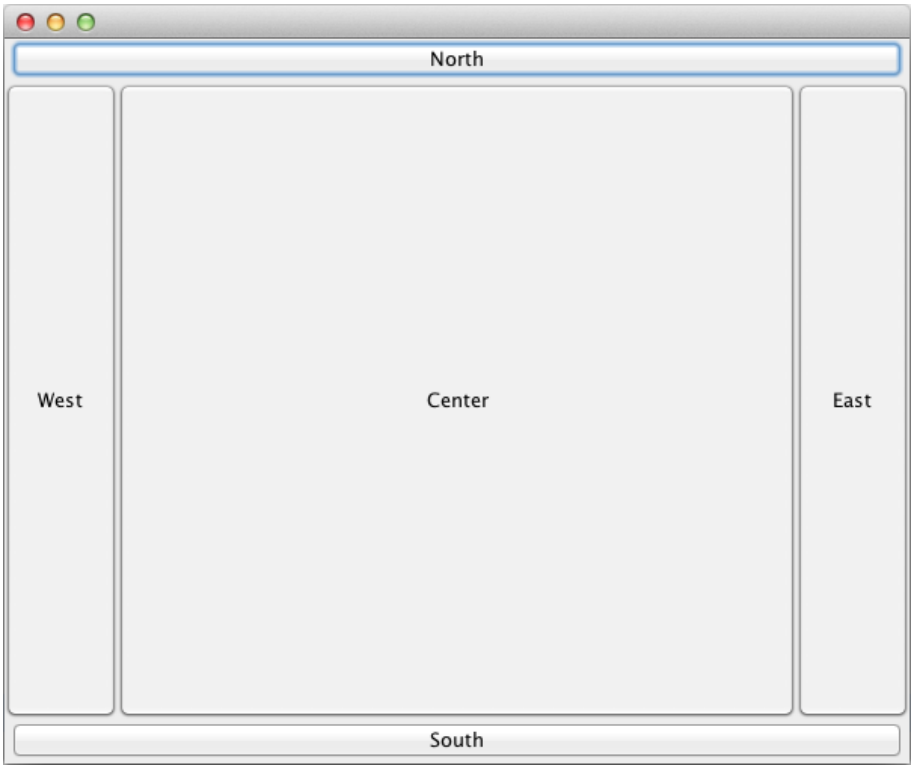
```
JPanel jp=new JPanel();
jp.setLayout(new BorderLayout());
```

Une fois le gestionnaire de mise en page initialisé, l'ajout des composants sur le panneau peut être effectué.

Avec un gestionnaire `BorderLayout`, l'emplacement d'un composant est indiqué comme une direction géographique : nord, sud, est, ouest et centre. Cette direction est précisée sous la forme d'une contrainte lors de l'appel à la méthode `add(Component composant,int contrainte)` des panneaux. Il suffit d'utiliser les constantes de classe entières telles que `BorderLayout.NORTH`, `BorderLayout.CENTER`, ...

```
import java.awt.*;
import javax.swing.*;
public class ExempleBorderLayout {
    public static void main(String[] args) {
        JPanel jp=new JPanel();
        jp.setLayout(new BorderLayout());
        jp.add(new JButton("North"), BorderLayout.NORTH);
        jp.add(new JButton("South"), BorderLayout.SOUTH);
        jp.add(new JButton("East"), BorderLayout.EAST);
        jp.add(new JButton("West"), BorderLayout.WEST);
        jp.add(new JButton("Center"), BorderLayout.CENTER);
        JFrame f=new JFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jp.setPreferredSize(new Dimension(600,480));
        f.setContentPane(jp);
        f.pack();
        f.setVisible(true);
    }
}
```

produit l'interface graphique suivante :



5.4.3 Gestionnaire `GridLayout`

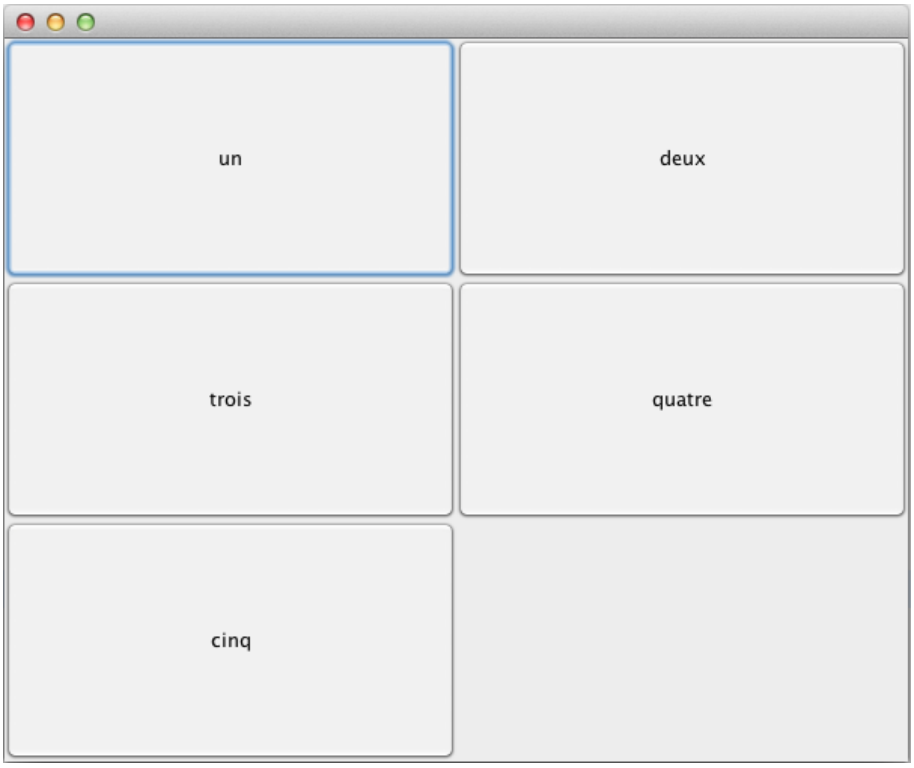
Ce type de mise en page (grille) offre une plus grande maîtrise de la disposition de composants dans un panneau. `GridLayout` divise le panneau en lignes et en colonnes. Chaque composant ajouté au panneau est placé dans une cellule de la grille de gauche à droite puis de haut en bas.

Exemple :

```
import java.awt.*;
import javax.swing.*;
public class ExempleGridLayout {
    public static void main(String[] args) {
        JPanel jp=new JPanel();
        jp.setLayout(new GridLayout(3,2));
        jp.add(new JButton("un"));
        jp.add(new JButton("deux"));
        jp.add(new JButton("trois"));
        jp.add(new JButton("quatre"));
        jp.add(new JButton("cinq"));
        JFrame f=new JFrame();
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jp.setPreferredSize(new Dimension(400,250));
        f.setContentPane(jp);
        f.pack();
        f.setVisible(true);
```

```
    }
}
```

produit l'interface graphique suivante :



5.4.4 Mise en page hiérarchique

Un panneau est lui-même un composant graphique. Il est donc possible de l'ajouter à un autre panneau. En attribuant aux panneaux contenants et contenus différents gestionnaires de mise en page, on peut obtenir une grande variété de dispositions de composants. L'exemple `DessinDiagonaleComposants` peut ainsi être mieux disposé de la manière suivante.

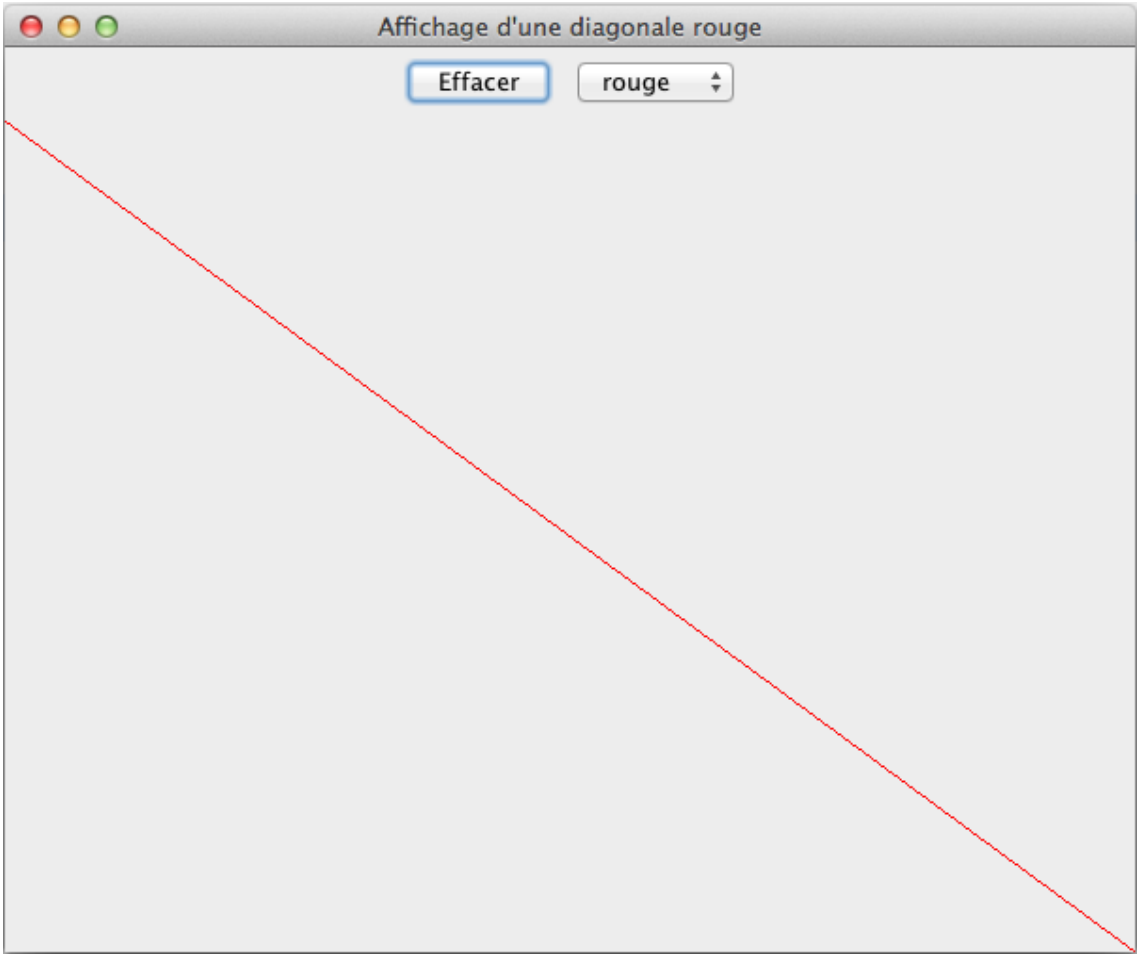
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class DessinDiagonaleComposantsHierarchiques extends JPanel {
    private Color couleur;
    public DessinDiagonaleComposantsHierarchiques() {
        couleur=Color.red;
        final JPanel zone_dessin=new JPanel() {
                public void paintComponent(Graphics g) {
                    super.paintComponent(g);
                    int h=getHeight();
                    int w=getWidth();
                    g.setColor(couleur);
```

```
                g.drawLine(0,0,w-1,h-1);
            }
        };
        // bouton "Effacer"
        JButton b=new JButton("Effacer");
        b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    zone_dessin.getGraphics().clearRect(0,0,getWidth(),getHeigh
                }
            });
        // choix de la couleur de la diagonale
        final JComboBox c=new JComboBox(new String[] {"rouge","vert","jaune","b
        c.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    switch (c.getSelectedIndex()) {
                    case 0 :
                        couleur=Color.red;
                        break;
                    case 1 :
                        couleur=Color.green;
                        break;
                    case 2 :
                        couleur=Color.yellow;
                        break;
                    case 3 :
                        couleur=Color.blue;
                        break;
                    default :
                        couleur=Color.black;
                    }
                    repaint();
                }
            });
        // les composants sont places au-dessus de la zone de dessin
        JPanel haut=new JPanel();
        haut.add(b);
        haut.add(c);
        this.setLayout(new BorderLayout());
        this.add(haut,BorderLayout.NORTH);
        this.add(zone_dessin,BorderLayout.CENTER);
    }
}
public class AffichageDiagonaleComposantsHierarchiques {
    public static void main(String[] args) {
        JFrame fenetre=new JFrame("Affichage␣d'une␣diagonale␣rouge");
```

```
        fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        DessinDiagonaleComposantsHierarchiques dessin=new DessinDiagonaleCompos
        dessin.setPreferredSize(new Dimension(600,480));
        fenetre.setContentPane(dessin);
        fenetre.pack();
        fenetre.setVisible(true);
    }
}
```

Dans ce programme, les composants prédéfinis sont placés dans le `JPanel haut` qui dispose d'un gestionnaire `FlowLayout`. L'affichage de la diagonale se fait dans le `JPanel zone_dessin`. Le panneau principal est muni d'un gestionnaire `BorderLayout`. On lui ajoute au nord le `JPanel haut`, et au centre le `JPanel zone_dessin`.

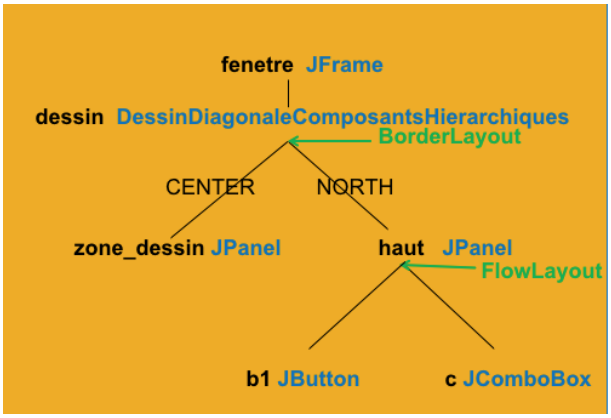
N.B. : dans cette version, on évite que le menu déroulant s'efface momentanément avec la diagonale, puisque l'effacement ne se fait que sur le `JPanel zone_dessin`.



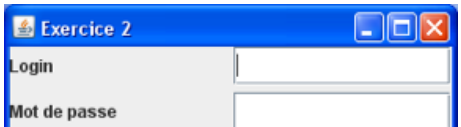
Arborescence des composants d'une interface graphique Afin de faciliter la compréhension de l'organisation des composants dans une interface graphique, il est conseillé, avant de passer à

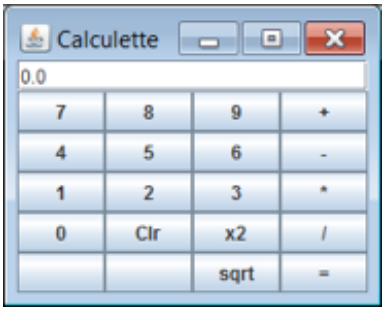
la programmation, de faire *l'arborescence de ses composants* en indiquant, au niveau de chaque composant `Container`, le gestionnaire de mise en page utilisé.

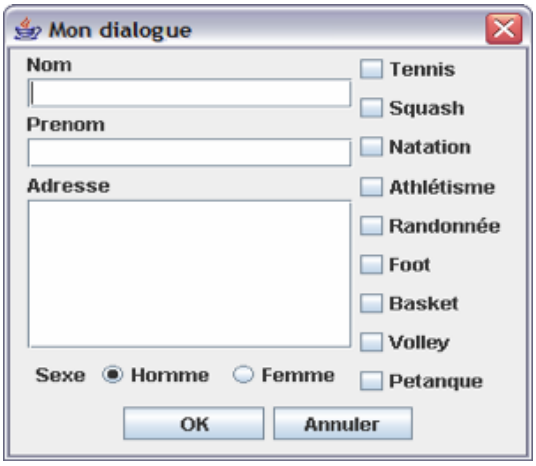
La racine de l'arbre est la fenêtre de l'interface, les nœuds sont les containers (souvent des `JPanel`) et les feuilles sont les composants figurant dans le composant `Container` parent. L'arborescence des composants de l'interface de l'exemple précédent est représentée sur la figure ci-dessous.



Exercice : Faire l'arborescence des composants des interfaces graphiques représentées ci-dessous.







6 Approche Modèle - Vue - Contrôleur

Pre-requis

Pour aborder cette partie, vous devez
— maîtriser le cours sur les interfaces graphiques sous JAVA
— savoir comment fonctionne une JFrame, un JPanel, un Listener, ,...
— bien faire la différence entre tous les constituants d'une interface

Contenu

Dans cette partie, on expliquera
— l'approche modèle vue, contrôleur et le design Observeur/Observé
— comment ce design est implémenté en java
— ce que sont les design patterns

6.1 Problème lié aux Interfaces graphiques

Dans l'absolu, une application n'est pas dépendante de son interface. Il devrait être possible d'écrire un logiciel puis de penser son interface après coup (voire de proposer plusieurs interfaces).

L'objectif de ce chapitre est de vous proposer une solution élégante, l'approche **MVC (Modèle-vue-contrôleur)**, classiquement utilisée et facile à mettre en oeuvre en JAVA pour proposer et développer des logiciels correctement structurés.

A l'issue de ce chapitre, vous devrez avoir une réponse aux questions suivantes :
— comment ajouter ou retirer une interface à l'exécution ?
— comment développer des interfaces différentes pour une même application ?
— comment ne pas mélanger le code lié à l'interface et le programme chargé de modifier les données ?
— comment écrire du code sans se préoccuper de l'interface ?

6.2 Modèle Vue Contrôleur

6.2.1 Principe

L'approche MVC cherche à séparer une application en trois blocs
— Le **Modèle** qui se charge de gérer l'ensemble des données et leur évolution (mise à jour des données, dynamique, règles dans un jeu vidéo, ...)
— la **Vue** qui se charge d'afficher les données
— le **Contrôleur** qui se charge de gérer les actions de l'utilisateur et d'appeler les actions correspondantes sur le modèle

6.2.2 Les différents blocs

Le Modèle. Le **modèle** gère les données et peut être crée indépendamment du reste. Il doit juste fournir des méthodes publiques correspondant aux actions possibles de l'utilisateur et aux lois

d'évolution du système à modéliser.

Par exemple, si l'objectif est de construire un logiciel de gestion, le modèle doit stocker l'ensemble des données (ou les accès à une BDD) et toutes les méthodes permettant de modifier certaines données et d'effectuer des calculs.

De la même manière, si l'objectif est de construire un jeu de stratégie temps réel (RTS), le modèle doit stocker la position des différentes troupes, gérer leurs positions et les déplacements, gérer les combats, gérer la destruction d'unité, la construction de bâtiments, ...

Par contre, le *Modèle* ne se charge pas d'afficher les données, c'est à la *Vue* de le faire. Le *Modèle* doit simplement connaître la *Vue* pour pouvoir lui demander de mettre à jour l'affichage dès qu'il y a des modifications.

La Vue. La **Vue** gère l'affichage des données. En théorie (mais cela peut être plus compliqué), elle ne s'occupe pas de la manière dont les données peuvent évoluer mais a pour objectif de rendre compte sur l'écran des données à un instant t.

Par exemple, si l'objectif est de construire un logiciel de gestion, la vue aura pour objectif de présenter les différentes données dans un `JPanel` à des endroits spécifiques en utilisant certaines couleurs, ...

Si l'objectif est de construire un jeu RTS, la vue se chargera d'afficher la carte du monde, les unités en fonction de leur position et de leur état, les bâtiments visibles, ...

La *Vue* est appelée par le *Modèle* dès que celui-ci estime que l'affichage doit être mis à jour. La *Vue* connaît le modèle puisqu'elle utilise les données du modèle pour construire ce qui sera affiché à l'écran.

Le Contrôleur. Le **contrôleur** a pour objectif de gérer l'ensemble des actions de l'utilisateur. Il ne modifie pas directement les données, mais appelle une méthode du modèle avec les paramètres correspondant à ce que souhaite faire l'utilisateur.

Par exemple, si l'objectif est de construire un logiciel de gestion, le contrôleur réagira aux clics de souris de l'utilisateur et appellera les méthodes du modèle permettant d'effectuer des modifications dans les données ou de lancer des calculs.

Si l'objectif est de construire un jeu RTS, le contrôleur se chargera d'analyser les clics de souris et d'appeler les méthodes correspondant aux actions des utilisateurs comme donner un ordre de déplacement à une unité, sélectionner une unité, etc ...

Le contrôleur est appelé par l'utilisateur et connaît le modèle pour pouvoir y appeler des méthodes.

La distinction entre *Vue* et *Contrôleur* est parfois délicate puisque le contrôleur peut s'appuyer sur la vue pour gérer les demandes des utilisateurs (exemple cliquer à un endroit de la Vue pour effectuer un changement). Le plus souvent, le contrôleur correspond à un `Listener` alors que la Vue est le `JComponent` lui-même.

6.2.3 Diagramme d'interaction

Le modèle MVC fonctionne de la manière suivante :

1. l'utilisateur déclenche une action interprétée par le *contrôleur*
2. le *contrôleur* analyse l'action de l'utilisateur et appelle une méthode spécifique du modèle en fonction des paramètres de l'action de l'utilisateur
3. le *modèle* modifie les données en conséquence et demande ensuite à la vue de se mettre à jour
4. la *vue* accède aux données du modèle en rend compte à l'écran des modifications effectuées

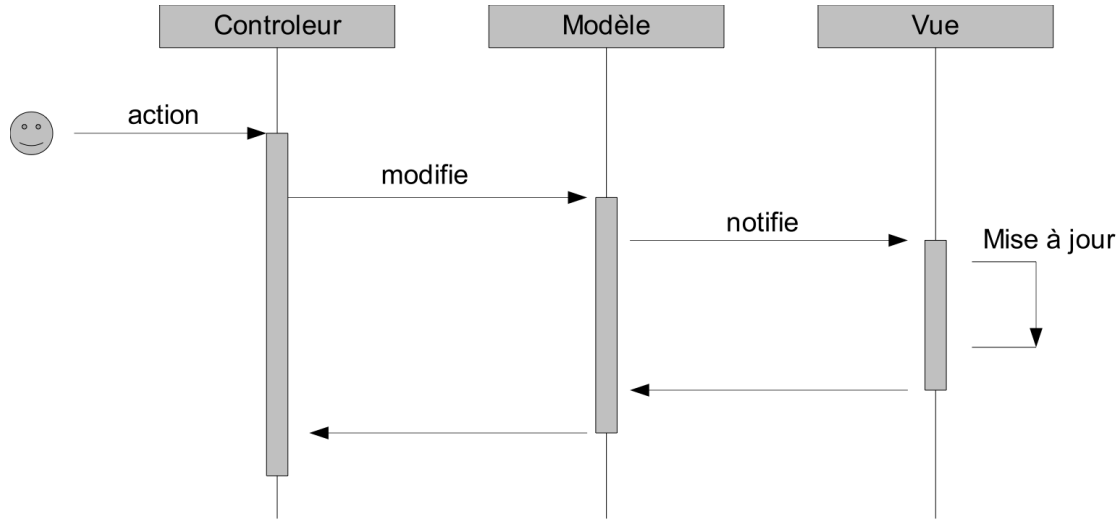


FIGURE 13 – Diagramme d'interaction de l'approche MVC

6.3 Observer / Observable

L'approche MVC est la combinaison de plusieurs '*design patterns*' [1]. Le *design pattern* Observeur/Observé est chargé de gérer le lien entre le modèle et la vue :

— Observé désigne l'objet à afficher et correspond au modèle

— Observeur désigne ce qui va observer l'objet et correspond aux différentes vues

Ce *design pattern* est mis en œuvre en Java par les classes `Observable` et l'interface `Observer` que nous décrirons en détail dans la partie 6.3.2.

1. Un **design pattern** est une solution éprouvée à un problème d'architecture logicielle rencontré fréquemment en informatique, cf fin de ce chapitre pour plus d'informations.

6.3.1 Le design pattern Observeur-Observé

Le design pattern Observeur/Observé fonctionne de la manière suivante :

Observé. Observé désigne l'objet à être affiché. Il correspond au modèle et doit proposer plusieurs méthodes

- une méthode publique `attache` permettant d'associer un Observeur supplémentaire affichant les données
- une méthode privée `notifie` qui demande à tous les Observeurs associés de mettre à jour leur affichage si cela est nécessaire

Observeur. Observeur désigne une Vue et doit proposer

- une méthode `miseAJour()` permettant d'afficher correctement l'Observé à qui il est associé

Diagramme de Classe simplifié. Un *design pattern* est une solution réutilisable. Une telle solution se structure donc en deux parties :

- la partie supérieure (`Sujet` correspondant à Observé et `Observeur`) correspond aux classes générales permettant de mettre en place le design Observeur-Observé
- la partie inférieure (`Sujet_reel` et `Observeur_reel`) correspond à l'utilisation particulière de ces classes générales pour mettre en œuvre le *design* Observeur-Observe sur un cas spécifique.

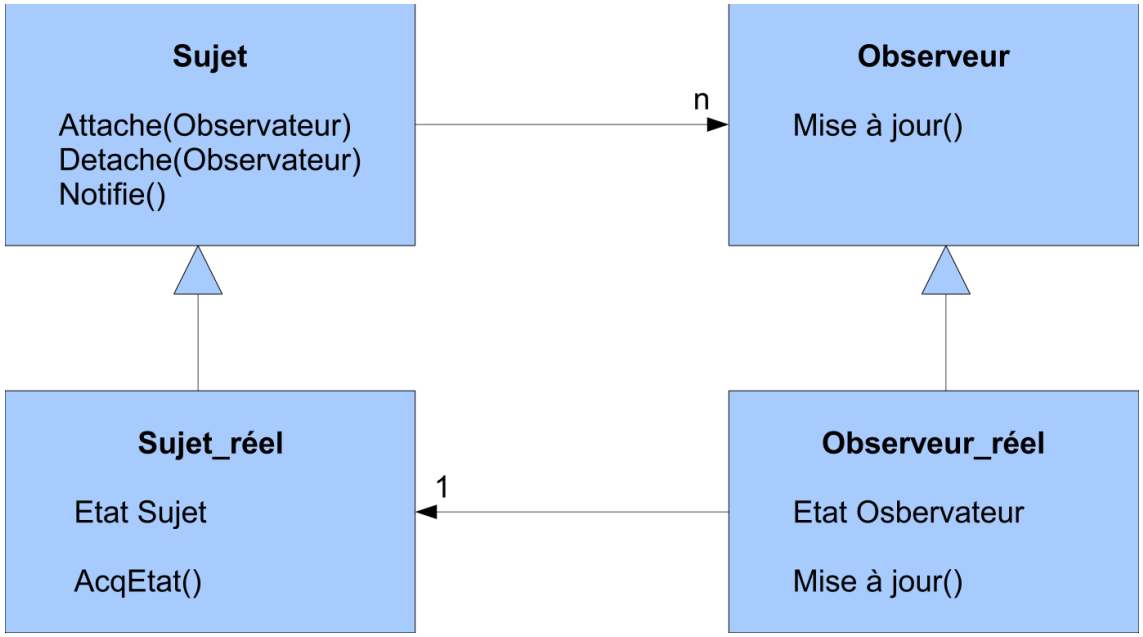


FIGURE 14 – Diagramme classe design Observateur/Observe

Les classes présentées dans la figure précédente sont organisées de la manière suivante :

- **Sujet.** désigne la classe de laquelle hérite tous les observés. `Sujet` possède la méthode `Attache` pour ajouter un Observeur et la méthode `notifie` pour prévenir d'une mise à jour

— la méthode `notifie` effectue une boucle sur tous les observeurs et appelle la méthode `MiseAjour` sur chacun d'entre eux.
— **Observeur.** désigne l'interface qu'implémente tous les observeurs. `Observeur` possède la méthode abstraite `miseAJour` chargée de mettre à jour l'affichage quand l'observeur est notifié.
— **Sujet réel.** désigne les données spécifiques que l'on souhaite observer. `Sujet Réel` hérite de `Sujet` et dispose d'un état interne correspondant aux données à traiter. `Sujet réel` dispose par héritage des méthodes `notifie` et `attache` définies dans `Sujet`.
— **Observeur réel.** désigne la vue que l'on souhaite mettre en oeuvre. Il faut redéfinir la méthode abstraite `miseAjour` de la classe `Observeur` pour gérer l'affichage réel des données du modèle. `Observeur reel` doit en outre possèder un lien vers le modèle `Sujet reel` pour accéder aux données à afficher.

6.3.2 Les classes Observer et Observable

JAVA propose des classes `Observer` et `Observable` qui mettent en œuvre ce *design*.

6.3.3 La classe Observable

La classe `java.util.Observable` correspond à la classe `Sujet`. Elle permet d'obtenir par héritage les différentes classes pouvant être observées. La classe `Observable` définit les méthodes
— `notifyObservers()` permettant de demander aux `Observer` enregistrés de se mettre à jour (par la méthode `update()`)
— `setChanged()` permettant de marquer l'objet comme devant être mis à jour (si la méthode n'est pas appelée auparavant, `notifyObservers()` ne fait rien)
— `addObserver(Observer O)` permettant d'enregistrer un Observer supplémentaire lié à l'Observable.

6.3.4 L'interface Observer

L'interface `Observer` correspond à la classe `Observeur` du design pattern. Elle permet d'obtenir par implémentation les `Observer`. Cette interface ne dispose que d'une méthode à implémenter
— `void update(Observable o, Object arg)` : cette méthode est appelée par la méthode `notifyObservers()` de `Observable` pour la mise à jour de l'affichage. `o` désigne l'observable à afficher et `arg` des arguments qu'il est possible de transmettre lors de l'appel de `notifyObservers()`

6.3.5 Mise en place d'un MVC avec Observer et Observable

Mettre en place le design Observeur-Observé se fait en trois étapes :

a. Le modèle. Il faut dans un premier temps créer le modèle par héritage de la classe `Observable` :
— récupérer par héritage les méthodes de `Observable` : `addObserver`, `notifyObservers` et `setChanged`
— ajouter tous les attributs permettant de stocker les données
— ajouter les méthodes permettant de modifier ces données

b. La vue. Il faut ensuite créer la vue en implémentant l'interface `Observer`
— Pour ce faire, il est nécessaire de redéfinir la méthode `update`

c. Liaison Modèle-Vue. Enfin, il reste à lier le modèle à la vue.
— il faut ajouter les appels à `notifyObservers` et `setChanged` dans le modèle pour demander les mises à jour de la vue.
— Il faut en outre faire un `main` ajoutant la vue au modèle avec la méthode `addObserver` héritée de `Observable`

6.4 Exemple

On souhaite faire une application permettant de modifier la taille d'un disque.

6.4.1 Description des éléments MVC

— le modèle correspond aux données, c'est à dire la description du disque à modifier
  — Le modèle est constitué d'une classe `Disque` de type `Observable`.
  — Cette classe possède un attribut `taille`.
— la vue correspond à l'affichage du disque. On considérera deux vues chacune implémentant `Observer`
  — une vue purement textuelle affichant la taille du disque
  — une vue graphique dessinant le disque sur un `JPanel`
— le contrôleur correspond à l'interaction avec l'utilisateur. on considérera deux contrôleurs
  — un contrôleur textuel avec la classe `Scanner`
  — un contrôleur graphique fondé sur un `JSlider`

6.4.2 Mise en place du modèle

Le modèle correspond au Disque. La classe `Disque` hérite de `Observable` et possède un attribut `taille`.

Pour gérer le lien avec la vue, la classe `Disque` doit pouvoir fournir les informations utiles à l'affichage. Elle dispose ainsi d'une méthode publique `getTaille()` retournant la taille du disque.

Pour gérer le lien avec le contrôleur, la classe doit proposer des méthodes publiques de modification. La classe disque doit donc disposer d'une méthode `setTaille()`

Pour gérer la mise à jour de l'affichage, la classe `Disque` doit appeler la mise à jour de l'affichage à chaque fois que sa taille est modifiée.

La classe `Disque` est donc la suivante

```
import java.util.Observable;

// la classe herite de Observable
public class Disque extends Observable {

    // declaration de l'attribut
    int taille;

    // constructeur
```

```
    public Disque() {
      taille = 10;
    }

    // getter pour la vue
    public int getTaille() {
      return (taille);
    }

    //setter pour le controleur
    public void setTaille(int t){
      if (t>0) taille=t;
      //prevenir la modification, methode de Observable
      setChanged();
      //notifier Observer, methode de Observable
      notifyObservers();
    }
}
```

6.4.3 Mise en place de la Vue Textuelle

La vue textuelle se charge simplement d'afficher la valeur de l'attribut `taille`.

Il suffit de créer une classe `VueTexte` qui implémente l'interface `Observer` et de surcharger la méthode `update(Observable o)`.

```
import java.util.Observable;
import java.util.Observer;


public class VueTexte implements Observer {

    //mise à jour de l'affichage
    public void update(Observable o, Object arg) {
      //consiste simplement à afficher la taille de l'observable passé
      System.out.println("vue texte :"+((Disque)o).getTaille());
    }
}
```

6.4.4 Mise en place de la vue Graphique

La vue Graphique va se charger d'afficher un cercle de la bonne taille lorsque l'affichage est mis à jour. La classe `VueGraph`, hérite de `JPanel` (pour être un composant affichable) et implémente `Observer` (pour pouvoir constituer une Vue). Elle dispose en outre d'un attribut correspondant au modèle à afficher.

Il suffit donc de créer la classe `VueGraph`, de surcharger les méthodes

— `update()` de `Observer` pour demander la mise à jour de l'affichage avec un `repaint()`

— `paint()` de `JPanel` pour afficher les informations que l'on souhaite

```
import java.awt.Graphics;
```

```
import java.util.Observable;
import java.util.Observer;

import javax.swing.JPanel;

public class VueGraph extends JPanel implements Observer{

    Disque modele;

    //pour répondre à une demande d'affichage
    public void update(Observable o, Object arg1) {
      //mise à jour du lien vers le modele à afficher
      modele=(Disque)o;
      //appel de la fonction d'affichage paint()
      repaint();
    }

    //surcharge de la méthode paint()
    public void paint(Graphics g) {
      super.paint(g);
      //affichage du cercle si il est reference
      if (modele!=null)
      {
          g.drawOval(0, 0, modele.getTaille(), modele.getTaille());
      }
    }
}
```

6.4.5 Mise en place du contrôleur textuel

L'objectif du contrôleur textuel est de vérifier rapidement que tout fonctionne. Il se limite simplement à un `main` qui crée les vues, fait le lien entre les vues et le modèle et modifie le modèle pour vérifier que la vue se met correctement à jour.

```
import java.awt.Dimension;
import java.util.Scanner;

import javax.swing.JFrame;

public class ControlText {

    public static void main(String[] args) {
      //création du modèle
      Disque d=new Disque();

      //création de la vue textuelle
      VueTexte vt=new VueTexte();

      //creation de la vue graphique
      VueGraph vg=new VueGraph();
      vg.setPreferredSize(new Dimension(100,100));
```

```
      JFrame f=new JFrame();
      //on ajoute le jpanel à la frame
      f.setContentPane(vg);
      f.pack();
      f.setVisible(true);

      //on fait le lien entre modèle et les vues
      d.addObserver(vt);
      d.addObserver(vg);

      //on fait la boucle de modification
      Scanner sc=new Scanner(System.in);
      //demande la nouvelle taille
      int choix=sc.nextInt();
      while(choix>0)
        {
        //modification
        //conduit à la mise à jour des vues
        d.setTaille(choix);
        choix=sc.nextInt();
        }
      System.exit(1);
    }
}
```

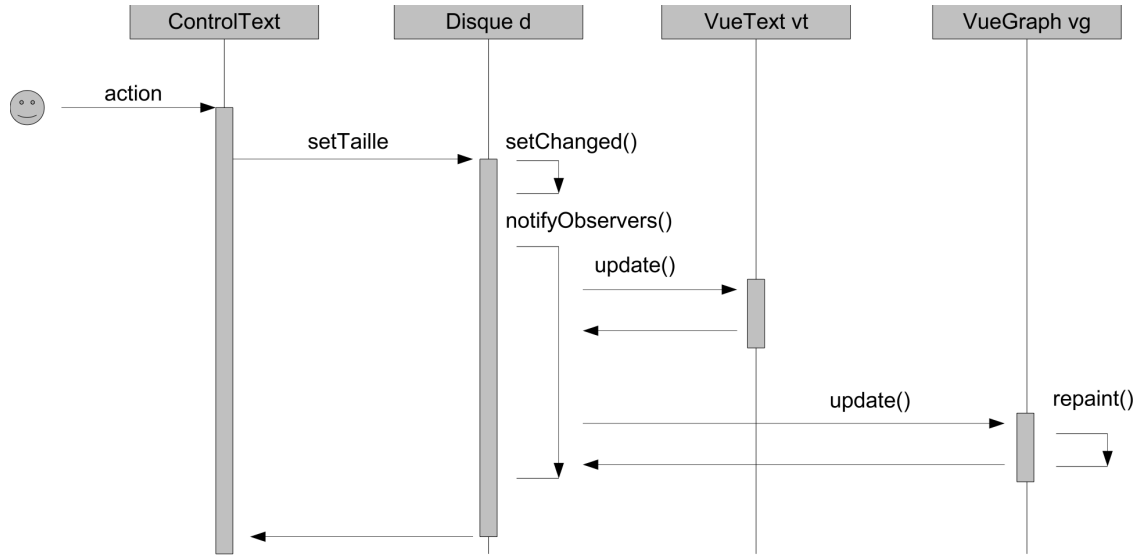


FIGURE 15 – Diagramme de séquence pour l'exécution de la méthode setTaille()

L'appel de `d.setTaille(choix)` dans la boucle déclenche les appels suivants :
— la méthode `setTaille()` de `Disque` qui modifie la taille du disque
  — la méthode `setTaille` appelle `setChanged()` qui specifie que le modèle a été modifié
  — la méthode `setTaille` appelle `notifyObservers()` qui appelle la mise à jour sur les observers référencés (cf `addObserver`)

        - Mise à jour de la vue textuelle
            - appel à la méthode `update` de `VueText` qui affiche la valeur de `taille`
        - mise à jour de la vue graphique
            - appel de la méthode `update` de `VueGraph` qui appelle `repaint` de `VueGraph`
            - la méthode `repaint` de `VueGraph` appelle la méthode `paint` de `VueGraph`
            - la méthode `paint` de `VueGraph` affiche le disque
        - remise de `changed` à false puisque l'affichage correspond au modèle
    - sortie de la méthode `setTaille`

6.4.6 Mise en place du contrôleur graphique

Le contrôleur Graphique sera représenté par un `JSlider`.

L'interaction avec l'utilisateur se fera donc en utilisant un `ActionListener`. Dès que la position associée au Slider change, le `JSlider` appelle la méthode `setTaille` du modèle et le modèle demande automatiquement à l'affichage de se mettre à jour.

Le contrôleur a besoin d'un attribut référençant le modèle pour savoir sur quel objet appeler la méthode.

```
import javax.swing.JSlider;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;


public class Controleur extends JSlider{

  //lien vers le modèle
  Disque modele;

  //constructeur
  public Controleur(Disque d)
  {
    super();
    //lien avec le modèle
    modele=d;

    //initialisation du Slider
    setMaximum(100);
    setMinimum(1);

    //ajout d'un listener pour suivre l'évolution du curseur
    addChangeListener(new ChangeListener(){
        public void stateChanged(ChangeEvent arg0) {
           modele.setTaille(getValue());
        }
    });
  }
}
```

6.4.7 Agencement des blocs

Il ne reste plus qu'à écrire un `main` pour lier tous les éléments. Le `main` se charge
— de créer le modèle
— de créer les vues
— de lier les vues au modèle
— de créer le contrôleur
Une fois ces éléments mis en relation, le contrôleur appelle les méthodes de modifications du modèle qui appellent automatiquement la mise à jour des vues.

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import javax.swing.JFrame;

public class Test {

    public static void main(String[] args) {
      // creer un modèle
      Disque d = new Disque();

      // creer les vues
      VueTexte vt = new VueTexte();
      VueGraph vg = new VueGraph();

      // attacher la vue au modèle
      d.addObserver(vt);
      d.addObserver(vg);

      // créer le contrôleur
      Controleur c = new Controleur(d);

      // ranger tout dans une frame
      JFrame frame = new JFrame();
      frame.setLayout(new BorderLayout());
      c.setPreferredSize(new Dimension(100, 50));
      vg.setPreferredSize(new Dimension(200, 200));
      frame.getContentPane().add(vg, BorderLayout.NORTH);
      frame.getContentPane().add(c, BorderLayout.SOUTH);
      frame.pack();
      frame.setVisible(true);
      d.setTaille(10);
      c.setValue(10);
    }
}
```

6.5 Synthèse de l'approche MVC

L'objectif de l'approche MVC est de séparer les données, de l'affichage et du contrôle.

6.5.1 Démarche

Pour produire un modèle MVC

1. il faut dans un premier temps identifier le modèle
2. il faut développer le modèle en proposant toutes les méthodes utiles
   — méthodes privées internes pour gérer la dynamique du modèle
   — méthodes publiques extérieures correspondant aux actions possibles de l'utilisateur
3. il faut construire les vues
   — déterminer les informations du modèle à afficher et proposer des méthodes pour récupérer l'information pertinente
   — construire les vues différentes en fonction de ce que l'on souhaite
4. il faut construire le contrôleur
   — déterminer l'interface entre le modèle et l'utilisateur
   — écrire le contrôleur qui analyse les actions de l'utilisateur et appelle les méthodes adaptées du modèles
5. créer un `main` qui lie les blocs M, V et C.

6.5.2 Intérêt

Le modèle MVC présente différents intérêts

— la possibilité de développer indépendamment Modèle, Vue et Contrôleur si les méthodes utiles aux uns et aux autres sont bien spécifiées
— la possibilité de développer plusieurs interfaces graphiques (vues) ou plusieurs contrôleurs pour des utilisateurs différents (multi-modalité) ou en changer au cours d'utilisation
— avoir un code correctement structuré et facile à analyser
— pouvoir développer et tester un modèle (qui est souvent le coeur d'un projet) indépendamment de son interface graphique.
— pouvoir facilement développer de nouvelles interfaces graphiques sans remettre en cause le code existant.

6.6 Une autre approche pour MVC

Ce chapitre s'est concentré sur une approche MVC fondée sur le design Observeur/Observé. Cette approche est particulièrement adaptée au langage JAVA pour lequel ce design est implémenté dans les classes fournies avec le langage.

L'approche MVC peut être abordée différemment dans d'autres contextes [2]. Dans cette approche, le controleur fait office de chef d'orchestre. Il contacte le modele pour effectuer les modifications et récupérer l'information puis transmet le résultat à la vue qui se charge d'afficher les données transmises.

2. Comme par exemple dans le cadre d'applications web (cf cours de pragrammation web) où il est important de séparer l'affichage, la gestion des données et le controleur

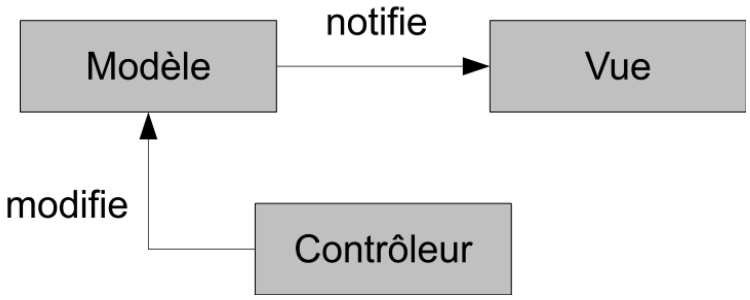


FIGURE 16 – Approche MVC fondée sur le design Observeur/Observé

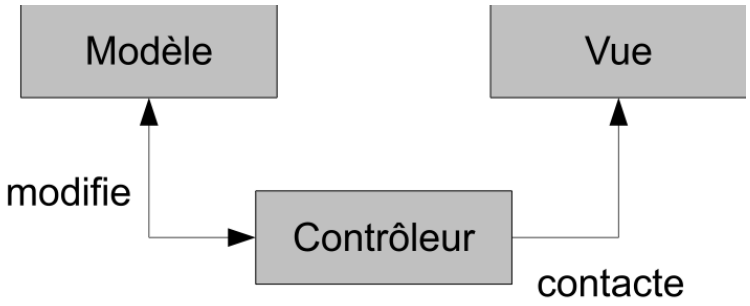


FIGURE 17 – Approche MVC où le controleur est central

6.7 Ouverture : Les patrons de conception

Les patrons de conception ou *design pattern* (comme le design Observeur/Observé) sont des solutions éprouvées destinées à résoudre des problèmes récurrents d'architecture et de conception en informatique.

Un *design* est indépendant d'un langage de programmation. Il correspond à une structure particulière à mettre en place et se présente sous la forme de diagramme de classe.

Il existe de nombreux *design patterns* adaptés à de nombreux problèmes. Il s'agit de solutions élégantes et pratiques. N'hésitez pas à y jeter un coup d'œil de temps en temps, vous trouverez peut être la solution à un problème sur lequel vous bloquez et vous aurez une meilleure vision des bonnes pratiques.

Objectif pédagogique

A l'issue de cette section, vous devrez savoir comment
— construire une interface graphique modulaire
— structurer une application flexible en terme de modèle, contrôle et vue

Références

— pour les design pattern : la page de wikipedia en résume un certain nombre
`http://fr.wikipedia.org/wiki/Patron_de_conception`
— le livre de référence sur les design pattern
`'Design Patterns - Catalogue de modèles de conceptions' de E. Gamma, R. Helm, R. Johnson, J. Vlissides`
— un livre très facile d'accès sur les design pattern (avec un chapitre dédié au MVC) `'Design patterns, tête la première', chapitre 3, Oreilly, 2005`

7 Introduction à l'ergonomie en informatique

L'ergonomie informatique a pour objectif d'améliorer l'interaction homme-machine et de faciliter l'utilisation et l'apprentissage des logiciels. Cette pratique cherche à concevoir ou modifier des interfaces utilisateurs afin qu'elles soient en adéquation avec les caractéristiques de leurs utilisateurs potentiels. Pour développer un logiciel ergonomique, il faut suivre des directives pour rendre les logiciels plus faciles à utiliser, et d'utiliser le bon sens pour créer des interfaces utiles et efficaces.

7.1 Principes de base des interfaces utilisateurs

Métaphores

Il s'agit d'utiliser les connaissances des utilisateurs pour transmettre des concepts et fonctionnalités du logiciel. Les métaphores permettent de mimer le comportement de l'interface sur celui d'un objet de la vie courante et donc déjà maitrisé par l'utilisateur. Par exemple, on utilise la métaphore de dossiers de fichiers pour le stockage des documents, l'utilisateur peut organiser son disque dur d'une manière qui est analogue à la façon dont il organise des classeurs. Un logiciel de gestion de photos représente les photos dans un album. La métaphore la plus connue est celle du bureau.

Modèle mental

L'utilisateur d'un logiciel dispose d'un certain nombre d'idées, de connaissances et de représentations qui lui permettent d'utiliser plus ou moins bien le logiciel. Cette image ou représentation est souvent désignée sous le nom de modèle mental. Connaître le modèle mental de l'utilisateur permet de connaître ses attentes et le logiciel doit refléter le modèle mental de l'utilisateur. Pour refléter le modèle mental de l'utilisateur, le logiciel doit avoir les caractéristiques suivantes :

- **Familiarité :** Le modèle mental se base principalement sur l'expérience. Les éléments d'une interface doivent être familiers.
- **Simplicité :** Il ne faut pas encombrer les éléments d'une interface de nombreuses fonctionnalités. Il faut garder l'interface simple.
- **Disponibilité :** Les éléments sont facilement accessibles.
- **Découverte :** Il faut encourager l'utilisateur à découvrir les éléments de l'interface.

Contrôle utilisateur

Il est recommandé de permettre à l'utilisateur, pas le logiciel, de lancer et de contrôler les actions. Certains logiciels tentent d'aider l'utilisateur en proposant des alternatives jugées bon pour l'utilisateur ou pour protéger l'utilisateur d'avoir à prendre des décisions détaillées. Cette approche donne le contrôle au logiciel, et non pas à l'utilisateur. Il faut fournir le niveau de contrôle à l'utilisateur selon son niveau (novice, expert, ..). Il est recommandé de fournir aux utilisateurs les fonctionnalités dont ils ont besoin tout en les aidant à éviter des actions irréversibles et dangereuses. Par exemple, dans le cas où l'utilisateur veut supprimer des données, il faut toujours donner un avertissement, mais permettre à l'utilisateur de procéder s'il le souhaite.

Communication et retour d'information

La communication entre le logiciel et l'utilisateur doit dépasser l'affichage des alertes uniquement en présence de danger. Au lieu de cela, il faut tenir l'utilisateur informé de ce qui se passe en donnant

un retour approprié lui permettant de communiquer avec le logiciel. Lorsqu'un utilisateur lance une action, il faut toujours fournir une indication que le logiciel a reçu la demande de l'utilisateur et qu'il est en train de la traiter. En effet, l'utilisateur veut savoir que sa demande est en cours de traitement. Si une demande ne peut pas être réalisée, il veut savoir pourquoi et ce qui peut être fait à la place.

- Lorsqu'une opération dure très longtemps, et si le logiciel ne répond pas aux événements pendant 2 secondes, le système affiche automatiquement le curseur indiquant une attente (le sablier, par exemple). L'utilisateur qui voit ce curseur sans d'autres retours pourrait penser que le logiciel a planté et le quitte par force.
- Pour les opérations potentiellement longues, il est recommandé d'utiliser un indicateur de progression qui fournit des informations utiles sur la durée de l'opération. L'utilisateur n'a pas besoin de savoir précisément combien de secondes une opération prendra, mais une estimation est utile. Il peut également être utile de communiquer le nombre total d'étapes nécessaires pour compléter une tâche, par exemple, un texte qui dit : "Copie de 20 fichiers sur 710".
- Il est recommandé de fournir les informations d'une manière directe et simple pour que les utilisateurs puissent comprendre. Par exemple, les messages d'erreur doivent préciser exactement ce qui a provoqué l'erreur ("Il n'y a pas assez d'espace sur ce disque pour enregistrer le document") et les actions possibles que l'utilisateur peut prendre pour y remédier ("Essayez d'enregistrer le document dans un autre endroit").

Cohérence

La cohérence dans l'interface permet aux utilisateurs de transférer leurs connaissances et compétences d'un logiciel à un autre. Il est recommandé d'utiliser les éléments standard de l'interface utilisateur proposé par SWING, dans le cas de java, pour assurer la cohérence au sein d'un logiciel que l'on développe et de bénéficier de la cohérence à travers d'autres logiciels. Les questions suivantes permettent de vérifier la cohérence :

- Y-a-t-il une cohérence au sein du logiciel lui-même ? Utilise-t-on une même terminologie pour les *toolbars* et les menus ? Les icônes signifient-elles la même chose à chaque fois qu'elles sont utilisées ? Les concepts sont-ils présentés de la même façon dans tous les modules ?
- Cohérence avec des versions précédentes : Utilise-t-on la même terminologie à travers les différentes versions ?
- L'interface utilisateur, correspond-elle aux attentes de l'utilisateur sans à avoir expliquer le fonctionnement du logiciel en détail ?

Stabilité

- L'interface utilisateur doit fournir un environnement compréhensible, familier et prévisible. Pour donner aux utilisateurs un sens visuel de stabilité, l'interface définit de nombreux éléments graphiques standards, tels que les menus, la barre de contrôle de la fenêtre, etc. Ces éléments standards offrent aux utilisateurs un environnement familier dans lequel ils savent comment ils se comportent.
- Pour donner aux utilisateurs un sens conceptuel de la stabilité, l'interface offre un ensemble fini et clair d'objets et un ensemble d'actions à effectuer sur ces objets. Par exemple, quand une commande de menu ne s'applique pas à un objet sélectionné ou de l'objet dans son état

actuel, la commande est grisée plutôt que omise.
— Pour aider à transmettre la perception de stabilité, il faut préserver les paramètres qui ont été modifiés par l'utilisateur tels que les dimensions et l'emplacement de la fenêtre. Lorsqu'un utilisateur configure son environnement à l'écran pour avoir une certaine mise en page, les réglages devraient le rester jusqu'à ce que l'utilisateur les modifie.

Tolérance

Il est recommandé d'encourager l'utilisateur à explorer le logiciel en utilisant le principe de tolérance, c'est-à-dire, rendre la plupart des actions facilement réversible. L'utilisateur a besoin de sentir qu'il peut essayer des choses sans endommager le système ou compromettre ses données. Il est possible, par exemple, d'annuler une action et de revenir à des commandes enregistrées, de sorte que l'utilisateur se sent confiant en utilisant le logiciel et cela permet de mieux se familiariser avec le logiciel.

L'utilisateur doit être averti quand il lance une tâche qui va causer la perte irréversible des données. Cependant, si les alertes apparaissent très souvent, cela peut signifier que le logiciel a des défauts de conception. Lorsque les solutions de repli sont présentées clairement et les retours d'information sont présentés au bon moment, l'utilisation du logiciel devrait être relativement exempt d'erreurs.

Intégrité esthétique

Intégrité esthétique signifie que l'information est bien organisée et présente une bonne conception visuelle. Le logiciel doit être agréable à regarder sur l'écran. Il est recommandé que les graphismes soient simples, et de les utiliser seulement quand ils améliorent réellement l'utilisabilité (la facilité d'utilisation). Il est inutile de surcharger les fenêtres et les boîtes de dialogue avec des dizaines d'icônes ou de boutons. Il faut éviter d'utiliser des symboles pour représenter des concepts arbitraires ; ils peuvent confondre ou distraire les utilisateurs. Les comportements des éléments graphiques doivent correspondre aux attentes de l'utilisateur. Il ne faut pas modifier le sens ou le comportement des éléments standards. Par exemple :
— Il faut toujours utiliser les *checkbox* pour des choix multiples.
— Les boutons doivent être utilisés pour une action immédiate (par exemple, “ouvrir” ou “jouer”).
— Il faut éviter d'utiliser des boutons pour en servir d'onglets.
Il est recommandé d'apporter le soin nécessaire aux graphismes de l'interface utilisateur d'un logiciel. Il ne faut pas sous-estimer l'impact des éléments graphiques de bonne qualité sur les utilisateurs. Une interface graphique de mauvaise qualité donne une mauvaise impression et peut affecter négativement la perception de l'utilisateur de la qualité globale d'un logiciel.

Réactivité

La réactivité est la façon dont l'utilisateur mesure la performance d'un logiciel. Un logiciel peut utiliser les meilleurs algorithmes de traitement des données et les meilleures techniques qui augmentent les performances, mais s'il ne répond pas instantanément à l'utilisateur, il semblera lent et l'utilisateur aura l'impression que c'est un mauvais logiciel.
— Reconnaître instantanément les demandes de l'utilisateur au logiciel.

— Ne pas attendre qu'une longue tâche se termine pour afficher des résultats à l'utilisateur. Si rien n'est affiché jusqu'à ce que tous les résultats soient terminés, l'utilisateur peut interpréter cela comme de la lenteur. Au lieu de cela, il est recommandé d'afficher des résultats partiels dès que possible afin de montrer qu'il y a une progression dans le traitement.
— Utiliser une barre de progression pour aider à estimer la durée du processus. L'utilisateur n'a pas toujours besoin de savoir précisément combien de temps une tâche prendra, mais il est important de lui donner une estimation.

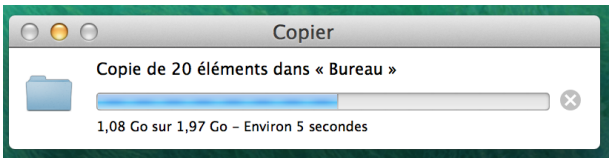


FIGURE 18 – Une barre de progression

Assister l'utilisateur

Idéalement, l'utilisateur devrait pouvoir facilement comprendre comment utiliser un logiciel sans jamais avoir besoin de lire le mode d'emploi. Mais parfois, l'utilisateur a besoin d'un peu d'aide pour comprendre comment utiliser une fonction avancée ou la façon d'effectuer une variation de la tâche principale. Il est recommandé de fournir de l'aide qui décrit comment utiliser les éléments d'interface. Le texte d'aide peut apparaître lorsque l'utilisateur déplace le pointeur sur un élément de l'interface utilisateur pendant quelques secondes. En java, il est possible d'utiliser le ToolTip (la classe ToolTipManager), par exemple.

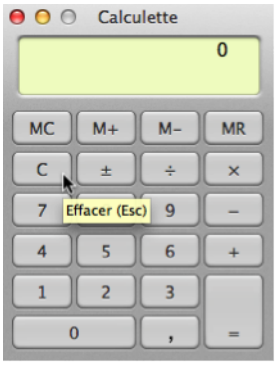


FIGURE 19 – Aider l'utilisateur

Exemples de recommandations pour quelques éléments d'interface utilisateur

Dans ce qui suit, nous présentons quelques exemples de recommandations d'ergonomie pour quelques éléments de l'interface utilisateur. Il existe des livres qui donnent des recommandations très détaillées.

Menus

Un menu doit avoir un titre qui indique clairement le type des éléments de la liste. Il faut choisir des titres de menu aussi courts que possible sans réduire la clarté du contenu du menu. Les éléments du menu peuvent avoir des raccourcis clavier et des liens vers des sous-menus. Il est recommandé d'éviter les menus trop longs. Il faut également réduire le nombre de niveaux de sous-menus (généralement, un seul niveau est largement suffisant).

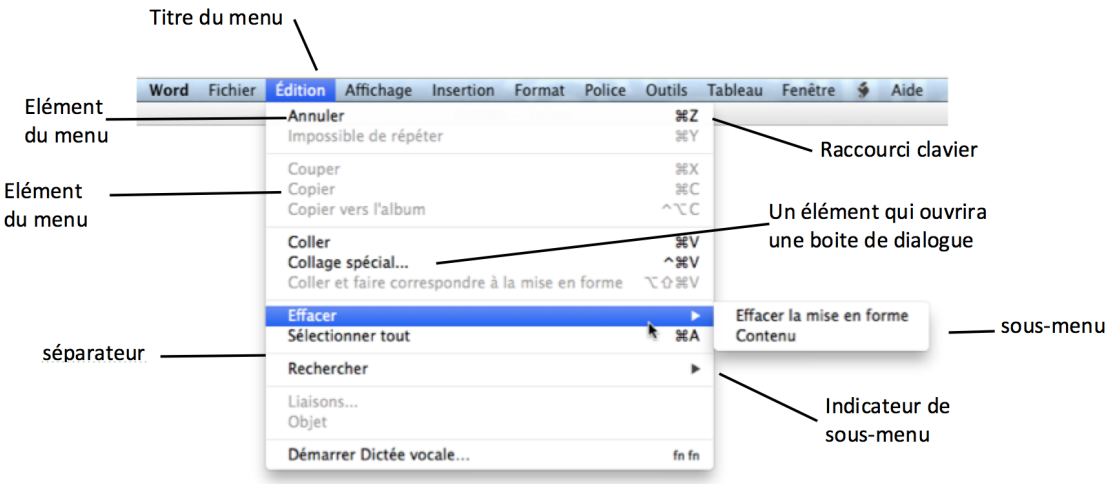


FIGURE 20 – Eléments d'un menu

Champs de saisie de texte

Un champ de saisie de texte accepte le texte saisi par l'utilisateur (en java, "JTextField"). L'utilisateur peut entrer du texte ou modifier un texte existant. Il faut effectuer des contrôles de

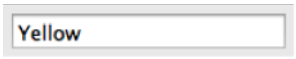


FIGURE 21 – Un champ de saisie de texte

validation appropriée après que l'utilisateur saisit son texte. Par exemple, si la seule valeur permise pour un champ est un nombre, le logiciel doit émettre une alerte si l'utilisateur tape des caractères autres que des chiffres. Il est conseillé également d'afficher un texte explicatif (avec "Jlabel", en java) avec le champ de saisie de texte. Ce texte aide l'utilisateur à comprendre quel type d'information il devrait entrer.

Indicateur de progression

Une barre de progression informe l'utilisateur sur l'état d'une longue opération. Une barre de progression est utilisée quand la longueur d'une opération peut être déterminée. Par exemple, une barre de progression peut montrer l'évolution d'une conversion de fichier. L'avancement de la barre de progression doit être associé avec précision avec le temps. Une barre de progression qui devient

90% complet en 5 secondes, mais prend 5 minutes pour les 10 pour cent restants, par exemple, serait inacceptable et conduire à penser que quelque chose ne va pas.