Chapitre 5

Programmation de l'interface graphique en Java

Nous avons vu, lors de l'initiation à l'informatique, les deux façons d'utiliser l'interpréteur de commandes : la méthode console, en écrivant des lignes de commandes, ou grâce à une interface graphique (la seule façon de faire pour MacOs jusqu'à MacOs X, grâce à Windows sur PC ou à X-Window avec Unix). Le moment est venu pour nous de programmer une telle interface graphique pour les programmes que nous concevons.

5.1 Étude générale

5.1.1 L'interface homme machine

Notion.- Jusqu'à maintenant, nous nous sommes surtout consacrés, en programmation, aux calculs que réalisait notre programme, sans trop nous préoccuper de la façon d'entrer les données et de sortir les résultats. L'**interface homme machine** (on utilise également l'acronyme **IHM**) s'intéresse à la façon d'entrer les données et de sortir les résultats.

Évolution de l'interface homme machine.- Depuis l'apparition des ordinateurs à la fin des années 1940, l'interface homme machine a évolué :

- Pour les tout premiers ordinateurs, il fallait entièrement *recâbler* la machine pour chaque programme, mais également pour entrer les données. Il s'agit visiblement du niveau zéro de l'interface homme machine.
- Une première évolution fut l'apparition de *panneaux d'interrupteurs* pour les entrées et de *panneaux de points lumineux* pour les sorties : on a, par exemple, un panneau d'un certain nombre de lignes de 32 interrupteurs ; on entre les données par mots de 32 bits ; on convertit, à la main, le mot en binaire et on manipule les interrupteurs pour l'entrer (disons avec la manette vers le haut pour 1 et vers le bas pour 0). Une petite sonnette avertit que l'on peut récupérer les résultats : ils sont affichés, également en binaire, mot par mot (par exemple avec lampe allumée pour 1 et éteinte pour 0, en espérant qu'aucune lampe n'est grillée).
  Cette interface homme machine apparaît si fantasmagorique pour le grand public qu'un film de James Bond la montre encore alors qu'elle avait disparue depuis longtemps. L'histoire n'étant qu'un éternel recommencement, le premier micro-ordinateur, l'Altaïr de 1977, utilisera cette interface.
- Lorque IBM s'intéressera aux ordinateurs, au milieu des années 1950, elle améliorera cette interface avec les moyens qu'elle connaît bien : les entrées se feront désormais avec des *bandes perforées* puis des *cartes perforées* ; les résultats peuvent être sortis sur les mêmes supports, remplacés très vite par des *imprimantes rapides*.
- Au début des années 1970 apparaissent les *terminaux* avec un clavier pour les entrées et un *écran* pour les résultats, ce qui exige également un support pour la conservation des données (disques durs et disquettes). Il s'agit d'un écran texte, généralement de 25 lignes de 80 colonnes.
  Ce genre de terminal sera utilisé pour le Minitel français.
- Au début des années 1980, le *projet Alto* du laboratoire de recherche de Xerox à Palo Alto va conduire à l'**interface graphique** telle que nous la connaissons : écran graphique haute définition, système de fenêtres, utilisation d'une souris.
  La société Apple sortira les premiers ordinateurs profitant de cette interface : d'abord le *Lisa* au succès mitigé puis le *Macintosh*, en 1984, au succès fulgurant. Sortiront ensuite pour les compatibles PC, d'abord *GEM* puis *Windows*, et X-Window pour Unix. L'histoire de l'origine de l'interface graphique est contée dans [LEV-94].
- Au début des années 1990, les ordinateurs à commandes vocales apparaissent mais ils ne sont encore utilisés que dans des situations particulières (amélioration des conditions de vie des handicapés, pilotes d'avion de chasse...).
- À la fin des années 1990, les PDA acceptent les commandes à écriture manuscrite, précédé des écrans tactiles.

L'interface homme machine la plus utilisée à l'heure actuelle (2010) est l'interface graphique.

5.1.2 Programmation de l'interface graphique

Prise en charge de l'interface graphique.- L'interface graphique peut être vue de trois façons :

- l'*aspect utilisateur* consistant à l'utiliser, tout simplement ;
- l'*aspect programmeur* consistant à programmer cette interface pour les programmes utilisateur, en utilisant une bibliothèque de fonctions ;
- l'*aspect concepteur* consistant à créer cette bibliothèque de fonctions.

Nous supposons ici que le lecteur domine l'aspect utilisateur depuis longtemps. Nous avons intérêt à connaître l'aspect programmeur avant d'aborder l'aspect concepteur, réservé à une élite restreinte. C'est cet aspect programmeur que nous allons aborder ici.

Outils de programmation.- Comme nous venons de le dire, la programmation de l'interface graphique se fait en utilisant une bibliothèque de fonctions utilisateur.

Prenons le cas de Windows. On peut programmer l'interface graphique en utilisant des outils plus ou moins élaborés :

- La *programmation brute* se fait en langage C en utilisant la bibliothèque dynamique de fonctions utilisateurs `win32.dll`. PETZOLD est l'auteur phare d'introduction à ce type de programmation, qui donne les résultats les plus performants.
- La programmation en langage orienté objet se fait en C++ avec les **MFC** (pour *Microsoft Foundations Classes*), qui se contentent d'encapsuler les fonctions de la bibliothèque précédente (et qui donne des programmes moins optimisés).
- Les compilateurs de Microsoft, Visual C++ (plutôt pour les personnes ayant reçu une éducation en programmation) et Visual Basic (plutôt pour les néophytes en programmation), possèdent des outils de mise en place (mais conduisent à des programmes moins optimisés).
- L'inconvénient de l'utilisation de ces derniers compilateurs se rencontre lors des projets : si certains programment une partie en Visual C++ et d'autres en Visual Basic, cette façon de faire conduit à un blocage. Microsoft a introduit la *technologie .NET* pour pallier, entre autre, à cet inconvénient : la compilation en C# (lire « si sharp » en anglais et « ces dièze » en français) ou en Visual Basic .NET conduit, non pas à un programme objet, mais à un programme en un code intermédiaire qui est le même quel que soit le compilateur choisi ; on peut donc regrouper des morceaux de programmes d'un même projet.

Dans le cas de X-Window, nous avons également plusieurs niveaux d'outillages de programmation :

- La programmation brute se fait en langage C avec la bibliothèque Xlib, quelquefois appelée le langage d'assemblage du graphisme.
- La programmation avec une boîte à outils plus élaborée se fait également en langage C avec la bibliothèque *Xt Intrinsics* (pour *X-window Toolkit* avec fonctions intrinsèques).
- La programmation avec Motif permet d'utiliser un certain nombre de **widgets** (pour *WIndowing gaDGET*). Cette bibliothèque est commercialisée mais une version libre existe (de nom *lesstif*).

Nous ne dirons rien de MacOS pour lequel on a des outils de programmation analogues.

Prototypage.- Développer un programme en tenant compte de l'interface graphique pour chacune des plate-formes est un projet lourd. Pour le tester on a intérêt à utiliser un **prototype**, conduisant à un utilitaire éventuellement moins performant sur une plate-forme donnée mais permettant de tester le programme sur toutes les plates-formes à la fois.

C'est là que réside tout l'intérêt de Java : on programme une seule fois l'interface graphique pour toutes les plates-formes. De plus on apprend les concepts de la programmation de l'interface graphique : en effet la façon de programmer avec la bibliothèque adéquate pour une plate-forme est très proche de celle pour une une autre, puisque toutes héritent du projet Alto. L'inconvénient est que le programme objet est lent (principalement parcequ'il est interprété et non compilé), mais on peut le voir comme une première étape avant le programme brut pour la plate-forme voulue. N'oublions pas également que Java est réellement utilisé sur Internet et donc que la programmation Java possède un intérêt intrinsèque.

5.1.3 Le cas de Java

Il existe deux paquetages de programmation de l'interface graphique pour Java :

- Les concepteurs de Java avaient conçu, dès le kit JDK 1.0, une boîte à outils dénommées **Abstract Windowing Toolkit** (c'est-à-dire *boîte à outils de fenêtrage abstrait*, les fenêtres apparaissant comme la primitive principale, abstrait puisqu'il s'agit d'une surcouche), se trouvant dans le paquetage `java.awt`, s'appuyant très fortement sur les boîtes à outils des interfaces des plates-formes sur lesquelles les programmes sont exécutés en définitive. Le compilateur donne le même code intermédiaire quelle sue soit le plate-forme ; l'interpréteur traduit en code du système graphique de la plate-forme.
- Puisque `awt` utilise la boîte à outils de la plate-forme sur laquelle on exécute le programme, l'aspect peut être assez différent d'une plate-forme à l'autre (y compris, par exemple, pour la césure d'une ligne). Les concepteurs de Java ont donc conçu un autre paquetage, appelé `swing`, moins dépendant de la plate-forme : on utilise une fenêtre totalement blanche de la plate-forme sur laquelle on se trouve, sur laquelle on peint entièrement l'aspect que l'on désire. Ajouté pour le kit JDK 1.1.18, les kits depuis JDK 1.2 comprennent le paquetage `javax.swing`.

Ces deux bibliothèques font désormais parti des **JFC** (pour *Java Foundation Classes*).

5.2 Graphisme de base

Nous appellerons **graphisme de base** la mise en place d'interfaces graphiques ne faisant pas intervenir *a priori* la gestion des événements (clic de souris...). Il s'agit essentiellement de la mise en place d'une fenêtre et de son remplissage statique.

Comme nous l'avons vu lors de notre étude de l'utilisation d'une interface graphique, celle-ci est une collection de **fenêtres**, ou plus exactement une hiérarchie de fenêtres. L'écran lui-même est une fenêtre qui est mère de toutes les autres. On ne peut pas accéder à la fenêtre de base qu'est l'écran en Java ; on doit nécessairement définir une fenêtre fille de l'écran, appelée **cadre** (car cette fenêtre a un encadrement).

5.2.1 Les cadres

5.2.1.1 Définition

On appelle **cadre** (en anglais *frame*) une fenêtre qui n'est pas contenue dans une autre fenêtre (autre que la fenêtre mère qu'est l'écran en son entier).

Traditionnelement une telle fenêtre est représentée avec des **bords** et un **bandeau de titre**. C'est la fenêtre de base pour l'utilisateur, mais elle est déjà complexe pour le concepteur de l'interface, car en fait constituée de plusieurs fenêtres pour les bords et pour le bandeau.

5.2.1.2 Mise en place

Commençons par un exemple que nous expliquerons ensuite.

Exemple.- Écrivons une application Java faisant apparaître un cadre.

```
import java.awt.*;

class PremierCadre extends Frame
   {
   public PremierCadre()
      {
      setTitle("Premier cadre");
      setSize(300, 200);
      }
   }

public class Cadre
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new PremierCadre();
      frame.show();
      }
   }
```

Exécution.- Lorsqu'on fait exécuter ce programme, on voit apparaître (au bout d'un certain temps) une fenêtre positionnée en haut à gauche. Son aspect est celui de la plate-forme sur laquelle l'application est exécutée. Sa taille est de 300 pixels horizontalement sur 200 verticalement. Le

FIG. 5.1 – exemple de cadre

titre de la fenêtre est 'Premier cadre'. On peut utiliser presque toutes les actions habituelles sur un cadre (déplacer, agrandir...), mais on remarquera, par contre, qu'essayer de fermer la fenêtre n'a pas d'action.

On utilisera CTRL-C pour arrêter l'application. On peut également tuer le processus.

Commentaires.- 1^o) Un cadre est un objet de la classe `Frame` du paquetage `java.awt` ou, plus exactement d'une classe dérivée de celle-ci (en effet par défaut la taille est de 0 pixel sur 0 pixel et donc le cadre serait invisible). Il faut donc commencer par définir une classe dérivée.

2^o) La classe `Frame` fait partie de la boîte à outils `awt`. Il faut donc importer ce paquetage.

3^o) La classe dérivée doit contenir au minimum un constructeur par défaut indiquant la taille de la fenêtre (si on veut quelque chose de visible). On utilise pour cela la méthode `setSize()`.

4^o) Nous avons ajouter en plus un titre à la fenêtre, en utilisant la méthode `setTitle()`.

5^o) Pour montrer le cadre, il faut instancier un objet de la classe dérivée et utiliser la méthode `show()` de la classe `Frame`.

6^o) Par défaut la fenêtre est positionnée en haut à gauche de l'écran.

7^o) La méthode `show()` est maintenant désavouée. On utilisera donc la méthode `setVisible(boolean)` avec la constante `true`.

5.2.2 Actions sur une fenêtre : fermeture

Nous avons vu que, par défaut, notre cadre ne se ferme pas de la façon habituelle. Pour que le cadre puisse être fermé de façon habituelle, il faut dire ce qui doit se passer lorsqu'on clique sur le bouton adéquat ou sur la ligne adéquate du menu déroulant. Nous devons donc traiter un événement, bien que nous avons dit au début que ce n'est pas ce qui nous intéresse pour l'instant.

5.2.2.1 Mise en place

Principe.- Pour pouvoir réaliser une action sur une fenêtre, il faut :
- définir un objet *adaptation de fenêtre* ;
- lier cet objet à la fenêtre.

La classe d'adaptation des fenêtres.- La classe `WindowAdapter` contient sept méthodes permettant d'adapter le comportement d'une fenêtre :

```
public void windowActivated(WindowEvent e)
public void windowDeactivated(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowClosing(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowOpened(WindowEvent e)
```

dont la signification est (presque) claire : que doit-il se passer lorsque la fenêtre est activée, désactivée, fermée, lorsqu'on appuie sur l'icône de fermeture (le X tout à fait à droite du bandeau de titre), lorsqu'on demande à l'iconifier (le premier bouton à droite du bandeau de titre), lorsqu'on clique sur l'icône ainsi obtenue et, enfin, au moment de l'ouverture.

Le corps de ces méthodes est vide, comme d'habitude. On doit donc surcharger les méthodes qui nous intéressent, en définissant une classe dérivée.

Lien avec la fenêtre.- On utilise la méthode :

```
void addWindowListener(WindowAdapter) ;
```

pour lier une instantiation d'une classe dérivée de `WindowAdapter` à une fenêtre.

5.2.2.2 Exemple : cadre fermant

Programme.- Écrivons une application Java faisant apparaître un cadre pouvant se fermer de façon habituelle.

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

class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }
```

```
class CadreFermant extends Frame
   {
   public CadreFermant()
      {
      setTitle("Cadre fermant");
      setSize(300, 200);
      addWindowListener(new Fermeture());
      }
   }

public class cadref
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadreFermant();
      frame.show();
      }
   }
```

Commentaires.- 1^o) Nous devons importer le paquetage `java.awt.event` contenant la classe `WindowEvent`. On remarquera au passage, qu'en Java, importer un paquetage n'importe pas les paquetages qui sont des sous-répertoires de celui-ci.

2^o) La méthode `exit(int)` de la classe `System` a pour effet de terminer le programme que l'on est en train d'exécuter et de sortir de l'interpréteur Java. Il est convenu de renvoyer 0 lorsqu'on sort normalement.

3^o) Pour implémenter l'interface, nous utilisons ici un raccourci :

```
addWindowListener(new Fermeture());
```

au lieu de, par exemple :

```
Fermeture ferme;
ferme = new Fermeture();
addWindowListener(ferme);
```

5.2.3 Affichage d'un texte

Nous allons voir comment remplir notre cadre et, dans une première étape, comment afficher un texte dans ce cadre.

5.2.3.1 Utilisation d'un composant

Composant.- Pour remplir le cadre, il faut utiliser un composant de la classe `Component` du paquetage `java.awt`, ou plutôt, comme d'habitude un objet d'une classe dérivée.

On place un tel objet dans un objet de la classe `Frame` en utilisant la méthode :

```
void add(Component);
```

de la classe `Frame` (en fait d'une super-classe de celle-ci, mais cela n'a pas d'importance pour l'instant).

Méthode fondamentale.- La méthode fondamentale de la classe `Component` est la méthode :

```
public void paint(Graphics);
```

que nous avons déjà rencontrée à propos des applets. Il faut donc surcharger cette méthode pour obtenir quelque chose.

Affichage d'un texte.- Nous avons également vu, à propos des applets, comment afficher un texte en utilisant la méthode :

```
void drawString(String, int, int) ;
```

de la classe `Graphics`.

5.2.3.2 Exemple

Écrivons une application Java permettant d'afficher un cadre (fermant) dans lequel est écrit 'Bonjour'.

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

class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }
class Message extends Component
   {
   public void paint(Graphics g)
      {
      g.drawString("Bonjour", 75, 100);
      }
   }
class CadreTexte extends Frame
   {
   public CadreTexte()
      {
      setTitle("Cadre avec texte");
      setSize(300, 200);
      addWindowListener(new Fermeture());
      add(new Message());
      }
   }
public class cadret
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadreTexte();
      frame.show();
      }
   }
```

5.2.4 Affichage d'un dessin

5.2.4.1 Dessin et image

Faisons d'abord une distinction. En informatique, un **dessin** se construit en temps réel alors qu'une **image** est prête à être affichée.

On construit un dessin en utilisant quelques **primitives de dessin** dont, bien entendu, l'affichage d'un pixel (coloré) qui est la seule primitive nécessaire. Bien entendu, en général on utilise d'autres primitives telles que le tracé d'un segment, d'une ligne brisée, d'un polygone, d'un arc de cercle, d'un rectangle à bords arrondis, etc.

5.2.4.2 Primitives de dessin

Nous allons donner ci-dessous quelques primitives de dessin implémentées comme méthodes de la classe `Graphics`, sans être exhaustif bien sûr.

Repère considéré.- En Java, le repère a pour origine le coin en haut à gauche de la fenêtre (on dit aussi nord-ouest), l'axe des abscisses étant horizontal et orienté de la gauche vers la droite, l'axe des ordonnées étant vertical et orienté de haut en bas (contrairement à ce qui se passe habituellement en mathématiques mais ce qui est courant en physique). L'unité est le pixel, les coordonées étant donc exprimées par des entiers.

Segment de droite.- Pour tracer un segment de droite on utilise la méthode :

```
void drawLine(int x1, int y1, int x2, int y2)
```

de la classe `Graphics`, reliant les points de coordonnées $(x1, y1)$ et $(x2, y2)$.

Ligne brisée et polygone.- Commençons par faire une distinction. En informatique, une **ligne brisée** (en anglais *polyline*) est déterminée par une suite de points appelés **sommets** ; elle est affichée comme une suite de segments, celui reliant le premier point au deuxième, celui reliant le deuxième point au troisième et ainsi de suite jusqu'à celui reliant l'avant-dernier point au dernier.

Un **polygone** est également déterminé par une suite de sommets mais il est affiché différemment : comme la ligne brisée associée plus le segment reliant le dernier point au premier, ce qui en fait une ligne brisée « fermée ».

Polygone.- Il existe une classe `Polygon` du paquetage `awt` ayant pour méthodes le constructeur par défaut et `addPoint()`. La méthode :

```
void addPoint(int, int) ;
```

permet d'ajouter un point au polygone, les points étant insérés dans l'ordre séquentiel d'insertion. Une fois le polygone défini, on peut l'afficher grâce à la méthode :

```
void drawPolygon(Polygon) ;
```

de la classe `Graphics`.

Affichage direct d'un polygone.- On peut également afficher directement un polygone, sans passer par la classe `Polygon`, grâce à la méthode (surchargée) :

```
void drawPolygon(int X[], int Y[], int n) ;
```

de la classe `Graphics`, où `X` et `Y` sont des tableaux de `n` entiers représentant respectivement les abscisses et les ordonnées de `n` points.

Ligne brisée.- Il n'existe pas de classe correspondante. Pour afficher une ligne briséee, on utilise la méthode :

```
void drawPolyline(int X[], int Y[], int n);
```

de la classe `Graphics` analogue à celle pour les polygones.

5.2.4.3 Premier exemple

L'application Java suivante fait apparaître un texte et un triangle.

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

class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }
class Poly extends Component
   {
   public void paint(Graphics g)
      {
      Polygon p = new Polygon();
      g.drawString("Exemple de polygone", 15, 15);
      p.addPoint(30, 20);
      p.addPoint(30, 150);
      p.addPoint(100, 50);
      g.drawPolygon(p);
      }
   }
class CadreDessin extends Frame
   {
   public CadreDessin()
      {
      setTitle("Cadre avec dessin");
      setSize(300, 200);
      addWindowListener(new Fermeture());
      add(new Poly());
      }
   }
public class dessin
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadreDessin();
      frame.show();
      }
   }
```

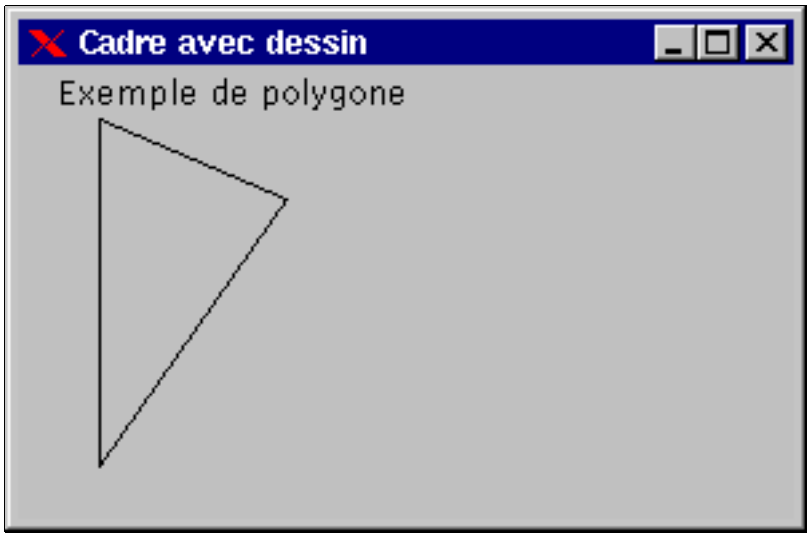

FIG. 5.2 – Cadre avec dessin

5.2.4.4 Deuxième exemple

Pour construire la ligne brisée correspondant à notre polygone, nous utiliserons l'application Java suivante :

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

class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }
class Poly2 extends Component
   {
   public void paint(Graphics g)
      {
      int X[] = {30, 30, 100};
      int Y[] = {20, 150, 50};
      g.drawString("Exemple de ligne brisee", 15, 15);
      g.drawPolyline(X, Y, 3);
      }
   }
class CadreDessin extends Frame
   {
   public CadreDessin()
      {
      setTitle("Cadre avec dessin");
```

```
      setSize(300, 200);
      addWindowListener(new Fermeture());
      add(new Poly2());
      }
   }
public class dessin2
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadreDessin();
      frame.show();
      }
   }
```

5.2.5 Incrustation d'image

5.2.5.1 Principe

Introduction.- La différence entre une `image` et un dessin est qu'une image est un tableau bidimensionnel de points colorés. Le point principal est qu'elle est entreposée et qu'on ne la reconstruit pas. Qu'importe comment on l'a obtenue : dessin, grâce à un scanner, à un appareil photo numérique...

Une image est *a priori* un tableau de caractéristiques des pixels à afficher. En fait, on a besoin également d'un **en-tête** indiquant au minimum le nombre de lignes et de colonnes. De plus les données (le tableau) sont la plus souvent compressées pour diminuer la taille du fichier. L'en-tête doit donc indiquer également la méthode de compression et les paramètres de compression.

Les images peuvent se présenter comme un tableau brut mais elles prennent alors beaucoup de place mémoire. Elles sont le plus souvent compressées dans un **format** donné. Il existe de nombreux formats mais Java n'en reconnaît que deux : **GIF** et **JPEG**.

Images.- Les images sont des objets de la classe `Image` du paquetage `java.awt` sur laquelle il n'y a pas grand chose à dire.

Récupération d'une image.- Pour récupérer une image, on se sert de la classe `Toolkit`.

- 1^o) La classe `Toolkit` interagit entre Java et le système pour obtenir de l'information sur le système.

- 2^o) C'est une classe abstraite, on ne peut donc pas instantier d'objet de cette classe.

- 3^o) La méthode statique :

```
Toolkit getDefaultToolkit()
```

de cette classe permet d'obtenir l'objet `Toolkit` par défaut du système.

- 4^o) La méthode :

```
Image getImage(String)
```

de la classe `Toolkit` permet d'obtenir une image. Cette image est désignée par le nom physique de son fichier : le nom court si elle se trouve dans le même répertoire que l'endroit où on exécute l'application, le nom complet, avec son chemin, si ce n'est pas le cas.

Affichage d'une image.- Pour afficher une image, on se sert de la méthode :

```
void drawImage(Image, int x, int y, this)
```

de la classe `Graphics`, où x et y sont les coordonnées, en pixels, du coin en haut à gauche de l'image.

5.2.5.2 Exemple d'application

Écrivons une application Java permettant d'afficher une image se trouvant dans le même répertoire que l'application.

FIG. 5.3 – Affichage d'une image

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

class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }
class Img extends Component
   {
   public void paint(Graphics g)
      {
      Toolkit tk;
      Image img = null;
      g.drawString("Exemple d'image", 15, 15);
      tk = Toolkit.getDefaultToolkit();
```

```
      img = tk.getImage("PCegielski.gif");
      g.drawImage(img, 40, 50, this);
      }
   }
class CadreImage extends Frame
   {
   public CadreImage()
      {
      setTitle("Cadre avec image");
      setSize(250,300);
      addWindowListener(new Fermeture());
      add(new Img());
      }
   }
public class image
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadreImage();
      frame.show();
      }
   }
```

5.2.5.3 Exemple d'applet

On peut adapter l'application précédente pour la transformer en une applet.

```
import java.awt.*;
import java.applet.Applet;

public class image2 extends Applet
   {
   public void init()
      {
      resize(250, 300);
      }
   public void paint(Graphics g)
      {
      Toolkit tk;
      Image img;
      g.drawString("Exemple d'image", 15, 15);
      tk = Toolkit.getDefaultToolkit();
      img = tk.getImage("PCegielski.gif");
      g.drawImage(img, 40, 50, this);
      }
   }
```

5.2.5.4 Récupération d'une image sur internet

Principe.- La classe `Applet` possède la méthode :

```
Image getImage(URL, String)
```

permettant de récupérer une image sur internet. Elle a deux arguments : un URL et le nom du fichier graphique.

URL par défaut.- La méthode `getDocumentBase()` renvoie la base URL dans laquelle l'applet réside.

Exemple.- Voici une variation de l'exemple précédent :

```
import java.awt.*;
import java.applet.Applet;

public class image3 extends Applet
   {
   public void init()
      {
      resize(250, 300);
      }
   public void paint(Graphics g)
      {
      Image img;
      g.drawString("Exemple d'image", 15, 15);
      img = getImage(getDocumentBase(), "PCegielski.gif");
      g.drawImage(img, 40, 50, this);
      }
   }
```

5.3 Programmation événementielle

Nous venons de voir une première partie pour mettre en place une interface graphique, à savoir du graphisme statique. Dans une seconde étape nous allons voir comment interagir avec l'interface, par exemple placer un bouton et effectuer une action lorsqu'on clique dessus avec la souris. On parle alors de **programmation événementielle** puisque le déroulement d'une session ne dépend pas que du programme : des **événements** son attendus et il y a des réactions suivant la nature de ceux-ci.

Nous allons illustrer la programmation événementielle par un exemple, simple du point de vue de l'utilisateur : afficher deux boutons portant des noms (par exemple 'bouton 1' et 'bouton 2') et indiquer sur quel bouton on clique. Cet exemple est simple du point de vue de l'utilisateur, mais il est quand même suffisamment difficile, du point de vue de la programmation, pour que nous le mettions en place petit à petit seulement.

5.3.1 Affichage d'un bouton

5.3.1.1 Définition d'un bouton

Un **bouton** apparaît, comme son nom l'indique, sous la forme d'un bouton, au graphisme plus ou moins réaliste.

figure

Lorsqu'on appuie dessus (c'est-à-dire, en fait, lorsqu'on place le curseur de la souris sur la partie active du bouton et que l'on clique), il déclenche une action.

Dans une première étape, ne nous occupons pas de la programmation événementielle (et donc du clic), contentons-nous d'afficher un bouton.

5.3.1.2 Mise en place d'un bouton

Déclaration d'un bouton.- Un bouton est un objet de la classe `Button`, du paquetage `java.awt`, classe dérivée de la classe `Component`. On déclarera donc un bouton sous la forme :

```
Button b ;
```

Instantiation d'un bouton.- Un bouton sera instantié, comme n'importe quel objet, de la façon habituelle avec `new`. Le constructeur le plus intéressant est certainement celui qui permet d'afficher un texte dans ce bouton, le nom du bouton pour l'utilisateur :

```
b = new Button("bouton") ;
```

Affichage d'un bouton.- On fait afficher le bouton en l'ajoutant au cadre, en utilisant la méthode déjà vue :

```
void add(Component) ;
```

on aura donc :

```
add(b) ;
```

le bouton apparaît alors dans la fenêtre avec comme nom le nom qui lui a été attribué.

5.3.1.3 Exemple d'application

Le programme.- Écrivons une application faisant apparaître un bouton, de nom 'bouton'.

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

class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }

class CadreBouton extends Frame
   {
   public CadreBouton()
      {
      setTitle("Cadre avec bouton");
      setSize(300, 200);
      addWindowListener(new Fermeture());
      Button b;
      b = new Button("Bouton");
      add(b);
      }
   }

public class cadreb
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadreBouton();
      frame.show();
      }
   }
```

Exécution.- Nous obtenons un gros bouton occupant toute la fenêtre (même si nous agrandissons celle-ci), là où nous attendions un petit bouton pour pouvoir mettre autre chose. Du coup, nous ne pouvons rien faire afficher d'autre.

Nous pouvons cliquer sur le bouton : il change d'aspect (on a l'impression qu'il s'enfonce et qu'il revient à sa position initiale) mais aucune autre action n'est exécutée. C'est normal puisque nous ne lui avons pas associé d'action.

5.3.1.4 Exemple d'une applet

Le programme.- Écrivons une applet qui, *a priori*, fait la même chose.

```
import java.applet.Applet;
import java.awt.*;

public class bouton2 extends Applet
   {
   Button b;

   public void init()
      {
      b = new Button("Bouton");
      add(b);
      }

   public void paint(Graphics g)
      {
      g.drawString("Bonjour", 10, 50);
      }
   }
```

Exécution.- Cette fois-ci nous avons bien un petit bouton d'affiché ainsi qu'un texte.

5.3.2 Les panneaux

5.3.2.1 Notion

Dans l'application précédente, nous n'avons vu qu'un seul (gros) bouton. Ceci est dû au fait qu'un cadre ne peut afficher qu'un seul **composant** à la fois. Pour insérer plusieurs composants, nous avons besoin de la notion de composants acceptant eux-mêmes des composants. On les appelle, en Java, des **panneaux** (d'affichage).

5.3.2.2 Mise en place

En Java on utilise la classe `Panel` qui est une classe dérivée de la classe `Component`. On peut donc utiliser la méthode `paint()` mais aussi lui ajouter autant de composants que l'on veut.

5.3.2.3 Cas des applets

La classe `Applet` est une classe dérivée de la classe `Panel`, ce qui explique pourquoi nous pouvions y placer sans problème un bouton lors de la section précédente.

5.3.2.4 Exemple d'application

Le programme.- Écrivons une application Java permettant d'afficher un (petit) bouton ainsi qu'un texte.

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

```
class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }

class Panneau extends Panel
   {
   Button b;

   public Panneau()
      {
      b = new Button("bouton");
      add(b);
      }

   public void paint(Graphics g)
      {
      g.drawString("Exemple de bouton", 15, 55);
      }
   }

class CadrePanneau extends Frame
   {
   public CadrePanneau()
      {
      setTitle("Essai de panneau");
      setSize(300, 200);
      addWindowListener(new Fermeture());
      add(new Panneau());
      }
   }

public class panneau1
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadrePanneau();
      frame.show();
      }
   }
```

Exécution.- Cette fois-ci le bouton et le message apparaissent. Remarquons cependant que, pour l'instant, nous ne dominons ni la taille ni l'emplacement du bouton.

5.3.3 Traitement des événements

Le traitement des événements est différent pour le JDK 1.0 et le JDK 1.1 (et les suivants). Nous avons déjà vu un exemple avec le JDK 1.0, grâce à la méthode `action()`. La mise en place en JDK 1.1 (et suivants) est plus complexe *a priori*, surtout pour les cas simples, mais elle permet de traiter des cas plus sophistiqués.

Nous ne nous occuperons donc ici que de la nouvelle manière de faire depuis le JDK 1.1.

5.3.3.1 Philosophie générale

Événement et source d'événements.- Dans cette nouvelle façon de faire on distingue les **événements**, par exemple les clics de souris (sur le bouton gauche), et la **source d'événements**, par exemple tel bouton.

Les événements sont représentés par des objets de la classe `EventObject` du paquetage `java.awt.event`. Une des classes dérivées est `ActionEvent`, comprenant les clics de souris et l'appui des touches du clavier.

Prise en compte des événements.- Pour prendre en compte un événement, il faut effectuer les trois étapes suivantes :

- **Se mettre à l'écoute de l'événement**.- Contrairement à la philosophie du JDK 1.0, si on veut qu'une classe tienne compte d'événements, c'est-à-dire si on veut que le comportement de certaines de ses méthodes dépendent de l'apparition d'un ou de plusieurs types d'événements, elle doit être explicitement conçue de façon à être **à l'écoute des événements** dont on veut qu'elle tienne compte.
  Pour ce faire cette classe doit *implémenter* une **interface d'écoute**.
  Par exemple si l'une des méthodes de la classe dépend du fait que l'on a appuyé ou non sur un bouton à l'aide de la souris (événement du type `action`), l'en-tête de la classe devra être suivi de :

  ```
  implements ActionListener
  ```

- **Rendre active une source d'événements**.- De même, il faut indiquer explicitement que l'on désire qu'une source (potentielle) d'événements soit active, c'est-à-dire qu'elle soit à l'écoute d'un événement.
  Ceci se fait grâce à une ligne de code du modèle suivant :

  ```
  ObjetSourceEvenement.addEvenementListener(ObjetEcouteEvenement) ;
  ```

  par exemple :

  ```
  bouton.addActionListener(this) ;
  ```

  pour que le bouton de la classe que nous sommes en train de définir soit à l'écoute des clics de souris.

- **Implémenter l'interface**.- Il faut enfin dire ce qu'il faut faire lorsqu'un tel événement est survenu. Pour cela on surcharge l'une des méthodes de l'interface.
  Par exemple, dans le cas d'un bouton, il faut surcharger la méthode :

  ```
  public void actionPerformed(ActionEvent evt) ;
  ```

5.3.3.2 Exemple d'activation d'un bouton

Problème.- Écrivons une application Java affichant un bouton ainsi que le nombre de fois que l'on a cliqué dessus.

Principe.- Nous venons de voir que, pour les boutons, les événements sont des objets de la classe `ActionEvent`, les sources d'événements des objets de la classe `Button`, l'interface d'écoute `ActionListener` avec la méthode `actionPerformed()` et, enfin, que la détermination de la source d'événement s'effectue grâce à la méthode `addActionListener()`. Ceci est suffisant pour écrire notre programme.

Un programme.- Ceci conduit au programme suivant :

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

class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }

class Panneau extends Panel implements ActionListener
   {
   Button b;
   String s;
   int count;
   public Panneau()
      {
      s = "";
      count = 0;
      b = new Button("Bouton");
      add(b);
      b.addActionListener(this);
      }
   public void paint(Graphics g)
      {
      g.drawString(s, 10, 55);
      }
   public void actionPerformed(ActionEvent evt)
      {
      count++;
      s = "le bouton a ete appuye " + count + " fois";
      repaint();
      }
   }

class CadreBouton extends Frame
   {
   public CadreBouton()
      {
      setTitle("Essai de bouton actif");
      setSize(300, 200);
```

```
      addWindowListener(new Fermeture());
      add(new Panneau());
      }
   }

public class bouton3
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadreBouton();
      frame.show();
      }
   }
```

5.3.4 Choix de la source d'événement

Jusqu'à maintenant nous ne nous sommes pas vraiment préoccupé de la source d'événement de l'événement capturé, puisque celle-ci était unique. Ceci change si on a plusieurs sources possibles d'événements, par exemple deux boutons.

5.3.4.1 Principe

Si on introduit plusieurs boutons, il faut en général déterminer celui sur lequel on a appuyé. On utilise, pour cela, la méthode :

`Object getSource()`

de la classe `EventObject` qui nous donne la source de l'événement. On compare ensuite le résultat aux identificateurs des boutons.

5.3.4.2 Exemple d'utilisation de plusieurs boutons

Problème.- Écrivons une application Java faisant apparaître un cadre avec deux boutons et indiquant le nombre de fois que l'on a cliqué sur chacun d'eux.

Un programme.- Ceci nous conduit au programme suivant :

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

class Fermeture extends WindowAdapter
   {
   public void windowClosing(WindowEvent e)
      {
      System.exit(0);
      }
   }

class Panneau extends Panel implements ActionListener
   {
   Button b1, b2;
```

```
   String s1, s2;
   int count1, count2;
   public Panneau()
      {
      s1 = "";
      s2 = "";
      count1 = 0;
      count2 = 0;
      b1 = new Button("Rouge");
      add(b1);
      b1.addActionListener(this);
      b2 = new Button("Bleu");
      add(b2);
      b2.addActionListener(this);
      }
   public void paint(Graphics g)
      {
      g.drawString(s1, 10, 55);
      g.drawString(s2, 10, 75);
      }
   public void actionPerformed(ActionEvent evt)
      {
      Object source;
      source = evt.getSource();
      if (source == b1) count1++;
      if (source == b2) count2++;
      s1 = "Le bouton rouge a ete appuye " + count1 + " fois";
      s2 = "le bouton bleu " + count2 + " fois";
      repaint();
      }
   }
class CadreBouton extends Frame
   {
   public CadreBouton()
      {
      setTitle("Essai de deux boutons actifs");
      setSize(300, 200);
      addWindowListener(new Fermeture());
      add(new Panneau());
      }
   }
public class bouton4 \\
   {
   public static void main(String args[])
      {
      Frame frame;
      frame = new CadreBouton();
      frame.show();
      }
   }
```

5.3.4.3 Cas d'une applet

Principe général.- Jusqu'à nouvel ordre on a intérêt, dans le cas d'une applet, à utiliser l'ancienne méthode de programmation événementielle, celle du JDK 1.0, pour être sûr d'être visible sur tous les navigateurs.

Exemple.- L'applet qui réalise la même chose que notre programme sera donc écrite de la façon suivante :

```
import java.applet.Applet;
import java.awt.*;

public class bouton extends Applet
   {
   Button b1, b2;
   String s;

   public void init()
      {
      b1 = new Button("Bouton1");
      add(b1);
      b2 = new Button("Bouton 2");
      add(b2);
      s = new String("");
      }

   public boolean action(Event e, Object o)
      {
      if ("Bouton1".equals(o))
      s = "Vous avez appuye sur le bouton 1";
      if \= ("Bouton2".equals(o))
      s = "Vous avez appuye sur le bouton 2";
      repaint();
      return true;
      }

   public void paint(Graphics g)
      {
      g.drawString(s, 10, 50);
      }
   }
```