CM 6 (BIS)

INTERFACES GRAPHIQUES EN SWING

PROGRAMMATION EVENEMENTIELLE VIA LE PATTERN OBSERVER

UNIVERSITÉ CAEN NORMANDIE

1. Présentation

Les interfaces graphiques en Java s'appuyaient originellement sur l'AWT (Abstract Window Toolkit). Cette bibliothèque graphique s'appuie sur les différents composants de chaque système d'exploitation, et en propose une approche suffisamment abstraite, générique, pour fonctionner de façon unique sur tout système. L'avantage est la rapidité d'exécution, puisque le code exploite (presque) directement des éléments natifs de chacun des systèmes. Mais il a un inconvénient majeur, celui du nivellement par le bas : pour être compatible avec toutes les plate-formes, on ne peut conserver que la partie commune à toutes ces dernières. D'où une bibliothèque finalement pauvre en termes de fonctionnalités. Un autre inconvénient est le fait que l'aspect graphique est directement lié au système : une même application Java aura un rendu différent sous tel ou tel système… Ce qui est contraire aux objectifs de ce langage.

Swing est apparu avec la version 1.2 de Java, et propose une couche au-dessus de l'AWT. Il prend le parti de redéfinir lui-même ses différents composants, en "dessinant" lui-même ces derniers. Il est ainsi possible de définir toute sorte d'objet graphique indépendamment de la plate-forme, et avec un rendu identique sur toutes ces dernières. La contre partie est un coût non négligeable en temps d'exécution.

Etant plaçé "au-dessus" de l'AWT, ne soyez pas étonnés de trouver des imports de package AWT lorsque vous faites du Swing.
Les packages de base pour les applications que nous allons voir sont :
javax.swing.*
java.awt.*
java.awt.event.*

2. Deux niveaux d'entités

Swing distingue deux types d'entités dans les interfaces graphiques. Des contenants et des contenus, que nous appellerons composants.

1. Les composants (contenus)

Ce sont en quelque sorte les briques élémentaires des interfaces graphiques, tels les différents boutons, boîtes de dialogue, etc.

- JButton (bouton)
- JLabel (étiquette)
- JList (liste)
- JTextField (champ de saisie)

2. Les containers (contenants)

Il permettent de contenir et d'organiser les uns par rapport aux autres les composants. En effet, un composant ne peut être utilisé que dans un container. Il est par exemple impossible de créer une interface graphique à partir d'un bouton seul. Il faut mettre ce dernier dans un container
On distingue deux niveux de containers :

Top-level containers (containers de haut niveau)

- JWindow (fenêtre simple, sans titre ni menu)
- JFrame (fenêtre classique, la plus utilisée)
- JDialog (boîte de dialogue éventuellement modale)
- JApplet (panneau swing intégré au sein d'un navigateur web)

Toute interface graphique possède au moins un container de haut niveau (éventuellement plusieurs), à partir duquel on va pouvoir bâtir le reste.

Intermediate containers (containers de niveau intermédiaire)

- JPanel (panneau simple)
- JScrollPane (panneau avec barres de défilement)
- JTabbedPane (panneau avec onglets)
- Etc…

Ces derniers containers sont eux-mêmes des composants, et tiennent ainsi un double rôle : ils peuvent contenir d'autres éléments (on peut mettre plusieurs boutons dans un JPanel par exemple), et peuvent par ailleurs, en tant que composants, être intégrés dans donc containers. Ils sont donc les piliers d'une construction récursive des interfaces graphiques.

3. Le top-level container JFrame

Etant le top-level container le plus populaires, et les autres fonctionnant de façon assez similaire, nous détaillons le fonctionnement de JFrame.

Ce frame comprend différentes couches, permettant notamment d'intégrer un ContentPane, i.e. le panneau conteneur, et un MenuBar (menus déroulants).
L'accès au ContentPane se fait via les accesseurs associés :
`Container getContentPane()`
`void setContentPane(Container c)`

De même pour le menu déroulant :

`JMenuBar getJMenuBar()`
`void setJMenuBar(JMenuBar m)`

Constructeurs

JFrame() : permet de construire une fenêtre sans titre
JFrame(String t) : permet de construire une fenêtre avec titre

Ajout de composants dans un JFrame :

On ajoute des composants au ContentPane du JFrame, jamais directement. Erreur classique…
Le ContentPane par défaut est un conteneur de niveau intermédiaire, héritant de Container, et géré par défaut par un BorderLayout manager (voir plus loin).

Il y a deux possibilités créer le contenu du JFrame. Soit on ajoute des composants à son ContentPane. Soit on change son ContentPane (via le mutateur setContentPane) avec un ContentPane que l'on a créé par ailleurs.

Affichage

`pack()` demande à la fenêtre et a ses composants, de façon éventuellement récursive, de s'ajuster
`setVisible(boolean v)` permet d'afficher ou de cacher la fenêtre. Attention, la valeur par défaut est false !

4. Les gestionnaires de mise en page (LayoutManagers)

Principe

Les LayoutManager permettent d'agencer les composants les uns par rapport au autres au sein d'un container.
Tout container utilise donc une instance de LayoutManager (il en existe de différentes sortes). Un cas particulier est celui où l'on fixe null comme référence, auquel cas le positionnement des composants se fait de façon absolue, au pixel près.
L'attribution d'un LayoutManager se fait via la méthode :
`void setLayout(LayoutManager l)`

Par ailleurs, n'oublions pas que grâce aux conteneurs de niveau intermédiaire, il est possible d'utiliser une stratégie d'emboîtement des objets, un JPanel pouvant contenir plusieurs autre JPanel disposés via un certain LayoutManagers, lesquels JPanel contiennent d'autres entités en les organisant avec tel autre LayoutManager, etc…

Pour disposer correctement et automatiquement les composants les uns par rapport aux autres, un LayoutManager va interroger ces derniers quant à leur taille, et ce de manière éventuellement récursive. Ainsi, en partant du JFrame, on va interroger toutes les entités contenues dans son ContentPane, qui vont elles-mêmes interroger leur contenu, etc. Les différentes valeurs seront additionnées à la remontée des appels.

Ajout de composants

L'ajout de composants se fait via une méthode add (surchargée…) qui dépend du LayoutManager utilisé.

Rappel : dans le cas d'un container de haut niveau, il faut ajouter les composants au ContentPane, et non directement au container.

Les différents LayoutManagers

null : ce n'est pas à proprement parler un LayoutManager, c'est plutôt l'absence de ce dernier, mais qui donne lieu à un agencement particulier, en positionnement absolu. Ex :

```
contentPane.setLayout(null);
Rectangle r=new Rectangle(10,10,100,40);
e=new JLabel("test",SwingConstants.LEFT);
e.setBounds(r );
contentPane.add(e );
```

FlowLayout : positionne les composants les uns derrière les autres
BorderLayout : positionne les composants dans 5 endroits prédéfinis (NORTH, SOUTH, EAST, WEST, CENTER) en maximisant l'espace occupé dans CENTER.
GridLayout : positionne les composants dans les cellules d'un tableau
GridBagLayout : idem, mais autorise les composants à s'étaler sur plus d'une cellule, avec des cellules qui ne sont pas forcément de la même taille
BoxLayout : permet un alignement vertical ou horizontal des composants.

Quelques éléments à considérer pour choisir un LayoutManager, selon que l'on souhaite :

- un composant prenant le plus de place possible : BorderLayout en plaçant ce composant au CENTER ou un BoxLayout
- des composants sur une seule ligne et gardant leur taille naturelle : FlowLayout ou BoxLayout
- des composants rangés dans un tableau, tous de taille identique : GridLayout
- des composants rangés en ligne ou en colonne en variant les alignements et les espaces : BoxLayout
- des composants disposés de façon plus riche : GridBagLayout

5. Programmation événementielle via le pattern Observer

Présentation

Vous venez de voir comment construire une interface graphique via des composants organisés au sein de containers, le tout placé dans (au moins) un container de haut niveau. Mais, en l'état, votre interface n'est pas réactive aux sollicitations de l'utilisateur. C'est ce que va permettre la programmation événementielle.

Cette dernière est :
- dirigée par les événements
- corollairement, elle est non séquentielle (ou du moins semble ne pas l'être)

En effet, plutôt que de considérer qu'un programme suit l'ordre naturel d'une suite d'instruction, la programmation événementielle permet de scruter les événements se produisant, qui sont souvent des sollicitation souris ou clavier de l'utilisateur, et d'associer l'exécution de telle partie de code (une méthode) à tel ou tel événement. On ne peut donc pas prévoir lors de l'écriture du code quelles méthodes, dans quel ordre, et à quels moments vont être successivement exécutées au cours du programme. C'est son aspect non séquentiel (même si de façon interne, il y a bien séquentialité).

Evénement

Ce peut être :
- une action sur un objet graphique (clic souris, drag, etc.)
- une action interne au programme (timer, etc.)

Listener (EventListeners)

Le modèle de gestion des événements en Swing délègue le traitement de ces derniers à des objets implémentant des interfaces appelées génériquement des EventListener.

Selon le type d'événement, ce peut être :
- ActionLister (clic sur un bouton, choix dans un menu, etc.)
- KeyListener (action sur le clavier)
- MouseListener (action sur la souris concernant un objet particulier)
- WindowListener (activation, fermeture, icônification, etc.)
- Etc… (les interfaces sont très nombreuses)

Chacune de ces interfaces définit une ou plusieurs méthodes relatives aux différents événements prévus, à voir dans l'API. Par exemple, ActionListener ne définit qu'une méthode,

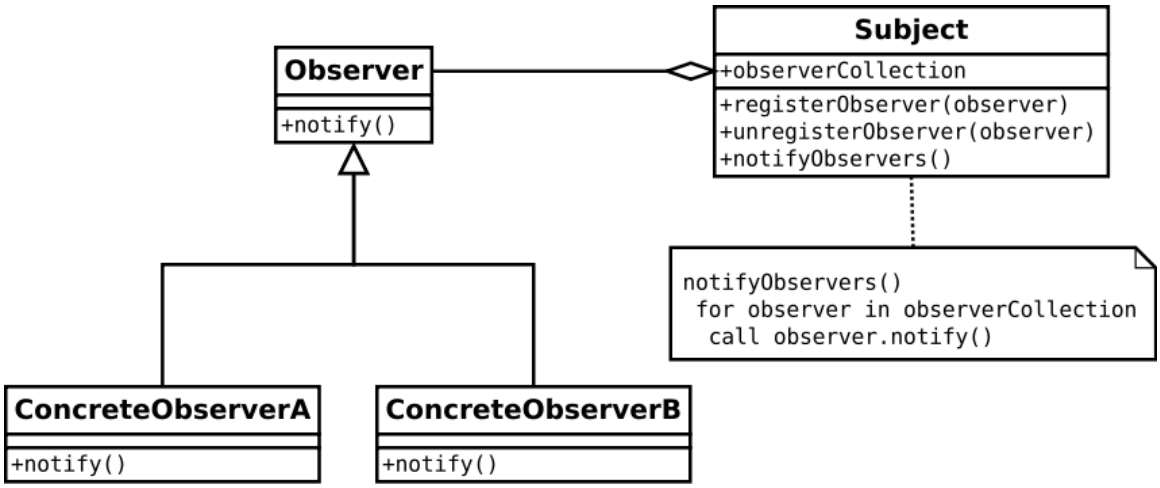
```
void actionPerformed(ActionEvent e)
```

qui sera appelée lorsque, par exemple, un certain bouton est cliqué.

Principe

Le principe général des la programmation événementielle s'appuie sur le design pattern Observer, qui consiste à faire en sorte qu'une instance puisse être "observée" par d'autres, ses Listeners, qui se sont enregistrés auprès d'elle, lesquels sont avertis lorsque cette instance déclenche un événement.

Dans le diagramme ci-dessous fourni par Wikipedia, Subject représente une classe qui souhaite être observée. Observer représente le type abstrait (interface) des objets qui souhaitent écouter un Subject. Une instance de ConcreteObserverA ou ConcreteObserverB peut donc écouter une instance de Subject (ou de ses sous-classes) en s'abonnant auprès d'elle via la méthode registerObserver(Observer observer). Dès lors, à chaque fois que le Subject invoquera sa méthode notifyObservers(), toutes les instances d'Observer enregistrées seront prévenus grâce à l'invocation de leur méthode notify().



Analyse critique du diagramme Wikipedia :

- notifyObservers() devrait être protected et non public : c'est l'objet lui-même qui doit décider qu'il faut prévenir ceux qui l'observe
- la méthode notify() devrait recevoir une instance du Subject qui est à l'origine de l'appel (call-back), afin que l'on puisse savoir qui nous prévient lorsque l'on écoute simultanément plusieurs Subjects.

La force de ce pattern réside dans le fait que l'on enregistre auprès de l'entité observée (par exemple, un bouton) des instances vues en tant qu'interface (une sous interface de EventListener), si bien que toute classe peut se transformer en Listener (il lui suffit pour cela d'implémenter l'interface correspondante). Ainsi, notre bouton peut être éventuellement écouté par des instances de classes très différentes.

Ainsi, on peut déléguer la gestion des événements d'une entité à :

- la classe elle-même. Il suffit pour cela qu'elle implémente l'EventListener correspondant (par exemple un bouton peut s'écouter lui-même)
- une classe quelconque implémentant l'EventListener
- une classe interne (classe à l'intérieur de notre classe), implémentant l'EventListener

On voit que ces trois façons de procéder sont finalement un peu toujours la même : ce sont dans tous les cas des classes qui implémentent l'interface adéquate.

Le fait de disposer d'une classe sachant écouter une entité ne suffit pas. Il faut aussi enregistrer (abonner) une instance à l'objet à écouter, via une méthode du type :

```
monObjetEcouté.add<type d'événement>Listener(référenceListener);
```

Par exemple, pour un bouton qui s'écouterait lui-même :

```
Bouton1.addActionListener(this);
```

Enfin, comment associer un morceau de code à exécuter lorsqu'un événement se produit ? En implémentant la méthode correspondante dans la classe d'implémentation du listener.

Abonnements multiples, désabonnements

Avec ce mécanisme, vous pouvez enregistrer autant de listeners que vous voulez à un objet donné, en invoquant autant de fois que nécessaire le add<…>Listener correspondant. Ainsi, différents listeners seront prévenus à chaque fois.
Notons que l'on peut à loisir enregistrer/dés-enregistrer des listeners de façon dynamique au cours de l'exécution du programme.