



ALG2ir-TD03 : Applications à interface graphique : Swing (Aspects dynamiques)

1 Présentation du TD

Ce TD est une suite au TD02. Après avoir vu comment créer et mettre en page une application à interface graphique, voyons maintenant comment la faire réagir aux actions des utilisateurs. Nous obtiendrons ainsi une application complète.

Une des grandes particularités d'une application à interface graphique et de son interaction avec l'utilisateur est que les désirs de ce dernier peuvent être multiples (accès à un menu, clic sur un bouton, réduction, etc.) et peuvent survenir à tout moment dans n'importe quel ordre (en général du moins).

Ceci va *a priori* à l'encontre de la programmation impérative où les instructions que doit exécuter l'ordinateur lui sont fournies les unes après les autres. Dans ce contexte, la détection de l'enfoncement d'un bouton de la souris, par exemple, est envisageable au sein d'une boucle d'attente. Cette technique est² mise en œuvre dans le cours de langage d'assemblage de première année...

Un autre paradigme de programmation est ici envisagé : la programmation événementielle. Celle-ci s'appuie sur le polymorphisme et le modèle de conception (design pattern en anglais) « Observateur – Sujet d'observation » (*Observer* en anglais). Remarquez que ces concepts sont également abordés dans le cours et les laboratoires de C++.

2 Les événements

Pour l'instant, nos applications graphiques sont figées. Il ne se passe strictement rien de dynamique. Appuyer sur un bouton, par exemple, n'a aucun effet.

Associer une réaction à une action posée sur un composant se réalise via le *mécanisme d'événement*.

Dans les applications disposant d'une interface graphique (un grand nombre d'applications donc), le programme ne suit pas un flot prédéfini mais est dirigé par des événements : l'appui sur un bouton, le choix d'un élément de menu, etc. L'ordre des événements est donc très

¹ Le texte de ce TD s'inspire très largement de ceux des Ateliers Logiciels Java (2^e Gestion et 3^e Industrielle et Réseaux) créés principalement par MCD et VAK, mais auxquels ont également collaboré RFS et SMB (et peut-être d'autres que j'ignore, qu'ils me pardonnent).

² Ou plutôt : était.

souvent impossible à prédire puisqu'il dépend de l'utilisateur du programme. Cette forme de programmation (non déterminisme du flux, réaction aux événements) est à la base de ce qu'on appelle la *programmation événementielle*.

La notion d'événement y est donc primordiale. Une gestion d'événement implique deux objets.

Le premier est l'*observé*, la *source de l'événement*. C'est sur lui que des événements peuvent se produire.

Le deuxième est l'*observateur* (ou *listener* en anglais). Il s'est déclaré intéressé par la gestion des événements sur le premier objet.

Lorsqu'un événement se produit sur un objet, tous les objets qui se sont inscrits comme observateur de cet objet pour cet événement sont prévenus et peuvent ainsi réagir.

Les événements similaires sont regroupés dans une même classe. Par exemple, la classe `MouseEvent` regroupe les événements `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed` et `mouseReleased`.

Tous ces éléments interagissent selon un procédé défini par le modèle de conception « Observateur – Sujet d'observation » que nous allons expliciter tout au long de ce TD.

Événements de composants et événements sémantiques

Swing^a distingue deux familles d'événements.

Un *événement de composant* est de bas niveau et sa sémantique est indépendante du type de composant (`keyPressed`, `mouseClicked`, etc.).

Un *événement sémantique* est de plus haut niveau. Son sens précis dépend du composant. Ainsi, l'événement `Action` signifie « *presser* » pour un bouton et « *choisir* » pour une liste.

On privilégie autant que possible l'utilisation du deuxième type d'événements.

^a En fait Swing reprend tel quel le mécanisme de AWT.

3 Les observés (sources)

Une classe doit annoncer qu'elle peut produire des événements pour qu'on puisse l'observer. Cela se fait en étendant la classe `Component`. C'est bien le cas des composants que nous allons utiliser. Il n'y a donc rien de plus à faire du côté des sources d'événements.

4 Les observateurs (*listeners*)

Pour définir un observateur, il y a essentiellement deux étapes :

1. être reconnu comme un observateur valable ;
2. définir les méthodes qui sont appelées lorsque les événements surviennent.

4.1 Être reconnu comme observateur

Une classe (ou plutôt un objet de cette classe) qui veut devenir un observateur doit être reconnu comme étant capable de l'être. Pour cela, elle doit implémenter l'interface correspondante ou hériter de la classe *ad hoc*.

```
1 class Observateur implements MouseListener
2 class Observateur extends MouseAdapter
```

Pourquoi deux techniques? L'héritage offre l'avantage de fournir une implémentation par défaut (et sans effet) des méthodes associées aux événements. Cela facilite l'écriture lorsqu'on ne veut réagir qu'à un petit nombre d'événements parmi ceux proposés par la classe. Par contre, l'héritage étant simple en Java, cela empêche d'hériter d'une autre classe, ce qui est parfois inacceptable.

4.2 Fournir les méthodes adéquates

L'observateur doit alors fournir une méthode pour chaque événement géré. Cette méthode a un nom imposé. Exemple : pour l'événement `Action`, la méthode est `actionPerformed` :

```
1 public void actionPerformed(ActionEvent e) {
2     ...
3 }
```

Cette méthode reçoit un objet représentant l'événement. On peut interroger ce dernier si on veut en savoir un peu plus. Un même observateur peut, par exemple, observer un même type d'événement (donc implémenter un même méthode) sur *plusieurs* objets observés. Interroger l'objet `e` permet de savoir sur quel objet précis a eu lieu l'événement.

5 Lier un observateur à un sujet d'observation

Une fois qu'on a un « observé » et un « observateur », il faut les mettre en contact, créer la dynamique.

1. Il faut que l'« observateur » s'enregistre auprès de l'objet qu'il « observe ». À cette fin, tout observé potentiel fournit une méthode qui permet cet enregistrement.

Exemple : pour la classe `ActionEvent`, on a la méthode :

```

1 public void addActionListener( ActionListener l ) ;
2 ...
3 observé.addActionListener(observateur) ; // Exemple d'utilisation

```

2. Lorsqu'un événement surgit sur un composant, celui-ci prévient tous ses observateurs enregistrés (par un appel à la méthode *ad hoc*, cf. 4.2).

6 Mise en pratique

Synthétisons les différentes notions introduites. Où et comment écrire l'observateur ? Qui va s'occuper de relier un observateur à son observé ? En pratique, il y a quatre façons de procéder. L'observateur peut être :

1. une classe différente et indépendante de celle de l'observé, avec implémentation d'une interface *listener* ;
2. une classe différente et indépendante de celle de l'observé, avec héritage d'un *adapter* ;
3. une classe interne à la classe de l'observé (ou interne au conteneur de l'observé). Cela permet de bien montrer que cette classe ne sert qu'à cela. En général via héritage ;
4. une classe interne anonyme à la classe de l'observé (ou de son conteneur). Cela réduit le code, mais il devient moins facile à lire. Presque toujours par héritage.

Exemple : Prenons une simple fenêtre vide et ajoutons le code nécessaire pour que la fenêtre se ferme lorsqu'on clique sur son bouton de fermeture³. Testons les 4 possibilités énumérées plus haut.

L'événement recherché fait partie de la classe `WindowEvent` et s'appelle `windowClosing`. L'interface à implémenter s'appelle `WindowListener`.

6.1 Solution 1 : Observateur dans une classe à part via une interface

```

1 // MaFenetre.java
2 package nvs.alg2ir.td03swingdynamique.enonce ;
3
4 import javax.swing.JFrame ;
5 import javax.swing.WindowConstants ;
6
7 /**
8  * La fenêtre à observer
9  */
10 class MaFenetre extends JFrame {
11

```

³ Remarquons que, par défaut, une `javax.swing.JFrame` est cachée quand on clique sur son bouton de fermeture. Par contre, une `java.awt.Frame` ne réagit pas à un tel clic. Nous allons voir ici comment fermer une `javax.swing.JFrame` sans recourir à la méthode `setDefaultCloseOperation(int operation)`, mais par une technique plus générale, applicable également aux `java.awt.Frame`. Notons cependant que *l'opération de fermeture par défaut est quand même exécutée*. Fixons-la donc à `javax.swing.WindowConstants.DO_NOTHING_ON_CLOSE`.

```

12 public MaFenetre() {
13     setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE) ;
14     // indispensable car comportement par défaut a toujours lieu !
15 }
16 }

```

```

1 // MaFenetreObservateur.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 import java.awt.Window;
5 import java.awt.event.WindowEvent;
6 import java.awt.event.WindowListener;
7
8 /**
9  * L'observateur de fenêtre (par implémentation d'interface)
10  */
11 class MaFenetreObservateur implements WindowListener {
12
13     public void windowClosing(WindowEvent e) {
14         System.exit(0);
15         // ci-dessous : un gestion alternative de l'événement :
16         /*
17         Window win = e.getWindow() ;
18         win.setSize(2*win.getWidth(),win.getHeight()) ;
19         */
20         // ci-dessous : à essayer sans setDefaultCloseOperation et
21         // sans System.exit...
22         /*
23         try {
24             Thread.sleep(2000) ;
25         } catch (InterruptedException ie) {
26             System.out.println(ie);
27         }
28         */
29     }
30     public void windowActivated(WindowEvent e) {}
31     public void windowClosed(WindowEvent e) {}
32     public void windowDeactivated(WindowEvent e) {}
33     public void windowDeiconified(WindowEvent e) {}
34     public void windowIconified(WindowEvent e) {}
35     public void windowOpened(WindowEvent e) {}
36 }

```

```

1 // Test.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 /**
5  * Classe de test des classes MaFenetre (la fenêtre à observer)
6  * et MaFenetreObservateur (le listener de fenêtre)
7  */
8 class Test {
9
10     public static void main(String[] args) {
11         MaFenetre f = new MaFenetre();
12         f.addWindowListener( new MaFenetreObservateur() );

```

```

13     f.setVisible(true);
14 }
15 }

```

6.2 Solution 2 : Observateur dans une classe à part via héritage

Dans ce cas, on utilise un adaptateur (*adapter* en anglais). Celui-ci implémente l'interface pour nous en fournissant des méthodes vides par défaut. Dans cette solution, seul le code de l'observateur change : il se simplifie.

```

1 // MaFenetre.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6
7 /**
8  * La fenêtre à observer
9  */
10 class MaFenetre extends JFrame {
11
12     public MaFenetre() {
13         this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE) ;
14     }
15 }

```

```

1 // MaFenetreObservateur.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 import java.awt.event.WindowAdapter;
5 import java.awt.event.WindowEvent;
6
7 /**
8  * Observateur de fenêtre (par le biais de l'héritage)
9  */
10 class MaFenetreObservateur extends WindowAdapter {
11
12     public void windowClosing(WindowEvent e) {
13         System.exit(0);
14     }
15 }

```

```

1 // Test.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 /**
5  * Classe de test des classes MaFenetre et MaFenetreObservateur
6  */
7 class Test {
8
9     public static void main(String [] args) {
10         MaFenetre f = new MaFenetre();

```

```

11     f.addWindowListener( new MaFenetreObservateur() );
12     f.setVisible(true);
13 }
14 }

```

6.3 Solution 3 : L'observateur est une classe interne de l'observé

On reste proche de la deuxième solution mais la classe de l'observateur est une classe interne de l'observé. Les avantages sont une meilleure visibilité et la possibilité pour l'observateur d'accéder à la partie privée de l'observé. Dans notre exemple, l'association entre observateur et observé se fait à présent au moment de la construction de la fenêtre.

```

1 // MaFenetre.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 import java.awt.event.WindowAdapter;
5 import java.awt.event.WindowEvent;
6 import javax.swing.JFrame;
7
8 /**
9  * La fenêtre à observer et l'observateur comme classe interne
10 */
11 class MaFenetre extends JFrame {
12     /**
13      * Observateur == classe interne du sujet d'observation
14      */
15     class MaFenetreObservateur extends WindowAdapter {
16         public void windowClosing(WindowEvent e) {
17             System.exit(0);
18         }
19     }
20
21     MaFenetre() {
22         setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
23         addWindowListener( new MaFenetreObservateur() );
24     }
25 }

```

```

1 // Test.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 /**
5  * Classe de test des classes MaFenetre
6  */
7 class Test {
8
9     public static void main(String[] args) {
10         MaFenetre f = new MaFenetre();
11         f.setVisible(true);
12     }
13 }

```

Attention! Il n’y a plus que deux fichiers sources mais toujours trois classes. Quel est le nom de la troisième?

6.4 Solution 4 : L’observateur est une classe interne et anonyme de l’observé

L’observateur ne servant que dans une classe et n’étant repris qu’à un seul endroit du code (le constructeur), on peut en faire une classe anonyme. Comme pour la solution précédente, il y a deux fichiers sources. La classe anonyme entraîne bel et bien la création d’un fichier `.class`. Trouvez son nom.

```

1 // MaFenetre.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 import java.awt.event.WindowAdapter;
5 import java.awt.event.WindowEvent;
6 import javax.swing.JFrame;
7
8 /**
9  * Fenêtre avec observateur de classe interne et de classe anonyme
10 */
11 class MaFenetre extends JFrame {
12
13     MaFenetre() {
14
15         setDefaultCloseOperation(DO_NOTHING_ON_CLOSE) ;
16
17         // Création d'un observateur nommé de classe anonyme
18         WindowAdapter obs = new WindowAdapter() {
19             public void windowClosing(WindowEvent e) {
20                 fermer() ;
21             }
22         }; // noter le ;
23
24         // Enregistrement de l'observateur
25         addWindowListener( obs );
26     }
27
28     private void fermer() {
29         this.dispose();
30         //System.exit(0); // ici, alternative...
31     }
32 }

```

```

1 // Test.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 /**
5  * Classe de test de la classes MaFenetre
6  */
7 class Test {
8

```

```

9  public static void main(String[] args) {
10     MaFenetre f = new MaFenetre();
11     f.setVisible(true);
12 }
13 }

```

Remarquez que la méthode qui est appelée quand l'événement survient appelle elle-même une méthode contenant le code de gestion de cet événement. Cela rend le code plus lisible et permet d'appliquer la même action à plusieurs événements. Par exemple, le menu « File >> Exit » pourrait avoir le même effet.

6.5 En pratique, quel procédé utiliser ?

Les deux premiers sont les plus souples car le lien n'est ni dans l'observateur ni dans l'observé. Ceci permet de réutiliser le code de l'un et de l'autre. On peut ainsi facilement :

- relier l'observé à d'autres observateurs ;
- demander à l'observateur d'observer d'autres objets.

Au sein même d'une classe, et lorsque le problème de la réutilisation n'est pas central (car cette partie n'aurait pas de sens ailleurs), la quatrième solution est privilégiée car elle ne demande pas l'introduction explicite d'une troisième classe. Parfois même, dans ce cas, observateur et observé sont un seul et même objet.

Enfin, le code de la quatrième solution peut être encore plus concis en recourant à un observateur *anonyme* instance d'une classe interne anonyme.

```

1  // MaFenetre.java
2  package nvs.alg2ir.td03swingdynamique.enonce;
3
4  import java.awt.event.WindowAdapter;
5  import java.awt.event.WindowEvent;
6  import javax.swing.JFrame;
7
8  /**
9   * Fenêtre avec observateur anonyme de classe interne et de classe anonyme
10  */
11  class MaFenetre extends JFrame {
12
13     MaFenetre() {
14
15         setDefaultCloseOperation(DO_NOTHING_ON_CLOSE) ;
16
17         // Enregistrement d'un observateur anonyme de classe anonyme
18         this.addWindowListener( new WindowAdapter() {
19             public void windowClosing(WindowEvent e) {
20                 fermer() ;
21                 //MaFenetre.this.dispose(); // noter la position de this...
22             }
23         }); // noter le ;
24     }
25
26     private void fermer() {
27         this.dispose();

```

```

28     //System.exit(0); // ici, alternative...
29 }
30 }

```

```

1 // Test.java
2 package nvs.alg2ir.td03swingdynamique.enonce;
3
4 /**
5  * Classe de test de la classes MaFenetre
6  */
7 class Test {
8
9     public static void main(String [] args) {
10         MaFenetre f = new MaFenetre();
11         f.setVisible(true);
12     }
13 }

```

6.6 Exercices

Ex1. Ajoutez à l'exemple précédent un bouton « Terminer » qui termine l'application. Utilisez le quatrième (voire le cinquième) procédé (observateur comme classe anonyme dans l'observé).

Ex2. Une fenêtre contient deux boutons : « stop » et « encore ». Appuyer sur le bouton « encore » ouvre une fenêtre identique. Appuyer sur le bouton « stop » ferme la fenêtre.

7 Échange d'informations entre conteneurs

Cette partie est une mise en pratique du mécanisme « observateur – sujet d'observation » au travers d'un problème particulier : l'échange d'information entre fenêtres.

On identifie les différents cas de figures et les solutions possibles. Ces solutions sont en accord avec les bonnes pratiques de l'orienté objet. En particulier, on évite de rendre les attributs publics.

Deux cas peuvent survenir. Soit l'information transite de la fenêtre source vers la fenêtre appelée, soit c'est le contraire.

7.1 Flux d'information de la fenêtre source (appelante) vers la fenêtre créée (appelée)

Exemple : Une fenêtre contient un champ de saisie et un bouton. Appuyer sur le bouton ouvre une fenêtre modale⁴ reprenant le contenu du champ de saisie.

Solution : Une approche propre est de passer l'information par le constructeur de la fenêtre créée.

⁴ Ce point n'est pas essentiel.

Ex3. Mettez en œuvre cette solution.

7.2 Flux d'information de la fenêtre créée (appelée) vers la fenêtre source (appelante)

Exemple : Une boîte de dialogue a été ouverte. Elle contient un champ de saisie. Lorsqu'on ferme cette boîte, le texte saisi doit être affiché dans la fenêtre qui a initié le dialogue.

Solution 1 (mode modal uniquement) : Il suffit d'attendre la fermeture de la boîte de dialogue et d'alors l'interroger via les méthodes adéquates qui retournent l'information voulue.

Solution 2 (mode non modal) : De temps en temps, l'appelant utilise des méthodes de l'appelé pour en obtenir des informations. La difficulté est de savoir « quand » utiliser ces méthodes. Il s'agit d'une *mauvaise solution*, ne faisant pas usage de la programmation événementielle.

Solution 3 (mode non modal) : Via un événement, l'appelé signale à l'appelant qu'une information est disponible. L'appelant utilise alors la méthode idoine de l'appelé. C'est la solution la plus souple et la plus propre.

Solution 4 (mode non modal) : L'appelé utilise une méthode de l'appelant pour lui communiquer l'information. Il ne faut jamais oublier que l'objet appelé ne sait pas par qui ni comment il sera utilisé ; c'est le principe même de la réutilisabilité dans l'orienté objet. Tout doit se faire via une interface précise que l'appelant doit implémenter. De plus, l'appelant se passera lui-même par le constructeur de l'appelé. C'est plus direct à implémenter que la solution précédente mais moins souple. Pourquoi ?

Ex4. Mettez en œuvre ces solutions, à l'exception de la seconde.

8 Exercice récapitulatif

Ex5. En première année, vous avez peut-être réalisé une application permettant de calculer l'« Indice de Masse Corporelle⁵ » (*Body Mass Index* en anglais). Pour le coup, demandez en plus ici, via une boîte de dialogue, le nom de la personne. Affichez le résultat dans une nouvelle boîte de dialogue. Ces exigences sont purement pédagogiques ; le programme obtenu ne répond probablement pas aux critères ergonomiques habituels !

⁵ Pour rappel, $IMC = p/h^2$, où p est le poids en kilogrammes et h est la taille en mètres.