

## TD3

# Interface utilisateur graphique (aspects dynamiques)

### 1 Préalables

Ce TD fait suite à la présentation de Swing dans ces aspects statiques. Nous allons voir maintenant comment faire réagir une interface graphique aux actions des utilisateurs.

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 *à 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 la frappe d'une touche, par exemple, est envisageable au sein d'une boucle d'attente par exemple.

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*) « Observateur / Observé » (*Observer*)<sup>2</sup>.

### 2 Les événements

Pour l'instant, nos applications 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**.

<sup>1</sup>Le texte de ce TD s'inspire très largement de ceux des ateliers logiciels écrits jadis par différents collègues. Parmi les TDs que comprend ce cours, *MCD*, *NVS*, *RFS*, *SMB* et *VAK* ont contribué de manière diverses. Certains ont écrits entièrement un TD, d'autres ont contribué ou coécrits un TD, d'autres encore ont fait du travail de relecture. Je (*PBT*) laisse mon empreinte et en profite pour remercier tous ceux qui aiment être remerciés.

<sup>2</sup>Remarquez que ces concepts sont également abordés dans le cours et les laboratoires de C++.

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 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, nous insistons donc, 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** (*listener*). 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 (*listener*) 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 ce modèle de conception « Observateur / Observé » que nous allons expliciter tout au long de ce TD<sup>3</sup>.

<sup>3</sup>Ce concept d'Observateur / Observé est également détaillé dans le guide correspondant. Dans la suite de ce TD nous traiterons principalement des aspects « graphiques » de ce patron de conception (*design pattern*) bien qu'il soit plus général comme vous pouvez le lire dans le guide.

### Événements de composants et événements sémantiques

Swing<sup>a</sup> 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.

<sup>a</sup> 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*.

```
class Observateur implements MouseListener
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 lorsque l'on ne veut réagir qu'à un petit nombre d'événements parmi ceux proposés par la classe. Implémenter l'interface imposerait d'implémenter chaque méthode imposée.

Par contre, l'héritage étant simple en Java, cela empêche d'hériter d'une autre classe, ce qui est parfois inacceptable et impose alors l'option « implémentation »

### 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` :

```
public void actionPerformed(ActionEvent e) {
...
}
```

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 une même méthode) sur *plusieurs* objets observés. Interroger l'objet `e` permet de savoir sur quel objet précis a eu lieu l'événement.

Par exemple, on peut imaginer qu'un seul observateur observe une ensemble de boutons représentant les touches d'une calculatrice. L'action exécutée lorsque l'on presse un bouton est identique pour tous les chiffres à la valeur du chiffre près.

## 5 Lier un observateur à un sujet d'observation

Une fois que l'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 :

```
public void addActionListener( ActionListener l );
...
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 ;
- 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 est 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 fermeture<sup>4</sup>. 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

```
$ cat MaFenetre.java
```

```
// MaFenetre.java
package nvs.alg2ir;

import javax.swing.JFrame;
import javax.swing.WindowConstants;

/**
 * La fenêtre à observer
 */
class MaFenetre extends JFrame {
```

<sup>4</sup>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`.

```
public MaFenetre() {
    setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
    // indispensable car comportement par défaut a toujours lieu !
}
}
```

```
$ cat MaFenetreObservateur
```

```
// MaFenetreObservateur.java
package nvs.alg2ir;

import java.awt.Window;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

/**
 * L'observateur de fenêtre (par implémentation d'interface)
 */
class MaFenetreObservateur implements WindowListener {

    public void windowClosing(WindowEvent e) {
        System.exit(0);
        // ci-dessous : un gestion alternative de l'événement :
        /*
        Window win = e.getWindow();
        win.setSize(2*win.getWidth(),win.getHeight());
        */
        // ci-dessous : à essayer sans setDefaultCloseOperation et
        // sans System.exit...
        /*
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ie) {
            System.out.println(ie);
        }
        */
    }

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

```
$ cat Test
```

```
// Test.java
package nvs.alg2ir;

/**
 * Classe de test des classes MaFenetre (la fenêtre à observer)
 * et MaFenetreObservateur (le listener de fenêtre)
 */
class Test {
    public static void main(String[] args) {
        MaFenetre f = new MaFenetre();
        f.addWindowListener(new MaFenetreObservateur());
        f.setVisible(true);
    }
}
}
```

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

Dans ce cas, on utilise un adaptateur (*adapter*). 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 n'est plus nécessaire d'écrire toutes les méthodes de l'interface, seules les méthodes redéfinies.

```
$ cat MaFenetre.java
```

```
// MaFenetre.java
package nvs.alg2ir;

import javax.swing.JFrame;
import javax.swing.JLabel;

/**
 * La fenêtre à observer
 */
class MaFenetre extends JFrame {

    public MaFenetre() {
        this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    }
}
```

```
$ cat MaFenetreObservateur
```

```
// MaFenetreObservateur.java
package nvs.alg2ir;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

/**
 * Observateur de fenêtre (par le biais de l'héritage)
 */
class MaFenetreObservateur extends WindowAdapter {

    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

```
$ cat Test
```

```
// Test.java
package nvs.alg2ir;

/**
 * Classe de test des classes MaFenetre et MaFenetreObservateur
 */
class Test {

    public static void main(String[] args) {
        MaFenetre f = new MaFenetre();
        f.addWindowListener(new MaFenetreObservateur());
        f.setVisible(true);
    }
}
```

## 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.

```
$ cat MaFenetre.java
```

```
// MaFenetre.java
package nvs.alg2ir;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;

/**
 * La fenêtre à observer et l'observateur comme classe interne
 */
class MaFenetre extends JFrame {
    /**
     * Observateur == classe interne du sujet d'observation
     */
    class MaFenetreObservateur extends WindowAdapter {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    }

    MaFenetre() {
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
        addWindowListener(new MaFenetreObservateur());
    }
}
```

```
$ cat Test
```

```
// Test.java
package nvs.alg2ir;

/**
 * Classe de test des classes MaFenetre
 */
class Test {

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

**Notez bien** 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.

```
$ cat MaFenetre.java
```

```
// MaFenetre.java
package nvs.alg2ir;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;

/**
 * Fenêtre avec observateur de classe interne et de classe anonyme
 */
class MaFenetre extends JFrame {

    MaFenetre() {

        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE) ;

        // Création d'un observateur nommé de classe anonyme
        WindowAdapter obs = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                fermer() ;
            }
        }; // noter le ;

        // Enregistrement de l'observateur
        addWindowListener( obs );
    }

    private void fermer() {
        this.dispose();
        //System.exit(0); // ici, alternative...
    }
}
```

```
$ cat Test
```

```
// Test.java
package nvs.alg2ir;

/**
 * Classe de test de la classes MaFenetre
 */
class Test {

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

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.

```
$ cat MaFenetre.java
```

```
// MaFenetre.java
package nvs.alg2ir;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;

/**
 * Fenêtre avec observateur anonyme de classe interne et de classe anonyme
 */
class MaFenetre extends JFrame {

    MaFenetre() {

        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE) ;

        // Enregistrement d'un observateur anonyme de classe anonyme
        this.addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                fermer() ;
                //MaFenetre.this.dispose(); // noter la position de this...
            }
        }); // noter le ;
    }

    private void fermer() {
        this.dispose();
        //System.exit(0); // ici, alternative...
    }
}
```

```
$ cat Test
```

```
// Test.java
package nvs.alg2ir;

/**
 * Classe de test de la classes MaFenetre
 */
class Test {

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

## 6.6 Exercices

**Exercice 1** 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).

**Exercice 2** 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.

**Exercice 3** Reprenez l'exercice du *poulailler*.

Ajoutez-y les fonctionnalités suivantes (vous devrez plus que probablement faire des choix car l'énoncé est volontairement flou, renseignez-les dans la documentation) :

- ↪ cliquer dans le poulailler permet de créer des poules. Je clique sur un bouton "Ajout d'une poule" et ensuite je clique dans le poulailler pour ajouter la poule à cet endroit,<sup>5</sup>
- ↪ une zone d'édition permet de fixer la taille du tas de graines, on pourra éventuellement la modifier en cours,
- ↪ des zones d'affichage permettent de voir l'état du tas de graines ainsi que le moment (le nombre de « tours » écoulés),
- ↪ un bouton permettra de passer d'un tour au suivant ( $t \rightarrow t + 1$ ),
- ↪ lorsqu'une poule pond, ses petits sont placés à côté d'elle (si la case est déjà occupée par une autre poule, elle écrase le poussin qui meurt ... *C'est*

<sup>5</sup>Normalement chaque case du poulailler devrait être un composant graphique capable d'afficher une poule (ou pas), de réagir aux clics de souris, de signaler ses changements d'états au poulailler qui le contient, ... mais cela va au-delà de cette présentation. Le lecteur intéressé pourra lire le « Guide du développeur JavaBean »

*trop injuste*<sup>6,7</sup>,

- ↪ les poules se déplacent d'une position latéralement et aléatoirement à chaque tour, elles ne peuvent pas sortir de l'enclos,
- ↪ une poule morte disparaît de l'enclos, c'est plus propre

<sup>6</sup>Rappelez-vous notre ami Caliméro

<sup>7</sup>Une poule n'aura donc pas plus de 8 petits sur une portée