

Guide du *designer* Observateur-Observé

L'observateur/observé (observer/observable) est un design pattern permettant une communication entre objets du type:

- « Je suis observable, observez-moi, je vous signalerai mes changements. »
- « Je suis un observateur, je m'inscris à la liste de tes observateurs et je me mettrai à jour lorsque tu m'informerai d'un changement. »

1 PRÉALABLES

Ce design pattern est mis en œuvre dès que des objets veulent pouvoir se mettre à jour en fonction de l'état d'un autre objet. Les premiers objets sont appelés les **observables** ou **observés** (*observable*) tandis que les seconds sont appelés les **observateurs** (*observers*) ou encore **écouteurs** (*listeners*)¹.

Les **observables** doivent préciser qu'ils le sont et devront informer tous leurs observateurs dès que leur état change. L'observateur choisira de se mettre à jour ou pas en fonction de l'état de l'observé.

Les **observateurs** devront préciser qu'ils le sont, s'inscrire comme observateur auprès d'un observé et prévoir une méthode qui sera exécutée à chaque changement d'état d'un observé.

2 MISE EN ŒUVRE EN JAVA² DANS UN CONTEXTE GÉNÉRAL

2.1 Une classe observatrice

Pour être **observateur**,

- une classe Java doit implémenter l'interface `Observer` qui lui imposera la réécriture de la méthode `update(Observable o, Object arg)`. Cette méthode sera exécutée lors de chaque notification de changement de la part d'un des observables auquel l'observateur est inscrit.

Le paramètre `arg` est un argument qui peut être passé par l'observable lors de la notification. Puisqu'il peut être récupéré par l'observé, cet argument peut permettre la transmission d'information(s) entre l'observateur et l'observé.

- une classe Java doit être inscrite auprès de l'observable via sa méthode `addObserver`.

¹ Cedi dépendra du contexte.

² Le code de cet exemple est disponible au même endroit que ce guide (<http://esi.namok.be>), ça vous évitera même le copier/coller.

On peut³ écrire le code suivant:

```
package pbt.observerobservable;

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

/**
 * ObserverClass.java
 *
 * Classe ayant la faculté d'observer un objet "observable".
 *
 * Cette classe implémente la classe <code>Observer</code> et doit
 * réécrire la méthode <code>update</code> qui sera exécutée lorsqu'un
 * observable change.
 *
 * @author pbt
 */
public class ObserverClass implements Observer {

    /**
     * Constructeur de l'observateur ... qui s'ajoute
     * à la liste des observateur de l'objet observé passé
     * en paramètre.
     * @param whatIsee l'objet observé
     */
    public ObserverClass(ObservableClass whatIsee) {
        whatIsee.addObserver(this);
    }

    /**
     * Implements java.util.Observer
     * @param o observable source de la modification
     * @param arg les arguments éventuellement passé à la
     *           méthode <code>notifyObservers</code>
     */
    public void update(Observable o, Object arg) {
        if (o instanceof ObservableClass) {
            ObservableClass oc = (ObservableClass) o;
            System.out.println("Observer: what I see say "
                + oc.getVeryImportantAttribute());
        }
    }
}
```

2.2 Une classe observable

Pour être **observable**,

- une classe doit hériter de la classe `Observable` et de ses méthodes `setChanged` et `notifyObservers`.
- une classe doit signaler –via la méthode `setChanged`– lorsque son état a changé et doit notifier ses observateurs de ce changement –via la méthode `notifyObservers`– afin qu'ils puissent se mettre à jour.

³ L'emploi de « pouvoir » n'est pas dû au hasard, on peut choisir d'autres manière d'implémenter la chose.

On peut par exemple écrire:

```
package pbt.observerobservable;

import java.util.Observable;
/**
 * ObservableClass.java
 *
 * Classe ayant la faculté d'être observée.
 *
 * Pour ce faire elle doit hériter de la classe <code>Observable</code>
 * qui lui propose les méthodes <code>setChanged()</code> et
 * <code>notifyObservers()</code>.
 * Ces méthodes doivent être appelée dès que l'on veut
 * signaler (<i>notify</i>)
 * aux observateurs (<i>observers</i>) que l'état de la
 * classe a changé.
 *
 * @author pbt
 */
public class ObservableClass extends Observable {
    /** Un attribut de la classe suffisamment important que
     * pour être observé.
     */
    private int veryImportantAttribute ;

    /** constructeur */
    public ObservableClass() {
        this.veryImportantAttribute = 0 ;
    }

    /**
     * getter
     * @return l'attribut suffisamment important que ...
     */
    public int getVeryImportantAttribute() {
        return veryImportantAttribute;
    }

    /**
     * setter
     * @param veryImportantAttribute
     */
    public void setVeryImportantAttribute(int veryImportantAttribute) {
        this.veryImportantAttribute = veryImportantAttribute;
        notifyObservers();
    }

    /**
     * Change l'attribut suffisamment important que ... de manière
     * aléatoire pour les besoins de la démonstration.
     */
    public void change() {
        this.veryImportantAttribute = (int) (Math.random()*100) ;
        // Je positionne mon indicateur de changement
        setChanged();
        // Je notifie les observateurs
        notifyObservers();
    }
}
```

2.3 Une classe testant le tout

Il reste à tester le tout dans une classe de test à l'allure suivante:

```
package pbt.observerobservable;

/**
 * Main.java
 *
 * Classe de test du "patron de conception" (<i>design pattern</i>)
 * <b>Observateur/observé</b>
 *
 * @author pbt
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        ObservableClass observable = new ObservableClass() ;
        ObserverClass observer =new ObserverClass(observable) ;
        for (int i=0; i<5 ; i++) {
            observable.change();
        }
    }
}
```

3 MISE EN ŒUVRE EN JAVA DANS UN CONTEXTE DE COMPOSANTS

Un inconvénient de la première approche est que l'on brûle son héritage unique en utilisant la classe `Observable`. Nous allons voir que dans une approche « composants », une partie du travail est déjà fait⁴.

Pour ce faire, étudions l'approche *event / listener* où les protagonistes sont des écouteurs d'événements et des sources d'événements⁵.

3.1 Une classe observatrice

L'observateur dans ce cas est à l'écoute des événements, on parlera d'**écouteurs** (*listeners*).

- Il doit être étiqueté « *listener* ». Pour ce faire, la classe implémentera une interface du type *EventTypeListener* où *EventType* peut être *PropertyChange* par exemple qui lui imposera la réécriture d'un méthode de mise à jour. Cette méthode reçoit un `PropertyChangeEvent`,

Nous verrons plus loin –et dans le guide *JavaBean*— que les événements dans ce contexte sont liés aux événements *Swing*.

- Il devra également s'inscrire comme écouteur de l'objet ... qu'il écoute via une méthode du type *addEventListener*.

⁴ La suite sera parfois redondante avec le *Guide du développeur JavaBean* disponible sur <http://esi.namok.be> qui présente la notion de gestion d'événements dans le cadre des *JavaBeans* puisque le principe est identique.

⁵ Il doit être clair que le concept du *design pattern* est le même mais sa mise en œuvre –et le vocabulaire— est différente.

Nous pouvons écrire un code –sans aucune référence à l'API Swing— ayant l'allure suivante:

```
package pbt.listeners;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

/**
 * Listener.java
 *
 * Classe ayant la faculté d'en écouter une autre.
 * @author pbt
 */
public class Listener implements PropertyChangeListener {

    /**
     * Méthode permettant la mise à jour de l'écouteur.
     * Elle sera appelée par la source de l'évènement lors de la
     * notification de changement d'état.
     *
     * Remarque: propertyName devrait être testé
     * @param evt
     */
    public void propertyChange(PropertyChangeEvent evt) {
        System.out.println("Event \"" +
            evt.getPropertyName()
            + "\" has changed");
    }
}
```

3.2 Une classe observable

L'« observable » dans ce cas est la **source de l'évènement**. Il doit maintenir une liste de ses écouteurs –dans un `Vector` par exemple— qu'il préviendra de tous changements via leur méthode `propertyChange`.

- Il a comme attribut un vecteur d'écouteurs
- Il possède deux méthode `addEventListener` et `removeEventListener` permettant l'ajout et la suppression d'un de ses écouteurs.
- Il possède une méthode `firePropertyChange` qui parcourera les écouteurs afin de les notifier d'un changement.

Une classe observable peut avoir l'allure suivante:

```
package pbt.listeners;

import java.beans.PropertyChangeEvent;
import java.util.Vector;

/**
 * ObservableNoExtendNoImplement.java
 *
 * Cette classe est observable, elle maintient une liste d'écouteurs
 * et a la possibilité de les informer de tout changement de son état.
 * @author pbt
 */
```

```
public class ObservableNoExtendNoImplement {
    /** Ma propriété ... */
    public static final String MAPROPRIETE = "Propriété sans nom";

    /** Liste de mes écouteurs */
    private Vector<Listener> listeners;

    public ObservableNoExtendNoImplement() {
        this.listeners = new Vector();
    }

    /**
     * Permet l'ajout d'un écouteur
     * @param listener écouteur à ajouter
     */
    public void addPropertyChangeListener(Listener listener) {
        listeners.add(listener);
    }

    /**
     * Permet la suppression d'un écouteur
     * @param listener écouteur à supprimer
     */
    public void removePropertyChangeListener(Listener listener) {
        listeners.remove(listener);
    }

    /**
     * Notifie le changement à tous les écouteurs inscrits.
     * @param propertie la source de l'évènement
     * @param oldValue l'ancienne valeur de la propriété
     * @param newValue la nouvelle valeur
     */
    public void firePropertyChange(
        Object propertie,
        Object oldValue,
        Object newValue) {
        for (Listener listener : listeners) {
            listener.propertyChange(
                new PropertyChangeEvent(
                    propertie,
                    MAPROPRIETE,
                    oldValue,
                    newValue));
        }
    }

    /**
     * Changement (académique) de l'état de cet objet
     */
    public void change() {
        System.out.println("Observable: I change");
        firePropertyChange("this", false, true);
    }
}
```

3.3 Une classe testant le tout

Pour tester le tout, il suffit d'instancier un `Listener` et un `ObservableNoExtendNoImplement` et d'ajouter le *listener* à la liste de l'observable. Un peu comme suit:

```
package pbt.listeners;

/**
 * Main.java
 *
 * Classe permettant de tester la notion de listener.
 * @author pbt
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        ObservableNoExtendNoImplement o =
            new ObservableNoExtendNoImplement();
        Listener l = new Listener();
        o.addPropertyChangeListener(l);
        o.change();
    }
}
```

3.4 Swing et Component

Cette approche peut paraître plus lourde à mettre en œuvre que la précédente puisqu'il faut tout faire soi-même du côté de l'observable⁶. Il s'avère que l'API `awt`⁷ s'en charge dans la classe `Component`.

Il suffit donc qu'une classe hérite de `Component` pour que ceci soit mis en œuvre ... et je brûle mon héritage ! Mais dans un contexte Swing, ce n'est pas grave puisque `Component` est parent de `JComponent` qui est parent de ... `JButton` par exemple. Et ainsi, tous mes composants Swing sont observables.

De même lorsque l'on parle de `PropertyChangeListener`, cette notion se généralise dans un contexte Swing à `ActionListener`, `MouseListener`, `WindowListener`, `KeyListener`, ... où l'on écrira, par exemple, des classes internes anonymes implémentant ces interfaces. Mais ceci est une autre histoire ...

3.5 Et la composition ?

Il reste une dernière manière de faire si l'on ne veut ni tout écrire à la main ni utiliser l'héritage (parce que l'on en a besoin par ailleurs ou parce que ça n'a pas de sens d'hériter de `Component` si l'on n'est pas dans un contexte graphique), c'est d'utiliser un **PropertyChangeSupport** par composition. Cela impose de réécrire les méthodes `add`, `remove` et `fire` mais ce sera le `PropertyChangeSupport` qui fera le travail.

⁶ Mais pour la compréhension des concepts, c'est mieux.

⁷ C'est un peu l'ancêtre de Swing, version composants lourds.