La programmation par évènements

Exercice I : Observer des objets (*Extrait du médian P09*)

1. Donner le code source de l'interface Observer.

```
package fr.utt.sit.lo02.median;

public interface Observer {
    public void update (Object o);
}
```

2. Donner le code source de la classe Observable.

```
package fr.utt.sit.lo02.median;

public class Observable {

    public final static int MAX_OBSERVERS = 10;

    private Observer[] observers;
    private int numberOfObservers;
    private boolean hasChanged;

    public Observable() {
      observers = new Observer [Observable.MAX_OBSERVERS];
      numberOfObservers = 0;
      hasChanged = false;
    }

    public void addObserver(Observer o) {
      if (numberOfObservers < Observable.MAX_OBSERVERS) {
          observers[numberOfObservers] = o;
          numberOfObservers++;
      } else {
          System.err.println("Error: max number of observers reached.");
      }
    }

    public void deleteObserver(Observer o) {
      boolean foundObserver = false;
      for (int i = 0; i < numberOfObservers; i++) {
          if (foundObserver == false) {
            if (observers[i].equals(o)) {
                observers[i] = null;
                foundObserver = true;
            }
          } else {
            observers[i - 1] = observers[i];
          }
      }

      if (foundObserver == true) {
          numberOfObservers--;
      }
    }

    public void notifyObservers(Object o) {

      if (hasChanged == true) {
```

```
            for (int i = 0; i < numberOfObservers; i++) {
              observers[i].update(o);
            }
            hasChanged = false;
        }
    }

    public void setChanged() {
      hasChanged = true;
    }

    public void clearChanged() {
      hasChanged = false;
    }
}
```

II. Application de l'observation à la classe Lampe

3. Donner le code source d'une Lampe observable.

```
package fr.utt.sit.lo02.median;

public class Lampe extends Observable {

    public final static int PUISSANCE_STANDARD = 100;

    private int puissance;
    private boolean allumee;

    public Lampe(int puissance) {
      this.puissance = puissance;
      allumee = false;
    }

    public int getPuissance() {
      return puissance;
    }

    public boolean isAllumee() {
      return allumee;
    }

    public void allumer() {
      if (this.allumee == false) {
          this.allumee = true;
          this.setChanged();
          this.notifyObservers("La lampe est allumée");
      }
    }

    public void eteindre() {
      if (this.allumee == true) {
          this.allumee = false;
          this.setChanged();
          this.notifyObservers("La lampe est éteinte");
      }
    }

    public String toString() {
      StringBuffer sb = new StringBuffer();
      sb.append("Lampe (");
```

```
        sb.append("Puissance : " + puissance);
        sb.append("\t");
        sb.append("Allumée : " + allumee);
        sb.append(")\n");
        return sb.toString();
    }
}
```

4. Donner le code source d'un observateur de Lampe.

```
package fr.utt.sit.lo02.median;

public class LampeObserver implements Observer {

    private String name;

    public LampeObserver(String name) {
      this.name = name;
    }

    public void update(Object o) {
      System.out.println(name + " : " + o);
    }

    public static void main(String[] args) {

      Lampe l = new Lampe(Lampe.PUISSANCE_STANDARD);
      LampeObserver lo1 = new LampeObserver("Observer 1");
      LampeObserver lo2 = new LampeObserver("Observer 2");
      l.addObserver(lo1);
      l.addObserver(lo2);

      l.allumer();
      l.allumer();
      l.eteindre();
      l.eteindre();

      l.deleteObserver(lo1);
      l.allumer();
      l.eteindre();

    }
}
```

III. Extension du principe d'observation

5. La classe Observable possède une méthode `setChanged()` qui permet d'indiquer qu'un changement a effectivement eu lieu dans l'objet. Un appel à `notifyObservers` notifira alors les observateurs si et seulement si un changement a été signalé par le biais d'un appel préalable à `setChanged()`. Redonner le code de classe Observable avec ces nouveaux paramètres (inutile de redonner le code des méthodes qui ne changent pas).

Correction déjà intégrée dans la question 2.

6. Il possible de passer des informations relatives au changement - quelle que soit sa nature - d'un objet observé à ses observateurs. Pour cela, la méthode `notifyObservers` reçoit cet objet en paramètre et le transmet à la méthode `update` dans un paramètre aussi. Donner le code source de la méthode `notifyObservers` et la définition de la méthode `update` pour le support d'indications sur le changement.

Correction déjà intégrée dans la question 2.

IV Analyse et critique de la conception

7. Expliquer pourquoi utiliser une interface pour représenter un observateur ?

La capacité d'observateur est un aspect particulier d'une classe qui ne présume en rien de la fonction initiale de la classe. De plus, le comportement d'un observateur peut varier d'un observateur à un autre, donc le corps de sa méthode `update` en peut être connu à l'avance. Pour ces deux raisons, un observateur doit être une interface.

8. Expliquer pourquoi utiliser une classe pour représenter un objet observable ?

Un objet observable doit pouvoir notifier ses observateurs. Il doit définir et implémenter les méthodes de gestion de ses observateurs. Il doit donc être une classe.

9. Pourquoi utiliser une méthode `setChanged` en plus de la méthode `notifyObservers` ?

Indiquer qu'un changement a eu lieu est différent de notifier ses observateurs. L'indication du changement n'est connue que de la classe observable, par contre la notification est diffusée à tous les observateurs. L'intérêt de ces deux méthodes consiste justement à pouvoir séparer ces deux opérations et notamment pouvoir différer la notification aux observateurs.

10. Donner toutes les limites de cette conception.

La principale limite de cette conception réside dans l'utilisation d'une classe pour représenter un objet observable et ce pour deux raisons : étant donné l'héritage simple en Java, un objet observable ne pourra hériter d'une autre classe, ce qui sera souvent problématique. Ensuite, l'implémentation de la classe observable est figée et ne peut être changée par le développeur pour s'adapter à un comportement de notification particulier. Utiliser une interface aurait permis d'éviter ces deux inconvénients.

Exercice 2 : Vue graphique d'un compteur

Implanter les classes permettant d'implanter un compteur graphique tel qu'il a été vu en cours (avec le modèle MVC). Etudier l'impact des threads sur le fonctionnement de l'interface.

```
package fr.utt.sit.lo02.swing.compteur;

public class CompteurThread extends Compteur implements Runnable {

    public CompteurThread () {
        super();
    }

    public void compter() {
        Thread t = new Thread(this, "Compteur");
        t.start();
    }

    public void run() {
        while (true) {
                compteur++;
                setChanged();
                notifyObservers();
```

```
                attendre();
            }
        }

        public static void main(String[] args) {

            CompteurThread c = new CompteurThread();
            VueCompteur vue = new VueCompteur(c);
            c.addObserver(vue);
        }
}
```

```
package fr.utt.sit.lo02.swing.compteur;

import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.util.Observer;
import java.util.Observable;

class VueCompteur implements Observer {

    private Compteur compteur;
    private JFrame fenetre;
    private JLabel texte;
    private JButton demarrer;

    public VueCompteur (Compteur c) {
        compteur = c;

        fenetre = new JFrame ("Un compteur");
        Container reservoir = fenetre.getContentPane();

        texte = new JLabel("Compteur : " + c.getValeur());
        reservoir.add(texte, BorderLayout.NORTH);

        demarrer = new JButton("Demarrer");
        reservoir.add(demarrer, BorderLayout.SOUTH);

        // Controleur du bouton
        demarrer.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    compteur.compter();
                }
            });

        fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fenetre.pack();
        fenetre.setVisible(true);
    }

    public void update(Observable o, Object arg) {
        texte.setText("Compteur : " + compteur.getValeur());
    }
}
```