TD 9 : Design Pattern Observer

Objectifs pédagogiques :

- ***design pattern* Observateur**
- **liaison d'observateurs**

Dans cet exercice nous allons développer une API permettant d'avoir des variables observables. Ainsi, tout objet observateur peut s'abonner à ce type de variable et ensuite être notifié lorsque cette dernière change de valeur. Dans un premier temps, nous allons programmer le mécanisme générique permettant à n'importe quel objet observable de gérer une liste d'objets observateurs. Dans un second temps, nous définirons le code d'une variable observable. Enfin, nous traiterons la notion de liaison (*binding*) permettant de définir un graphe de dépendance d'observation entre plusieurs variables observables.

Voici deux interfaces `IObservable<T>` et `IObserver<T>` qui spécifient respectivement les méthodes que doit offrir un objet observable et un objet observateur. La variable de type `T` spécifie le type de la notification lorsque l'observable change d'état.

```java
package pobj.observer;

import java.util.Collection;

public interface IObservable<T> {
   void addObserver(IObserver<T> o);
   Collection<IObserver<T>> getObservers();
   void deleteObservers();
}
```

```java
package pobj.observer;

public interface IObserver<T> {
   void update(IObservable<T> o, T oldval, T newval);
}
```

Question 1. Donner le code de la classe abstraite `AbstractObservable<T>` qui implante l'ensemble des méthodes offertes par l'interface `IObservable<T>` et qui permet de gérer la liste des observateurs.

Question 2. Écrire la classe `ObservableVariable<T>` qui permet d'avoir une variable observable de type `T`. Cette classe offrira les méthodes `T getValue()` et `setValue(T newvalue)` qui permettent respectivement de lire le contenu de la variable et de changer sa valeur. Chaque changement de valeur doit donner lieu à une notification des observateurs.

Question 3. En utilisant la classe `ObservableVariable<T>` de la question précédente, écrire une classe `StringAffichable` qui affiche sa valeur sur la console avec un numéro de version incrémenté à chaque fois que la valeur change. Par exemple le programme ci-dessous :

```java
StringAffichable s = new StringAffichable("");
s.setValue("bonjour");
s.setValue("foo");
s.setValue("foo");
s.setValue("bar");
s.setValue("bar");
```

doit afficher :

```
Version 1 : bonjour
Version 2 : foo
Version 3 : bar
```

Nous souhaitons maintenant ajouter la méthode `bind(ObservableVariable<T> other)` à la classe `ObservableVariable` permettant de lier la valeur de l'instance à la valeur de la variable `other`. Ainsi, la liaison (*binding*) assure que la valeur de l'instance est toujours égale à la valeur de `other`. En revanche la liaison ne se fait ici que dans un sens : si on change la valeur de l'instance, la valeur de `other` n'est en rien modifiée. Une variable peut être liée au plus une fois. Ainsi la méthode `bind` renvoie un booléen qui vaut vrai si la liaison réussit (l'instance n'était pas liée à une variable), faux sinon.

Question 4. Donner le code de la méthode `bind`

Soit le programme suivant :

```
ObservableVariable<Integer> a = new ObservableVariable<Integer>(2);
ObservableVariable<Integer> b = new ObservableVariable<Integer>(5);
ObservableVariable<Integer> c = new ObservableVariable<Integer>(10);

a.bind(b);
b.bind(c);
c.bind(a);

a.setValue(-1);
```

Question 5. Est-ce que le fait d'avoir un cycle dans les dépendances pose problème ?

Question 6. Sur le même principe que la méthode `bind`, donner le code de la méthode `boolean bindBidirectionnal(ObservableVariable<T> other)` qui permet de lier deux variables dans les deux sens. La double liaison est valide si aucune des deux variables n'utilise déjà son lien.

Question 7. Donner le code de la méthode `boolean unbind(ObservableVariable<T> other)` qui permet de supprimer unidirectionnellement le lien entre l'instance et `other`. La méthode renvoie faux si l'instance n'est pas reliée à `other` ou bien si elle n'utilise pas son lien, vrai sinon.

Nous souhaitons à présent enrichir la notion de liaison en permettant à une variable entière d'avoir une valeur qui dépend non pas d'une, mais d'un ensemble de variables entières à travers une fonction arithmétique. Ici, nous allons programmer une classe `ArithmeticVariable` qui est une `ObservableVariable<Integer>` et qui offre la méthode `sum`. La méthode `sum` prend un nombre quelconque d'`ArithmeticVariable` en paramètre et assure que la valeur de l'instance est égale à la somme de toutes les valeurs des variables passées en paramètre. Voici un exemple de test qui illustre la spécification de cette classe :

```
ArithmeticVariable a = new ArithmeticVariable(0);
ArithmeticVariable b = new ArithmeticVariable(1);
ArithmeticVariable c = new ArithmeticVariable(10);
ArithmeticVariable d = new ArithmeticVariable(100);

d.sum(a, b, c);
assertEquals(11, d.getValue().intValue());

a.setValue(10);
assertEquals(21, d.getValue().intValue());
```

Question 8. Donner le code de la classe `ArithmeticVariable`