Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2020-2021**

TD – Séance n°12

Design Patterns

Observateurs et observés

On considère une classe `Pseudo` qui contient simplement une chaîne de caractères. On souhaite définir des objets `ChangePseudoHistory` qui retiendront l'ensemble des changements qui ont eu lieu sur certains pseudos (anciennes et nouvelles valeurs) qu'ils observent. Le fonctionnement attendu peut se comprendre sur l'exemple suivant :

```
1  public class Test {
     public static void main(String[] args) {
3      // partie declarative
       Pseudo a = new Pseudo("a");
5      Pseudo b = new Pseudo("b");
       Pseudo c = new Pseudo("c");
7      Pseudo d = new Pseudo("d");

9      ChangePseudoHistory h_ab = new ChangePseudoHistory();
       ChangePseudoHistory h_bc = new ChangePseudoHistory();

       // parametrages a faire pour que
13     // h_ab conserve l'historique des modifications de a et b
       // h_bc conserve l'historique des modifications de b et c
15     // ...
       // ...

       // modifications
19     a.set("a1");
       b.set("b1");
21     a.set("a2");
       c.set("c1");
23     d.set("d1");

25     // affichage des historiques observes
       System.out.println(h_ab);
27     System.out.println(h_bc);
     }
29 }
```

qui doit produire :

```
a-->a1
b-->b1
a1-->a2

b-->b1
c-->c1
```

Exercice 1 Proposez une solution avec vos connaissances en programmation.

Exercice 2 Faites-la évoluer pour ne plus prendre en compte de simples chaînes de caractères, mais un type générique qui serait encapsulé dans les pseudos au lieu de cette simple chaîne.

Exercice 3 En fait, le design pattern Observateur/Observé est tellement fréquent que les développeurs Java ont mis à disposition des éléments dans l'API pour aider à l'implémenter rapidement.

Dans cet exercice, on va adapter l'exemple des pseudos pour qu'il utilise les objets Observer et Observable comme ils sont définis dans `java.util`. Vous devez donc décrypter l'API et l'utiliser.

On va utiliser seulement un extrait de l'API de la classe `Observable`. Voilà une introduction générale :

Introduction.
- The Observable class represents an observable object.
- It can be subclassed to represent an object that the application wants to have observed.
- An observable object can have one or more observers.
- An observer may be any object that implements interface Observer.
- After an observable instance changes, an application calling the Observable's notifyObservers method causes all of its observers to be notified of the change by a call to their update method.
- The default implementation provided in the Observable class will notify Observers in the order in which they registered interest.
- When an observable object is newly created, its set of observers is empty.

Et voilà les méthodes de la classe `Observable` qu'on va utiliser :

Methods.
- `Observable()` : Construct an `Observable` with zero Observers.
- `void addObserver(Observer o)` : Adds an observer to the set of observers for this object.
- `void setChanged()` : Marks this `Observable` object as having been changed.
- `boolean hasChanged()` : Tests if this object has changed.
- `void clearChanged()` : Indicates that this object has no longer changed, or that it has already notified all observers of its most recent change, so that the `hasChanged` method will now return `false`.
- `void notifyObservers(Object arg)` : If this object has changed, according to `hasChanged`, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.

L'interface `Observer` définit une seule méthode :

Introduction. A class can implement the Observer functionnal interface when it wants to be informed of changes in observable objects.

Methods.
`void update(Observable o, Object arg)`
This method is called whenever the observed object is changed. An application calls an Observable object's notifyObservers method to have all the object's observers notified of the change.
Parameters :
- `o` – the observable object.
- `arg` – an argument passed to the notifyObservers method.

Exercice 4 En tant que développeuse ou développeur, vous devriez être capable de créer des classes et interfaces équivalentes au couple `Observer`/`Observable` de l'API de Java. Faites ce travail en définissant des classes ou interfaces `MonObserver`/`MonObservable` qu'on pourrait substituer au couple précédent.

Proxy

Exercice 5

1. Créez une implémentation de l'interface `Usine` ci-dessous, nommée `MonUsine`. **Attention : Le but de cet exercice est qu'on ne fait plus aucune modification à la classe `MonUsine` par la suite.**

Usine.java

```
1  public interface Usine {
2
3    void setName(String name);
4
5    String getName();
6
7    void setAddress(String address);
8
9    String getAddress();
   }
```

2. Créez la classe `UsineFactory` ayant une méthode `Usine createUsine()` permettant de créer une usine.
3. Ajoutez une méthode `setDebugMode(boolean)` à la classe `UsineFactory` permettant d'indiquer que cette instance de `UsineFactory` passe en mode débug ou non. Une usine créée par une `UsineFactory` en mode débug affichera, lors de la modification d'un attribut par une méthode `setXXX`, un message indiquant la variable changée, son ancienne valeur et sa nouvelle valeur.

   On rappelle que vous ne pouvez plus faire aucune modification à la classe `MonUsine`, il vous faut donc créer une nouvelle classe `UsineProxy` qui s'occupe de faire travailler une usine.

   (Si vous voulez : À la limite, vous pouvez concevoir cette classe de façon à ce qu'elle puisse fonctionner pour d'autres implémentations que `MonUsine`.)

   Votre réflexion devrait vous conduire à utiliser, sans le connaître, le design pattern «proxy».