TD – Séance nº 12 - Correction

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 :

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

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

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

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

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

qui doit produire :

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

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

Exercice 1 Proposez une solution avec vos connaissances en programmation.

Correction :

Pseudo.java

```java
import java.util.*;

public class Pseudo {
  private String content;
  private List<ChangePseudoHistory> observateurs;

  public Pseudo(String x) {
    content = x;
    observateurs = new LinkedList<>();
  }

  public void addObservateurs(ChangePseudoHistory... h_tab) {
    for (ChangePseudoHistory h : h_tab) observateurs.add(h);
  }

  public void set(String y) {
    for (ChangePseudoHistory x : observateurs) x.infoChange(content, y)
        ;
    content = y;
  }
}
```

ChangePseudoHistory.java

```java
import java.util.*;

public class ChangePseudoHistory {
  // classe interne
  private static class Pair {
    private String before;
    private String after;

    Pair(String x, String y) {
      before = x;
      after = y;
    }
  }

  List<Pair> modifications = new LinkedList<Pair>();

  public void infoChange(String x, String y) {
    modifications.add(new Pair(x, y));
  }

  public String toString() {
    String rep = "";
    for (Pair p : modifications) rep += p.before + "-->" + p.after + "\
        n";
    return rep;
  }
}
```

Parametrage dans le main

```
a.addObservateurs(h_ab);
b.addObservateurs(h_ab, h_bc);
c.addObservateurs(h_bc);
```

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.

Correction : En gros, il suffit de mettre des génériques partout. Seul problème : la méthode avec un nombre variable d'arguments n'est pas sûre (si on compile avec -Xlint, il y a un message d'erreur). Donc on utilise une méthode avec un seul argument.

Pseudo.java

```
import java.util.*;

public class Pseudo<T> {
  private T content;
  private List<ChangePseudoHistory<T>> observateurs;

  public Pseudo(T x) {
    content = x;
    observateurs = new LinkedList<>();
  }

  public void addObservateur(ChangePseudoHistory<T> h) {
    observateurs.add(h);
  }

  public void set(T y) {
    for (ChangePseudoHistory<T> x : observateurs) x.infoChange(content,
        y);
    content = y;
  }
}
```

ChangePseudoHistory.java

```
import java.util.*;

public class ChangePseudoHistory<T> {
  // classe interne
  private static class Pair<T> {
    private T before;
    private T after;

    Pair(T x, T y) {
      before = x;
      after = y;
    }
  }

  List<Pair<T>> modifications = new LinkedList<>();

  public void infoChange(T x, T y) {
```

```
    modifications.add(new Pair<T>(x, y));
  }

  public String toString() {
    String rep = "";
    for (Pair<T> p : modifications) rep += p.before + "-->" + p.after +
        "\n";
    return rep;
  }
}
```

Main.java

```
public class Main {
  public static void main(String[] args) {
    // partie declarative
    Pseudo<String> a = new Pseudo<>("a");
    Pseudo<String> b = new Pseudo<>("b");
    Pseudo<String> c = new Pseudo<>("c");
    Pseudo<String> d = new Pseudo<>("d");

    ChangePseudoHistory<String> h_ab = new ChangePseudoHistory<>();
    ChangePseudoHistory<String> h_bc = new ChangePseudoHistory<>();

    // parametrages a faire pour que
    // h_ab conserve l'historique des modifications de a et b
    // h_bd conserve l'historique des modifications de b et c
    a.addObservateur(h_ab);
    b.addObservateur(h_ab);
    b.addObservateur(h_bc);
    c.addObservateur(h_bc);

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

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

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. Il s'agit de la classe/interface `Observable/Observer`.

N.B. Remarquez que ces classes sont obsoletes depuis Java 9, car des modèles plus complexes de gestions des événements ont été mis en place, pour des applications avancées. Toutefois à but pédagogique il est utile d'étudier ces classes car leur simplicité permet de bien comprendre l' "Observer pattern".

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.

Correction :

Pseudo.java

```
1  import java.util.*;
2
3  public class Pseudo<P> extends Observable {
4    private P content;
5
```

```
  public Pseudo(P x) {
    content = x;
  }

  public P getContent() {
    return content;
  }

  public void set(P y) {
    P old = content;
    content = y;
    setChanged();
    notifyObservers(old);
  }
}
```

ChangePseudoHistory.java

```
import java.util.*;

public class ChangePseudoHistory<P> implements Observer {
  // classe interne
  private static class Pair<P> {
    private P before;
    private P after;

    Pair(P x, P y) {
      before = x;
      after = y;
    }
  }

  List<Pair<P>> modifications = new LinkedList<Pair<P>>();

  public void update(Observable o, Object arg) {
    // Nous ne pouvons pas utiliser instanceof pour des types géné
       riques,
    // donc on laisse échouer le type cast et capture l'exception.
    try {
      Pseudo<P> o_cast = (Pseudo<P>) o;
      P arg_cast = (P) arg;
      modifications.add(new Pair<P>(arg_cast, o_cast.getContent()));
    } catch (ClassCastException e) {
      throw new IllegalArgumentException();
    }
  }

  public String toString() {
    String rep = "";
    for (Pair<P> p : modifications) {
      rep += p.before + "-->" + p.after + "\n";
    }
    return rep;
  }
}
```

Main.java

```java
public class Main {
  public static void main(String[] args) {
    // partie declarative
    Pseudo<String> a = new Pseudo<>("a");
    Pseudo<String> b = new Pseudo<>("b");
    Pseudo<String> c = new Pseudo<>("c");
    Pseudo<String> d = new Pseudo<>("d");

    ChangePseudoHistory<String> h_ab = new ChangePseudoHistory<String
        >();
    ChangePseudoHistory<String> h_bc = new ChangePseudoHistory<String
        >();

    // parametrages a faire pour que
    // h_ab conserve l'historique des modifications de a et b
    // h_bd conserve l'historique des modifications de b et c
    a.addObserver(h_ab);
    b.addObserver(h_ab);
    b.addObserver(h_bc);
    c.addObserver(h_bc);

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

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

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.

Correction : On change le début des classes précédentes en :
`public class Pseudo extends MonObservable`
et
`public class ChangePseudoHistory implements MonObservateur`
et il faut aussi adapter la signature de la méthode `update` :
`public void update(MonObservable o, Object arg) {`

MonObservable.java

```java
import java.util.*;

public abstract class MonObservable {
  List<MonObservateur> observateurs = new LinkedList<>();
  boolean aChange = false;

```

```
  void setChanged() {
    aChange = true;
  }

  boolean hasChanged() {
    return aChange;
  }

  void clearChanged() {
    aChange = false;
  }

  void addObserver(MonObservateur o) {
    observateurs.add(o);
  }

  void notifyObservers(Object param) {
    for (MonObservateur o : observateurs) o.update(this, param);
    clearChanged();
  }
}
```

MonObservateur.java

```
public interface MonObservateur {
  void update(MonObservable o, Object arg);
}
```

Proxy

Exercice 5

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

   Usine.java

   ```
   public interface Usine {
     void setName(String name);
     String getName();
     void setAddress(String address);
     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 faudra donc créer une nouvelle classe `UsineProxy` qui s'occupera de faire travailler

en mode “debug” toute instance de `MonUsine`, et plus généralement toute implémentation de l'interface `Usine`.

Correction :

Main.java

```
public class Main {
  public static void main(String[] args) {
    UsineFactory f = new UsineFactory();
    Usine u1 = f.createUsine();
    u1.setName("voila");

    f.setDebugMode(true);
    Usine u2 = f.createUsine();
    u2.setName("quoi");
    u2.setName("qui");
  }
}
```

MonUsine.java

```
public class MonUsine implements Usine {
  String name;
  String adresse;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setAddress(String address) {
    this.adresse = address;
  }

  public String getAddress() {
    return adresse;
  }
}
```

UsineFactory.java

```
public class UsineFactory {
  boolean debug = false;

  public Usine createUsine() {
    if (!debug) return new MonUsine();
    return new UsineProxy(new MonUsine());
  }

  public void setDebugMode(boolean b) {
    debug = b;
  }
}
```

UsineProxy.java

```
public class UsineProxy implements Usine {
  Usine u;

  public UsineProxy(Usine u) {
    this.u = u;
  }

  public void setName(String name) {
    System.out.println("Ancien nom " + getName());
    u.setName(name);
    System.out.println("Nouveau nom " + getName());
  }

  public String getName() {
    return u.getName();
  }

  public void setAddress(String address) {
    System.out.println("Ancienne adresse " + getAddress());
    u.setAddress(address);
    System.out.println("Nouvelle adresse " + getAddress());
  }

  public String getAddress() {
    return u.getAddress();
  }
}
```