

Theme : Design Pattern Observer

Presente par :
Pape Amadou SAKHO
Khardiatou BASSE
Marieme AIDARA

- Introduction
- Définition et principe
- Cas d'utilisation
- Mise en œuvre
- Conclusion



Introduction

Un patron de comportement permet de résoudre les problèmes liés aux comportements, à l'interaction entre les classes.

Les différents patrons de comportement sont les suivants : Chaine de responsabilité, Commande, Interpréteur, Itérateur, Médiateur, Memento, Etat, Stratégie, Patron de méthode, Visiteur et Observateur dont se portera notre travail qui permet d'intercepter un évènement pour le traiter.



Definition et Principe(1)

Observer est un patron de conception qui permet de mettre en place un mécanisme de souscription pour envoyer des notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent. L'objet que l'on veut suivre est en général appelé sujet ou observable, mais comme il va envoyer des notifications pour prévenir les autres objets dès qu'il est modifié, nous l'appellerons diffuseur. Tous les objets qui veulent suivre les modifications apportées au diffuseur sont appelés des souscripteurs ou observateurs. Observer nous propose d'ajouter un mécanisme de souscription à la classe diffuseur pour permettre aux objets individuels de s'inscrire ou se désinscrire de ce diffuseur.



Definition et Principe(2)

Les notions d'observateur et d'observable permettent de limiter le couplage entre les modules aux seuls phénomènes à observer. Le patron permet aussi une gestion simplifiée d'observateurs multiples sur un même objet observable. Dans la pratique, quand un événement important arrive au diffuseur, il fait le tour de ses souscripteurs et appelle la méthode de notification sur leurs objets. Les applications peuvent comporter des dizaines de classes souscripteur différentes qui veulent être tenues au courant des événements qui affectent une même classe diffuseur.

Cas d'utilisation

L'Observer design pattern est tout particulièrement utile dans les applications basées sur des composants dont le statut est

- d'une part, fortement observé par les autres composants
- d'autre part, soumis à des modifications régulières.

Probleme pose

Nous mettons en place une application de conversion en plusieurs bases. Le principe de notre application c'est de convertir automatiquement un nombre entré en base décimale par l'utilisateur en base binaire, base octale et base hexadécimale.

Et nous devons disposer d'un convertisseur pour chacune de ses bases. Notre problème est comment signaler automatiquement à ces trois convertisseurs qu'il y a un nouveau nombre à convertir pour qu'il effectue la conversion.

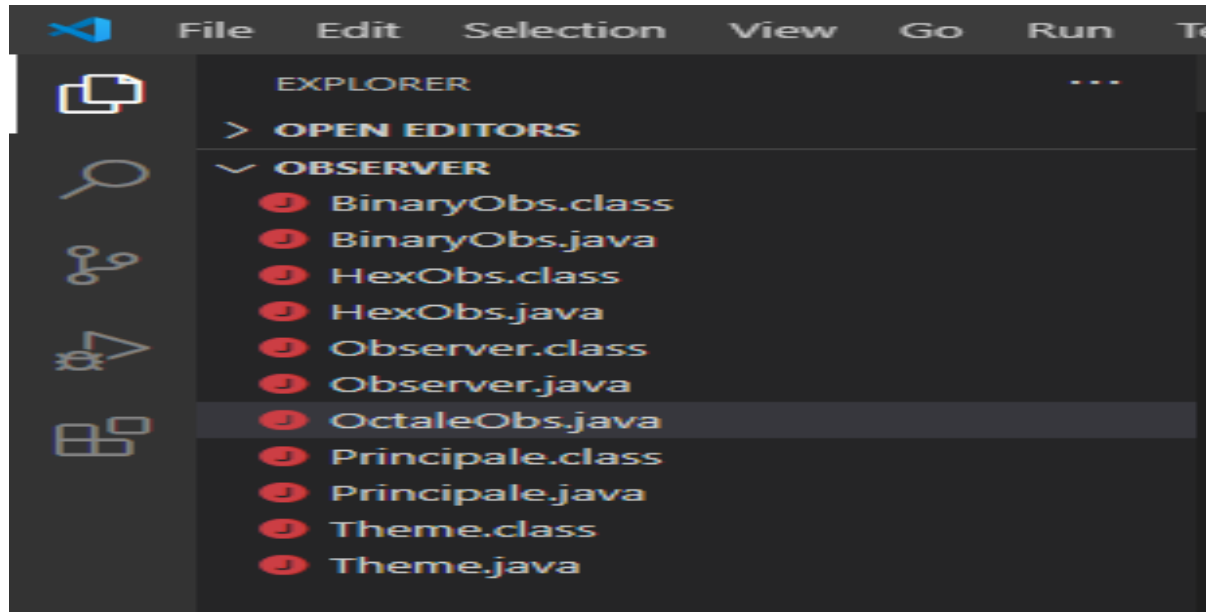
Solution

La réalisation de cette application nécessite de créer une classe abstraite `Observer` contenant une méthode `modifier ()`, dont vont héritées les trois concrètes `BinaryObs`, `OctaleObs` et `HexObs` qui vont définir la méthode `modifier ()`.

On disposera aussi d'une classe **Theme** qui va avoir tous ces observateurs dans un tableau, qui sera notre observable et qui notifiera tous ces convertisseurs c'est-à-dire les observateurs de l'arrivée d'un nouveau nombre entré à convertir.

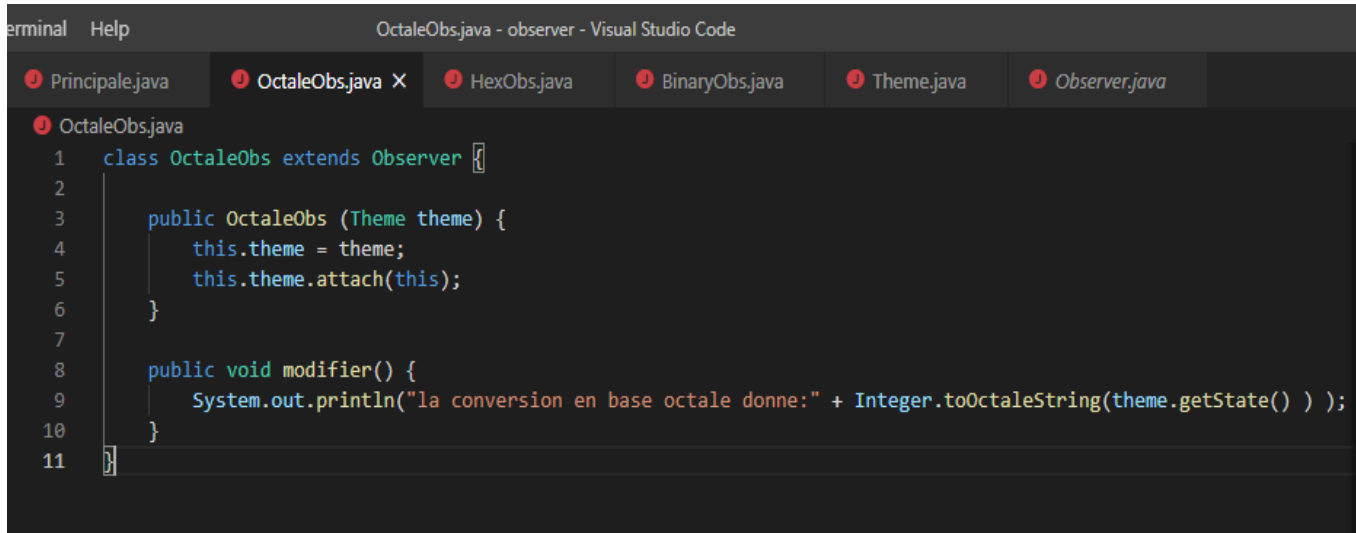
Mise en oeuvre (3)

On note les différentes classes dont on dispose



Mise en oeuvre (4)

On note la classe `OctaleObs.java` qui est un observateur. Il est encore appelé souscripteur. Cette observateur s'inscrit ou se désinscrit de l'observable (classe `Theme.java`). Cette classe reçoit une alerte lorsque la variable au niveau l'observable change, donc à chaque fois qu'une modification se fait au niveau de l'observable les observateurs sont modifiés.



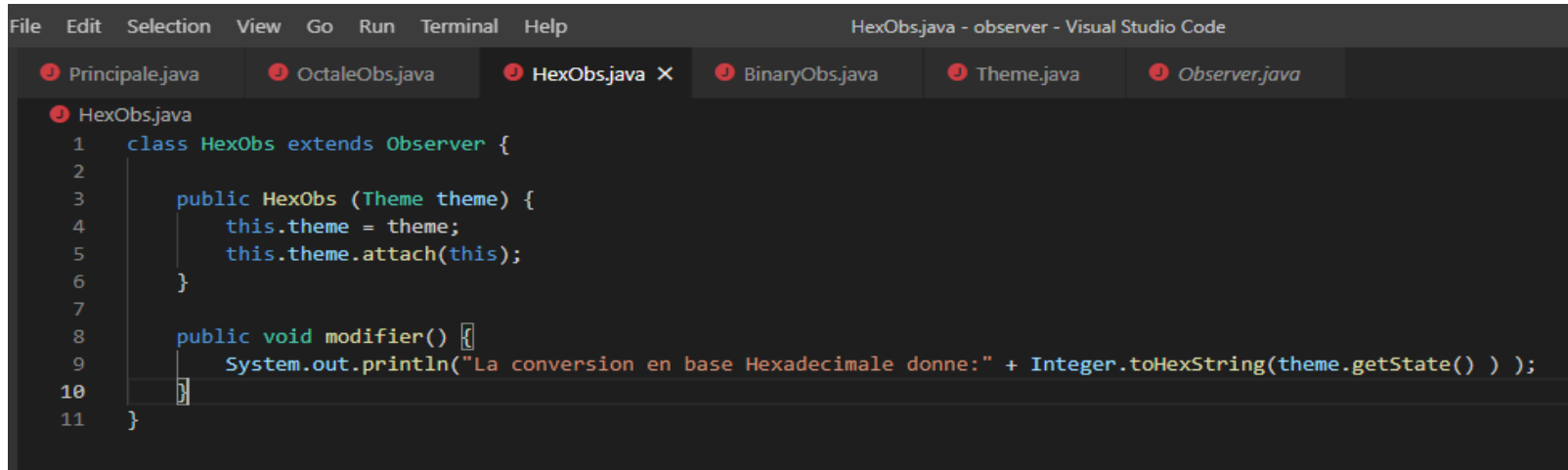
```
terminal  Help  OctaleObs.java - observer - Visual Studio Code

Principale.java  OctaleObs.java X  HexObs.java  BinaryObs.java  Theme.java  Observer.java

OctaleObs.java
1  class OctaleObs extends Observer {
2
3      public OctaleObs (Theme theme) {
4          this.theme = theme;
5          this.theme.attach(this);
6      }
7
8      public void modifier() {
9          System.out.println("la conversion en base octale donne:" + Integer.toOctalString(theme.getState() ) );
10     }
11 }
```

Mise en oeuvre (5)

On note la classe HexObs.java qui aussi est un observateur. Il est encore appele souscripteur. Cette observateur s'inscrit ou ce desinscrit de l'observable (classe Theme.java). Cette classe recoit une alerte lorsque la variable au niveau l'observable change, donc a chaque fois qu'une modification se fait au niveau de l'observable les observateurs sont modifies.



```
File Edit Selection View Go Run Terminal Help
HexObs.java - observer - Visual Studio Code

Principale.java OctaleObs.java HexObs.java X BinaryObs.java Theme.java Observer.java

HexObs.java
1 class HexObs extends Observer {
2
3     public HexObs (Theme theme) {
4         this.theme = theme;
5         this.theme.attach(this);
6     }
7
8     public void modifier() {
9         System.out.println("La conversion en base Hexadecimale donne:" + Integer.toHexString(theme.getState() ) );
10    }
11 }
```

Mise en oeuvre (6)

On note la classe BinaryObs.java qui est aussi un observateur. Il est encore appelé souscripteur. Cette observateur s'inscrit ou se désinscrit de l'observable (classe Theme.java). Cette classe reçoit une alerte lorsque la variable au niveau observable change, donc à chaque fois qu'une modification se fait au niveau de l'observable les observateurs sont modifiés.



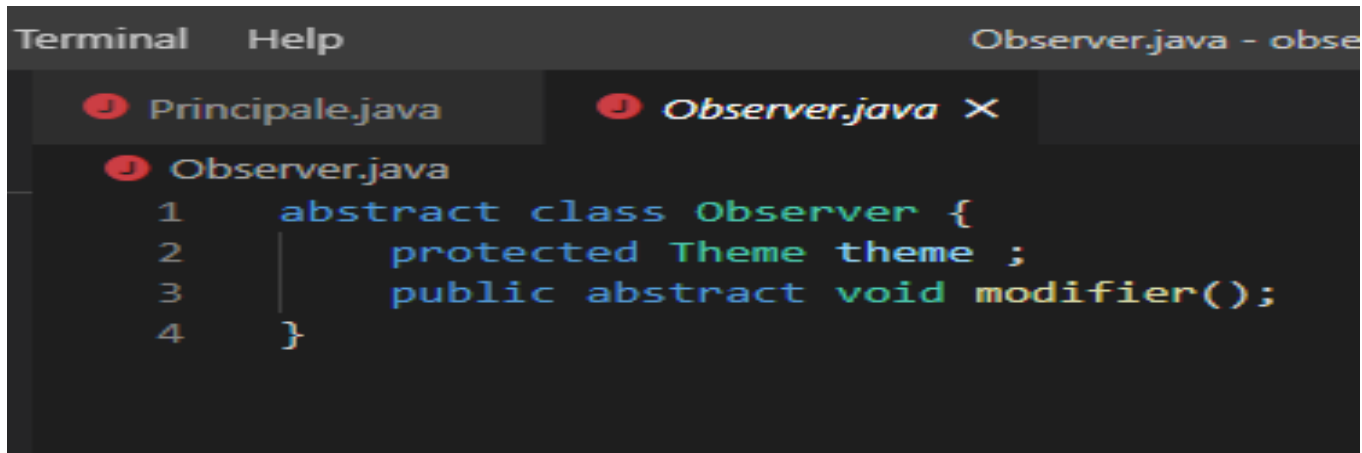
```
File Edit Selection View Go Run Terminal Help
BinaryObs.java - observer - Visual Studio Code

Principale.java OctaleObs.java HexObs.java BinaryObs.java X Theme.java Observer.java

BinaryObs.java
1 class BinaryObs extends Observer {
2
3     public BinaryObs( Theme theme) {
4         this.theme = theme; //
5         this.theme.attach(this);
6     }
7
8     public void modifier() {
9         System.out.println("La conversion en base binaire donne:" + Integer.toBinaryString(theme.getState() ) );
10    }
11 }
```

Mise en oeuvre (7)

On note la classe `Observer.java` qui est une classe abstraite, par composition a une classe observable qui joue souvent le role de sujet represente ici par la classe `Theme.java`. Elle implemente la methode `notifyAllObserver`, comme cest par composition, l'attribut migre. Si la classe `Theme` note un evenement, et la variable `theme` change au niveau de l'observable alors les autres observateurs sont au courant. Donc a chaque fois qu'une modification se fait au niveau de l'observable les observateurs sont modifies



```
Terminal  Help  Observer.java - observeur

Principale.java  Observer.java X

Observer.java
1  abstract class Observer {
2      protected Theme theme ;
3      public abstract void modifier();
4  }
```

Mise en oeuvre (8)

La classe Theme.java designe l'observable dispose d'une methode notifyAllObservers qui notifie les observateurs. Si l'observable a un evenement, la methode notifyAllObservers le lui signal automatiquement

```
1  import java.util.*;
2  import java.util.List;
3  import java.util.ArrayList;
4
5  class Theme {
6      private List<Observer> observers = new ArrayList<Observer>();
7      private int state;
8
9      public int getState() {
10         return state;
11     }
12
13     public void setState(int state) {
14         this.state = state;
15         notifyAllObservers();
16     }
17
18     public void attach (Observer observer) {
19         observers.add(observer);
20     }
21
22     public void notifyAllObservers() {
23         for(Observer observer : observers) {
24             observer.modifier();
25         }
26     }
27 }
28
```


Mise en oeuvre (9)

classe Principale.java

```
Terminal  Help  Principale.java - observer - Visual Studio Code

Principale.java X  Observer.java

Principale.java
1  import java.util.*;
2  import java.util.Scanner;
3
4  class Principale {
5      public static void main(String[] args) {
6          Scanner sc = new Scanner(System.in);
7          Theme theme = new Theme();
8
9          new BinaryObs(theme);
10         new OctaleObs(theme);
11         new HexObs(theme);
12
13         System.out.println("*****CONVERSION*****");
14         int number;
15         System.out.println();
16
17         for (int i=0; i<3; i++) {
18             System.out.println("*****le nombre a convertir*****");
19             number = sc.nextInt();
20             theme.setState(number);
21             System.out.println();
22         }
23     }
24 }
```



Conclusion

En résumé, le pattern observateur permet à un objet d'observer le changement d'état d'un autre objet et d'agir en conséquence.

Nous devons donc disposer d'un objet portant des informations d'état dont les autres objets doivent avoir connaissances. Lorsque ces informations d'état changent, tous ces objets doivent changer.