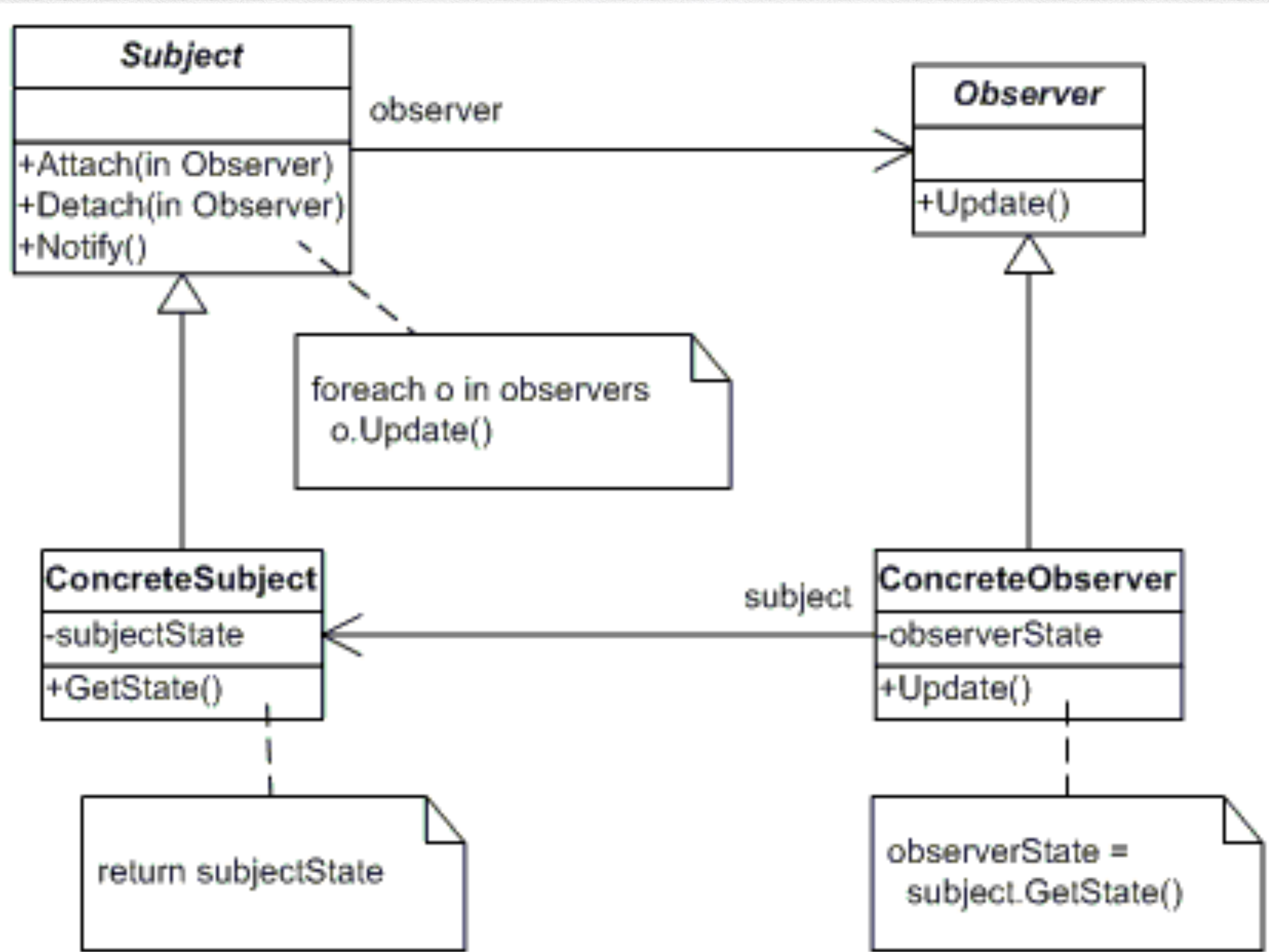
Design Pattern Observer

«Pattern Observer» : le problème

- ✓Problème : Mettre en oeuvre une relation de «un vers plusieurs» objets afin que plusieurs objets puissent être notifiés du changement d'état d'un objet et puisse réagir.

Il est très utilisé en IHM, mais peut être appliqué dans bien d'autres cas.

Pattern Observer : Solution



«Pattern Observer» : les rôles

✓ Rôles : Un sujet et des «observers» (le sujet doit devenir «observable»)

✓ Responsabilités :

➡ Le sujet :

- notifie les observeurs quand il «change»
- permet aux observeurs de s'(de-)enregistrer.

➡ Les observeurs

- acceptent les notifications

Pattern Observer : Solution

- **Sujet Abstrait** (Observable) :
  - gère les observeurs (*AddObserver(Observer)*)
  - notifie les observeurs *(notifyObservers)*

- **Sujet (Observable) :**
  - A chaque changement d'état, il «notifie» les observeurs (*notify*)
  - Il peut donner son état (*getState*)

- **Observeur**
  - Se met à jour quand il est notifié (update)

- **Observeur (Abstrait)**
  - Peut être notifié (*update*)

«Pattern Observer» et autres en action

- ✓Un forum
  - ➡On peut poster des messages dans le forum : un message à un titre.
- ✓Des messages
  - ➡Certains messages sont des alertes, d'autres des annonces,…
- ✓Des abonnés
  - ➡Dès qu'un message est posté sur le forum, tous les abonnés sont notifiés.
  - ➡Les abonnés enregistrent le message dans leur boite « inbox »
  - ➡Certains abonnés VIP filtrent les messages reçus en fonction du titre et les affectent aux boîtes.

«Pattern Observer» en java

- Le sujet abstrait : classe abstraite **java.util.observable**

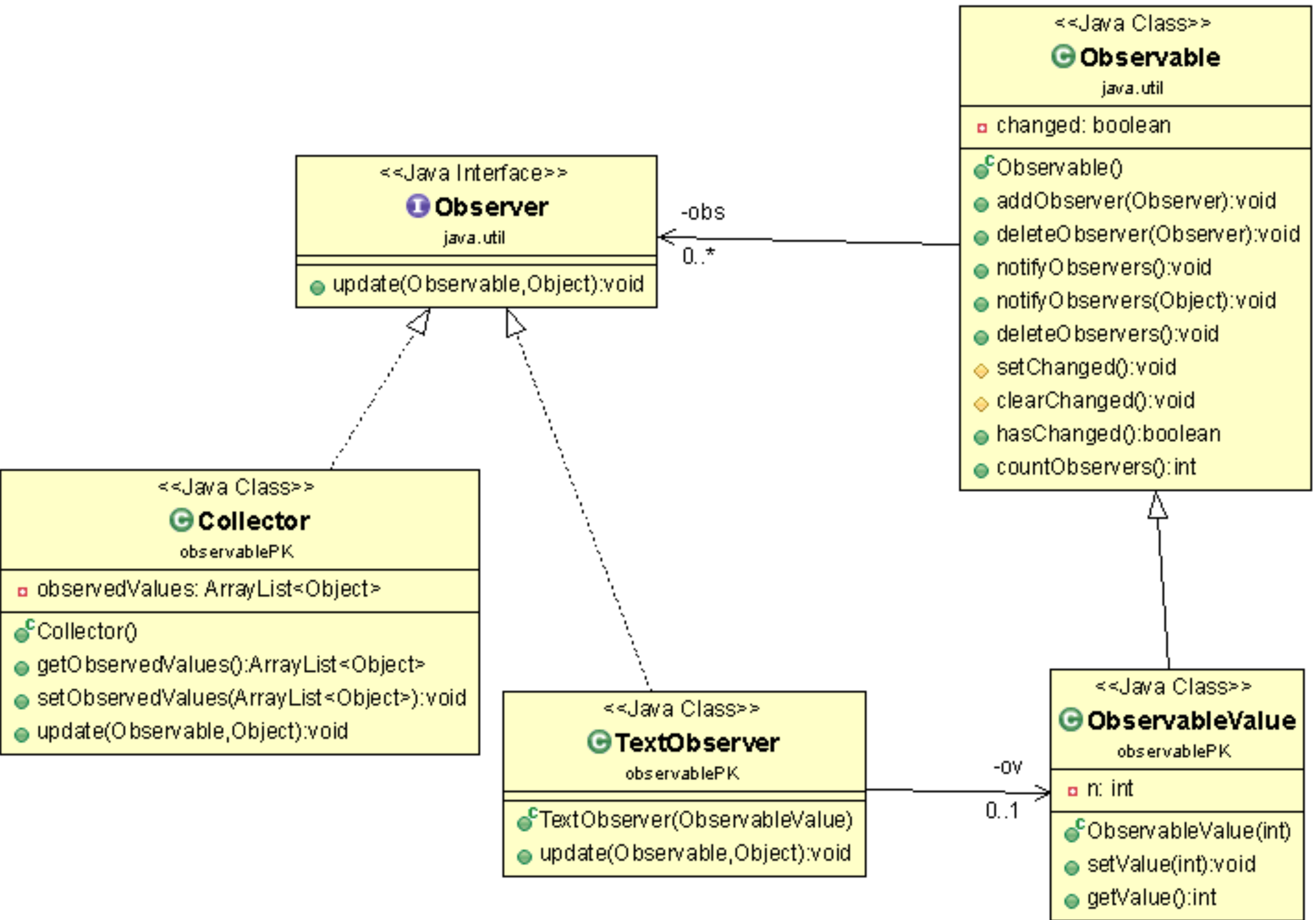
- Le sujet concret :
  - ➡Votre classe qui hérite de observable
  - ➡*C'est à vous d'appeler* **notifyObservers**

Observable
- changed
- obs
- Observable()
- addObserver(Observer) : void
- clearChanged() : void
- countObservers() : int
- deleteObserver(Observer) : void
- deleteObservers() : void
- hasChanged() : boolean
- notifyObservers() : void
- notifyObservers(Object) : void
- setChanged() : void

- L'observeur abstrait : Interface java.util.Observer

Observer
- update(Observable, Object) : void

- L'observeur concret :
  - ➡Votre classe qui «implémente» Observer
  - ➡*doit implémenter* **update**



<<Java Class>>
Observable
java.util
changed: boolean
Observable()
addObserver(Observer):void
deleteObserver(Observer):void
notifyObservers():void
notifyObservers(Object):void
deleteObservers():void
setChanged():void
clearChanged():void
hasChanged():boolean
countObservers():int
<<Java Interface>>
Observer
java.util
update(Observable,Object):void
-obs
0..*
<<Java Class>>
Collector
observablePK
observedValues: ArrayList<Object>
Collector()
getObservedValues():ArrayList<Object>
setObservedValues(ArrayList<Object>):void
update(Observable,Object):void
<<Java Class>>
TextObserver
observablePK
TextObserver(ObservableValue)
update(Observable,Object):void
-ov
0..1
<<Java Class>>
ObservableValue
observablePK
n: int
ObservableValue(int)
setValue(int):void
getValue():int

«Pattern Observer» en java exemple

```
import java.util.Observable;

public class ObservableValue extends Observable
{
    private int n = 0;
    public ObservableValue(int n)
    {
        this.n = n;
    }
    public void setValue(int n)
    {
        this.n = n;
        setChanged();
        notifyObservers(n);
    }
    public int getValue()
    {
        return n;
    }
}
```

http://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html

«Pattern Observer» en java exemple

```java
import java.util.Observer;
import java.util.Observable;
public class TextObserver implements Observer
{
   private ObservableValue ov = null;

   public TextObserver(ObservableValue ov)
   {
      this.ov = ov;
   }


   public void update(Observable obs, Object obj)
   {
      if (obs == ov)
      {
         System.out.println(ov.getValue() + " :" +  obj);
      }
   }
}
```

http://www.javaworld.com/article/2077258/learn-java/observer-and-observable.html

Autre Observer en java exemple

```
import java.util.ArrayList;
import java.util.Observable;
import java.util.Observer;

public class Collector implements Observer {
   private ArrayList<Object> observedValues = new ArrayList<>();

   public ArrayList<Object> getObservedValues() {
      return observedValues;
   }

   public void setObservedValues(ArrayList<Object>
observedValues) {
      this.observedValues = observedValues;
   }

   public void update(Observable obs, Object obj) {
      observedValues.add(obj);
   }

}
```

«Pattern Observer» en action

```
public static void main(String [] args)
{
      ObservableValue ov = new ObservableValue(0);
      TextObserver to = new TextObserver(ov);
      Collector collector = new Collector();
      ov.addObserver(collector);
      ov.addObserver(to);
      ov.setValue(10);
      ov.setValue(9);
      System.out.println(collector.getObservedValues());
      System.out.println("*******");
      ObservableValue ov2 = new ObservableValue(3);
      ov2.addObserver(collector);
      ov2.setValue(5);
      ov.setValue(6);
      System.out.println(collector.getObservedValues());

}
```

```
10 :10
9 :9
[10, 9]
*******
6 :6
[10, 9, 5, 6]
```

DP Observateur : Résumé

✓**Intention** :

➡ Définit une interdépendance de type un à plusieurs, de telle façon que quand un objet change d'état, tous ceux qui en dépendant en soient notifiés et automatiquement mis à jour.

✓**Applicabilité** : Utilisez l'Observateur dans les situations suivantes :

➡Quand un concept a deux représentations, l'une dépendant de l'autre. Encapsuler ces deux représentations dans des objets distincts permet de les réutiliser et de les modifier indépendamment.

➡Quand la modification d'un objet nécessite de modifier les autres, et que l'on ne sait pas combien sont ces autres.

➡- Quand un objet doit être capable de faire une notification à d'autres objets sans faire d'hypothèses sur la nature de ces objets. En d'autres termes, quand ces objets ne doivent pas être trop fortement couplés.

http://www.goprod.bouhours.net/?page=pattern&pat_id=16