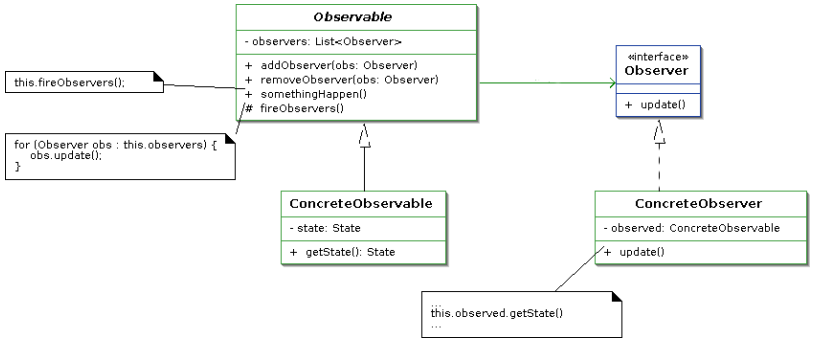
Observer/Observable

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



- Observer/Observable ou Abonneur/Abonné ou "event generator" idiom

Problème

Un objet (acteur) doit réagir lorsqu'un autre objet l'informe de l'occurrence d'un événement, mais ce dernier n'a pas nécessairement conscience de l'existence de l'acteur (ou **des** acteurs).

Les (ré)acteurs peuvent :

- être de différentes natures
- changer sans que l'émetteur de l'événement en soit conscient

Exemple :

- la gestion des événements dans awt/Swing

Les contraintes

- plusieurs objets peuvent être avertis des événements émis par une source
- le nombre et la nature des objets avertis ne sont potentiellement pas connus à la compilation et peuvent changer dans le temps.
- l'émetteur de l'événement et le récepteur ne sont pas fortement liés.

⇒ Mettre en place un mécanisme de délégation entre l'émetteur de l'événement (*event generator*) et le récepteur (*listener*)

Scénario **(d'après B. Venners)**

Un téléphone sonne, différentes personnes ou machines peuvent potentiellement être concernées par l'appel. Les personnes peuvent sortir ou entrer dans la pièce et ainsi sortir ou entrer du "champ d'intéressement" du téléphone.

Les étapes de design en Java

1. définir la classe d'événements ;
2. définir l'interface des `listeners` ;
3. définir la classe émettrice (génératrice des événements)
4. définir les classes réceptrices (les `listeners`)

Étape 1 : définir les classes d'événements

- définir une classe par type d'événements qui peuvent être émis par le générateur d'événements
- faire hériter ces classes de `java.util.EventObject`
- concevoir les classes d'événements de telle sorte qu'un événement englobe toute l'information qui doit être transmise à l'observateur
- donner aux classes d'évenement un nom de la forme `XXXEvent`

Étape 1 : définir les classes d'événements

```
package essais.observer;
public class TelephoneEvent extends java.util.EventObject {
   public TelephoneEvent(Telephone source) {
      super(source);
   }
}
```

- `EventObject.getSource()` : disposer de la source permet par exemple d'être abonné à plusieurs sources pour le même type d'évenements et de distinguer la source émettrice

Étape 2 : définir les interfaces des `listeners`

- pour chaque type d'événements, définir une interface qui hérite de `java.util.EventListener` et contient, pour chaque événement, une déclaration de méthode qui sera déclenchée lors de la notification de l'occurrence d'un événement
- le nom de cette interface est obtenu à partir du nom de la classe de l'événement en remplaçant `Event` par `Listener`
- les noms des méthodes de l'interface sont construits à partir d'un verbe au passé indiquant la situation d'activation.
- chaque méthode retourne `void` et prend un paramètre qui est de la classe de l'événement

```
package essais.observer;
public interface TelephoneListener extends java.util.EventListener {
   public void telephoneRang(TelephoneEvent e);
   public void telephoneAnswered(TelephoneEvent e);
}
```


Étape 3 : définir la classe émettrice

C'est la classe génératrice des événements (la « source »).

- pour chaque type d'événements émis, définir un couple de méthodes add/remove
- le nom de ces méthodes est de la forme : `add`*listener-interface-name*`()` et `remove`*listener-interface-name*`()`.
  = méthodes d'*abonnement* et *désabonnement*
- pour chacune des méthodes des interfaces des `listeners`, définir une méthode `private` de création et propagation de l'événement.
  Cette méthode est nommée : `fire`*listener-method-name*.
- déclencher la propagation des événements depuis les méthodes où ils sont détectés, en invoquant la méthode `fireXXX` appropriée.

Étape 3 : définir la classe émettrice

```
package essais.observer;
import java.util.*;
public class Telephone {
  private ArrayList<TelephoneListener> telephoneListeners = new ArrayList<TelephoneListener>();
  public synchronized void addTelephoneListener(TelephoneListener l) {
    if (telephoneListeners.contains(l)) { return ; }
    telephoneListeners.add(l);
  }
  public synchronized void removeTelephoneListener(TelephoneListener l){
    telephoneListeners.remove(l);
  }
  private void fireTelephoneRang() {
    ArrayLis<TelephoneListener>t tl = (ArrayList<TelephoneListener>) telephoneListeners.clone();
    if (tl.size() == 0) { return; }
    TelephoneEvent event = new TelephoneEvent(this);
    for (TelephoneListener listener :  tl) {
      listener.telephoneRang(event);
    }
  }
  private void fireTelephoneAnswered() { ...  }
  public void ringPhone() { fireTelephoneRang(); }
  public void answerPhone() { fireTelephoneAnswered(); }
}
```


Étape 4 : définir les classes réceptrices

- les classes réceptrices doivent implémenter l'interface `listener` associée

```
package essais.observer;
public class AnsweringMachine implements TelephoneListener {
   public void telephoneRang(TelephoneEvent e) {
      System.out.println("AM hears the phone ringing.");
   }
   public void telephoneAnswered(TelephoneEvent e) {
      System.out.println("AM sees that the phone was answered.");
   }
}
```

```
package essais.observer;
class MyTelephoneListener implements TelephoneListener {
   public void telephoneRang(TelephoneEvent e) {
      System.out.println("I'll get it!");
   }
   public void telephoneAnswered(TelephoneEvent e) {}
}
// abonnement "indirect"
public class Person {
   public void listenToPhone(Telephone t) {
      t.addTelephoneListener(new MyTelephoneListener());
   }
}
```

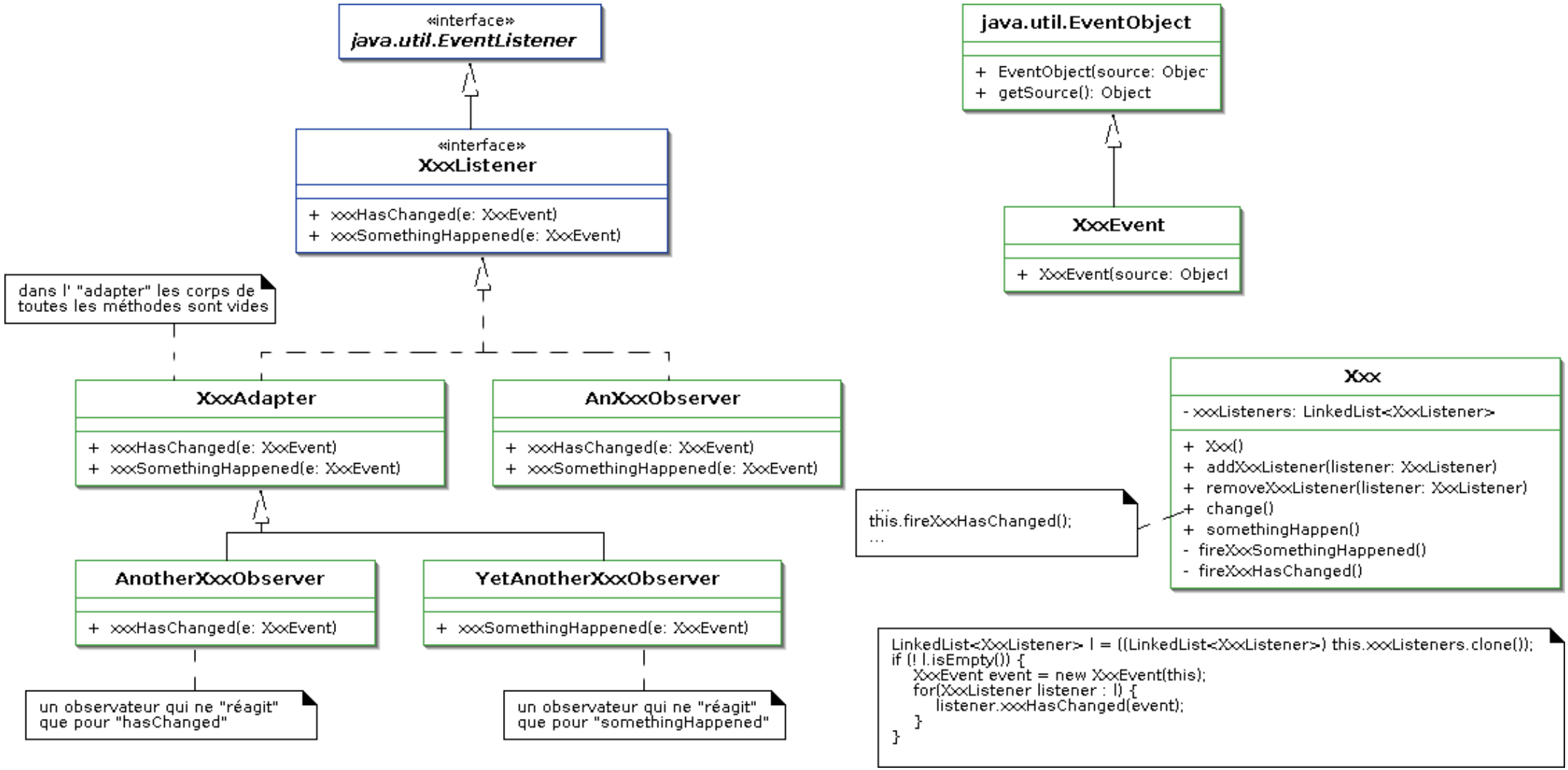

Exploitation

```
package essais.observer;

public class Example1 {
  public static void main(String[] args) {
    Telephone ph = new Telephone();                  // émetteur
    AnsweringMachine am = new AnsweringMachine();    // listener
    Person bob = new Person();

    ph.addTelephoneListener(am);                     // abonnement
    bob.listenToPhone(ph);                           // abonnement "indirect"

    ph.ringPhone();                  // provoquent l'émission
    ph.answerPhone();                // d'un événement
                                     // et la notification des listeners
  }
}
```


«interface»
java.util.EventListener
«interface»
XxxListener
+ xxxHasChanged(e: XxxEvent)
+ xxxSomethingHappened(e: XxxEvent)
java.util.EventObject
+ EventObject(source: Objec
+ getSource(): Object
XxxEvent
+ XxxEvent(source: Object
dans l' "adapter" les corps de toutes les méthodes sont vides
XxxAdapter
+ xxxHasChanged(e: XxxEvent)
+ xxxSomethingHappened(e: XxxEvent)
AnXxxObserver
+ xxxHasChanged(e: XxxEvent)
+ xxxSomethingHappened(e: XxxEvent)
Xxx
- xxxListeners: LinkedList<XxxListener>
+ Xxx()
+ addXxxListener(listener: XxxListener)
+ removeXxxListener(listener: XxxListener)
+ change()
+ somethingHappen()
- fireXxxSomethingHappened()
- fireXxxHasChanged()
...
this.fireXxxHasChanged();
...
AnotherXxxObserver
+ xxxHasChanged(e: XxxEvent)
YetAnotherXxxObserver
+ xxxSomethingHappened(e: XxxEvent)
un observateur qui ne "réagit" que pour "hasChanged"
un observateur qui ne "réagit" que pour "somethingHappened"
LinkedList<XxxListener> l = ((LinkedList<XxxListener>) this.xxxListeners.clone());
if (! l.isEmpty()) {
XxxEvent event = new XxxEvent(this);
for(XxxListener listener : l) {
listener.xxxHasChanged(event);
}
}