Il Design Pattern Observer - 1

Intento:

Definire una dipendenza tra un particolare oggetto (detto ***subject***) ed altri oggetti detti osservatori, cosicché quando il subject cambia stato, gli osservatori sono notificati ed aggiornati.

Es. Un oggetto contiene le coordinate di un punto e la sua variazione deve essere notificata ad un oggetto che effettua una rappresentazione grafica del punto.

Problema (Forze):

Il subject deve essere indipendente dal numero e dal tipo degli osservatori. In altre parole, il subject non fa assunzioni sugli oggetti che dipendono da esso. Oggetti lascamente accoppiati sono più facili da riusare.

Deve essere possibile aggiungere nuovi osservatori durante l'esecuzione dell'applicazione.

Soluzione:

Il subject rende disponibili delle operazioni che consentono ad un osservatore di dichiarare il proprio interesse per un cambiamento di stato.

Il Design Pattern Observer - 2

Componenti:

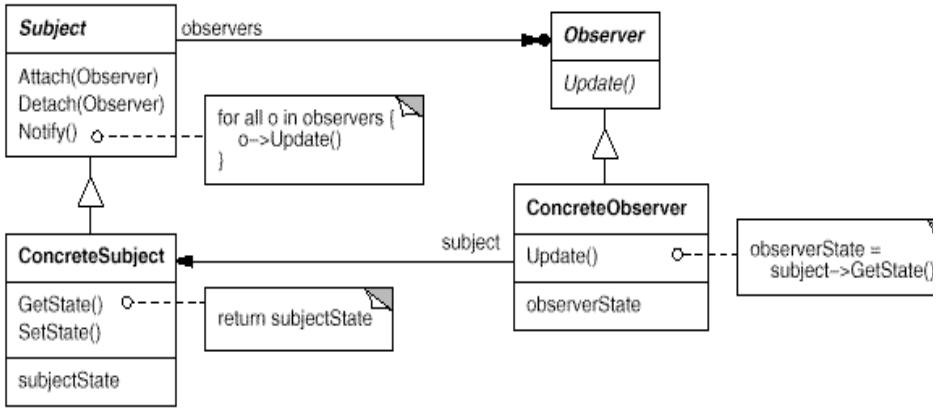
Subject è una classe che fornisce operazioni ad oggetti `Observer`, con metodi come: `attach()`, `detach()` e `notify()`; conosce solo la classe `Observer`

Observer è una classe che fornisce una interfaccia (operazione `update()`) comune a tutti gli oggetti che necessitano notifica

ConcreteSubject eredita da `Subject`, è la classe il cui stato deve essere notificato, conosce solo la classe `Subject`

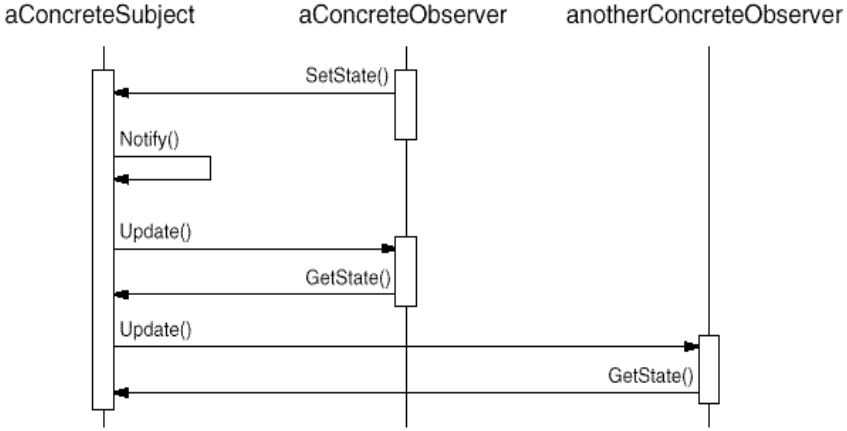
ConcreteObserver eredita da `Observer`, è la classe che vuole essere notificata, conosce solo la classe `Observer`

Struttura (class diagram):



Il Design Pattern Observer - 3

Comportamento a runtime



Conseguenze:

Il `Subject` conosce solo la classe `Observer` e non ha bisogno di conoscere le classi `ConcreteObserver`. `ConcreteSubject` e `ConcreteObserver` non sono accoppiati quindi più facili da riusare e modificare.

Esempi di utilizzo:

Smalltalk Model View Controller (framework dell'interfaccia nell'ambiente Smalltalk)

X Window (ma non ad oggetti), ogni finestra registra il suo interesse ad un particolare evento e tutte le informazioni sono inviate indietro (callback) quando l'evento accade.

Design Pattern Observer in Java

Il problema affrontato dal design pattern Observer è così comune che la sua soluzione è parte della libreria `java.util`

In tale libreria ci sono due classi Java che permettono di implementare l'Observer: `Observable` ed `Observer`.

La classe `Observable` tiene traccia di tutti gli oggetti che vogliono essere informati quando accade un cambiamento. La classe `Observable` notifica il cambiamento di stato e permette agli osservatori di aggiornare il proprio stato, attraverso il metodo `notifyObservers()`.

La classe `Observable` ha una variabile (flag) che indica se lo stato è cambiato. Questo è settato dal metodo `setChanged()`. La chiamata a `setChanged()` è da implementare nella classe che la eredita, secondo la logica del programma.

`Observer` è una interfaccia che ha solo il metodo `update()`.

Questo metodo è chiamato dall'oggetto osservato. `update()` può avere un argomento che indica quale oggetto ha causato l'aggiornamento.

Esempio con codice: vedi Address Book

Design Pattern Observer

```
// AddrBook e' un ConcreteSubject
public class AddrBook extends Observable {
  private static AddrBook instance = new AddrBook();
  private Vector<Person> nomi = new Vector<Person>();

  public static AddrBook getInstance() {
    return instance;
  }

  public void clear() {
    nomi.clear();
    setChanged();
    notifyObservers(nomi);
  }

  public boolean insert(Person p) {
    if (nomi.contains(p)) return false;
    nomi.addElement(p);
    setChanged();
    notifyObservers(nomi);
    return true;
  }
  ...
}
```

```
public class Dialog {
  public static AddrBook book;

  public static void main(String args[]) {
    Store st = new Store();
    book = AddrBook.getInstance();
    book.addObserver(st);
    // aggiungo st come observer di book
    ...
  }
}
```

```
// Store e' un ConcreteObserver
public class Store implements Observer {
  private FileWriter f;

  public void update(Observable s, Object c) {
    System.out.println("Writing AddrBook");
    Vector<Person> v = (Vector<Person>) c;
    ...
  }
}
```