19. Observer

(GoF pag. 293)

19.1. Descrizione

Consente la definizione di associazioni di dipendenza di molti oggetti verso di uno, in modo che se quest'ultimo cambia il suo stato, tutti gli altri sono notificati e aggiornati automaticamente.

19.2. Esempio

Ad un oggetto (Subject) vengono comunicati diversi numeri. Questo oggetto decide in modo casuale di cambiare il suo stato interno, memorizzando il numero ad esso proposto. Altri due oggetti incaricati del monitoraggio dell'oggetto descritto (un Watcher e un Psychologist), devono avere notizie di ogni suo singolo cambio di stato, per eseguire i propri processi di analisi.

Il problema è trovare un modo nel quale gli eventi dell'oggetto di riferimento, siano comunicati a tutti gli altri interessati.

19.3. Descrizione della soluzione offerta dal pattern

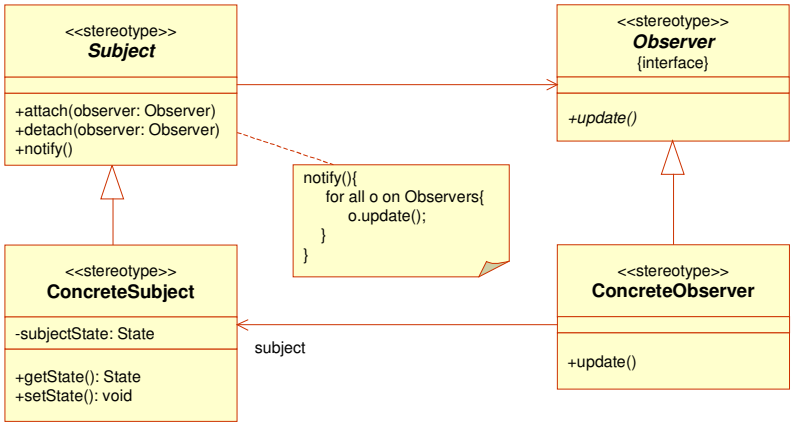
Il pattern "Observer" assegna all''oggetto monitorato (Subject) il ruolo di registrare ai suoi interni un riferimento agli altri oggetti che devono essere avvisati (ConcreteObservers) degli eventi del Subject, e notificarli tramite l'invocazione a un loro metodo, presente nella interfaccia che devono implementare (Observer).

Questo pattern è stato proposto originalmente dai GoF per la manutenzione replicata dello stato del ConcreteSubject nei ConcreteObserver, motivo per il quale sono previsti una copia dello stato del primo nei secondo, e la esistenza di un riferimento del ConcreteSubject nel ConcreteObserver.

Nonostante lo espresso nel parafo precedente, si deve tenere in conto che questo modello può servire anche a comunicare eventi, in situazioni nelle quali non sia di interesse gestire una copia dello stato del Subject. Da un'altra parte si noti che non è necessario che ogni ConcreteObserver abbia un riferimento al Subject di interesse, oppure, che i riferimenti siano verso un unico Subject. Le Java API offrono un modello esteso in funzionalità, allineato in questa direzione, che sarà l'approccio utilizzato in questo esempio.

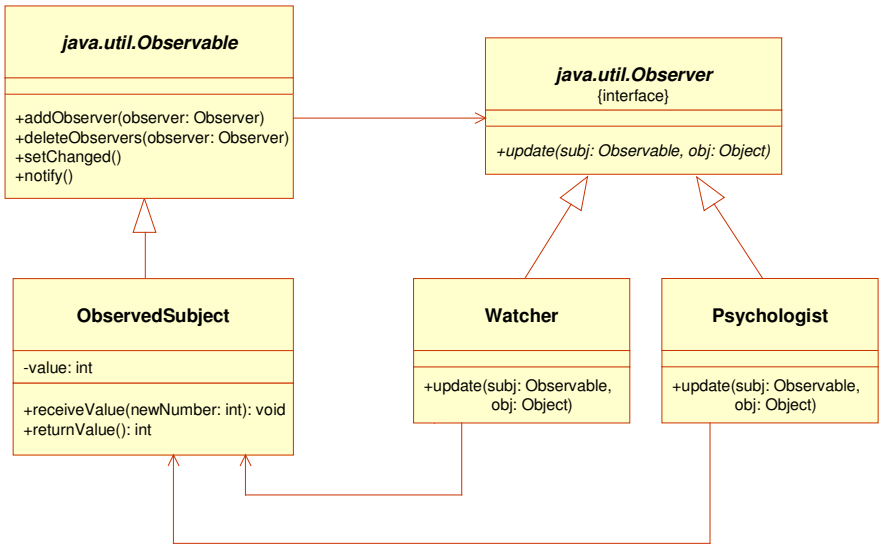
Un'altra versione del pattern Observer, esteso con una gestione più completa degli eventi, è implementato dentro l'ambiente di sviluppo G++ [16]. In questo modello è consentita la registrazione del Observer presso il Subject, indicando addizionalmente il tipo di evento davanti al quale l'Observer deve essere notificato, e la funzione del Observer da invocare.

19.4. Struttura del pattern



19.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Subject**: classe Observable.
  - Ha conoscenza dei propri **Observer**, i quali possono esserci in numero illimitato.
  - Fornisce operazioni per l'addizione e cancellazione di oggetti **Observer**.
  - Fornisce operazioni per la notifica agli **Observer**.
- **Observer**: interfaccia Observer.
  - Specifica una interfaccia per la notifica di eventi agli oggetti interessati in un **Subject**.

- **ConcreteSubject**: classe ObservedSubject.
  - Possiede uno stato dell'interesse dei **ConcreteSubject**.
  - Invoca le operazioni di notifica ereditate dal **Subject**, quando devono essere informati i **ConcreteObserver**.
- **ConcreteObserver**: classi Watcher e Psychologist.
  - Implementa l'operazione di aggiornamento dell'Observer.

Descrizione del codice

Si presenta di seguito l'applicazione di questo pattern utilizzando come costruttori di base quelli forniti dalle Java API, che sono l'interfaccia `java.util.Observer`, e la classe `java.util.Observable`:

- Interfaccia `java.util.Observer`: specifica l'interfaccia che devono implementare i **ConcreteObserver**. Specifica il singolo metodo:
  - `void update(Observable o, Object arg)`: è il metodo che viene chiamato ogni volta che il **Subject** notifica un evento o cambiamento nel proprio stato i **ConcreteObserver**. Al momento dell'invocazione il **Subject** passa come primo parametro un riferimento a sé stesso, e come secondo parametro un oggetto `Object` qualunque (utile a trasferire informazioni aggiuntive all'Observer)
- classe `java.util.Observable`: serve come classe di base per l'implementazione dei **Subject**. Ha questo costruttore:
  - `Observable()`

    Costruisce un `Observable` senza Observers registrati.

  E questi metodi d'interesse:

  - `void addObserver( Observer o )`: registra l' **Observer** nel suo elenco interno di oggetti da notificare.
  - `protected void setChanged()`: segna sé stesso come "cambiato", in modo che il metodo `hasChanged` restituisce `true`.
  - `boolean hasChanged()`: restituisce true se l'oggetto stesso è stato segnato come "cambiato".
  - `void notifyObservers()`: se l'oggetto è stato segnato come "cambiato", come indicato dal metodo `hasChanged`, fa una notifica a tutti gli **Observer** e poi chiama il metodo `clearChanged` per segnare che l'oggetto è adesso nello stato "non cambiato".
  - `void notifyObservers( Object arg )`: agisce in modo simile al metodo precedentemente descritto, ma riceve l'`Object arg` come argomento, che è inviato ad ogni **Observer**, come secondo parametro del metodo `update`.

- `protected void clearChanged()`: indica che l'oggetto non è cambiato, o che la notifica è già stata fatta agli **Observer**. Dopo l'invocazione a questo metodo, `hasChanged` restituisce `false`.
- `int countObservers()`: restituisce il numero di **Observer** registrati.
- `void deleteObservers()`:cancella l'elenco degli **Observer** registrati.
- `void deleteObservers(Observer o)`: cancella un particolare **Observer** dall'elenco dei registrati.

In questo esempio si implementa la classe `ObservedSubject` (**ConcreteSubject**) che estende la classe `Observable` (**Subject**). Ogni istanza di questo **ConcreteSubject** riceve semplicemente un numero tramite il metodo `receiveValue`, e con una scelta a caso decide di copiare o meno questo valore nella propria variabile di stato `value`. Dato che possono esserci **Observer** interessati a monitorare questo cambiamento di stato, quando accadde ciò, viene invocato `setChanged` per abilitare la procedura di notifica.

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

public class ObservedSubject extends Observable {

   private int value = 0;

   public void receiveValue( int newNumber ) {
     if (Math.random() < .5) {
        System.out.println( "Subject : I like it, I've changed my "
                            + "internal value." );
        value = newNumber;
        this.setChanged();
     } else
        System.out.println( "Subject : I have a number "+ value +
                            " now, and I not interested in the number "
                            + newNumber + "." );
     this.notifyObservers();
   }

   public int returnValue() {
     return value;
   }
}
```

Si noti che l'istruzione `notifyObservers` viene invocata, indipendentemente se si è registrato un cambio di stato nell'oggetto. Questo per dimostrare che la notifica agli **Observer** sarà eseguita soltanto se il cambio di stato è stato segnato da una invocazione al `setChanged`.

La classe `Watcher` implementa un **ConcreteObserver**, che ad ogni notifica di cambiamento nel **Subject**, invoca un metodo su quest'ultimo per conoscere il nuovo numero archiviato (stato del **Subject**).

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

public class Watcher implements Observer {

  private int changes = 0;

  public void update(Observable obs, Object arg) {
     System.out.println( "Watcher : I see that the Subject holds now a "
```

```
                          + ((ObservedSubject) obs ).returnValue() + ".");
      changes++;
   }

   public int observedChanges() {
      return changes;
   }

}
```

Un secondo **ConcreteObserver** è rappresentato dalla classe `Psycologist`, che esegue una operazione diversa dal `Watcher` nei confronti della stessa notifica:

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

public class Psychologist implements Observer {

   private int countLower, countHigher = 0;

   public void update(Observable obs, Object arg) {
      int value = ((ObservedSubject) obs ).returnValue() ;
      if( value <= 5 )
         countLower++;
      else
          countHigher++;
   }

   public String opinion() {
      float media;
      if( (countLower + countHigher ) == 0 )
        return( "The Subject doesn't like changes.");
      else
        if( countLower > countHigher )
           return( "The Subject likes little numbers.");
        else if ( countLower < countHigher )
           return( "The Subject likes big numbers.");
        else
           return( "The Subject likes little numbers and big numbers.");
   }

}
```

Si presenta di seguito l'applicazione che prima crea una istanza Subject e un'altra di ogni tipo di Observer, e registra questi Observers nel Subject, tramite l'invocazione al metodo addObserver che è ereditato da quest'ultimo, dalla classe Observable.

```
public class ObserverExample {

   public static void main (String[] args) {

       ObservedSubject s = new ObservedSubject() ;
       Watcher o = new Watcher();
       Psychologist p = new Psychologist();
       s.addObserver( o );
       s.addObserver( p );
       for(int i=1;i<=10;i++){
         System.out.println( "Main : Do you like the number " + i +"?" );
         s.receiveValue( i );
       }

       System.out.println( "The Subject has changed " +
                           o.observedChanges()
                           + " times the internal value.");
       System.out.println("The Psychologist opinion is:" + p.opinion() );
   }

}
```

Osservazioni sull'esempio

In questo esempio non è stato necessario replicare lo stato del **Subject** negli **Observers**. Nel caso di voler farlo, semplicemente basta di aggiungere una apposita variabile nell'**Observer**, e copiare in essa il valore del momento presente nel **Subject** ad ogni notifica eseguita.

Esecuzione dell'esempio

```
C:\Design Patterns\Behavioral\Observer>java ObserverExample

Main    : Do you like the number 1?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 1.

Main    : Do you like the number 2?
Subject : I have a number 1 now, and I not interested in the number 2.

Main    : Do you like the number 3?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 3.

Main    : Do you like the number 4?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 4.

Main    : Do you like the number 5?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 5.

Main    : Do you like the number 6?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 6.

Main    : Do you like the number 7?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 7.

Main    : Do you like the number 8?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 8.

Main    : Do you like the number 9?
Subject : I have a number 8 now, and I not interested in the number 9.

Main    : Do you like the number 10?
Subject : I have a number 8 now, and I not interested in the number 10

The Subject has changed 7 times the internal value.
The Psychologist opinion is: The Subject likes little numbers.
```

19.6. Osservazioni sull'implementazione in Java

Java estende il modello originalmente proposto dai GoF, in modo di poter associare un singolo **Observer** a più **Subject** contemporaneamente. Questo è consentito dal fatto che il metodo di `update` dell'`Observer` riceve come parametro un riferimento al **Subject** che fa la notifica, consentendo al primo di conoscere quale di tutti i **Subject** la ha originato. E' importante anche notare che questo meccanismo non predispone il modello alla sola gestione di una replica dello stato del **Subject** nell'**Observer**, dato che potrebbe anche essere utilizzato, ad esempio, per la sola comunicazione di eventi.

Le particolarità da tenere in conto se si vuole utilizzare le risorse fornite da Java sono:

- `Observable` è una classe che, tramite l'estensione, fornisce i metodi di base del **Subject**. Questo vieta la possibilità che il **Subject** possa essere implementato contemporaneamente come una estensione di un'altra classe.
- La notifica viene eseguita nello stesso thread del **Subject**, costituendo un sistema di notifica sincrono (il metodo update di un **Observer** deve finalizzare e restituire il controllo al metodo notify del **Subject**, prima che questo possa continuare a notificare un altro). In un sistema multi-threding potrebbe essere necessario avviare un nuovo thread ogni volta che un update è eseguito.

Altre idee interessanti riguardanti questo pattern sono presenti negli articoli di Lopez [11] e Bishop [2].