Azzolini Riccardo

Gestione degli eventi – Observer

1 Il pattern observer

Il pattern **observer** (noto anche come *publish-subscribe*, *callback*, o *dependents*) riassume e formalizza il meccanismo di gestione degli eventi visto in precedenza. Esso è infatti un pattern comportamentale che gestisce la comunicazione tra oggetti, definendo il modo in cui un certo numero di classi possono ricevere *notifiche di eventi* a cui sono interessate.

L'observer definisce una dipendenza *uno-a-molti* tra oggetti: per ogni oggetto osservato ci possono essere tanti osservatori, che ricevono tutti le notifiche per ogni evento.

1.1 Motivazione

La motivazione del pattern observer è quella di evitare il polling, cioè che gli osservatori debbano chiedere periodicamente all'oggetto osservato se si sia verificato un evento. Il polling, infatti, è:

- poco efficiente, perché molte interrogazioni vengono fatte a vuoto;
- poco efficace, poiché l'evento potrebbe verificarsi un microsecondo dopo che si è fatta l'interrogazione.

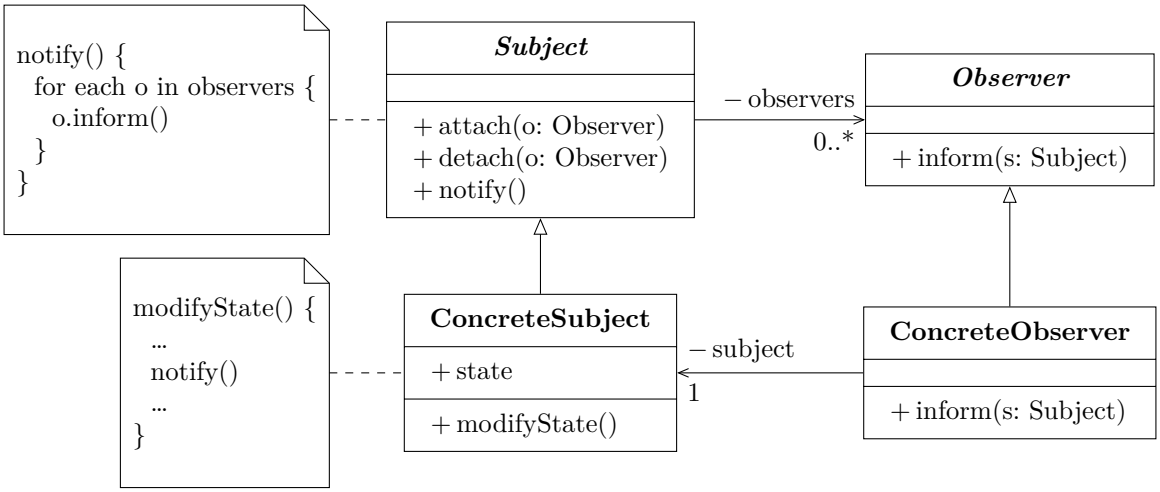
1.2 Partecipanti e comportamento

I partecipanti[1] del pattern observer sono:

- il **soggetto osservabile** dove accadono gli eventi (o che comunque è al corrente dell'accadere degli eventi);
- gli **osservatori**, che ricevono delle notifiche quando si verificano gli eventi.

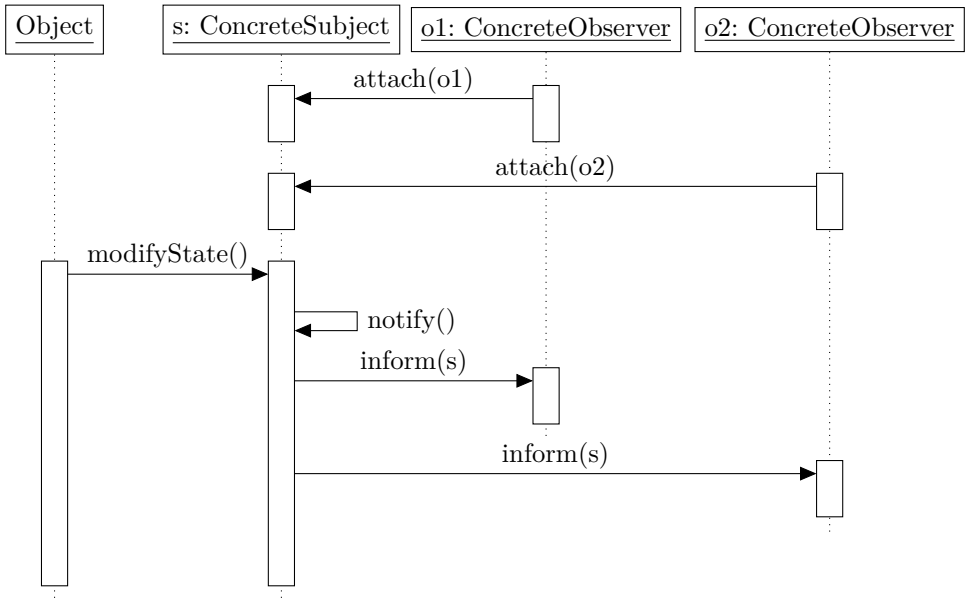
La struttura del pattern è quindi descritta dal seguente class diagram:

[1] Questi oggetti possono essere distribuiti in rete, oppure essere sulla stessa macchina: il pattern indica una situazione logica, non fisica.



- Innanzitutto, sono definiti i ruoli astratti di soggetto osservabile (la classe `Subject`, talvolta chiamata `Observable`) e di osservatore (`Observer`). Poi, si creano delle classi concrete che ricoprono questi ruoli implementando la logica specifica per l'applicazione.
- La classe `Observer` è dotata di un metodo `inform` (o `update`), che viene invocato per notificare l'oggetto osservatore quando si verifica un evento.
- `Subject` prevede invece i metodi:
  - `attach` e `detach`, per aggiungere e togliere oggetti osservatori;
  - `notify`, che notifica tutti gli osservatori (chiamando il metodo `inform` di ciascuno di essi).
- Il metodo `notify` viene tipicamente invocato per segnalare un qualche cambiamento di stato del `ConcreteSubject`.

La dinamica del pattern observer può essere esemplificata con un sequence diagram come il seguente:



1.3 Implementazione in locale

Quando si realizza il pattern in locale, un observer è semplicemente un oggetto che implementa l'omonima interfaccia:

```
public interface Observer {
    void inform(Subject subject);
}
```

La classe `Subject`, invece, contiene la lista degli osservatori e i metodi descritti in precedenza:

```
import java.util.ArrayList;
import java.util.List;

public abstract class Subject {
    protected List<Observer> observers = new ArrayList<>();

    public void attach(Observer o) {
        observers.add(o);
    }

    public void detach(Observer o) {
        observers.remove(o);
    }
```

```
    public void notify() {
        for (Observer o : observers) {
            o.inform(this);
        }
    }
}
```

In questo caso, per ottenere la soluzione più generale possibile, la classe `Subject` comunica agli osservatori un riferimento a sé stessa ogni volta che invia una notifica di un evento: attraverso tale riferimento, gli osservatori potranno esaminare il soggetto osservato per determinare come sia cambiato il suo stato.

2 Observer nella libreria standard di Java

In Java, non è necessario implementare da zero il pattern observer, poiché il package `java.util` mette già a disposizione un'interfaccia `Observer` e una classe `Observable`.[2]

- L'interfaccia `Observer` prevede un unico metodo,

  ```
  void update(Observable o, Object arg);
  ```

  che viene chiamato dall'oggetto `Observable` per notificare un evento all'observable. Esso ha due argomenti:

  - il primo è un riferimento allo stesso oggetto `Observable` dal quale proviene l'evento;
  - il secondo è un oggetto di tipo qualunque (la possibilità di passare anche questo rende più generale il meccanismo di notifica).

- I metodi principali della classe `Observable` sono:

  - `addObserver`: aggiunge un osservatore al soggetto osservabile.

    ```
    public void addObserver(Observer o);
    ```

  - `notifyObservers`: notifica tutti gli osservatori, chiamando il metodo `update` di ciascuno di essi (opzionalmente, con un oggetto qualunque come argomento), ma solo se il soggetto osservabile è *cambiato*.

    ```
    public void notifyObservers();
    public void notifyObservers(Object arg);
    ```

[2] A partire da Java 9, queste sono state deprecate, sostanzialmente perché si ritiene che forniscano un modello di eventi troppo limitato per le esigenze della maggior parte delle applicazioni; ciò nonostante, esse rimangono di fatto adeguate per esempi semplici come quelli che verranno presentati in seguito.

Questo metodo può essere chiamato dal soggetto stesso (quando sa di essere cambiato), oppure "dall'esterno".

- `setChanged`: marca come *cambiato* il soggetto.

  ```
  protected void setChanged();
  ```

- `clearChanged`: marca come *non cambiato* il soggetto.

  ```
  protected void clearChanged();
  ```

  Esso può essere chiamato manualmente, se necessario, ma altrimenti viene chiamato automaticamente da `notifyObservers` dopo l'invio delle notifiche: così, un osservatore non riceverà mai due notifiche relative allo stesso cambiamento.

- `hasChanged`: indica se il soggetto è cambiato o meno.

  ```
  public boolean hasChanged();
  ```

3 Esempio di utilizzo locale

Come esempio semplice ma significativo di realizzazione (in locale) dell'observer pattern, si vogliono gestire eventi relativi all'input da tastiera:

- il soggetto osservato gestisce l'input da tastiera;
- l'inserimento di ogni riga è considerato un evento, che deve essere notificato agli osservatori.

Il programma verrà realizzato usando l'implementazione dell'observer fornita dalla libreria standard: a ogni stringa letta da tastiera, si chiamerà `notifyObserver` per informare tutti gli osservatori dell'evento di lettura.

3.1 Osservatore concreto

L'osservatore concreto è la classe in grado di gestire l'input: esso viene ricevuto come argomento del metodo `update`, convertito in una stringa, e semplicemente stampato:

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

public class InputHandler implements Observer {
    public void update(Observable o, Object arg) {
        String info = (String) arg;
        System.out.println(info + " communicated");
```

```
    }
}
```

3.2 Soggetto concreto

Il soggetto osservabile è un thread che, per ogni stringa letta da tastiera (in un ciclo infinito), chiama `setChanged` e `notifyObserver` in modo da inviare la stringa a tutti gli osservatori:

```
import java.io.*;
import java.util.Observable;

public class EventSource extends Observable implements Runnable {
    public void run() {
        try (
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr)
        ) {
            while (true) {
                System.out.print("Enter text > ");
                System.out.flush();
                String str = br.readLine();
                setChanged();
                notifyObservers(str);
            }
        } catch (IOException e) {}
    }
}
```

Osservazione: Chiamando `setChanged` subito prima di `notifyObservers`, si è sicuri che verrà inviata una notifica agli osservatori.

3.3 Main

Nel main bisogna semplicemente istanziare il soggetto e l'osservatore, registrare l'osservatore presso il soggetto, e avviare un thread per il soggetto:

```
public class MyApp {
    public static void main(String[] args) {
        EventSource source = new EventSource();
        InputHandler handler = new InputHandler();
        source.addObserver(handler);
        Thread sourceThread = new Thread(source);
```

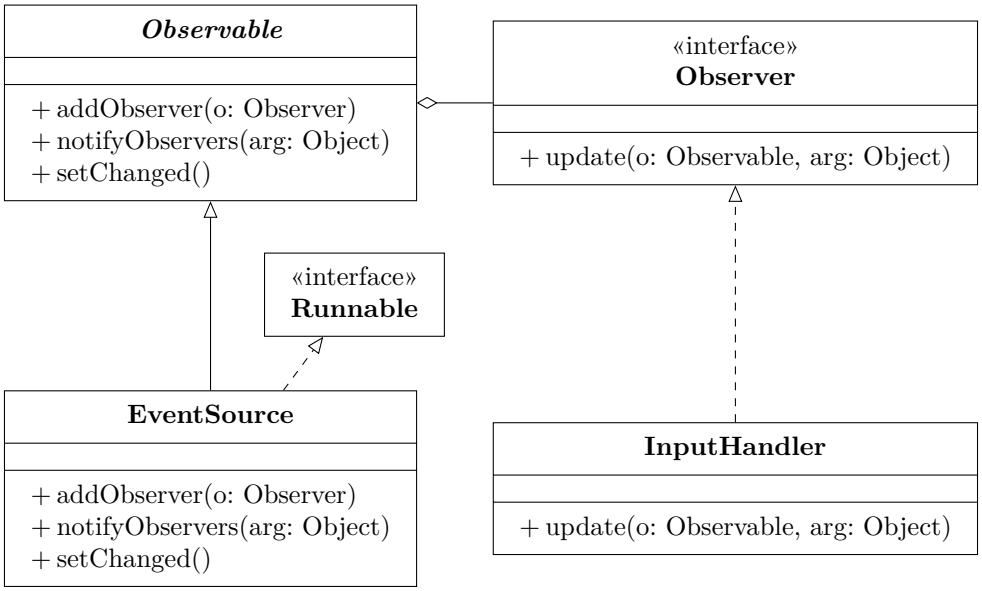
```
        sourceThread.start();

        // Adesso MyApp può proseguire la sua elaborazione senza
        // preoccuparsi di gestire l'input.
        // ...
    }
}
```

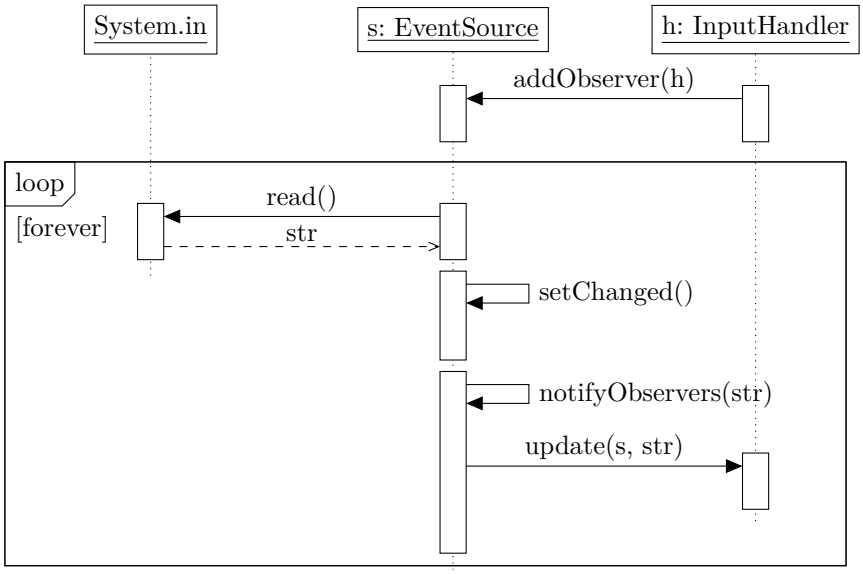
Osservazione: Questa situazione, nella quale cui un programma fa quel che deve fare e, intanto, un thread si preoccupa di gestire eventi (spesso di input), è molto comune, indipendentemente dall'uso del pattern observer.

3.4 Class e sequence diagram

Le classi che costituiscono l'applicazione sono:

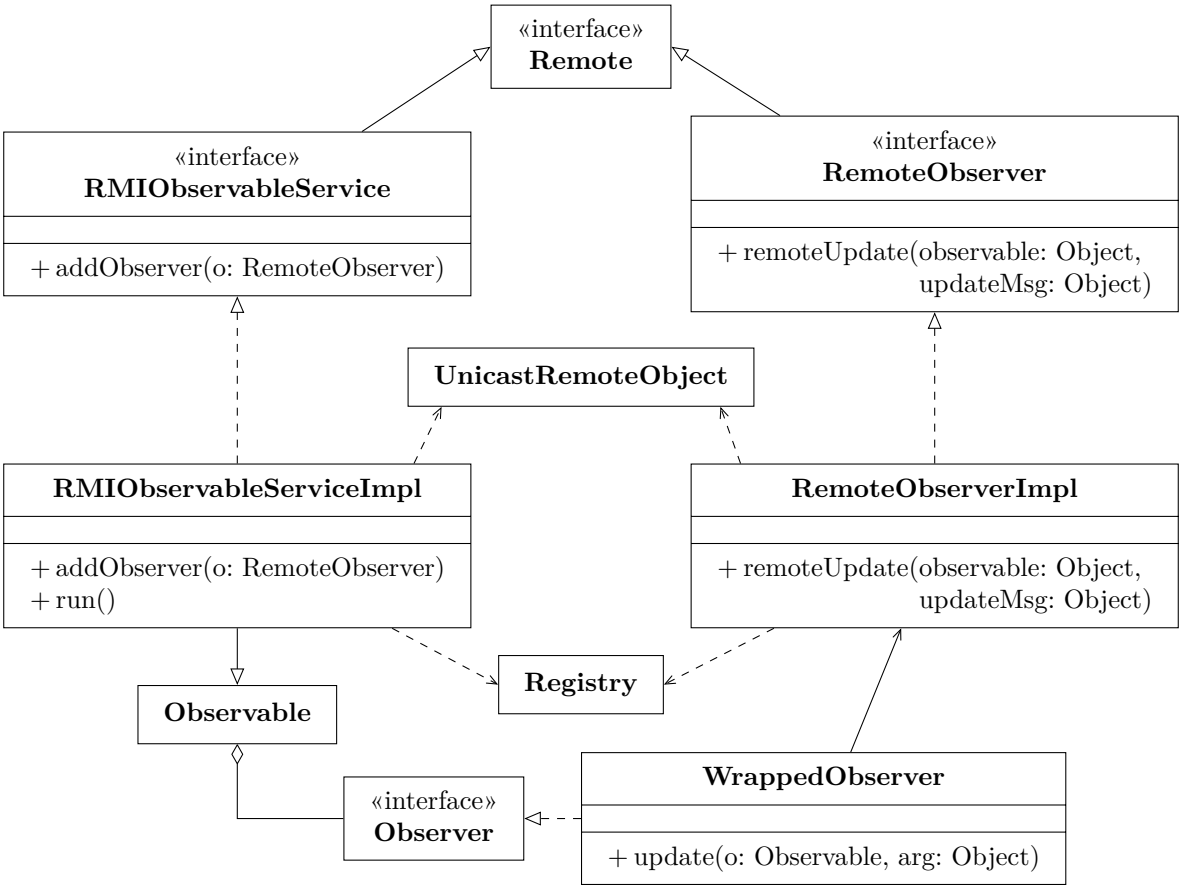


Il loro comportamento è il seguente:



4 Esempio di utilizzo con RMI

Per mostrare come il pattern observer può essere utilizzato in un'applicazione distribuita, realizzata con RMI, verrà creato un server che, ogni 5 secondi, invia l'ora esatta a tutti i client (che ne hanno fatto richiesta). Dunque, il server è un `Observable`, e i client sono `Observer`. Complessivamente, l'applicazione è composta dalle seguenti classi:



- Essendo necessario l'uso di callback, client e server sono entrambi oggetti remoti, come al solito.
- Le interfacce remote `RMIObservableService` e `RemoteObserver` sono l'equivalente distribuito di `Observable` e `Observer`.
- L'oggetto remoto server, `RMIObservableServiceImpl`, deve comportarsi da `Observable` all'interno della macchina server, e quindi deve aver associati degli `Observer` *locali*. Allora, si definisce una classe `WrappedObserver`, che contiene il riferimento all'observer remoto e inoltra a esso le notifiche degli eventi (in pratica, è un proxy).

4.1 Interfaccia `RemoteObserver`

L'interfaccia remota `RemoteObserver` assomiglia molto all'interfaccia `Observer`:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteObserver extends Remote {
```

```
    void remoteUpdate(Object observable, Object updateMsg)
        throws RemoteException;
}
```

4.2 Implementazione del client

Il client implementa l'interfaccia `RemoteObserver`, semplicemente stampando i messaggi ricevuti. Anche il main non è particolarmente interessante: dopo la solita parte di preparazione dei riferimenti remoti, il deve solo client registrarsi come osservatore presso il server.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class RemoteObserverImpl implements RemoteObserver {
    public void remoteUpdate(Object observable, Object updateMsg) {
        System.out.println("Got message: " + updateMsg);
    }

    public static void main(String[] args) throws Exception {
        Registry registry = LocateRegistry.getRegistry();
        RMIObservableService remoteService = (RMIObservableService)
            registry.lookup("Observable");
        RemoteObserver client = new RemoteObserverImpl();
        RemoteObserver clientStub = (RemoteObserver)
            UnicastRemoteObject.exportObject(client, 3949);
        remoteService.addObserver(clientStub);
    }
}
```

4.3 Interfaccia `RMIObservableService`

L'interfaccia remota `RMIObservableService` prevede solo il metodo `addObserver` (analogo all'omonimo metodo della classe `java.util.Observable`), che permette la registrazione di un osservatore remoto:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RMIObservableService extends Remote {
    void addObserver(RemoteObserver o) throws RemoteException;
}
```

4.4 Implementazione del server

Il server è un `Observable` che implementa `RMIObservableService`:

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;
import java.util.Observable;

public class RMIObservableServiceImpl
        extends Observable implements RMIObservableService {
    public void addObserver(RemoteObserver o) {
        WrappedObserver wo = new WrappedObserver(o);
        addObserver(wo);
        System.out.println("Added observer: " + wo);
    }

    private void run() throws InterruptedException {
        while (true) {
            Thread.sleep(5000);
            setChanged();
            notifyObservers(new Date());
        }
    }

    public static void main(String[] args)
            throws RemoteException, InterruptedException {
        RMIObservableServiceImpl obj = new RMIObservableServiceImpl();
        RMIObservableService stub = (RMIObservableService)
            UnicastRemoteObject.exportObject(obj, 3939);
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.rebind("Observable", stub);
        System.err.println("Server ready");
        obj.run();
    }
}
```

- L'implementazione di `RMIObservableService.addObserver` crea un wrapper locale per l'observer remoto e lo passa a `Observable.addObserver`.
- Il metodo `run` (che, a differenza di quanto si potrebbe pensare dal nome, qui non definisce un thread) contiene un ciclo infinito in cui, ogni 5 secondi, si usano i

metodi `setChanged` e `notifyObservers` di `Observable` per inviare la data e ora correnti ai `WrappedObserver`, che la inoltreranno agli osservatori remoti.

- Il main è uguale al solito.

4.5 Classe `WrappedObserver`

La classe `WrappedObserver`, che implementa `Observer`, funge da proxy locale al server per gli osservatori remoti:

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

public class WrappedObserver implements Observer {
    private final RemoteObserver remoteClient;

    public WrappedObserver(RemoteObserver ro) {
        remoteClient = ro;
    }

    public void update(Observable o, Object arg) {
        try {
            remoteClient.remoteUpdate(o.toString(), arg);
        } catch (RemoteException e) {
            System.err.println(
                "Remote exception; removing observer: " + this
            );
            o.deleteObserver(this);
        }
    }
}
```

Ogni volta che viene inviata una notifica a questo osservatore, tramite una chiamata locale del metodo `update`, esso prova a inoltrarla al client remoto, invocando il metodo `remoteUpdate` di quest'ultimo, e inoltrando anche gli argomenti della chiamata.[3]

Se l'invocazione remota fallisce, si presume che l'osservatore remoto si sia disconnesso, perciò si indica all'`Observable` di rimuovere anche il proxy locale.

[3] Per semplicità, l'argomento `Observable` viene passato come stringa: in teoria, non è detto che esso sia un oggetto remoto o serializzabile, quindi non è garantito che lo si possa passare direttamente in una chiamata remota; invece, una stringa è sempre serializzabile, dunque può essere passata senza problemi.