

Ingegneria del Software B

# Il pattern *Observer*

Ex presentazione realizzata  
dallo studente Davide Ferrari nell'a.a. 2009/2010

# Observer

**Noto anche come**

*Publisher-Subscriber, Delegation Event Model* (in Java)

**Classificazione**

Object behavioural

**Scopo**

Definisce una **relazione *uno a molti*** fra oggetti, in modo tale che, quando un **oggetto cambia stato**, tutti gli oggetti che **dipendono da esso ne ricevano notifica** (e vengano aggiornati) automaticamente

# Observer: Motivazione

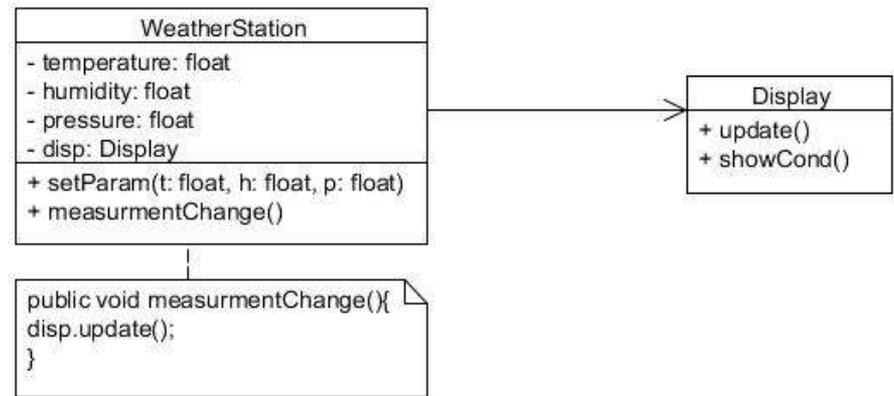
## Esempio

Realizzare una classe che rappresenti una **stazione meteorologica**, in grado di misurare *temperatura, umidità e pressione atmosferica*; realizzare, in seguito, una classe che rappresenti un **display**, in grado di **mostrare le condizioni atmosferiche** misurate dalla stazione.

# Observer: Motivazione

## Implementazione

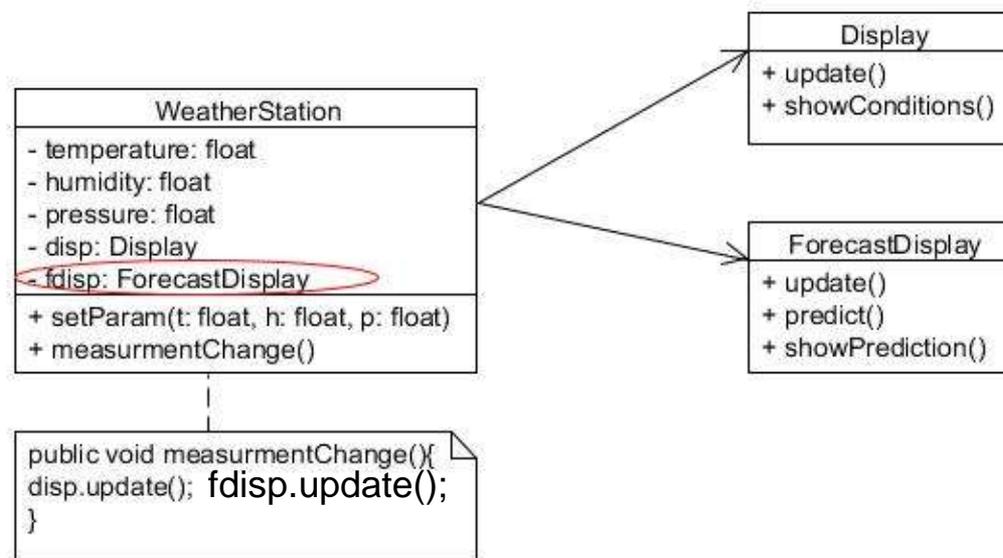
- la classe *WeatherStation* ha un **attributo** di tipo *Display*
- “ogni volta” che **cambiano le condizioni atmosferiche** si invoca (ad es. come ultima istruzione di *setParam*) *measurementChange()*, che **aggiorna** l’attributo *disp*



# Observer: Motivazione

## Estensione

Aggiungere una classe in grado di **eseguire** (e **visualizzare**) *previsioni meteorologiche* sulla base delle condizioni atmosferiche correnti (N.B. ogni volta che le condizioni cambiano, la previsione potrebbe cambiare).

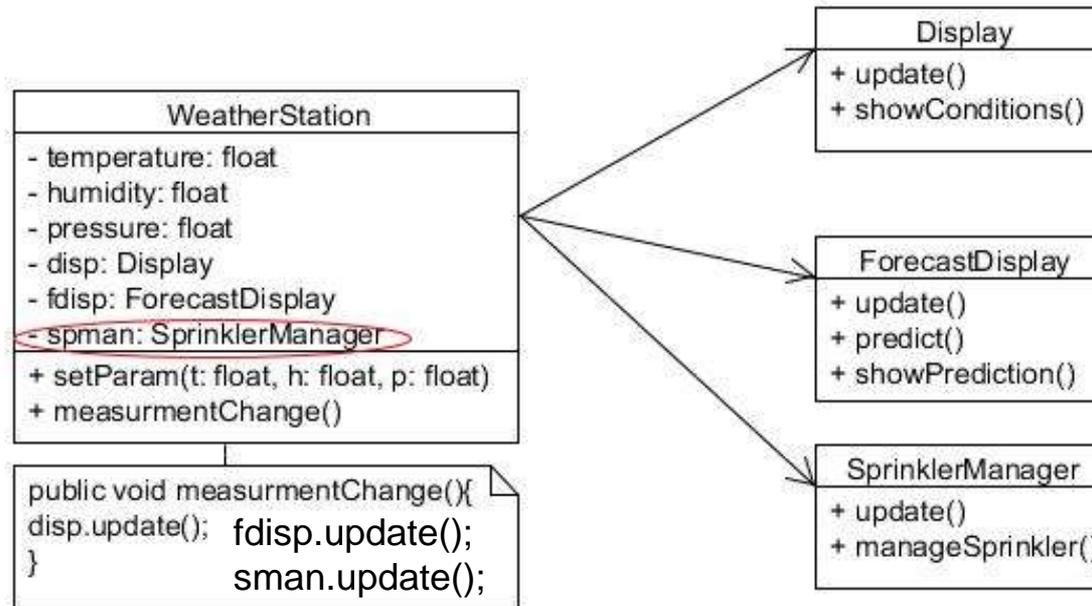


→ È stato necessario **modificare** la classe *WeatherStation*

# Observer: Motivazione

## Altra estensione

Aggiungere una classe che rappresenti un gestore di irrigatori, in grado di *attivare* (o *disattivare*) degli irrigatori, in base alle condizioni atmosferiche attuali.



→ Ancora **modifiche** alla classe *WeatherStation*

→ Da notare il metodo *update()* in tutte le classi dipendenti da *WeatherStation*

# Observer: Motivazione

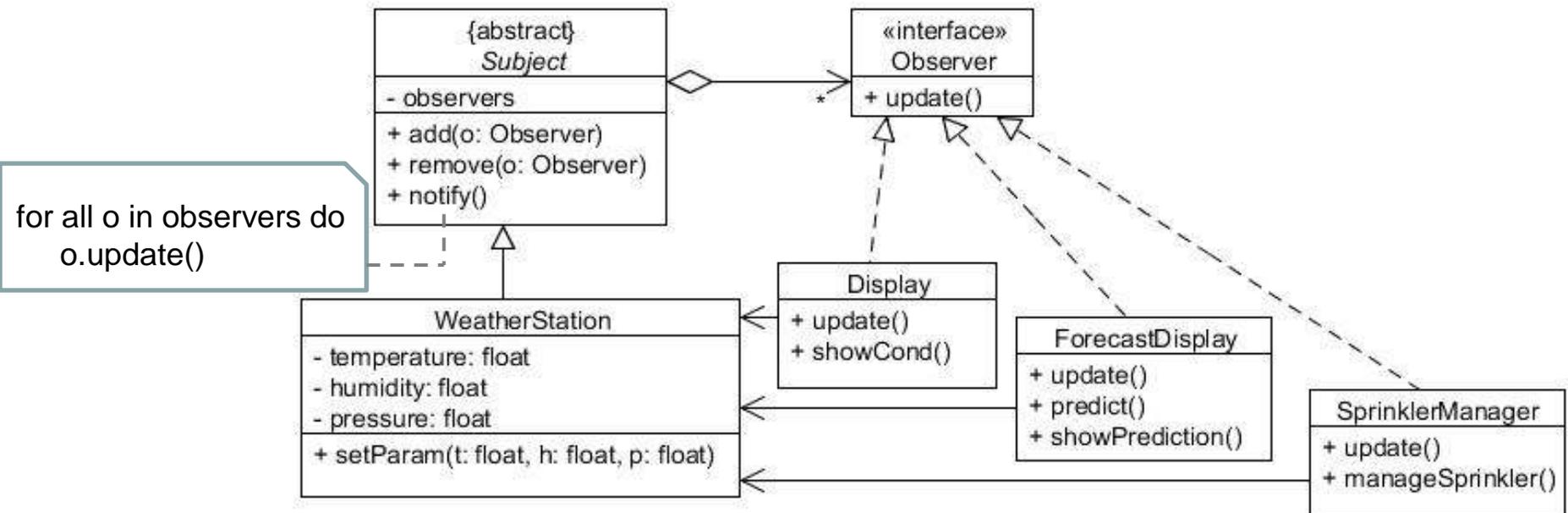
## Problemi:

- per ciascun **nuovo oggetto** interessato alle condizioni atmosferiche, è necessario **modificare la classe** *WeatherStation*;
- **non** è possibile **aggiungere dinamicamente oggetti** dipendenti dalle condizioni atmosferiche.

Vorremmo potere **aggiungere, dinamicamente,** *osservatori* delle condizioni atmosferiche, senza  
modificare le classi già esistenti

# Observer: Motivazione

## Implementazione con pattern *Observer*



- **Observer**: interface implementata dai vari “osservatori” (metodo *update()*)
- **WeatherStation** ha **attributi e metodi** (ereditati) per gestire l’aggiornamento degli “osservatori”
- Possibilità di **modificare, dinamicamente, il numero di “osservatori”**

# Observer: Motivazione

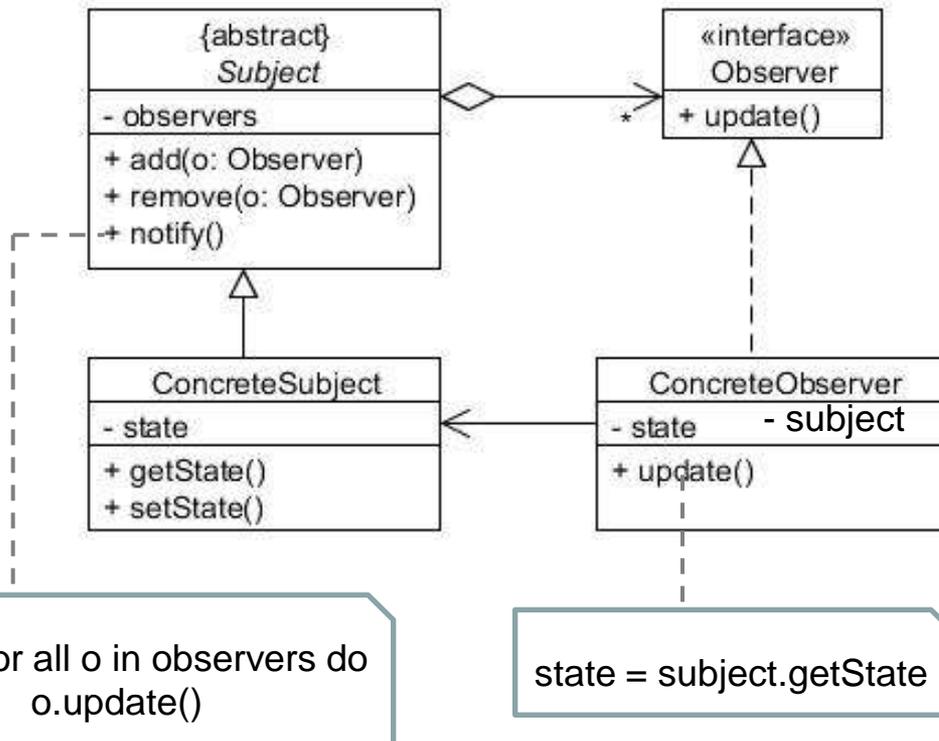
## Altri esempi (più *informatici*)

- Gestione di **interfacce grafiche**
  - Oggetti che dipendono dagli **eventi di interazione con l'interfaccia grafica**
- Dati nel **foglio di calcolo**
  - Dipendenza dal **valore** e dai **cambiamenti di valore di una cella** (grafici, funzioni, altre celle)

# Observer: Applicabilità

- quando **alcuni oggetti dipendono dai dati di un altro oggetto** e hanno necessità di **essere informati** quando questi dati cambiano;
- quando una classe presenta **molti attributi**, buona parte dei quali sono riferimenti a **oggetti che dipendono dal suo stato**.

# Observer: Struttura e Partecipanti



- Subject
  - classe **astratta**
  - contiene i metodi per la **gestione degli “osservatori”**
- ConcreteSubject
  - implementazione vera e propria dell’**oggetto da realizzare**
  - **eredita** attributi e metodi per la **gestione di “osservatori”**
- Observer
  - **Interfaccia** comune a tutti gli osservatori (metodo *update()*)
- ConcreteObserver
  - Implementa il metodo *update()* nel modo più opportuno

# Observer: Collaborazioni

- ConcreteSubject

- contiene un attributo, *observers*, che rappresenta **l'insieme degli oggetti interessati al suo stato**;
- mette a disposizione, degli oggetti interessati, i metodi *add(o:Observer)* e *remove(o:Observer)* che permettono di **aggiungere o togliere un riferimento a loro stessi** all'attributo *observers*;
- a “ogni cambiamento”, invoca *notify()* per informare tutti gli oggetti il cui riferimento appartiene a *observers*.

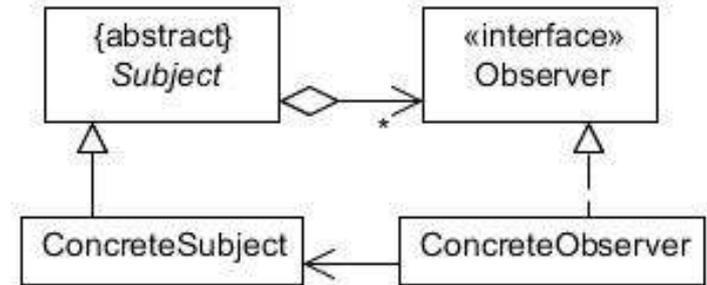
- ConcreteObserver

- si “**registra**” a uno specifico *ConcreteSubject*, qualora sia interessato ai cambiamenti di stato di quest'ultimo;
- **implementa** in modo specifico il metodo *update()* (interfaccia nota a *ConcreteSubject*).

# *NOTA: Significato dei nomi del pattern*

- **Observer**

- Ci sono oggetti interessati a *osservare* i **cambiamenti** dell'oggetto *ConcreteSubject*



- **Publisher-Subscriber**

- *Editore-Abbonato*
- Gli oggetti *ConcreteObserver* si *abbonano ai cambiamenti pubblicati* dall'oggetto *ConcreteSubject* (cfr. metodi *add(o:Observer)* e *remove(o:Observer)* ereditati da *Subject*)

- **Delegation Event Model (Java)**

- Il *publisher* delega la gestione degli eventi ai vari *subscriber* (detti *Listener*)

# Observer: Conseguenze

- Vantaggi

- **Accoppiamento lasco** fra un *ConcreteSubject* e i suoi *ConcreteObserver*
  - » Il subject sa **solo** che gli **observer** implementano **una certa interfaccia**
  - » Si possono **aggiungere dinamicamente nuovi observer**
  - » **Nessuna modifica statica** alla classe *ConcreteSubject* (almeno per quanto riguarda gli observer)
  - » Le **modifiche** (alle classi osservatrici) **non** devono essere **propagate**
  - » Ciascun **observer** può “**abbonarsi**” a **molti subject diversi**
- Possibilità di **costruire dinamicamente le collaborazioni**

# Observer: Conseguenze

- **Svantaggi**
  - **Prestazioni peggiori** (in particolare per alcune implementazioni)
  - **Difficoltà nella comprensione delle dipendenze** fra oggetti (in fase di **testing** e **debugging**)
    - » quanti osservatori dipenderanno dal soggetto
    - » a quali soggetti un osservatore potrà riferirsi

# Observer: Implementazione

- Un *observer* può essere abbonato a più *subject*
  - Come capisce a **quale subject** una **notifica ricevuta** si riferisce?
    - Il *subject* passa un **riferimento a se stesso** come parametro (*notify(this)*)
  - Come mantenere le **associazioni fra soggetti e osservatori**?
    - **Strutture associative**
      - » Permettono di usare **meno spazio** (ad es. soggetto senza osservatori)
      - » Comportano **svantaggi nel tempo** (ricerca dell'oggetto specifico)

# Observer: Implementazione

- Invio delle **notifiche**
  - Al verificarsi di un cambiamento nello stato del *subject*, **chi** deve **inviare la notifica agli observer**?
    - Il **client** del *subject* invoca *notify()* ogni volta che **ne modifica lo stato**
      - » Troppa **responsabilità al client**
      - » Conseguenze pesanti nel caso di **dimenticanze**
    - Il **subject** invoca *notify()* come **ultima istruzione dei metodi di modifica**
      - » Possibili **stati inconsistenti** nel caso di metodi innestati

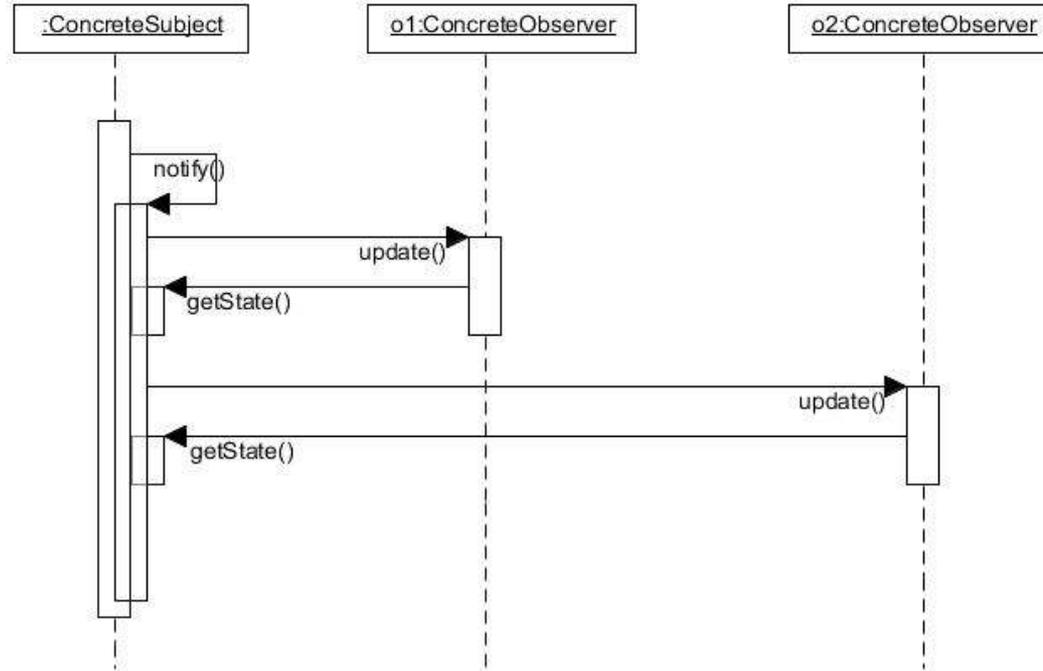
```
public A() {                public B() {
    //do sth                //do sth
    B();                    notify();
    //do sth else
    notify();}
```

# Observer: Implementazione

- Come **propagare le informazioni**
  - Come è possibile fare **conoscere lo stato del *subject* all'*observer*?**
    - **Modello PUSH**: il *subject* invia agli *observer* tutti i **dati**, riguardanti il suo stato, di cui essi necessitano
      - » il *subject* deve fare **assunzioni sulle informazioni utili** ai vari *observer* (deve conoscerli → **dipendenza**)
    - **Modello PULL**: dopo avere ricevuto una notifica, gli *observer* **richiedono al *subject* le informazioni** di cui necessitano (metodi *get()*)
      - » più **flessibile**

# Observer: Implementazione

- Come **propagare le informazioni**



# Observer: Implementazione

- Come gestire **cambiamenti ravvicinati**
  - **Quando propagare** i cambiamenti?
    - Troppo **presto** → lo stato può essere ancora modificato (**inefficienze**)
    - Troppo **tardi** → si rischia di avere **inconsistenze** (fra subject e observer)

è necessario un opportuno **change management**

# Observer: Implementazione

- Quali **cambiamenti significativi**

- Quale dev'essere la “**sensibilità**” del meccanismo di **notifica**?

(notifico ogni minima variazione o solo variazioni *significant*– cfr stazione meteo)

➤ Si utilizza un **flag *changed*** (posto a true quando si ha un cambiamento significativo rispetto all'ultima modifica)

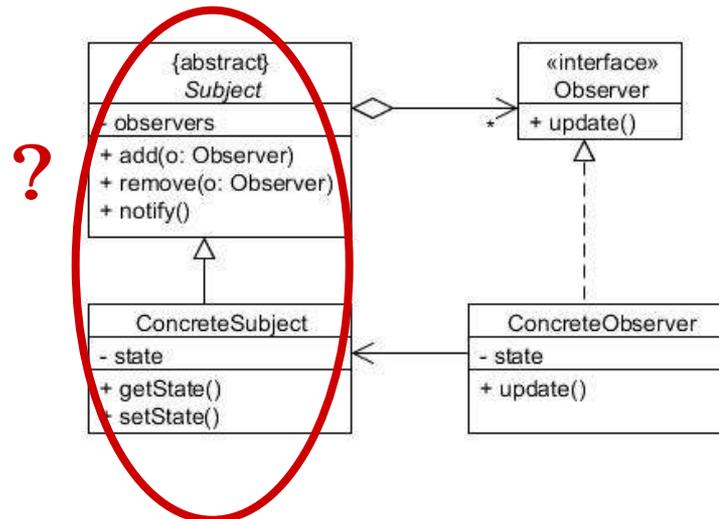
```
setChanged() { changed = true; }

notify() {
    if (changed) {
        for (int i=0; i<observers.size(); i++)
            (observers.elementAt(i)).update();
    }
    changed = false;
}
```

# Observer: Implementazione

- **Linguaggi con ereditarietà semplice**

- Come implementare il **pattern *Observer***, anche se il ***ConcreteSubject*** deve necessariamente **ereditare da altre classi**?



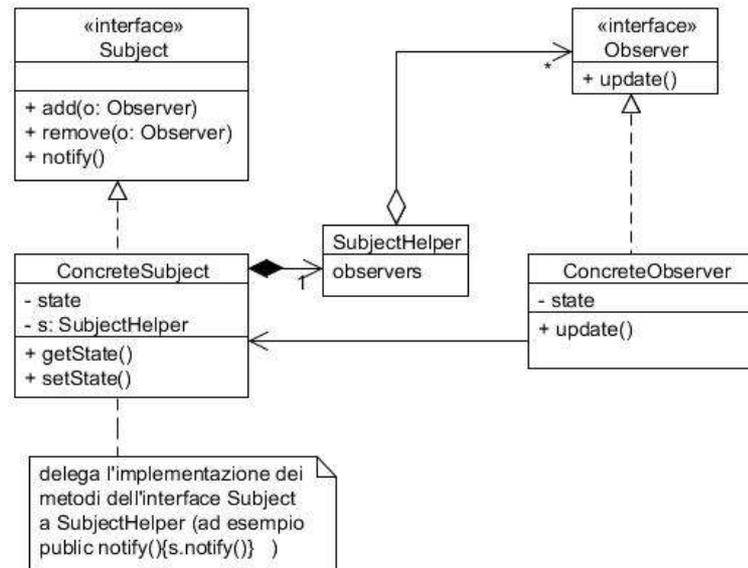
*ConcreteSubject* è già sottoclasse di *Subject*

# Observer: Implementazione

- **Linguaggi con ereditarietà semplice**

- **Prima soluzione:**

- » *Subject* è un **interface** implementato da *ConcreteSubject*
- » *ConcreteSubject* **delega a *SubjectHelper*** l'implementazione dei metodi di *Subject*



**Svantaggio:** il **codice** per la **gestione degli observer** va **ripetuto** in ogni classe `ConcreteSubject`

# Observer: Codice d'esempio

- Implementazione della *stazione meteorologica*

```
public abstract class Subject{
    private Vector observers;

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

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

    public void notify(){
        for(int i=0;i<observers.size();i++)
            (observers.elementAt(i)).update(this);
    }
}
```

# Observer: Codice d'esempio

```
public class WeatherStation extends Subject{
    private double temperature;
    private double humidity;
    private double pressure;

    public WeatherStation(){
        observers = new Vector();
        //acquisizione delle prime misurazioni
    }
    public void setParam(double t, double h, double p){
        temperature = t;
        humidity = h;
        pressure = p;
        notify();
    }

    public double getTemperature(){return temperature;}
    public double getHumidity(){return humidity;}
    public double getPressure(){return pressure;}
}
```

# Observer: Codice d'esempio

```
public interface Observer{public void update (Subject s);}

public class Display implements Observer{
    private double measuredTemp;
    private double measuredHum;
    private double measuredPress;

    public Display(...){
        //acquisizione delle misure}

    public void update (Subject s){
        measuredTemp = s.getTemperature();
        measuredHum = s.getHumidity();
        measuredPress = s.getPressure();
        showCond();}

    public void showCond(){
        //visualizzazione delle condizioni meteo
    }
}
```

# Observer: Codice d'esempio

```
public class ForecastDisplay implements Observer{
    private double measuredTemp;
    private double measuredHum;
    private double measuredPress;

    public ForecastDisplay(...){
        //acquisizione delle misure e previsione}

    public void update(Subject s){
        measuredTemp = s.getTemperature();
        measuredHum = s.getHumidity();
        measuredPress = s.getPressure();
        predict();showPrediction();}

    public void predict(){
        //si effettuano previsioni
        //sulla base delle condizioni attuali}
    public void showPrediction(){
        //visualizzazione delle previsioni meteo}}
```

# Observer: Codice d'esempio

```
public class SprinklerManager implements Observer{
    private double measuredTemp;
    private double measuredHum;
    private double measuredPress;
    //riferimenti agli irrigatori

    public Display(...){
        //acquisizione delle misure
        //inizializzazione riferimenti agli irrigatori}

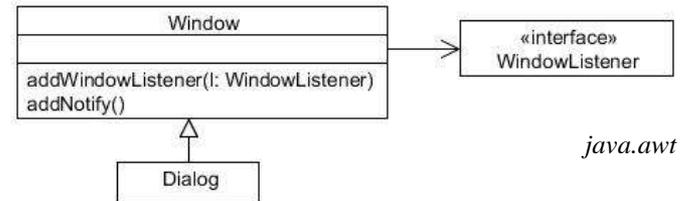
    public void update(Subject s){
        measuredTemp = s.getTemperature();
        measuredHum = s.getHumidity();
        measuredPress = s.getPressure();
        manageSprinkler();}

    public void manageSprinkler(){
        //decide cosa fare sulla base delle condizioni}
}
```

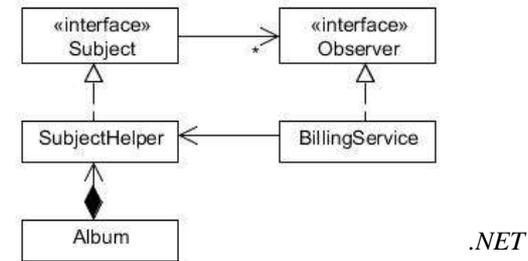
# Observer: Utilizzi noti

- Gestori di interfacce grafiche

- Librerie *Java AWT* e *Swing*  
(N.B. esistono *Observable* e *Observer*)



- Framework *Microsoft .NET*



- Altro

- Gestione di eventi di alarm clock
- Altri pattern

# Observer: Pattern correlati

- **Model-View-Controller**
  - La **relazione fra *model* e *view*** può essere espressa mediante una relazione tipo **pattern *observer***

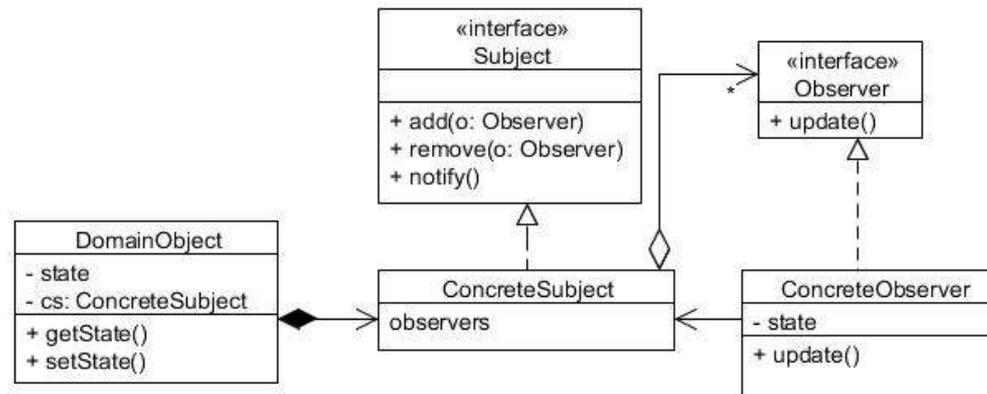
# Observer: Implementazione

- **Linguaggi con ereditarietà semplice**

- **Seconda soluzione:**

- » *Subject* è un **interface** implementato da *ConcreteSubject*

- » L'**oggetto del dominio** è rappresentato dalla classe *DomainObject*, che ha, fra i suoi **attributi**, un oggetto di tipo *ConcreteSubject* (si occupa della gestione degli *observer*)



**Solo *ConcreteSubject* implementa** i metodi di *Subject*

*DomainObject* può avere **più attributi di tipo *ConcreteSubject*** (può **differenziare gli *observer*** in base agli eventi cui sono interessati)