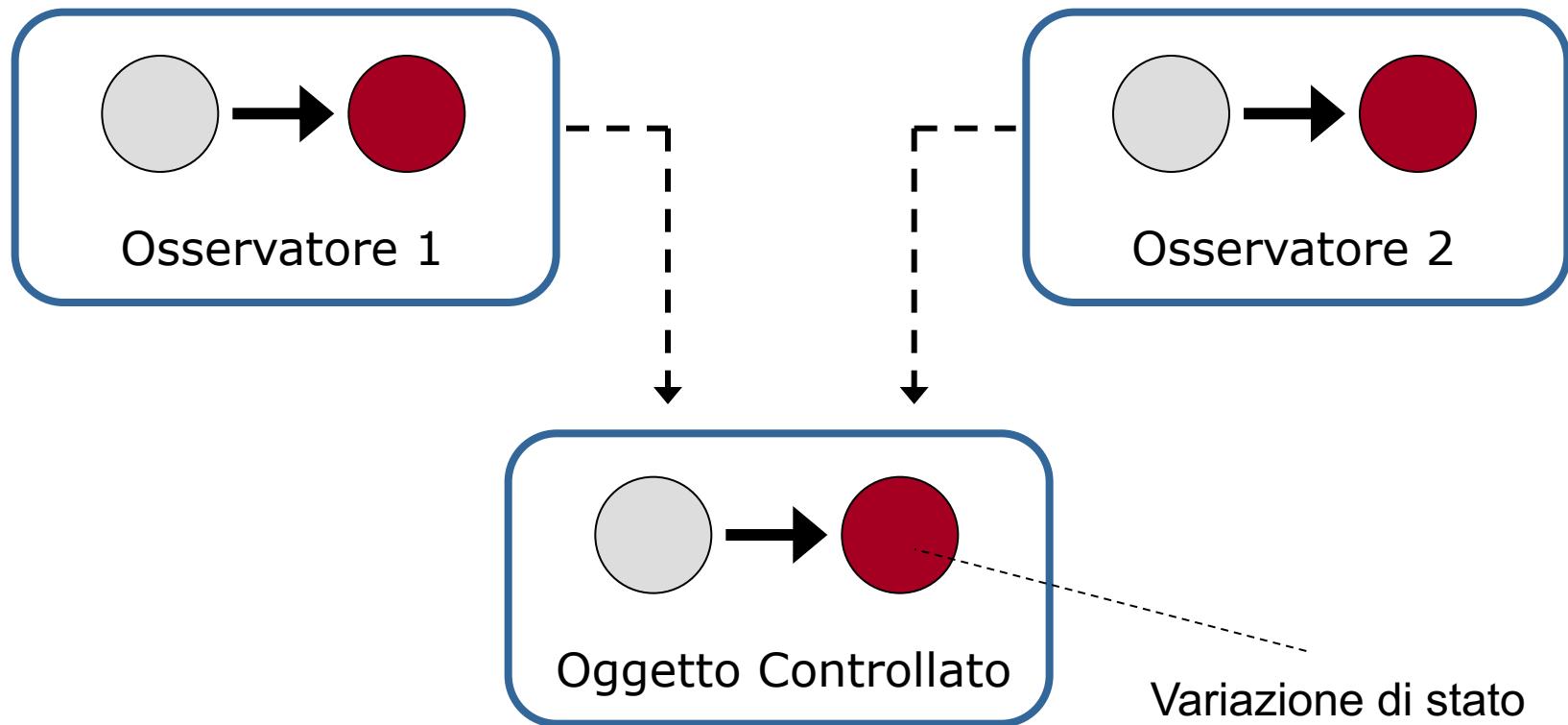


Observer

- ◆ **Scopo:** Definire una dipendenza uno a molti tra oggetti, mantenendo basso il grado di coupling. In altre parole la variazione dello stato di un oggetto deve essere osservata da altri oggetti, in modo che possano aggiornarsi automaticamente.
- ◆ **Motivazione:** Lo scenario classico è quello di applicazioni con GUI, realizzate secondo il paradigma Model-View-Control. Quando il Model cambia, gli oggetti che implementano la View devono aggiornarsi

Classificazione: comportamentale basato su oggetti

Observer: l'idea di fondo



Observer: una possibile soluzione

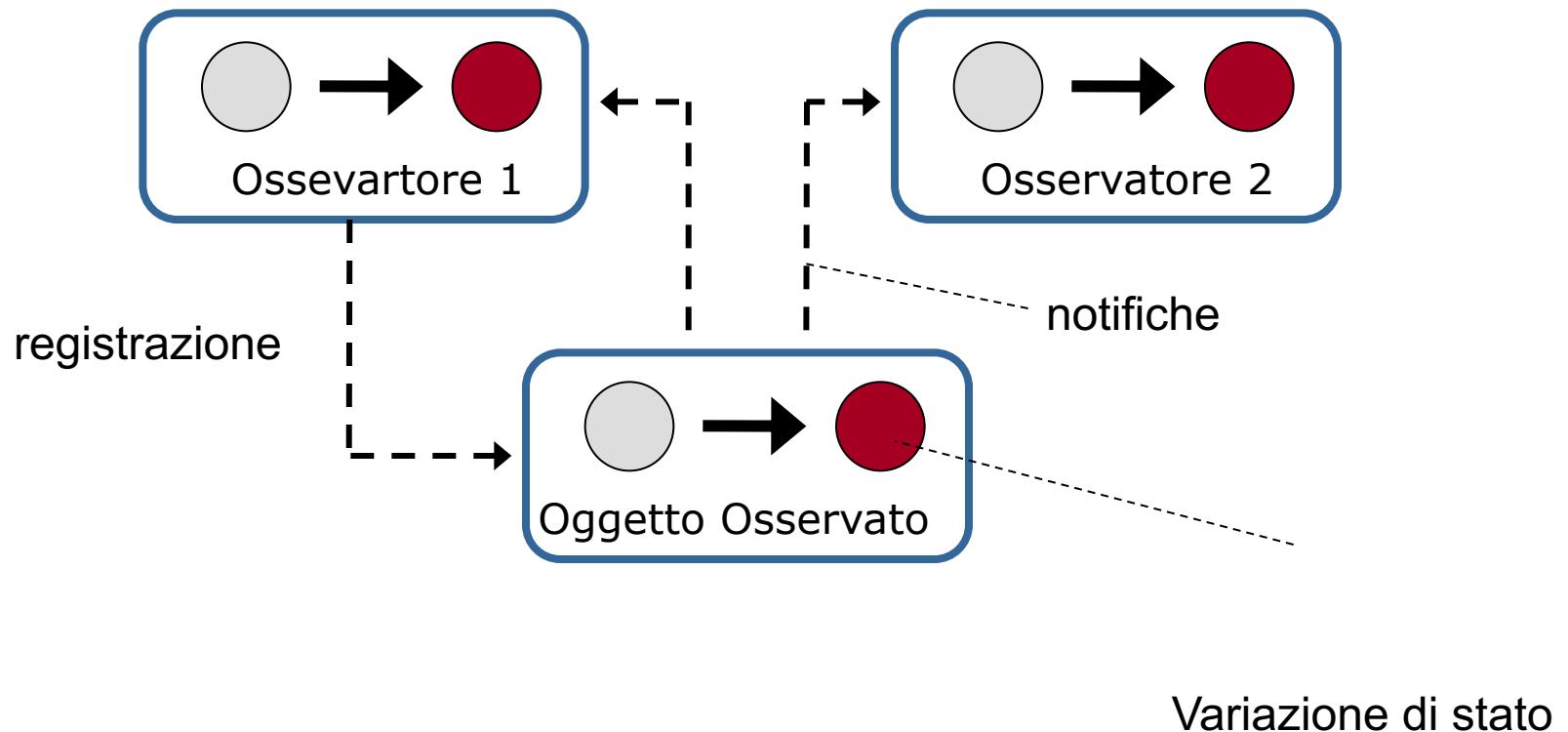
- ◆ Una prima soluzione potrebbe essere quella di utilizzare, nell'oggetto osservato, **attributi pubblici** oppure **metodi pubblici** che leggono il valore di un attributo protetto
- ◆ Non è una buona soluzione:
 - Non è scalabile
 - se aumentano troppo gli osservatori l'oggetto osservato è sovraccaricato dalle richieste
 - Gli osservatori dovrebbero continuamente interrogare l'oggetto osservato
 - Variazioni rapide potrebbero comunque non essere rilevate da qualche osservatore

Observer: l'approccio corretto

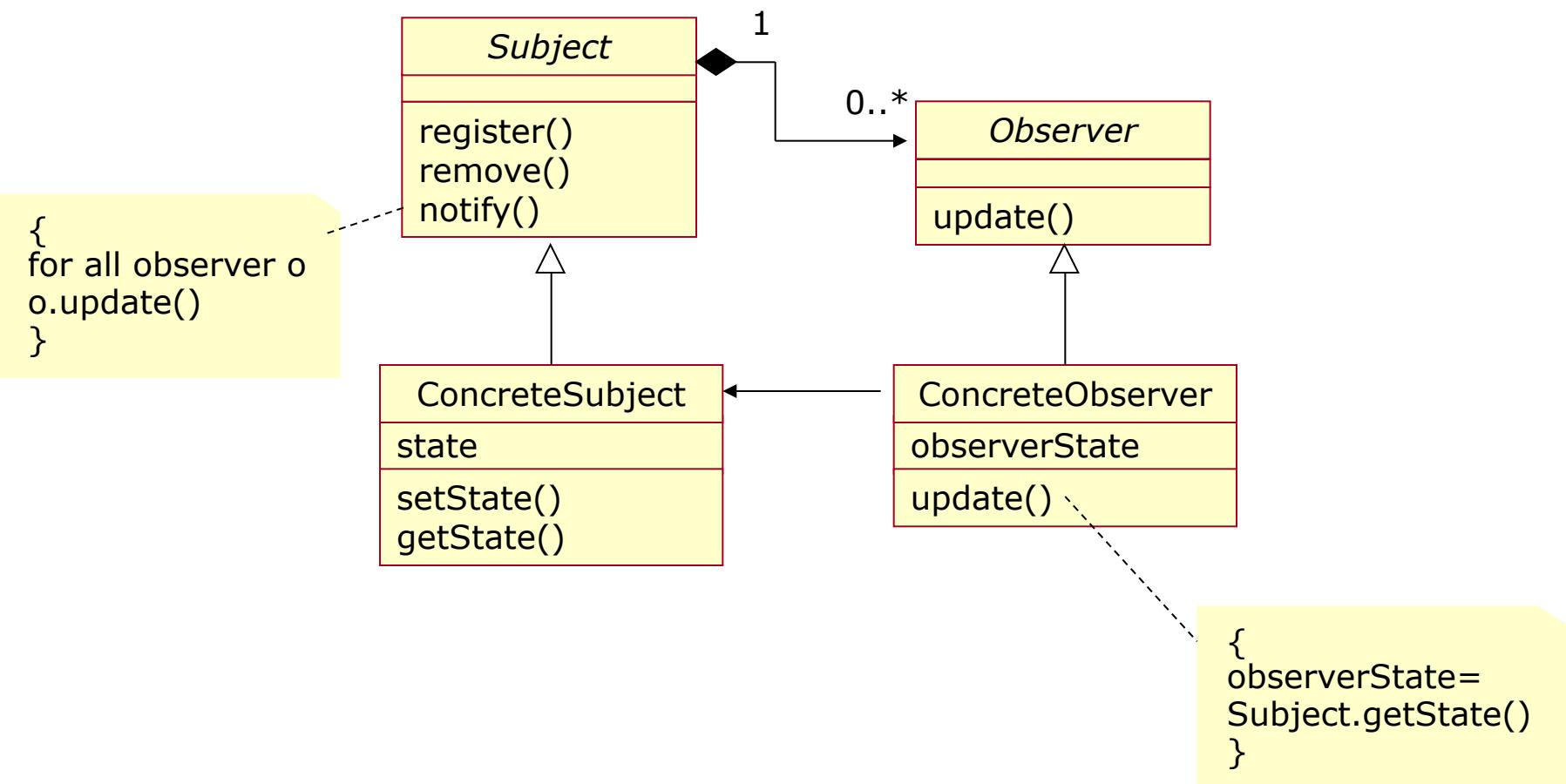
- ◆ Il pattern Observer prevede che gli osservatori **si registrino** presso l'oggetto osservato
- ◆ In questo modo è l'oggetto osservato che **notifica** ogni cambiamento di stato agli osservatori
- ◆ Quando l'osservatore rileva la notifica può interrogare l'oggetto osservato, oppure può svolgere altre operazioni indipendenti dal valore specifico dello stato

Observer: l'approccio corretto (2)

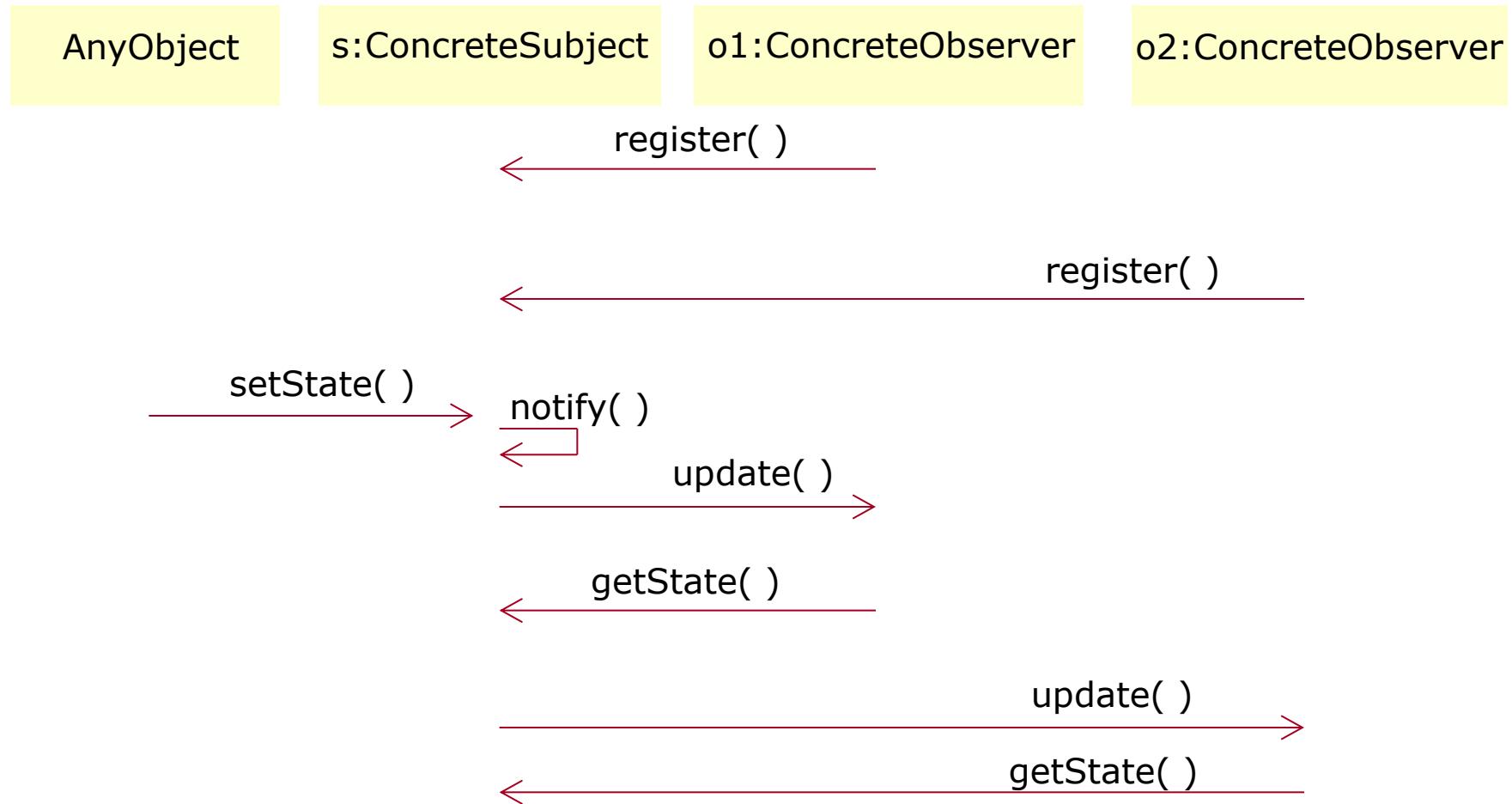
- ◆ Gli osservatori possono essere aggiunti a runtime



Observer: struttura



Observer: sequence diagram



Altre caratteristiche

◆ **Applicabilità**

- Si applica quando una azione può essere scomposta in due ambiti, ciascuno dei quali encapsulato in oggetti separati per mantenere basso il livello di coupling.
- Gestire le modifiche di oggetti consequenti alla variazione dello stato di un oggetto.

◆ **Partecipanti**

- Subject e ConcreteSubject
- Observer e ConcreteObserver

Altre caratteristiche (2)

◆ Conseguenze

- L'accoppiamento tra *Subject* ed *Observer* è astratto
 - il *Subject* conosce solo la lista degli osservatori
- La notifica è una comunicazione di tipo broadcast
 - il *Subject* non si occupa di quanti sono gli *Observer* registrati
- Attenzione perché una modifica al *Subject* scatena una serie di modifiche su tutti gli osservatori e su tutti gli oggetti da questi dipendenti

Template Method

- ◆ **Scopo:** Definire la struttura di un algoritmo all'interno di un metodo, delegando alcuni passi alle sottoclassi

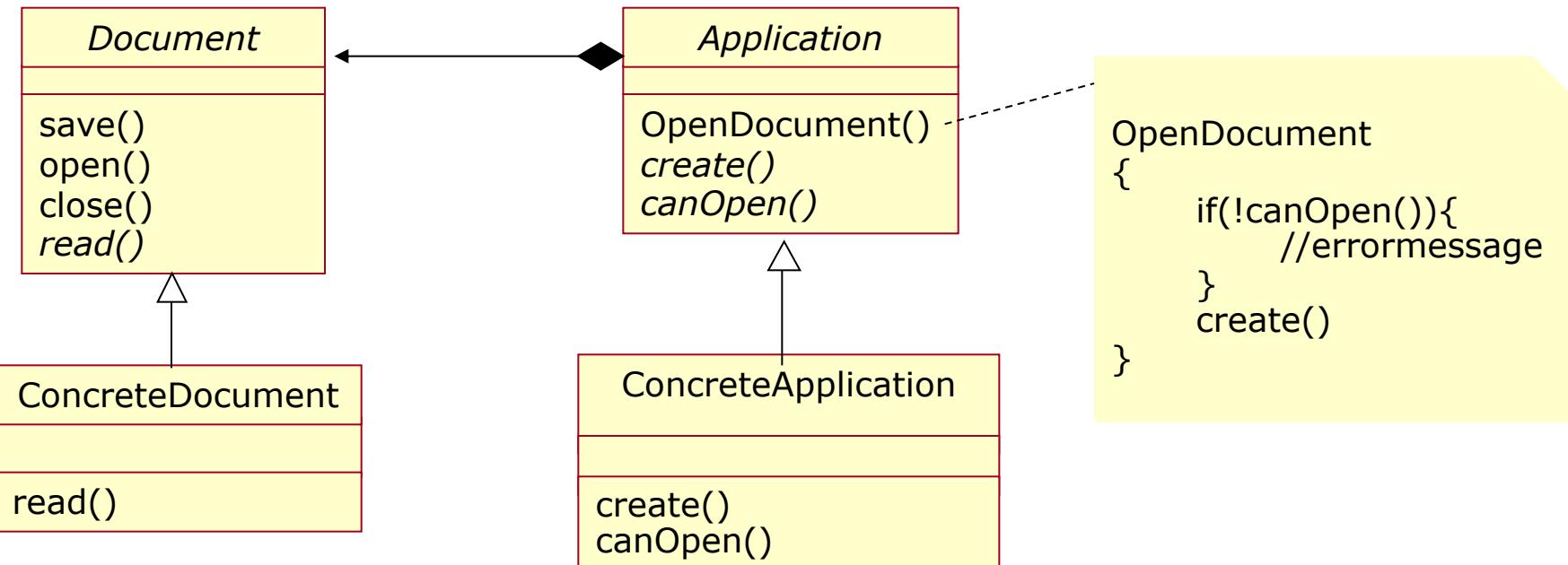
Le sottoclassi ridefiniscono solo alcuni passi dell'algoritmo ma non la sua struttura

- ◆ **Motivazione:** consideriamo un framework per costruire applicazioni in grado di gestire documenti diversi. Il Template Method definisce un algoritmo in base ad operazioni astratte che saranno definite nelle sottoclassi specifiche.

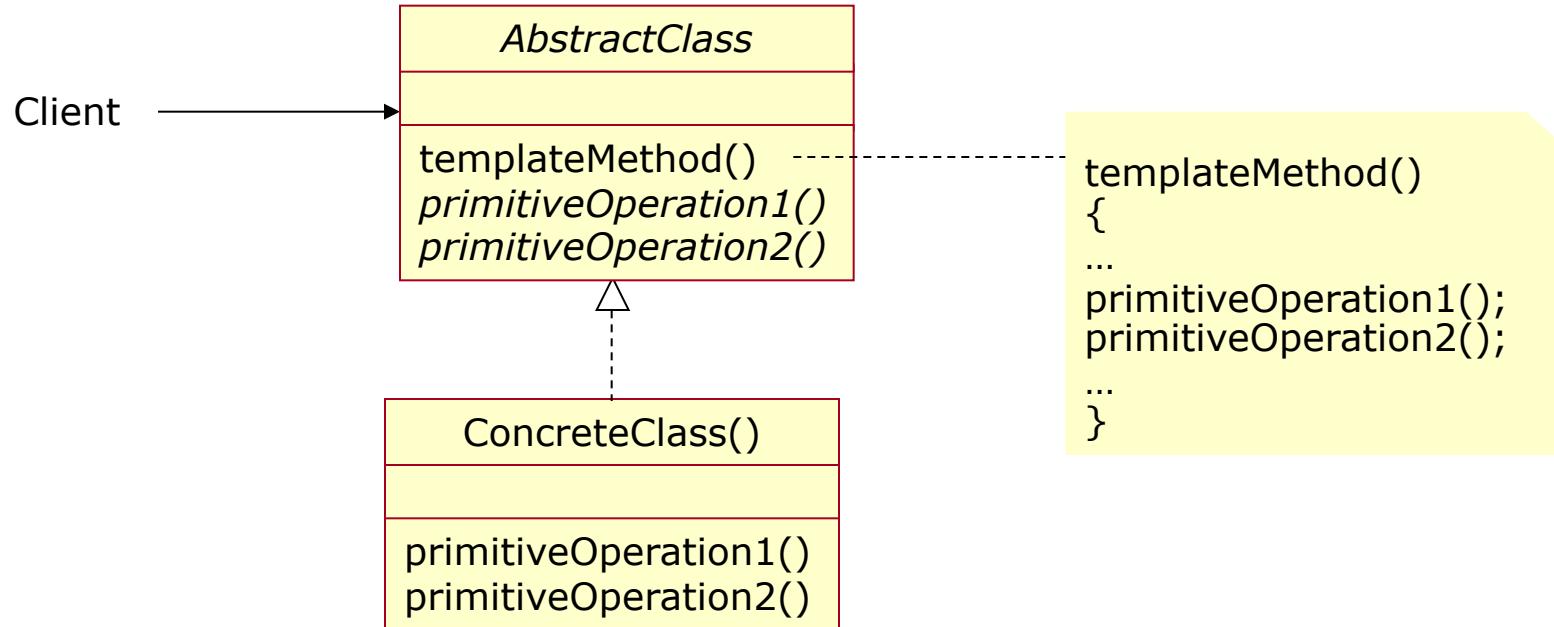
Classificazione: comportamentale basato su classi

Template Method: esempio

- ◆ Consideriamo quindi un semplice framework costituito da 2 classi: Application e Document.



Template Method: struttura



Altre caratteristiche

◆ **Applicabilità**

- E' utilizzato per implementare la parte invariante di un algoritmo, lasciando alle sottoclassi la definizione degli step variabili
- E' utile quando ci sono comportamenti comuni che possono essere inseriti nel template

◆ **Partecipanti**

- AbstractClass e ConcreteClass
- Client

Altre caratteristiche

◆ Conseguenze

- I metodi template permettono il riuso del codice
- Creano una struttura di controllo invertito dove è la classe padre che chiama le operazioni ridefinite nei figli e non viceversa
- Per controllare l'estendibilità delle sottoclassi, i metodi richiamati dal template sono chiamati metodi **gancio** (*hook*)
- I metodi *hook* possono essere implementati, offrendo un comportamento standard, che la sottoclasse può volendo ridefinire

Alcune note

- ◆ Il Template Method è simile al Factory Method
 - Invocazione di metodi astratti tramite interfaccia
 - Implementazione dei metodi rimandata a classi concrete non note
- ◆ Indirizzano però problemi diversi
 - Il Template Method è il metodo che invoca i metodi astratti, al fine di **generalizzare un algoritmo**
 - Il Factory Method è un metodo astratto che deve creare e restituire l'istanza di classe concreta, al fine di **sganciare il cliente dalla scelta del tipo specifico**

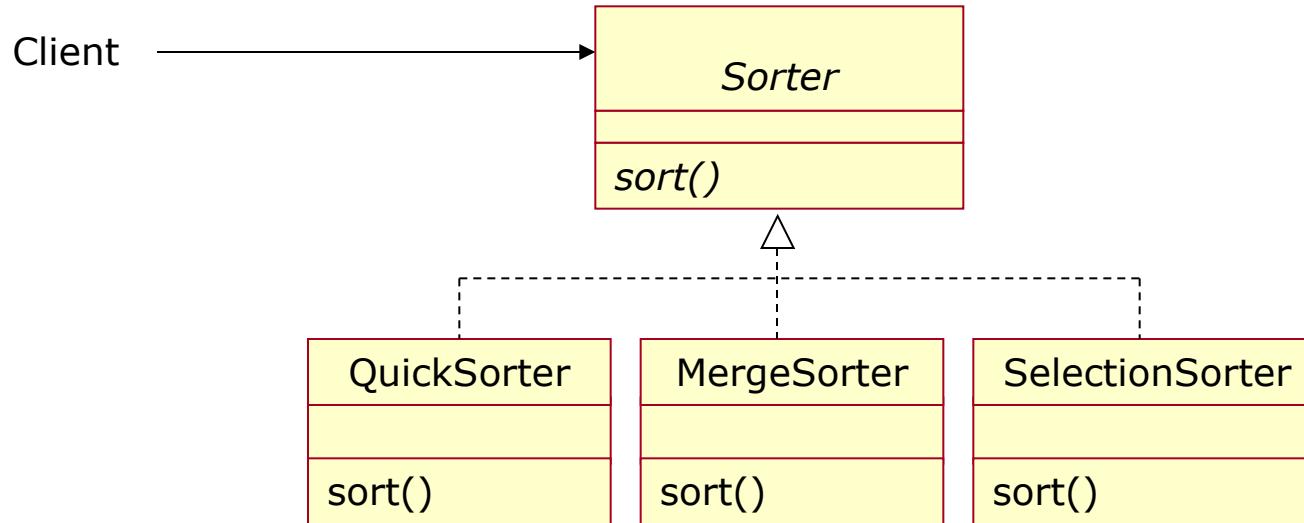
Strategy

- ◆ **Scopo:** Definire ed encapsulare una famiglia di algoritmi in modo da renderli intercambiabili indipendentemente dal client che li usa.
- ◆ **Motivazione:** Consideriamo la famiglia degli algoritmi di ordinamento. Ne esistono diversi (QuickSort, BubbleSort, MergeSort, etc). Costruiamo una applicazione che li supporti tutti, che possa essere facilmente estendibile, e che permetta una scelta rapida del tipo di algoritmo

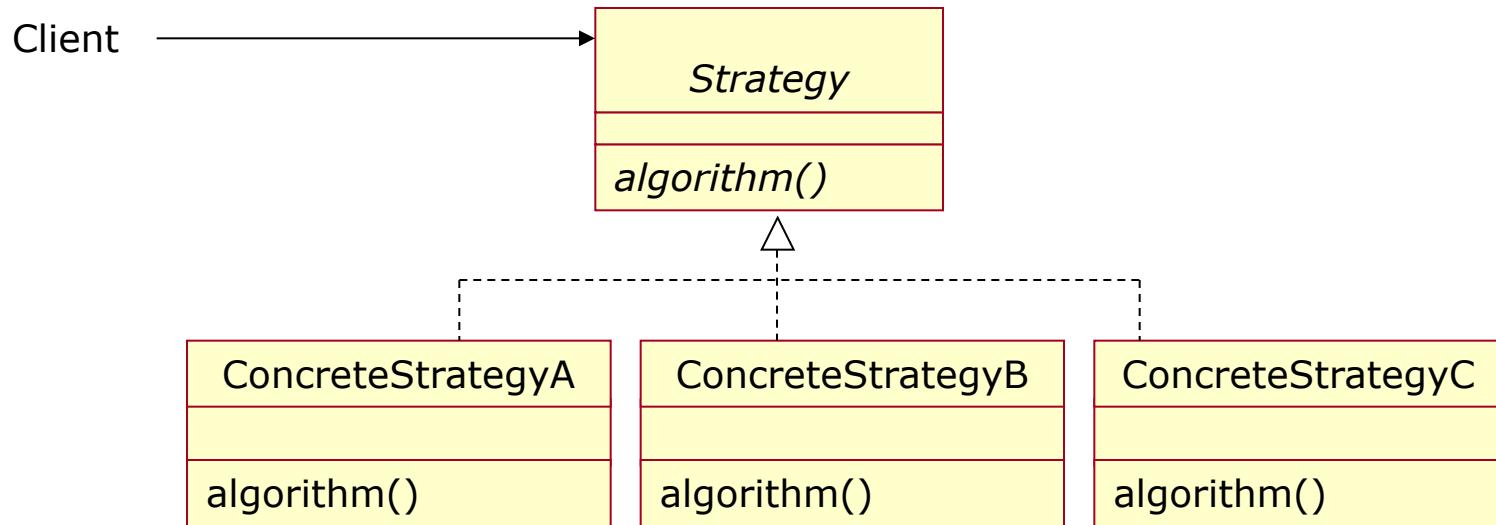
Classificazione: comportamentale basato su oggetti

Strategy : esempio

Gli algoritmi di ordinamento devono essere indipendenti dal vettore di dati su cui operano, e dal resto dell'implementazione dell'applicazione



Strategy: struttura



Altre caratteristiche

◆ **Applicabilità**

- Molte classi correlate differiscono solo per il comportamento
 - Il pattern fornisce un modo per avere una interfaccia comune.
- Sono necessarie più varianti di uno stesso algoritmo, a seconda dei tipi di dato in ingresso o delle condizioni operative.

◆ **Partecipanti**

- Strategy e ConcreteStrategy
- Client

Altre caratteristiche

◆ Conseguenze:

- Il pattern separa l'implementazione degli algoritmi dal contesto dell'applicazione

usare il *subclassing* della classe Client per aggiungere un algoritmo non sarebbe stata una buona scelta.

- Le diverse strategie eliminano i blocchi condizionali che sarebbero necessari inserendo tutti i diversi comportamenti in una unica classe
- Lo svantaggio principale è che i client devono conoscere le diverse strategie

Design Pattern - Esercizi

Esercizio 1

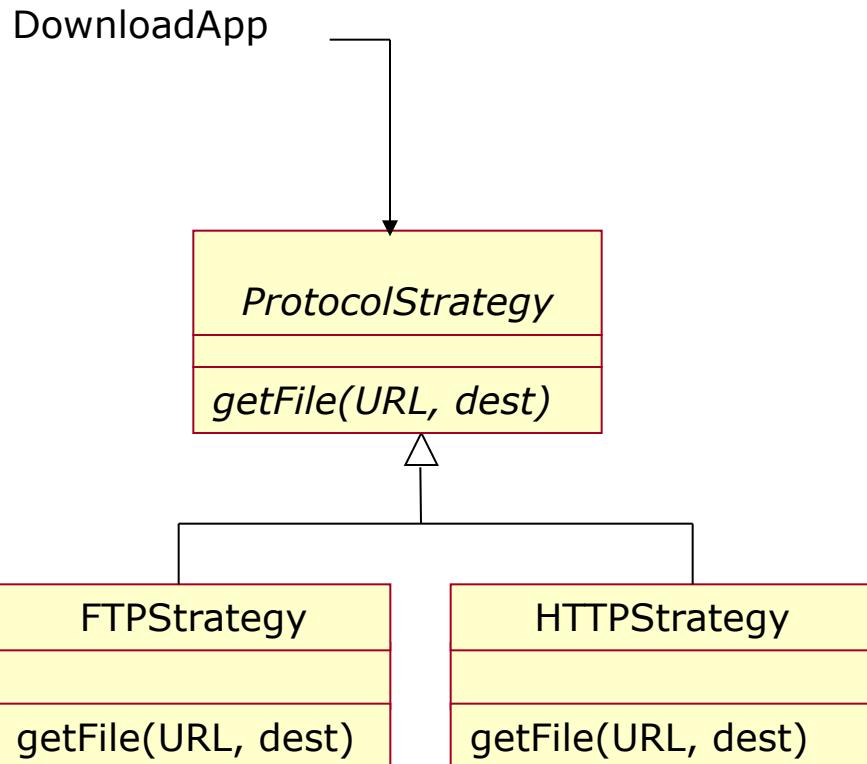
- ◆ Realizzare una applicazione per gestire il download da siti http e ftp. La selezione avviene in base all'inizio dell'url (ftp o http)

suggerimento: usare una factory e uno strategy pattern....

Soluzione 1/1

- ◆ L'applicazione deve gestire il download da siti supportando solo due protocolli
- ◆ E' ragionevole scrivere una applicazione che sia estendibile senza problemi (manutenzione evolutiva)
- ◆ Usiamo un pattern Strategy per implementare il download secondo i due protocolli

Soluzione 1/2

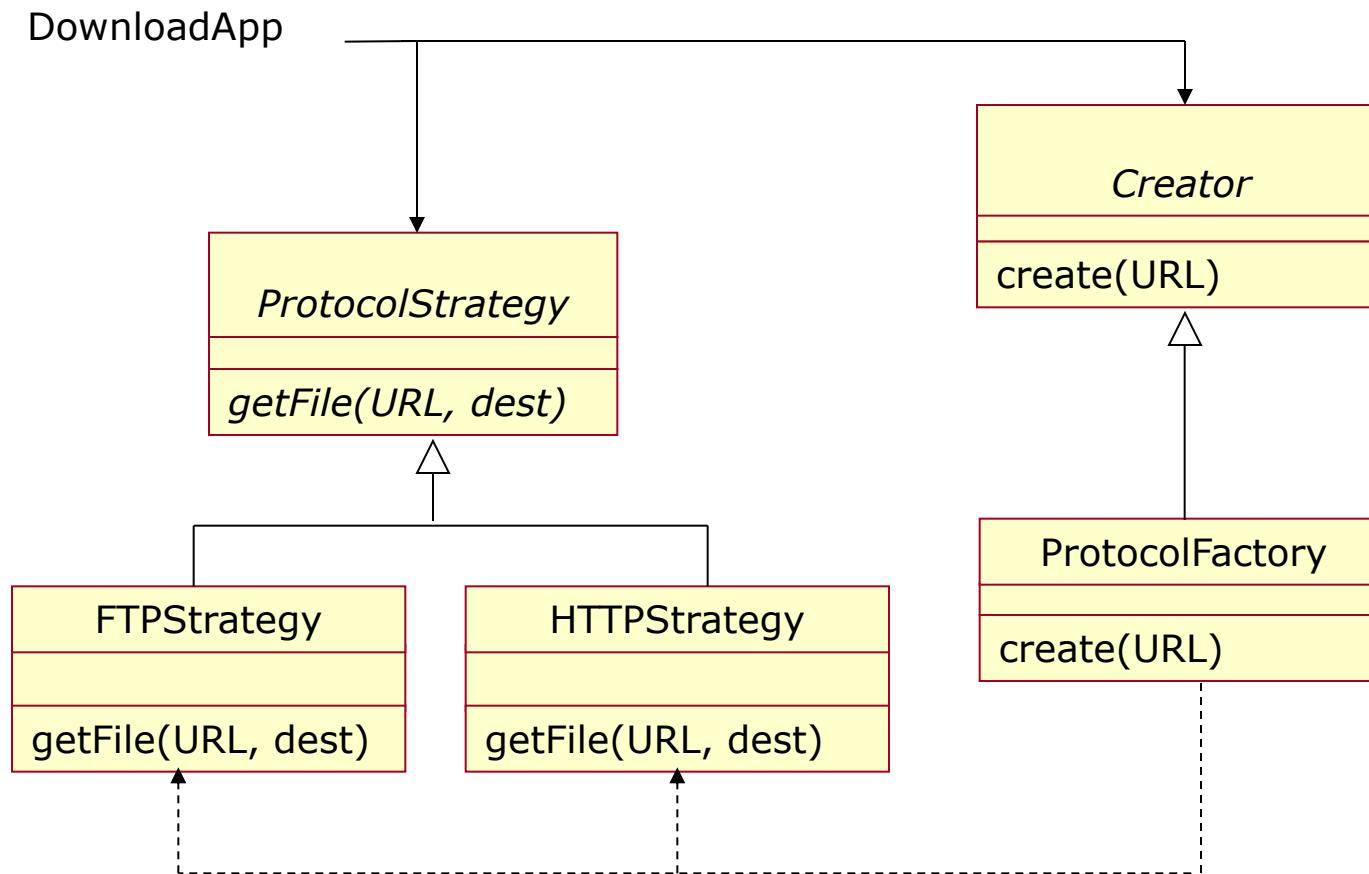


```
download( String url, String dest ) {
    ProtocolStrategy ps=null;
    url = url.toLowerCase();
    if( url.startsWith( "ftp" ) ) {
        ps = new FtpStrategy();
    }
    else if( url.startsWith( "http" ) ) {
        ps = new HttpStrategy();
    }
    else {
        //No operation available
    }
    ps.getFile( url, dest );
}
```

Soluzione 1/3

- ◆ In questo modo ogni volta che vogliamo aggiungere un nuovo protocollo dobbiamo modificare la classe DownloadApp
- ◆ DownloadApp in questo scenario deve conoscere alcuni dettagli implementativi: ad esempio deve sapere esattamente quali protocolli sono supportati e come si chiamano le relative classi di gestione
- ◆ Introduciamo quindi una Factory.

Soluzione 1/4



Soluzione 1/5

```
abstract class Creator {  
    public abstract ProtocolStrategy create( String url );  
}
```

La Factory Contiene
la logica per creare
la corretta
sottoclasse di
ProtocolStrategy

```
class ProtocolFactory extends Creator {  
    public ProtocolStrategy create( String ind ){  
        ind = ind.toLowerCase();  
        if( ind.startsWith( "ftp" ) ) {  
            return new FtpStrategy();  
        }  
        else if( ind.startsWith( "http" ) ) {  
            return new HttpStrategy();  
        }  
        else {  
            //throw an exception...  
        }  
    }  
}
```

Soluzione 1/6

```
abstract class ProtocolStrategy {  
    abstract void getFile(String ind, String dest);  
}
```

```
class FtpStrategy extends ProtocolStrategy {  
    public void getFile(String url, String dest) {  
        //...method implementation  
    }  
}
```

```
class Download{  
    ...  
    download( String url, String dest ) {  
        ProtocolFactory p = new ProtocolFactory();  
        ProtocolStrategy ps = p.create(url);  
        ps.getFile(url,dest);  
    }  
}
```

Esercizio 2

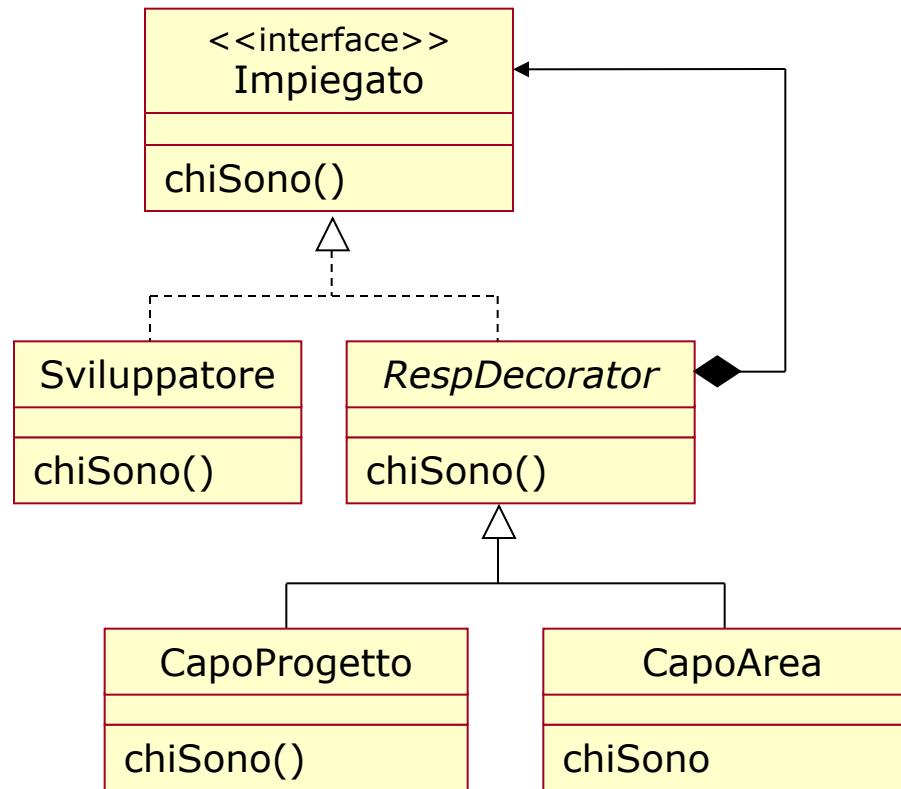
- ◆ Descrivere un modello a oggetti che rappresenti gli *Impiegati* di una azienda.
Gli impiegati hanno tutti il metodo *chiSono()* che visualizza il nome e la particolare responsabilità
- ◆ Gli impiegati possono avere delle responsabilità aggiuntive: possono essere *CapoArea* o *CapoProgetto*, in modo non esclusivo.
- ◆ Una particolare categoria di impiegati che ci interessa sono gli *Sviluppatori*

suggerimento: usare il Decorator Pattern per le responsabilità, e l'ereditarietà per contraddistinguere gli Sviluppatori dagli Impiegati

Soluzione 2/1

- ◆ Dall'esame dei requisiti si può provare a disegnare una struttura con queste caratteristiche:
 - Deve esistere una classe Impiegato che definisce una interfaccia comune (metodo *chiSono()*)
 - Deve esistere una classe Sviluppatore che eredita da Impiegato
 - Definiamo con il pattern Decorator due classi CapoArea e CapoProgetto

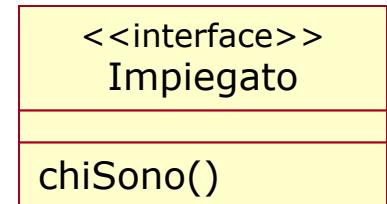
Soluzione 2/2



Soluzione 2/3

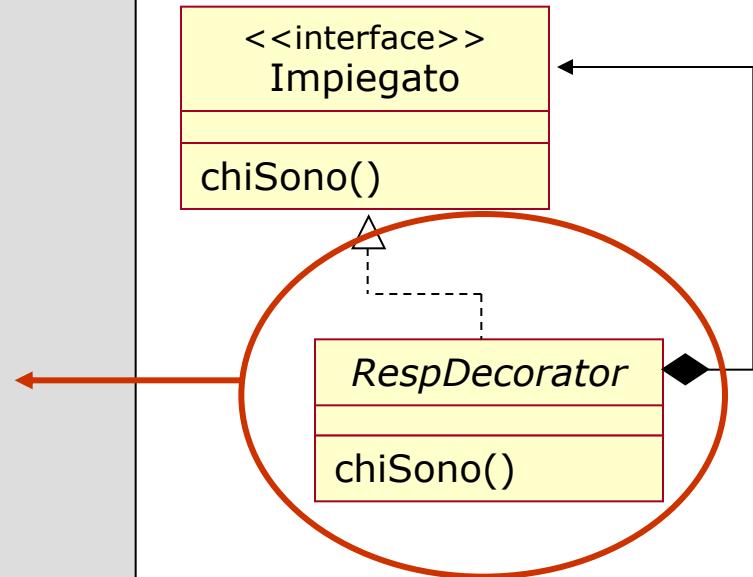
```
public interface Impiegato {  
    public void chiSono();  
    public void getName();  
}
```

```
public class Sviluppatore implements Impiegato {  
    private String nome;  
    public Sviluppatore( String nome) {  
        this.nome = nome;  
    }  
    public String getName () {  
        return nome;  
    }  
    public void chiSono() {  
        System.out.println( "Salve, sono lo  
sviluppatore " + getName());  
    }  
}
```



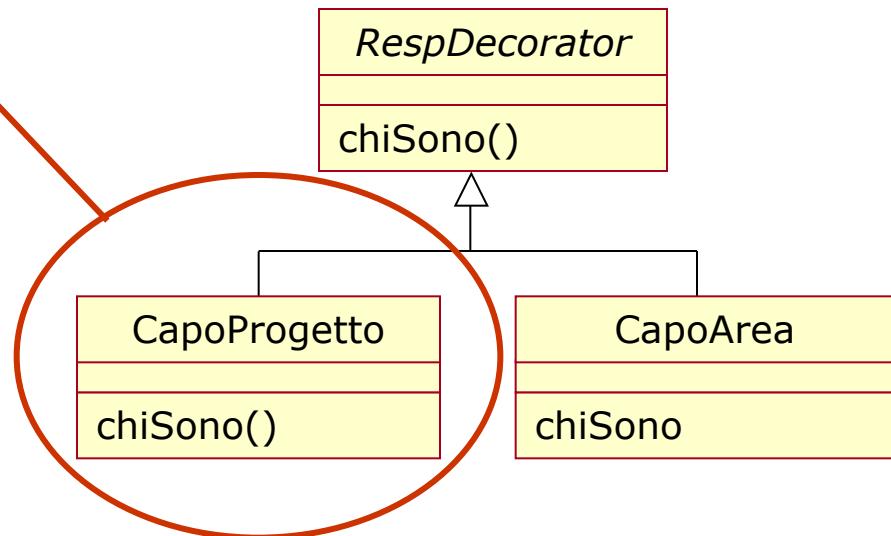
Soluzione 2/4

```
abstract class RespDecorator implements Impiegato {  
  
    protected Impiegato responsabile;  
  
    public RespDecorator(Impiegato imp) {  
        responsabile = imp;  
    }  
  
    public String getName() {  
        return responsabile.getName();  
    }  
  
    public void chiSono() {  
        responsabile.chiSono();  
    }  
}
```



Soluzione 2/5

```
public class CapoProgetto extends RespDecorator {  
    public CapoProgetto ( Impiegato imp ) {  
        super( imp );  
    }  
  
    public void chiSono() {  
        super.chiSono();  
        System.out.println("e sono anche un CapoProgetto!")  
    }  
}
```



Soluzione 2/6

```
Impiegato pippo = new Sviluppatore("Pippo");  
pippo.chiSono();
```

Salve, sono lo sviluppatore Pippo

```
pippo = new CapoProgetto(pippo));  
pippo.chiSono();
```

Salve, sono lo sviluppatore Pippo e sono anche un CapoProgetto!

```
Impiegato pluto = new CapoProgetto(new  
                    Sviluppatore("Pluto"));  
pluto.chiSono();
```

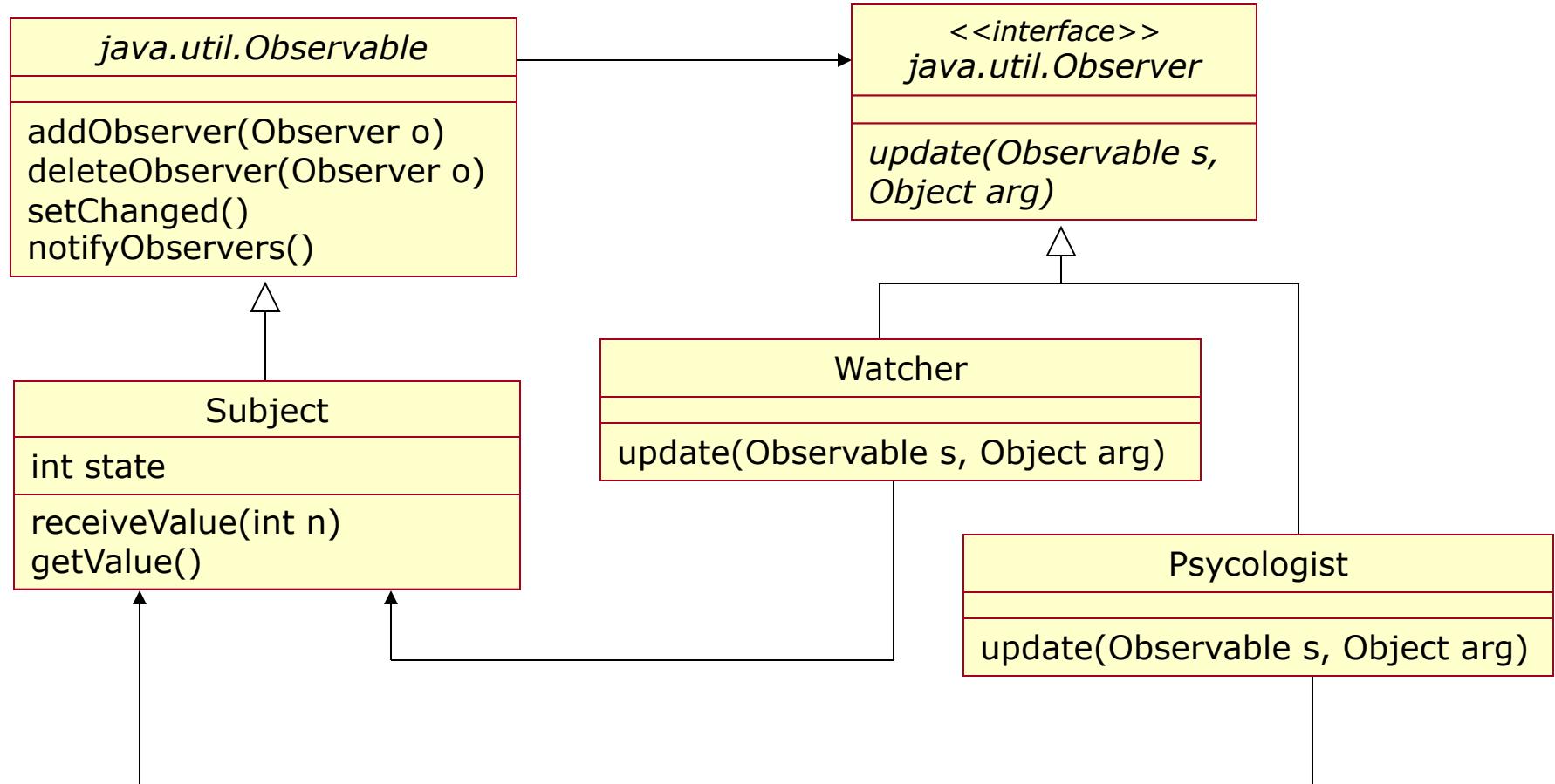
Salve, sono lo sviluppatore Pippo e sono anche un CapoProgetto!

Esercizio 3

- ◆ Consideriamo un oggetto Subject che riceve valori dall'esterno e che in modo casuale cambia il suo stato interno accettando o meno il valore ricevuto.
- ◆ Due oggetti Watcher e Psicologist devono monitorare il cambiamento di valore di Subject.
- ◆ Utilizzare il pattern Observer sfruttando le capability di Java

suggerimento: usare le interfacce Observer ed Observable

Soluzione 3/1



Soluzione 3/2

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

public class Subject extends Observable {
    private int value = 0;
    public void receiveValue( int newNumber ) {
        if (Math.random() < .5) {
            System.out.println( "Subject : Ho cambiato il mio stato" );
            value = newNumber;
            this.setChanged();
        } else
            System.out.println( "Subject : Stato interno invariato" );
        this.notifyObservers();
    }

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

Soluzione 3/3

```
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: Lo stato del subject è: "
            + ((ObservedSubject) obs).returnValue() + ".");
        changes++;
    }

    public int observedChanges() {
        return changes;
    }
}
```

Soluzione 3/4

```
public class Esempio {  
    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 : Invio il numero " + i );  
            s.receiveValue( i );  
        }  
        System.out.println( "Il subject ha cambiato stato " );  
        o.observe( s );  
        ...  
    }  
}
```

Main : Invio il numero 1
Subject : Stato interno invariato
Main : Invio il numero 2
Subject : Ho cambiato il mio stato
Watcher: Lo stato del subject è: 2.
...
Il subject ha cambiato stato 4 volte.

Esercizio 4

- ◆ Realizzare un semplice Editor in grado di gestire elementi grafici costituiti dai seguenti elementi elementari:
 - frame, testi, immagini
- ◆ L'editor deve supportare due diversi algoritmi di formattazione
- ◆ L'editor deve supportare elementi grafici complessi come le barre di scorrimento o dei bordi grafici

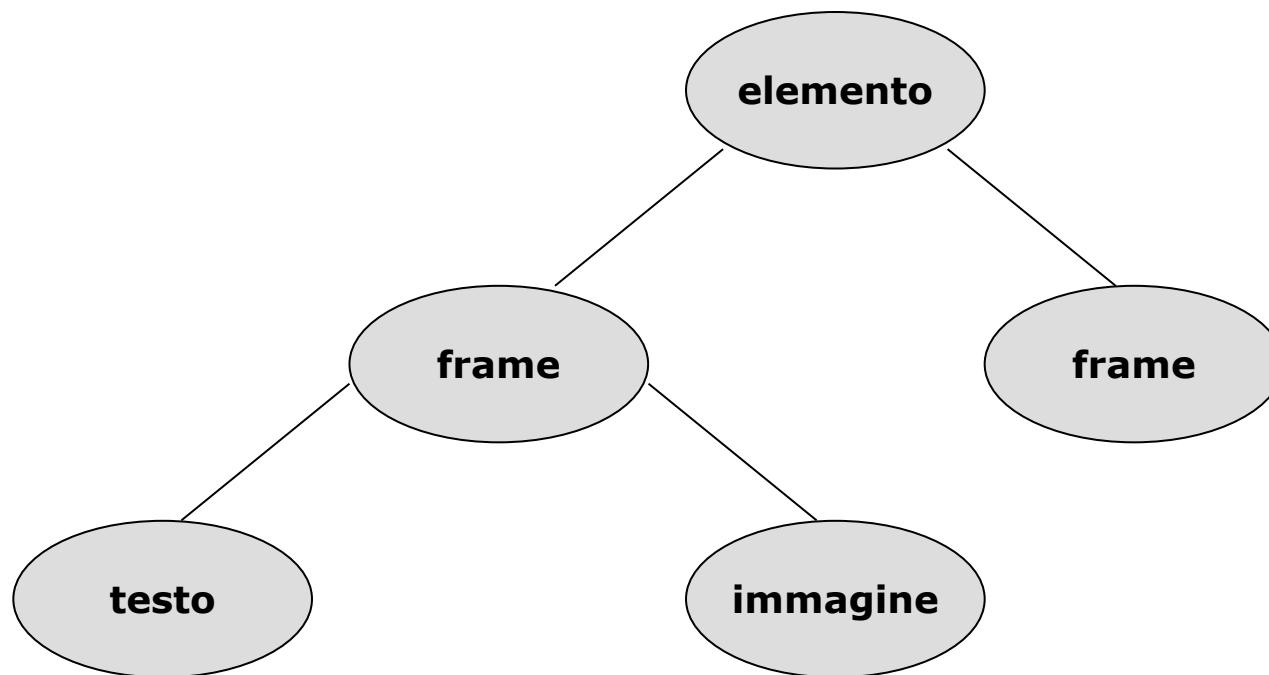
suggerimento: usare i pattern Composite, Strategy, Decorator....

Soluzione 4/1

- ◆ La struttura degli elementi tipografici può essere resa con un Composite Pattern
- ◆ La gestione degli algoritmi di formattazione può essere disaccoppiata ed estendibile utilizzando il pattern Strategy
- ◆ Il controllo delle funzionalità grafiche può essere ottenuto utilizzando il Decorator Pattern per gli elementi grafici

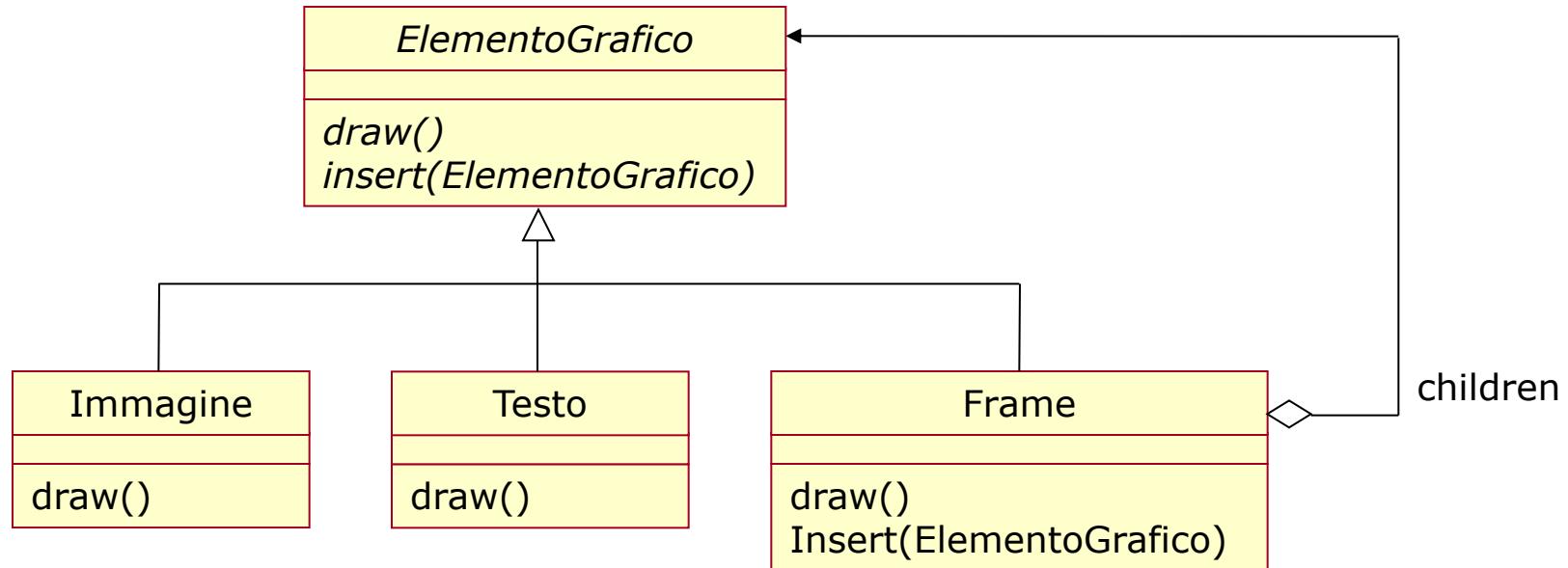
Soluzione 4/2

- ◆ Per semplicità consideriamo la rappresentazione di elementi semplici (frame, testo, immagine)



Soluzione 4/3

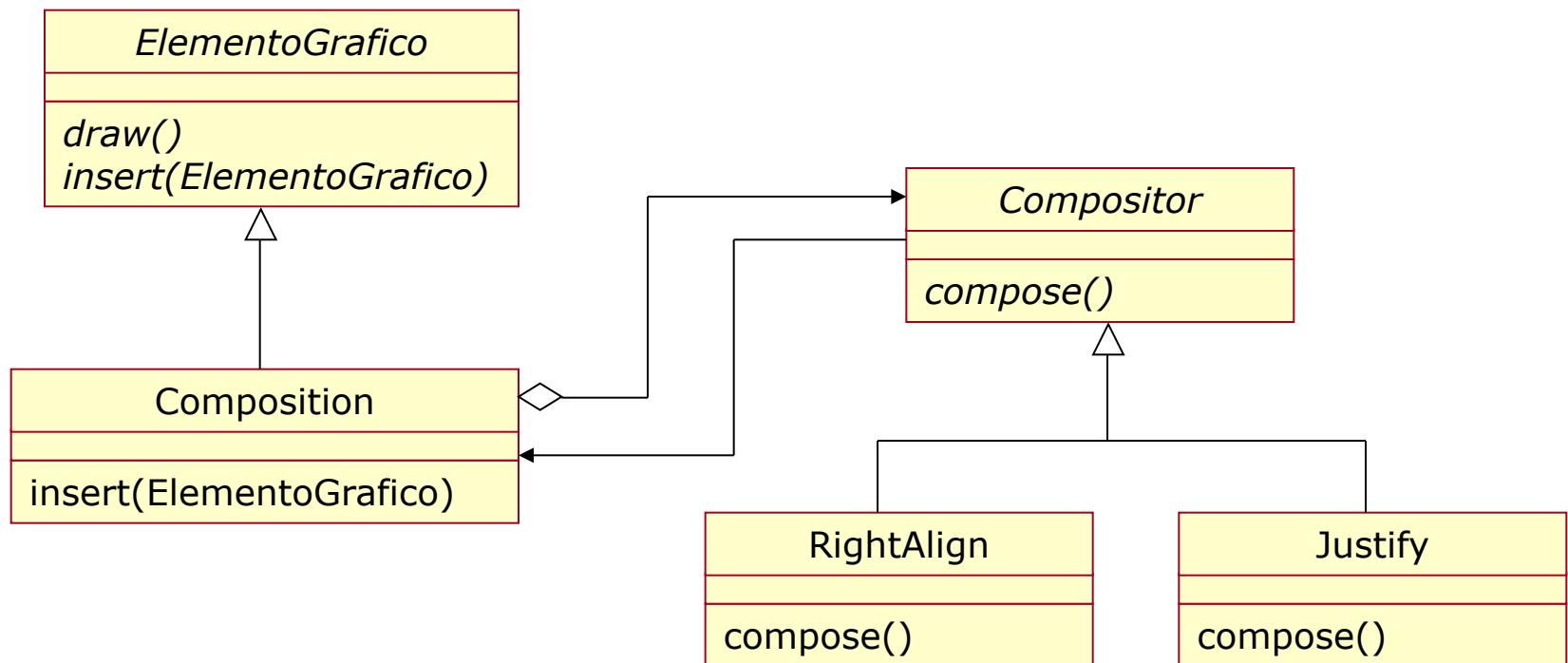
Rappresentazione degli elementi grafici: il pattern Composite



Nota: In Java le classi foglia (*Immagine* e *Testo*) devono necessariamente implementare il metodo *insert()* previsto dalla classe astratta. In questo caso una possibile soluzione è che queste generino una eccezione apposita

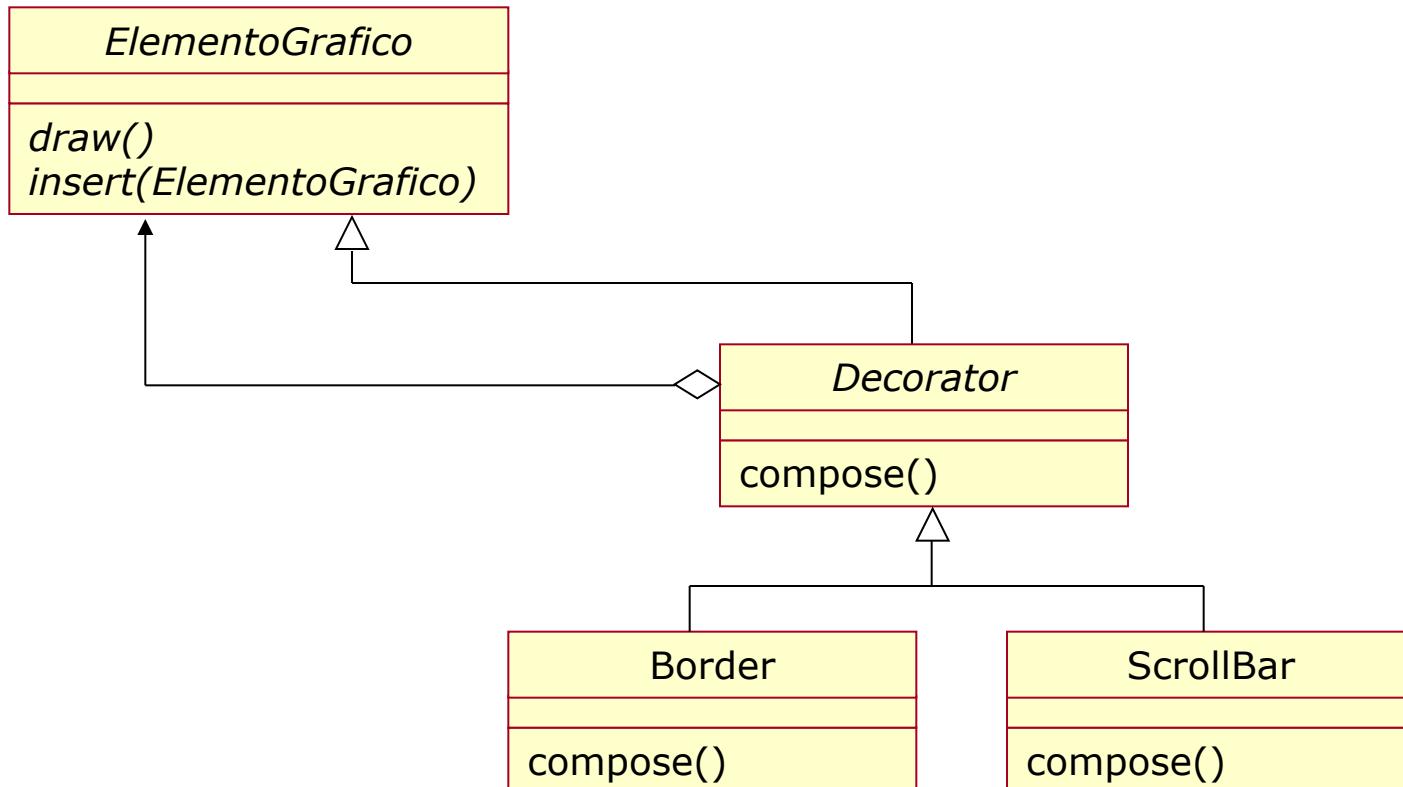
Soluzione 4/4

Rappresentazione degli algoritmi di formattazione:
il pattern Strategy



Soluzione 4/5

Rappresentazione delle proprietà grafiche: il pattern Decorator



Soluzione 4/6

Mettendo tutto insieme...

