



Design Patterns

Introduzione

- ♦ La progettazione di software orientato ad oggetti è una attività complessa
- ♦ Una delle maggiori difficoltà che il progettista deve affrontare è l'individuazione di un insieme di oggetti:
 - correttamente individuato
 - il più possibile riusabile
 - definendo al meglio le relazioni tra classi e le gerarchie di ereditarietà

Una lezione da imparare

"Ogni pattern describe un problema che si ripete più e più volte nel nostro ambiente, describe poi il nucleo della soluzione del problema, in modo tale che si possa riusare la soluzione un milione di volte, senza mai applicarla alla stessa maniera."

Christopher Alexander

- Architetto -

Design Pattern:

Alcune Caratteristiche

- ◆ Rappresentano soluzioni a problematiche ricorrenti che si incontrano durante le varie fasi di sviluppo del software.
- ◆ Organizzano l'esperienza di OOD favorendo il riuso.
- ◆ Evitano al progettista di 'reinventare la ruota' ogni volta
- ◆ Permettono di imparare dal lavoro degli altri (evitando errori comuni...)

Desing Pattern:

Alcune Caratteristiche (2)

- ◆ Permettono di definire un linguaggio comune che semplifica la comunicazione tra gli addetti ai lavori.
- ◆ Indirizzano verso la scrittura di codice che usa strutture valide.
- ◆ Portano di norma ad una buona progettazione: semplificano la manutenzione (adattiva, perfettiva, preventiva e correttiva).
- ◆ Non risolvono tutti i problemi!

Riferimenti Bibliografici

- ♦ E. Gamma, R. Helm, R. Johnson, J. Vlissides (GoF, Gang of Four), *Design Patterns – Elementi per il riuso di software a oggetti*, Addison Wesley
- ♦ Craig Larman *Applicare UML e i Pattern*, Prentice Hall

Classificazione

- ♦ Un primo criterio riguarda lo scopo (*purpose*).
 - **Creazionali**: i pattern di questo tipo sono relativi alle operazioni di creazione di oggetti.
 - **Strutturali**: sono utilizzati per definire la struttura del sistema in termini della composizione di classi ed oggetti. Si basano sui concetti OO di ereditarietà e polimorfismo.
 - **Comportamentali**: permettono di modellare il comportamento del sistema definendo le responsabilità delle sue componenti e definendo le modalità di interazione.

Classificazione (2)

- ♦ Un secondo criterio riguarda il raggio di azione (*scope*):
 - **Classi**: pattern che definiscono le relazioni fra classi e sottoclassi. Le relazioni sono basate prevalentemente sui concetti di ereditarietà e sono quindi **statiche** (definite a tempo di compilazione).
 - **Oggetti**: pattern che definiscono relazioni tra oggetti, che possono cambiare durante l'esecuzione e sono quindi più **dinamiche**.

Catalogo dei Pattern (GoF)

- | | |
|--------------|-----------------------------|
| 1. Adapter | 13. Memento |
| 2. Façade | 14. Interpreter |
| 3. Composite | 15. Iterator |
| 4. Decorator | 16. Visitor |
| 5. Bridge | 17. Mediator |
| 6. Singleton | 18. Template Method |
| 7. Proxy | 19. Chain of Responsibility |
| 8. Flyweight | 20. Builder |
| 9. Strategy | 21. Prototype |
| 10. State | 22. Factory Method |
| 11. Command | 23. Abstract Factory |
| 12. Observer | |

Classificazione completa

		Scopo		
		Creazionale	Strutturale	Comportamentale
Raggio d'azione	Classi	Factory Method	Adapter (class)	Interpreter Template Method
	Oggetti	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Iterator Mediator Memento Observer State Strategy Visitor

Descrizione dei Design Pattern

- ♦ **Nome e Classificazione:** il nome illustra **l'essenza di un pattern** definendo un vocabolario condiviso tra i progettisti; la classificazione lo identifica in termini di scopo e raggio di azione.
- ♦ **Motivazione:** scenario che descrive in modo astratto **il problema al quale applicare il pattern**; può includere la lista eventuali pre-condizioni necessarie a garantirne l'applicabilità.
- ♦ **Applicabilità:** Descrive le situazioni in cui il pattern può essere applicato.
- ♦ **Struttura:** descrive graficamente la **configurazione** di elementi che risolvono il problema (relazioni, responsabilità, collaborazioni).



E' uno schema di soluzione, non una soluzione per un progetto specifico

Descrizione dei Design Pattern(2)

- ♦ **Partecipanti:** **classi ed oggetti** che fanno parte del pattern con le relative responsabilità.
- ♦ **Conseguenze:** **risultati** che si ottengono applicando il pattern.
- ♦ **Implementazioni:** tecniche e suggerimenti utili **all'implementazione** del pattern.
- ♦ **Codice di esempio:** frammenti di **codice** che illustrano come implementare in un certo linguaggio di programmazione (es. java o c++) il pattern.
- ♦ **Usi conosciuti:** esempi di applicazione in **sistemi reali**
- ♦ **Pattern correlati:** altri pattern correlati

Un concetto utile: i Framework

- ♦ Un Framework non è una semplice libreria.
- ♦ Un Framework rappresenta il *design* riusabile di un sistema (o di una sua parte), definito da un insieme di classi astratte.
- ♦ Un framework è lo scheletro di un'applicazione che viene personalizzato (*customized*) da uno sviluppatore
- ♦ Il programmatore di applicazioni implementa le interfacce e classi astratte ed ottiene automaticamente la gestione delle funzionalità richieste.

I Framework (2)

- ♦ I Framework permettono quindi di definire lo scopo e la struttura statica di un sistema.
- ♦ Sono un buon esempio di progettazione orientata agli oggetti.
- ♦ Permettono di raggiungere due obiettivi:
 - Il riuso del *design*
 - Il riuso del codice

I Framework (3)

- ♦ **Classe Astratta:** una classe astratta è una classe che possiede almeno un metodo non implementato, definito "astratto".
- ♦ Una classe astratta è quindi un template per le sottoclassi da cui deriveranno le specifiche applicazioni.
- ♦ Un framework è rappresentato da un'insieme di classi astratte e dalle loro interrelazioni.
- ♦ I Pattern sono spesso mattoni per la costruzione di framework.

Abstract Factory

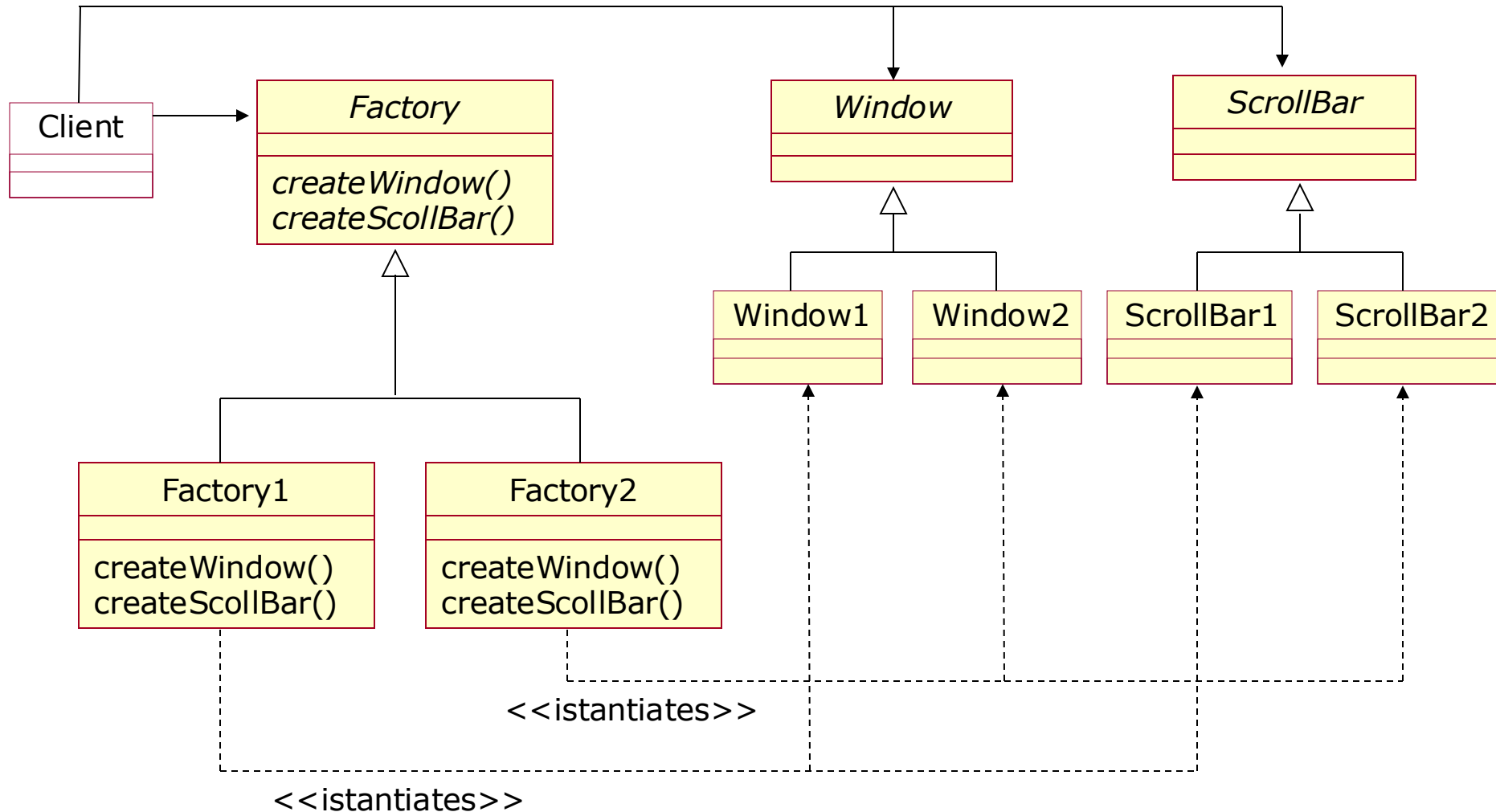
- ♦ **Scopo:** fornire una interfaccia per la **creazione di famiglie di oggetti tra loro correlati**
- ♦ **Motivazione:** realizzazione di uno strumento per lo sviluppo di *user interface* (UI) in grado di supportare diversi tipi di *look & feel*. Per garantire la portabilità di una applicazione tra look & feel diversi, gli oggetti non devono essere cablati nel codice.

Classificazione: creazionale basato su oggetti

Abstract Factory: esempio

- ◆ Consideriamo uno strumento per realizzare UI con due soli elementi, in grado di supportare due look & feel diversi
 - Look&Feel1 -> Window1 e ScrollBar1
 - Look&Feel2 -> Window2 e ScrollBar2
- ◆ L'applicazione client deve realizzare una UI che rispetti le relazioni tra gli oggetti e che sia portabile da un L&F ad un altro:
 - Il client non dovrebbe istanziare direttamente gli oggetti
 - Bisogna evitare che il client accoppi (sbagliando) **Window1** e **ScrollBar2**
- ◆ Usiamo una Abstract Factory

Abstract Factory: esempio (2)



Abstract Factory: esempio (3)

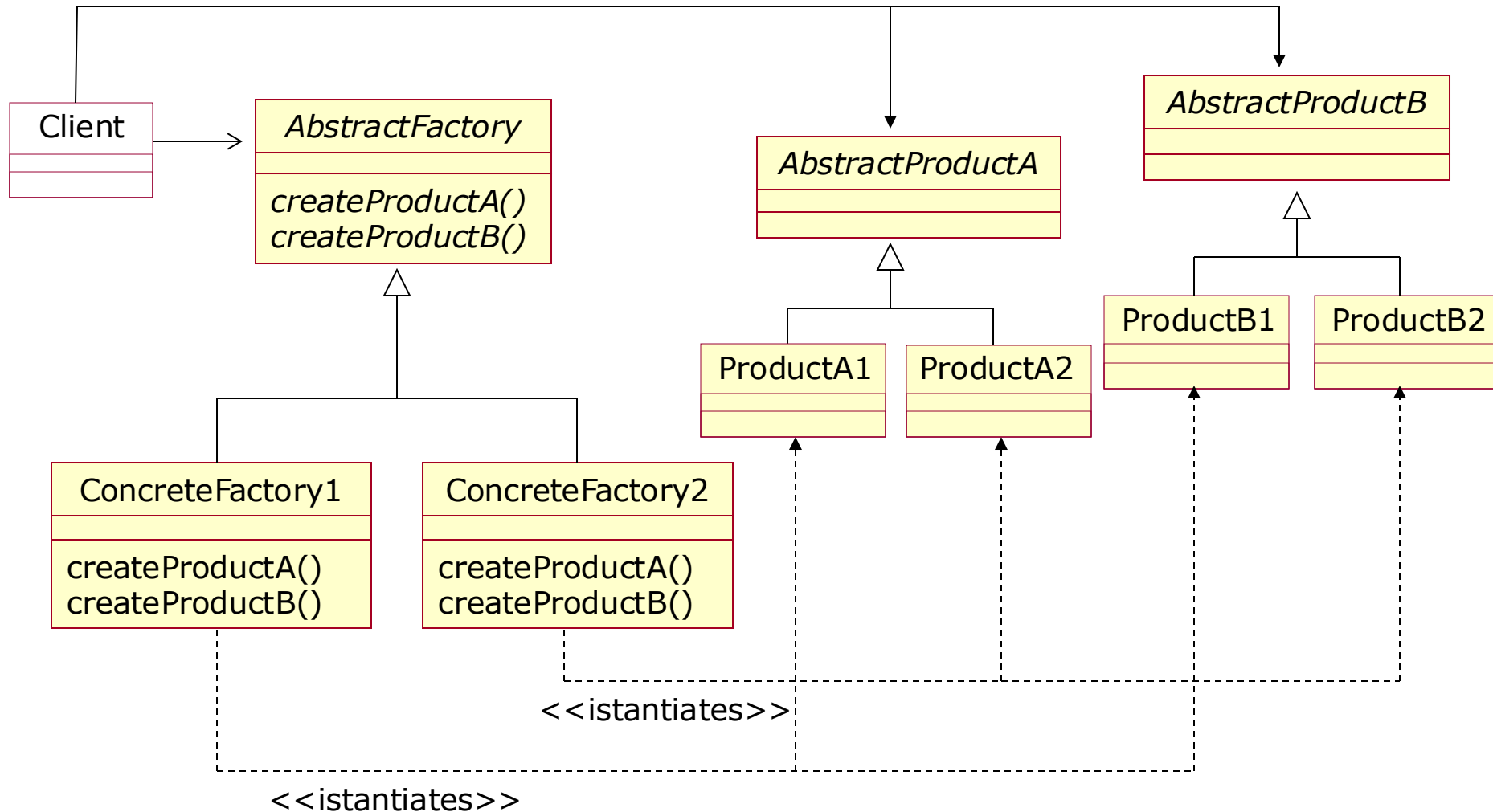
- ♦ Senza utilizzare l'Abstract Factory l'applicazione client deve esplicitamente istanziare gli oggetti.
- ♦ Il rispetto delle relazioni è cablato nel codice e deve essere noto al client.

```
Window w = new Window1();  
...  
ScrollBar s = new ScrollBar1();
```

- ♦ Con l'Abstract Factory la responsabilità è demandata alla Factory.

```
Factory f = new Factory1();  
Window w = f.createWindow();  
...  
ScrollBar s = f.createScrollBar();
```

Abstract Factory: struttura



Altre caratteristiche

♦ Applicabilità

- A sistema che deve essere **indipendente** dalle modalità di creazione dei prodotti con cui opera
- A sistema che deve poter essere configurato per usare **famiglie di prodotti diverse**
- Il client **non deve essere legato** ad una specifica famiglia

♦ Partecipanti

- AbstractFactory e ConcreteFactory
- AbstractProduct e ConcreteProduct
- Applicazione Client

Altre caratteristiche (2)

♦ **Conseguenze**

- Le classi concrete sono isolate e sotto controllo
- La famiglia di prodotti può essere cambiata rapidamente perché la factory completa compare in un unico punto del codice
- Aggiungere nuove famiglie di prodotti richiede ricompilazione perché l'insieme di prodotti gestiti è legato all'interfaccia della factory

Factory Method

- ♦ **Scopo:** Definire una interfaccia per la creazione di un oggetto, che consenta di decidere a tempo di esecuzione quale specifico oggetto istanziare.
- ♦ **Motivazione:** E' un pattern ampiamente usato nei framework, dove le classi astratte definiscono le relazioni tra gli elementi del dominio, e sono responsabili per la creazione degli oggetti concreti.

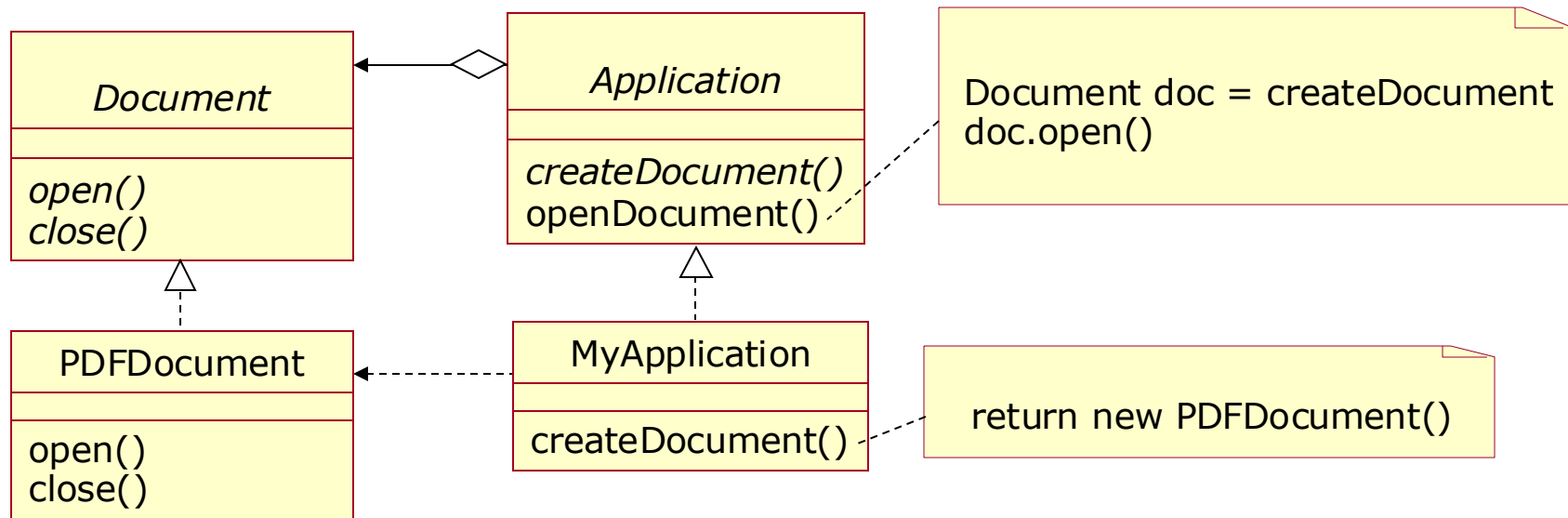
Classificazione: creazionale basato su classi

Factory Method: esempio

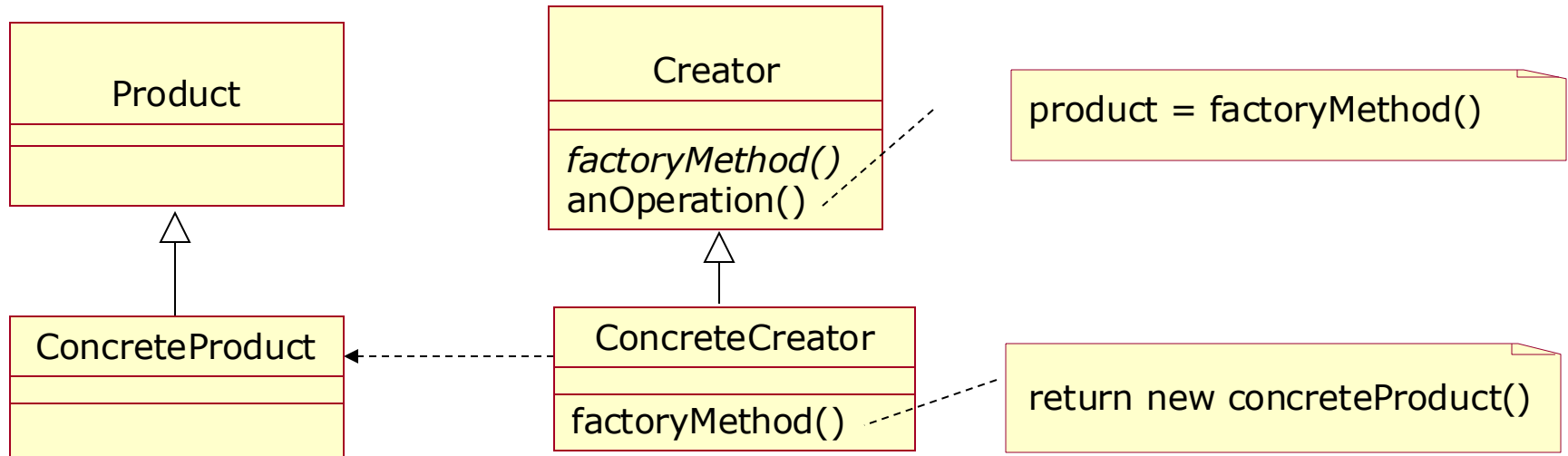
- ◆ Consideriamo un framework di gestione di documenti di tipo diverso.
- ◆ Le due astrazioni chiave sono Application e Document.
- ◆ Gli utilizzatori devono definire delle sotto classi per ottenere delle implementazioni adatte all'applicazione specifica.
- ◆ Application contiene la logica per sapere **quando** un nuovo documento sarà creato, ma non per sapere **quale** tipo di documento creare.

Factory Method: esempio (2)

- ♦ Il pattern Factory incapsula la conoscenza della specifica classe da creare al di fuori del framework



Factory Method: struttura



Altre caratteristiche

♦ Applicabilità

- Una classe non è in grado di sapere in anticipo le classi di oggetti che deve creare.
- Una classe vuole che le sue sottoclassi scelgano gli oggetti da creare.
- Le classi delegano la responsabilità di creazione.

♦ Partecipanti

- Product e ConcreteProduct
- Creator e ConcreteCreator

♦ Conseguenze

- Elimina la necessità di riferirsi a classi dipendenti dall'applicazione all'interno del codice.

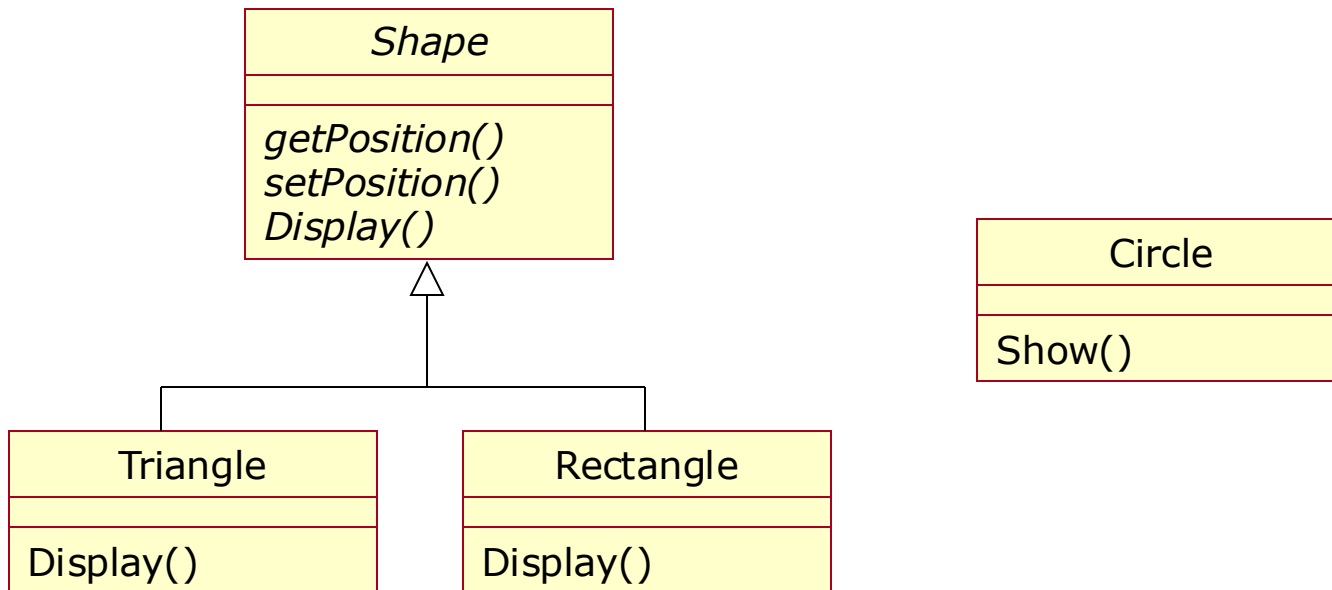
Adapter

- ♦ **Scopo:** Convertire l'interfaccia di una classe esistente **incompatibile** con un client, in una **compatibile**.
- ♦ **Motivazione:** Consideriamo un editor che consente di disegnare e comporre elementi grafici. L'astrazione chiave è un singolo oggetto grafico. Supponiamo di voler integrare un nuovo componente, ma che questo non abbia una interfaccia compatibile con l'editor.

Classificazione: strutturale basato su classi/oggetti

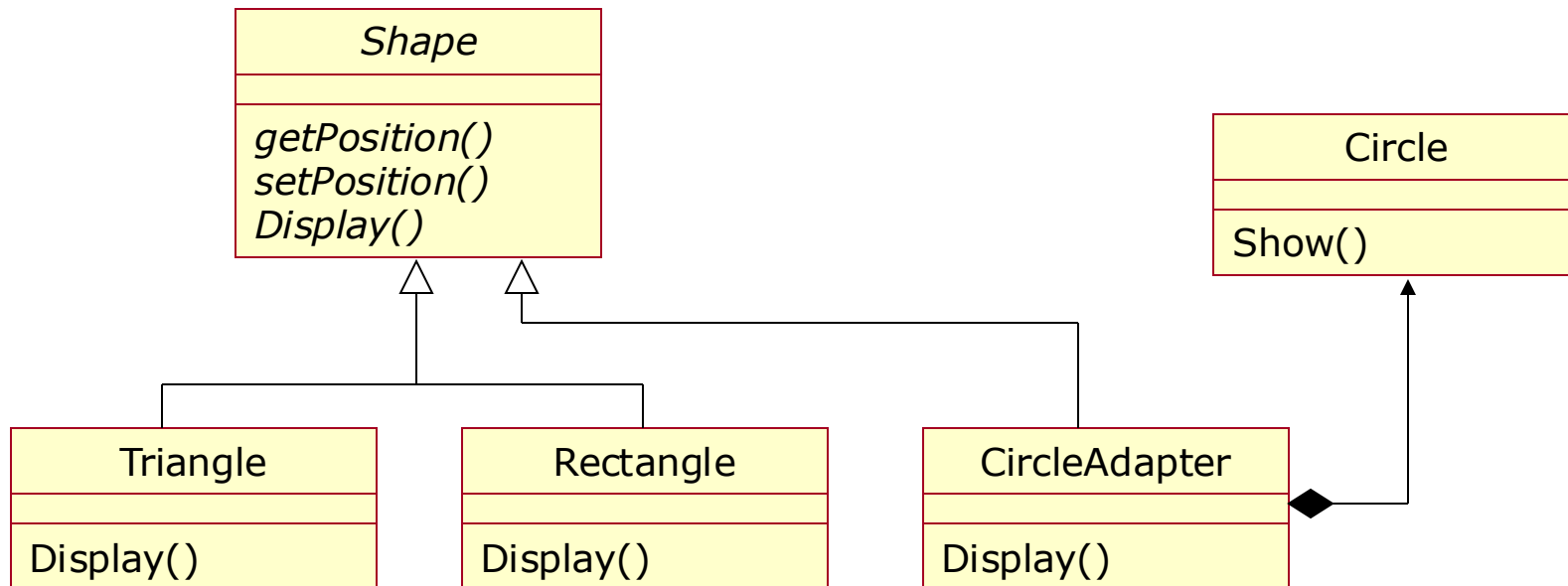
Adapter: esempio

- ◆ Supponiamo di voler integrare il componente Circle nell'editor che già supporta le forme Triangle e Rectangle.



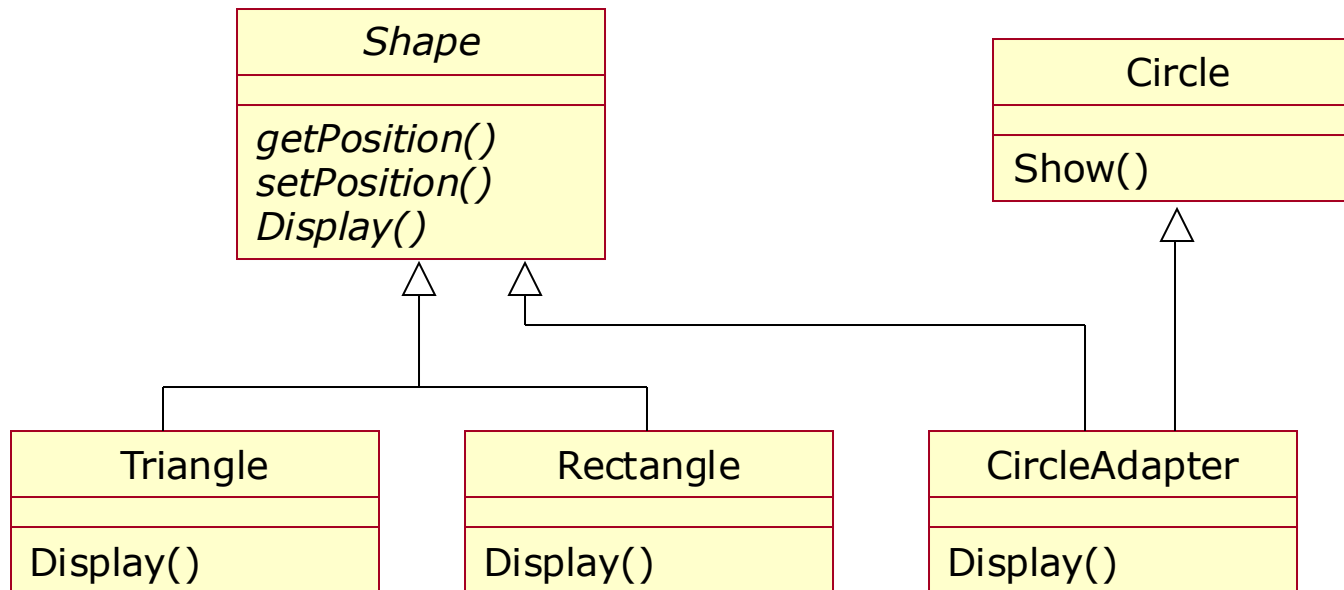
Adapter: esempio (2)

♦ Soluzione 1: Object Adapter

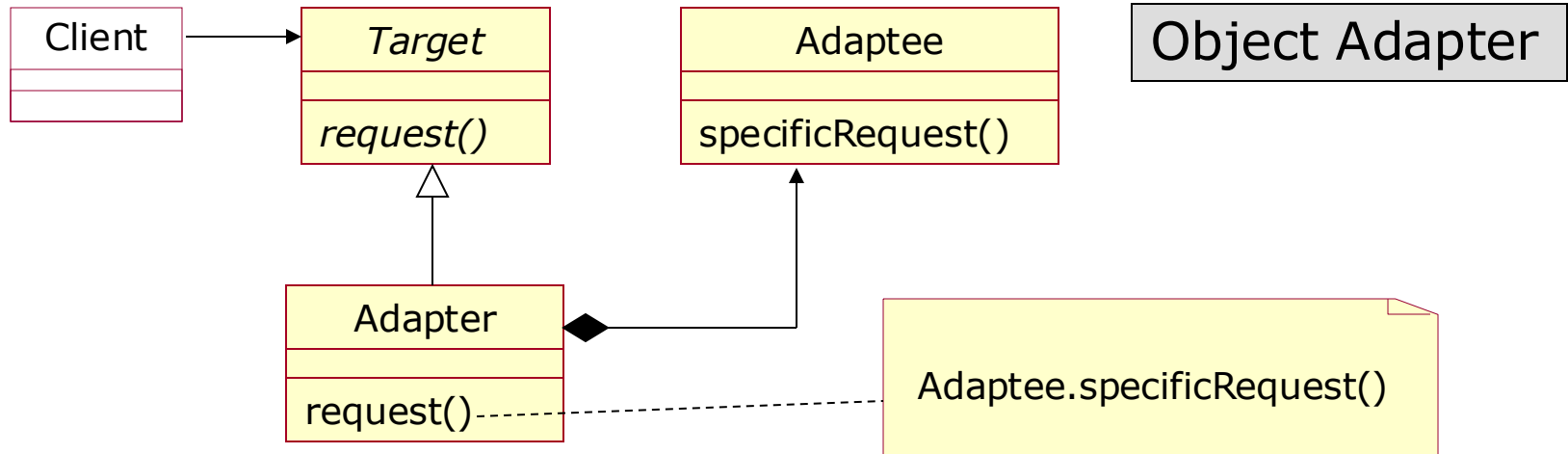
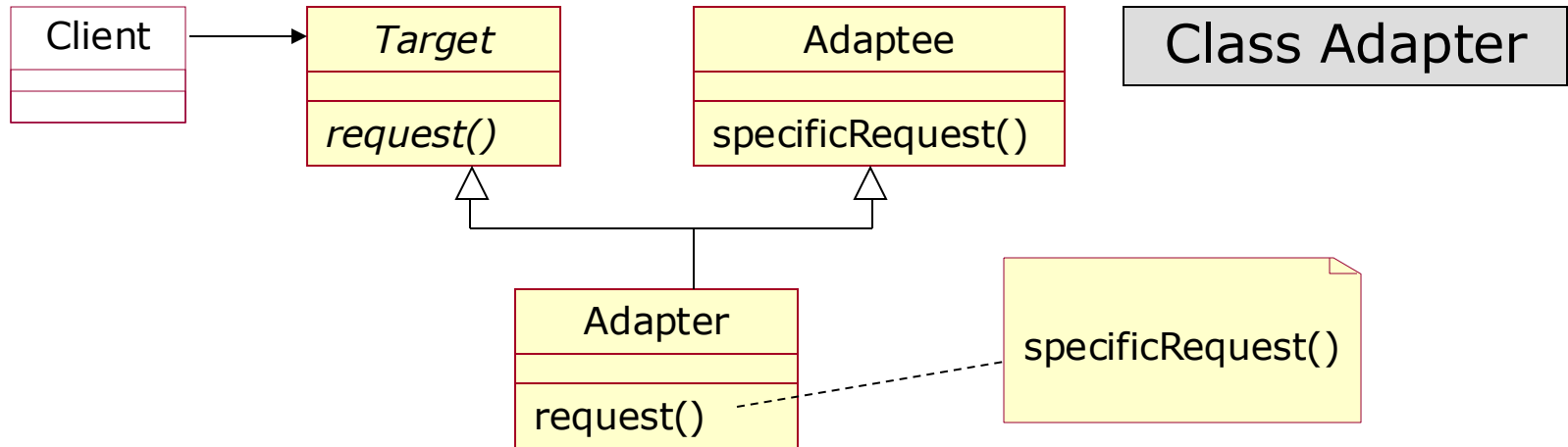


Adapter: esempio (3)

♦ Soluzione 1: Class Adapter



Adapter: struttura



Altre caratteristiche

♦ **Applicabilità**

- Si usa quando si vuole riusare una classe esistente, ma con interfaccia incompatibile con quella desiderata.

♦ **Partecipanti**

- Client
- Target
- Adapter ed Adaptee

♦ **Conseguenze:** E' necessario prendere in considerazione l'effort necessario all'adattamento

Composite

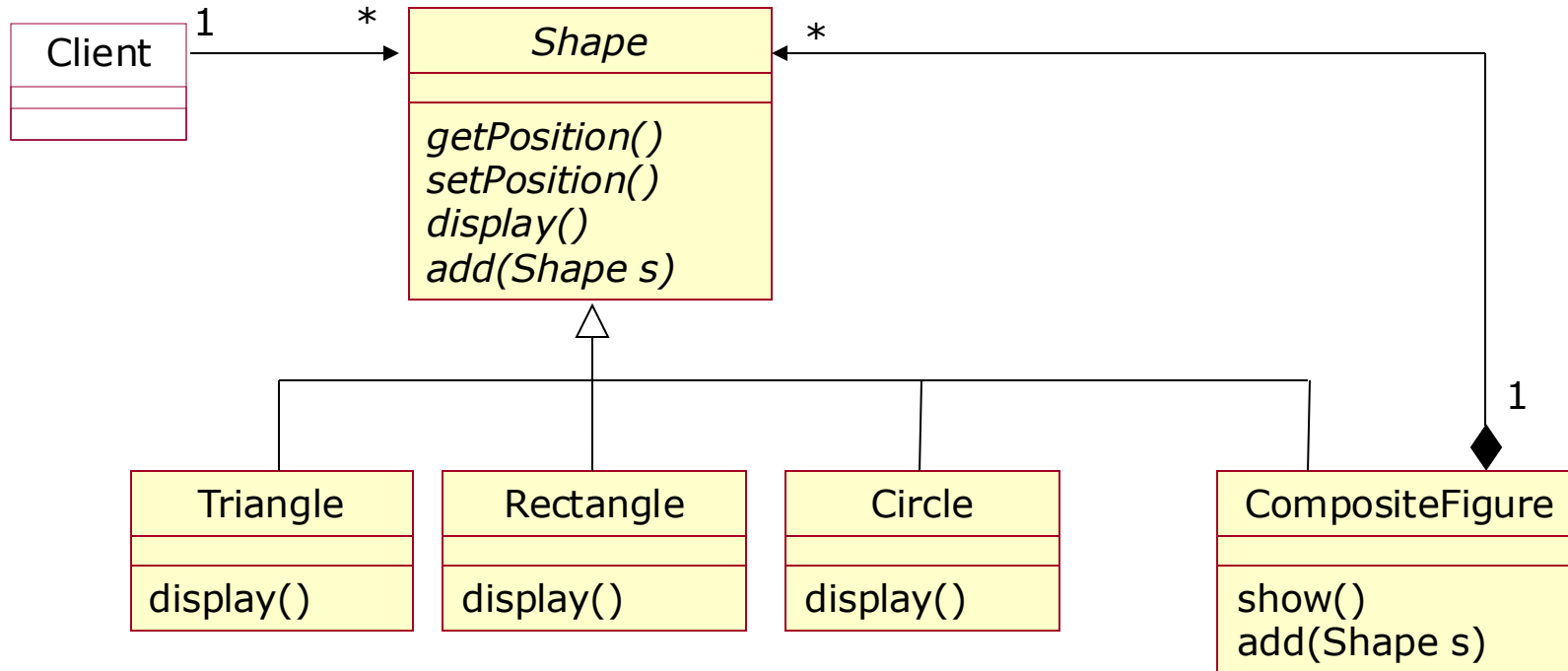
- ♦ **Scopo:** Comporre oggetti in strutture che consentano di trattare i singoli elementi e la composizione in modo uniforme.
- ♦ **Motivazione:** Le applicazioni grafiche consentono di trattare in modo uniforme sia le forme geometriche di base (linee, cerchi,...) sia gli oggetti complessi che si creano a partire da questi elementi semplici. Molti editor grafici ad esempio hanno la funzione *raggruppa*

Classificazione: strutturale basato su oggetti

Composite: esempio

- ◆ Consideriamo un applicazione grafica in grado di gestire gli oggetti elementari Triangle, Rectangle, e Circle.
- ◆ Un ulteriore requisito è che l'applicazione deve poter permettere il raggruppamento dinamico di oggetti elementari in oggetti compositi.
- ◆ Gli oggetti elementari e quelli compositi devono essere trattati in modo uniforme

Composite: esempio (2)

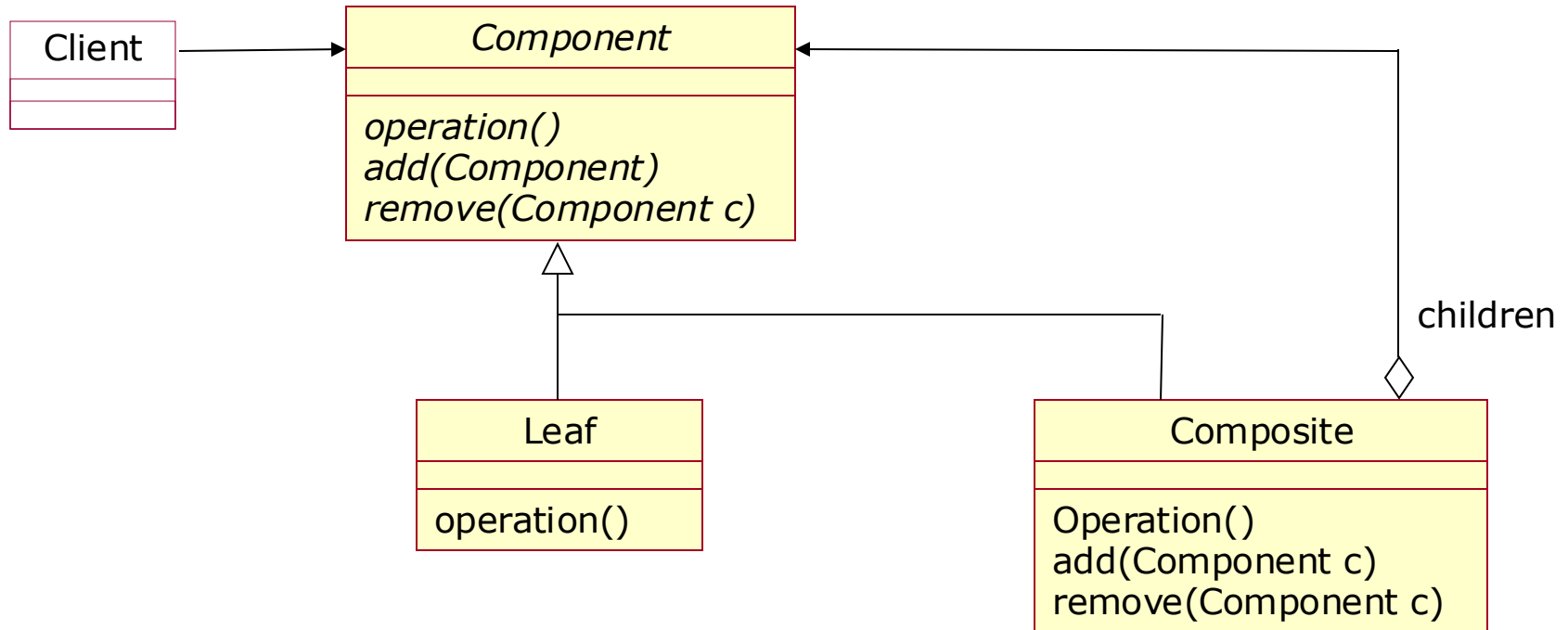


- La sola relazione di composizione non soddisfa tutti i requisiti: CompositeFigure è una collezione di oggetti elementari ma **non** è essa stessa una figura geometrica (Shape)



La relazione di ereditarietà permette di considerare CompositeFigure una figura geometrica

Composite: struttura



Altre Caratteristiche

♦ **Applicabilità**

- Si usa quando si vogliono rappresentare gerarchie di oggetti in modo che oggetti semplici e oggetti composti siano trattati in modo uniforme.

♦ **Partecipanti**

- Component e Composite
- Leaf
- Client

Altre Caratteristiche (2)

♦ **Conseguenze**

- I client sono semplificati perché gli oggetti semplici e quelli composti sono trattati allo stesso modo.
- L'aggiunta di nuovi oggetti Leaf o Composite è semplice, e questi potranno sfruttare il codice dell'applicazione Client già esistente.
- Può rendere il sistema troppo generico. Non è possibile fare in modo che un oggetto composto contenga solo un certo tipo di oggetti.

Decorator

- ♦ **Scopo:** Aggiungere dinamicamente funzionalità (responsabilità) ad un oggetto.

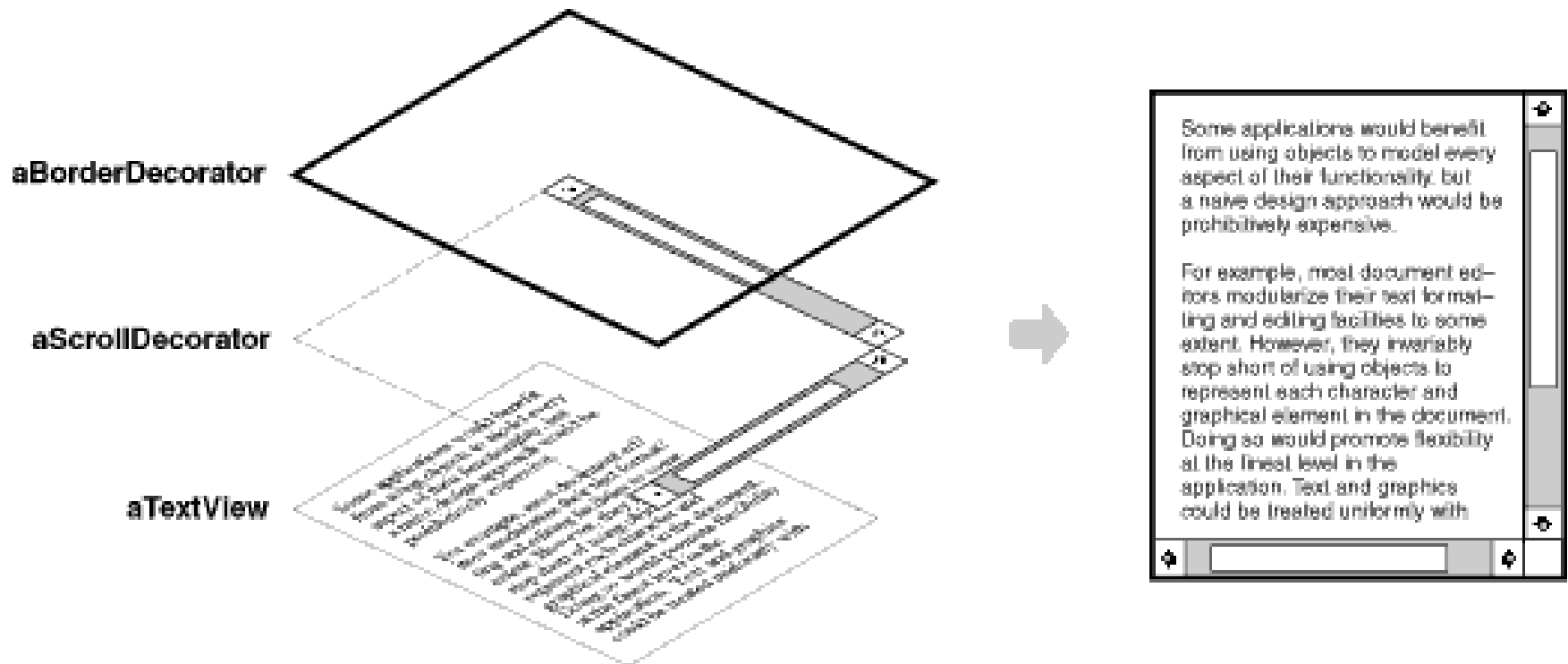


Il subclassing è una alternativa statica e il cui scope è a livello di classe e non di singolo oggetto.

- ♦ **Motivazione:** Uno scenario classico di applicabilità per questo pattern è la realizzazione di interfacce utente. Responsabilità quali il **testo scorrevole** o un particolare **bordo** devono poter essere aggiunti a livello di singolo oggetto.

Classificazione: strutturale basato su oggetti

Decorator: esempio

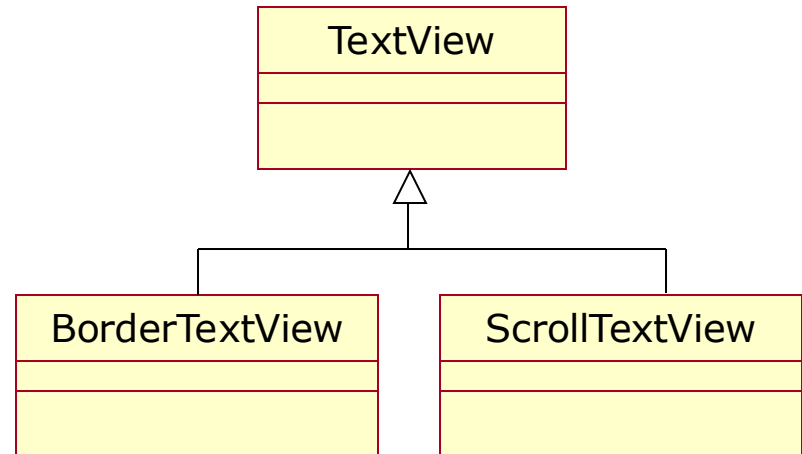


Come combinare i tre oggetti `aBorderDecorator`, `aScrollDecorator` e `aTextView` per raggiungere l'obiettivo della dinamicità?

Decorator: esempio (2)

Il primo approccio che scartiamo è quello del subclassing.

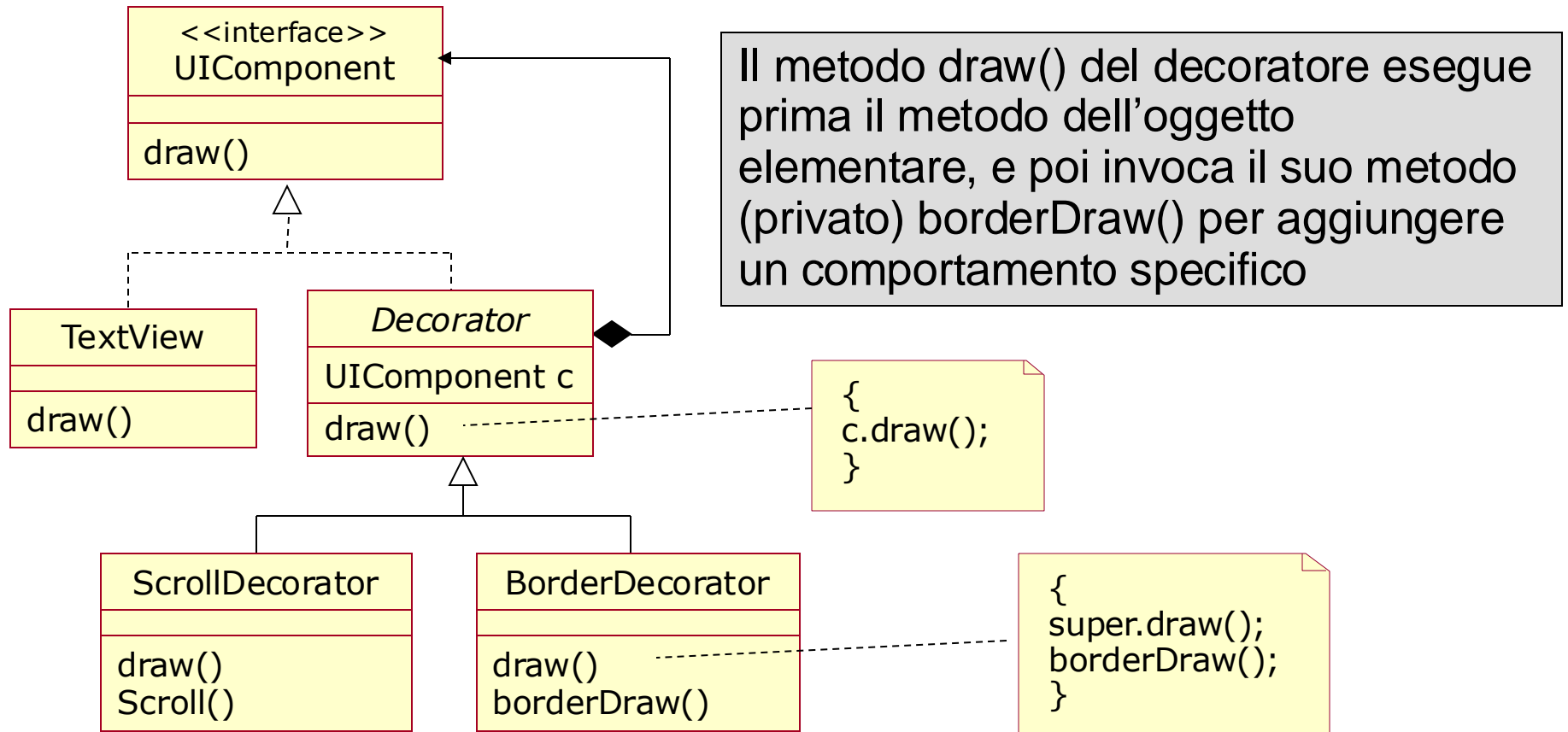
- ♦ E' un approccio statico. Una volta creato `BorderTextView` non posso più cambiarlo.
- ♦ Devo creare una sottoclasse per ogni esigenza
- ♦ E per creare una finestra che abbia sia il bordo, sia il testo scorrevole?
 - `BorderAndScrollTextView`?
 - `ScrollAndBorderTextView`?



Decorator: esempio (3)

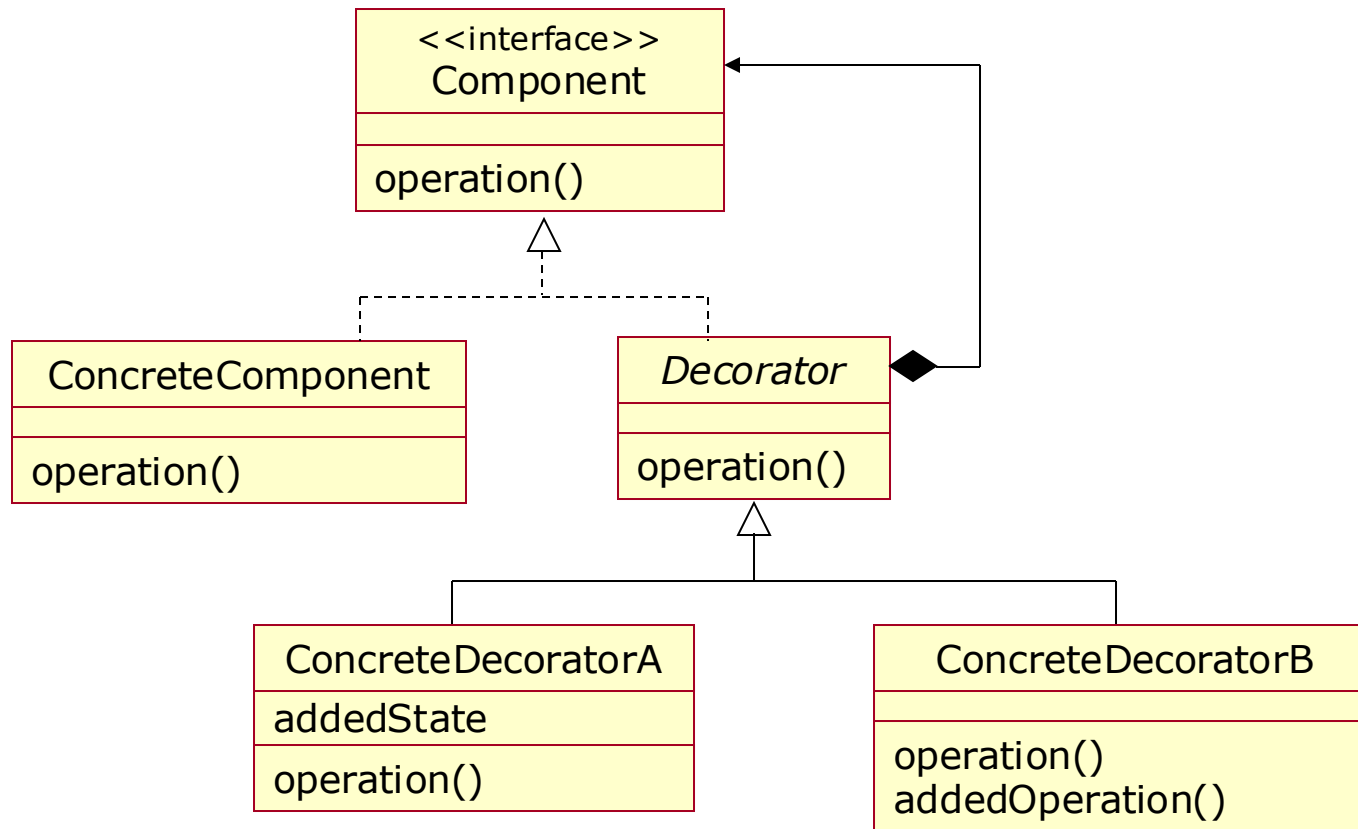
- ♦ Un approccio più flessibile è quello di racchiudere un oggetto elementare in un altro, che aggiunge una responsabilità particolare.
- ♦ L'oggetto contenitore è chiamato **Decorator**.
- ♦ Il Decorator ha una interfaccia conforme all'oggetto da decorare.
- ♦ Il Decorator trasferisce le richieste all'oggetto decorato ma può svolgere **funzioni aggiuntive** (ad esempio aggiungere un bordo) prima o dopo il trasferimento della richiesta.

Decorator: esempio (4)



Area di testo con bordo e barre di scorrimento:
`t = new BorderDecorator(new ScrollDecorator(new TextView()));`

Decorator: struttura



Altre caratteristiche

♦ **Applicabilità**

- Si applica quando è necessario aggiungere responsabilità agli oggetti in modo trasparente e dinamico.
- Si applica quando il subclassing non è adatto.

♦ **Partecipanti**

- Component e ConcreteComponent
- Decorator e ConcreteDecorator(s)

♦ **Conseguenze:**

- Maggiore flessibilità rispetto all'approccio statico
- Evita di definire strutture gerarchiche complesse

Decorator: alcune note

- ♦ In Java il Decorator è particolarmente usato nella definizione degli Stream di I/O:

```
BufferedInputStream bin = new BufferedInputStream( new FileInputStream(  
"test.dat"));
```

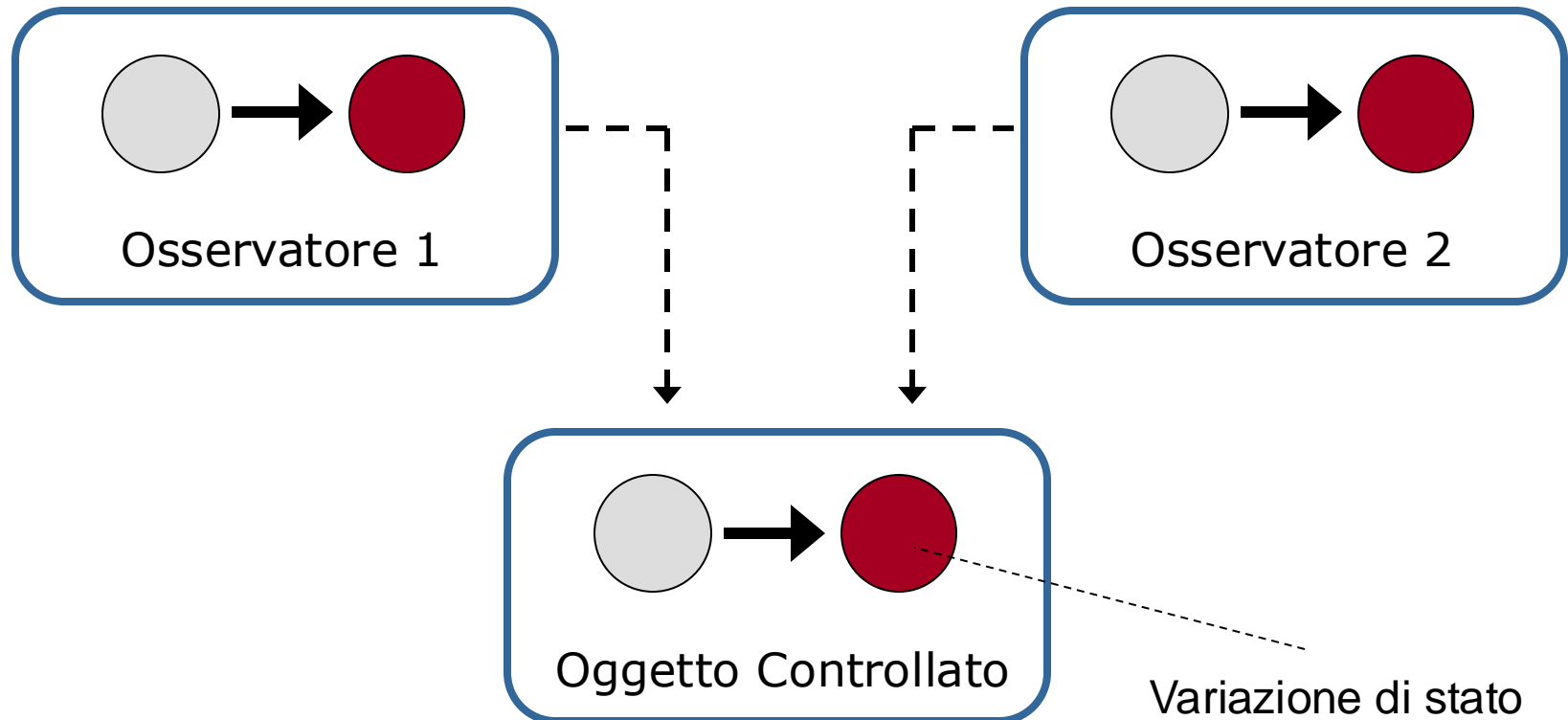
- ♦ E' simile al pattern **Composite**, ma la finalità è diversa. Decorator serve ad aggiungere responsabilità in modo dinamico.
- ♦ E' simile al pattern **Adapter**, ma quest'ultimo si limita ad un adattamento (limitato) di una interfaccia.

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

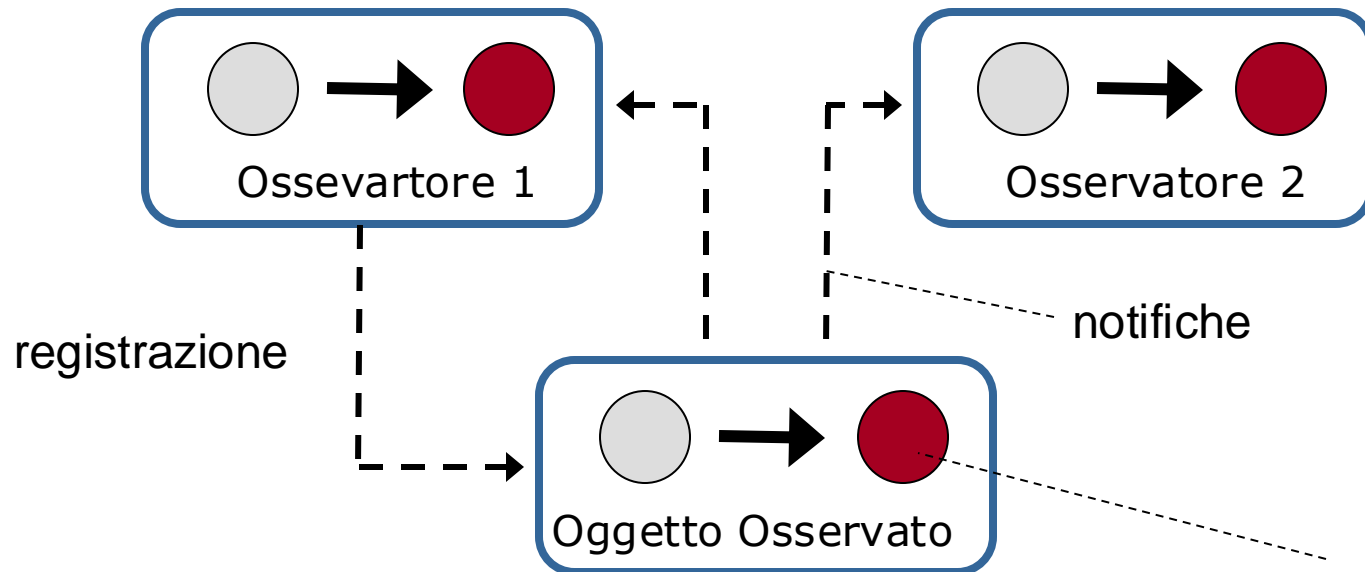
- ♦ Una prima soluzione potrebbe essere quella di utilizzare, nell'oggetto osservato, **attributi pubblici** oppure **metodi pubblici** che leggono il valore di un attributo protetto
- ♦ Non è una buona soluzione:
 - Non è scalabile
 - se aumentano troppo gli osservatori l'oggetto osservato è sovraccaricato 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

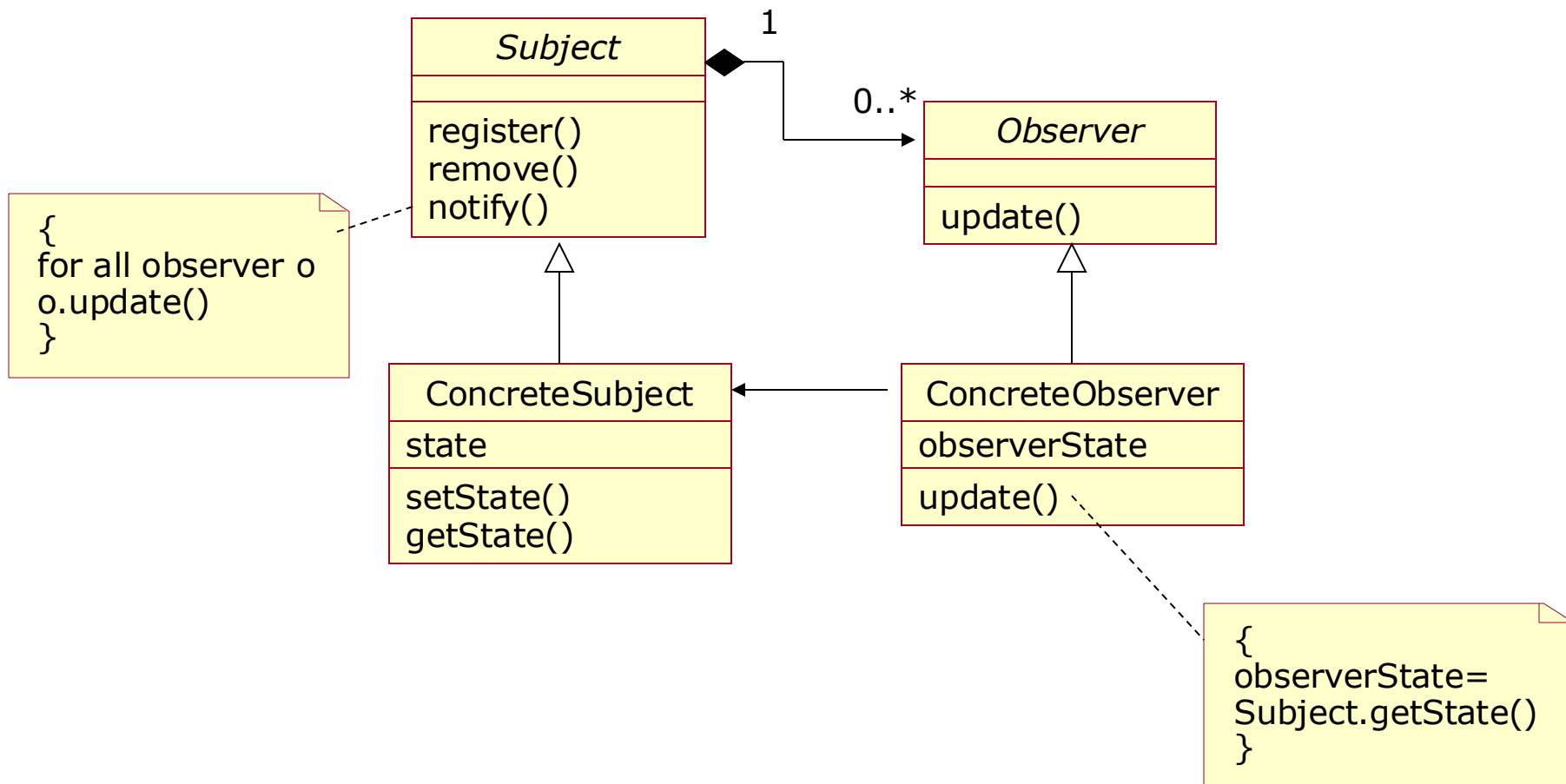
Observer: l'approccio corretto (2)

- ◆ Gli osservatori possono essere aggiunti a runtime

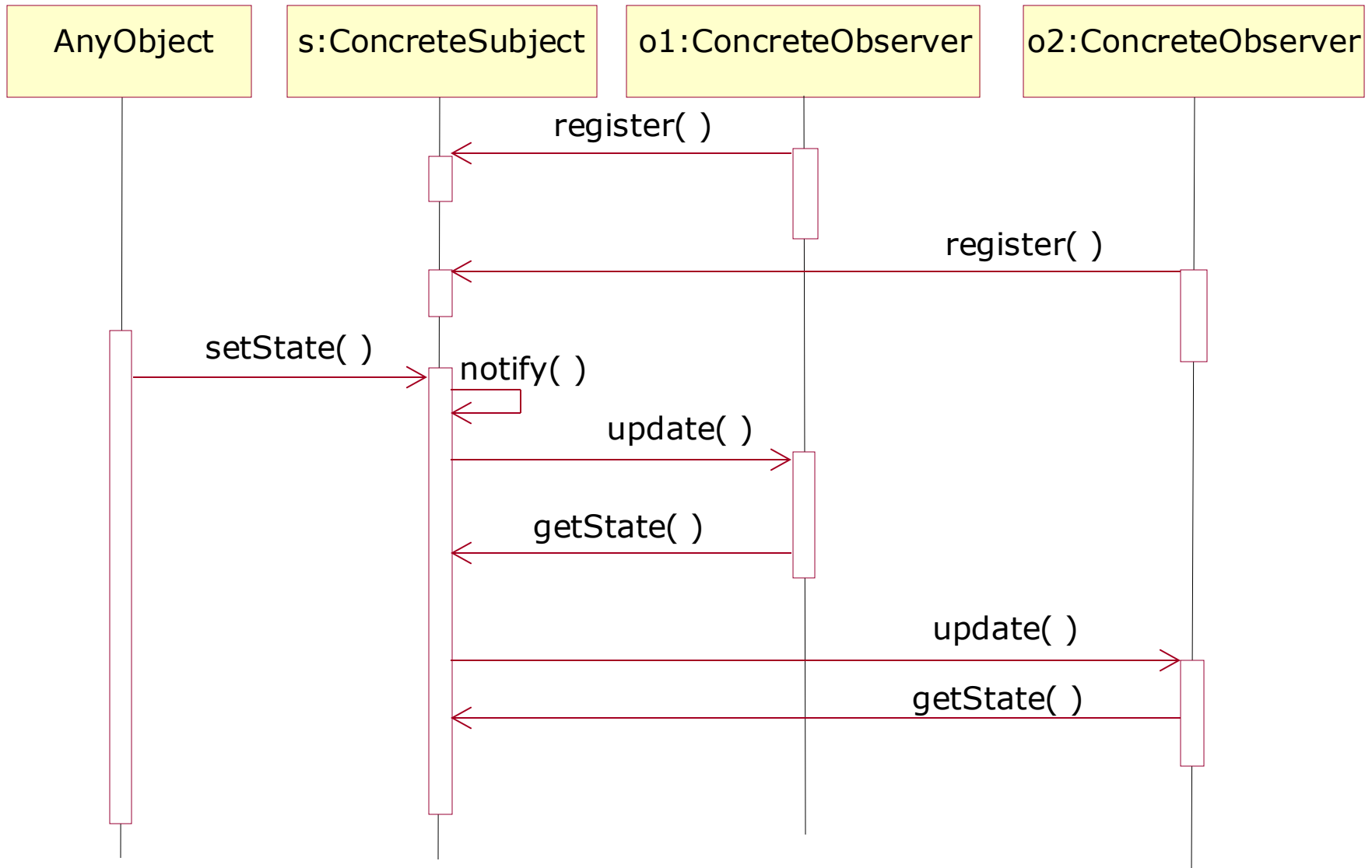


Variazione di stato

Observer: struttura



Observer: sequence diagram



Altre caratteristiche

♦ **Applicabilità**

- Si applica quando una azione può essere scomposta in due ambiti, ciascuno dei quali incapsulato in oggetti separati per mantenere basso il livello di coupling.
- Gestire le modifiche di oggetti conseguenti 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

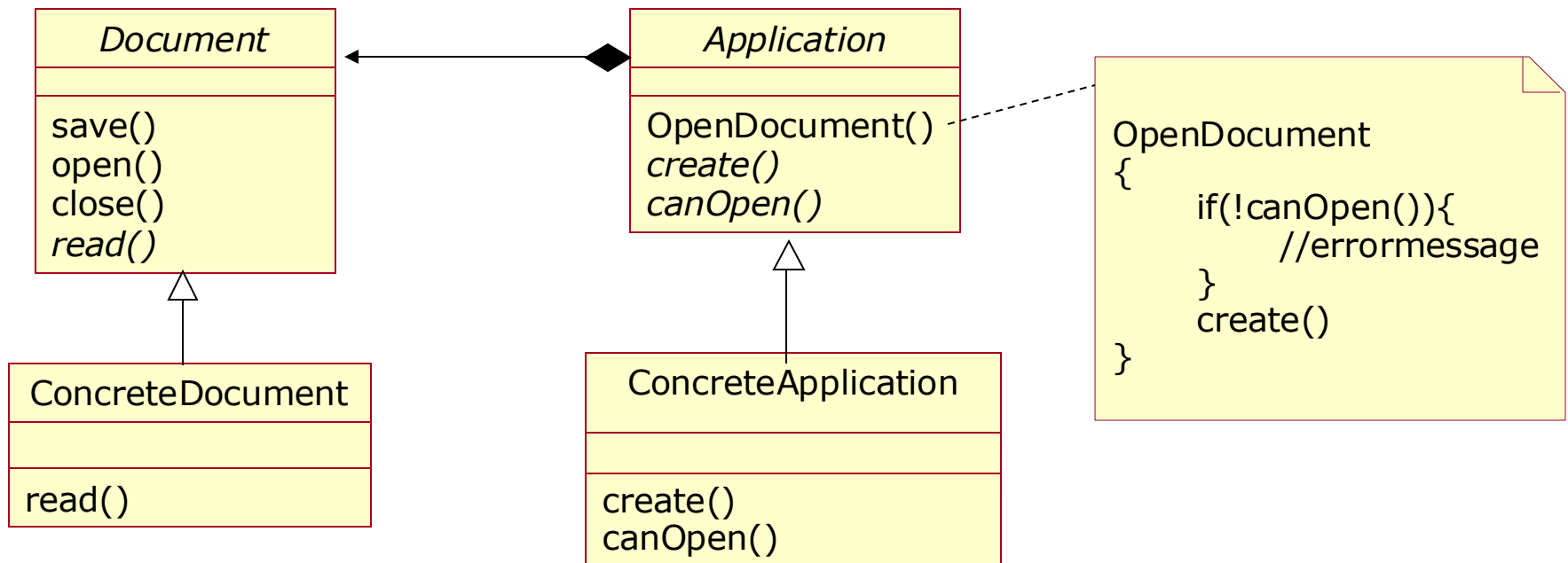
Le sottoclassi ridefiniscono solo alcuni passi dell'algoritmo ma non la sua struttura

- ♦ **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.

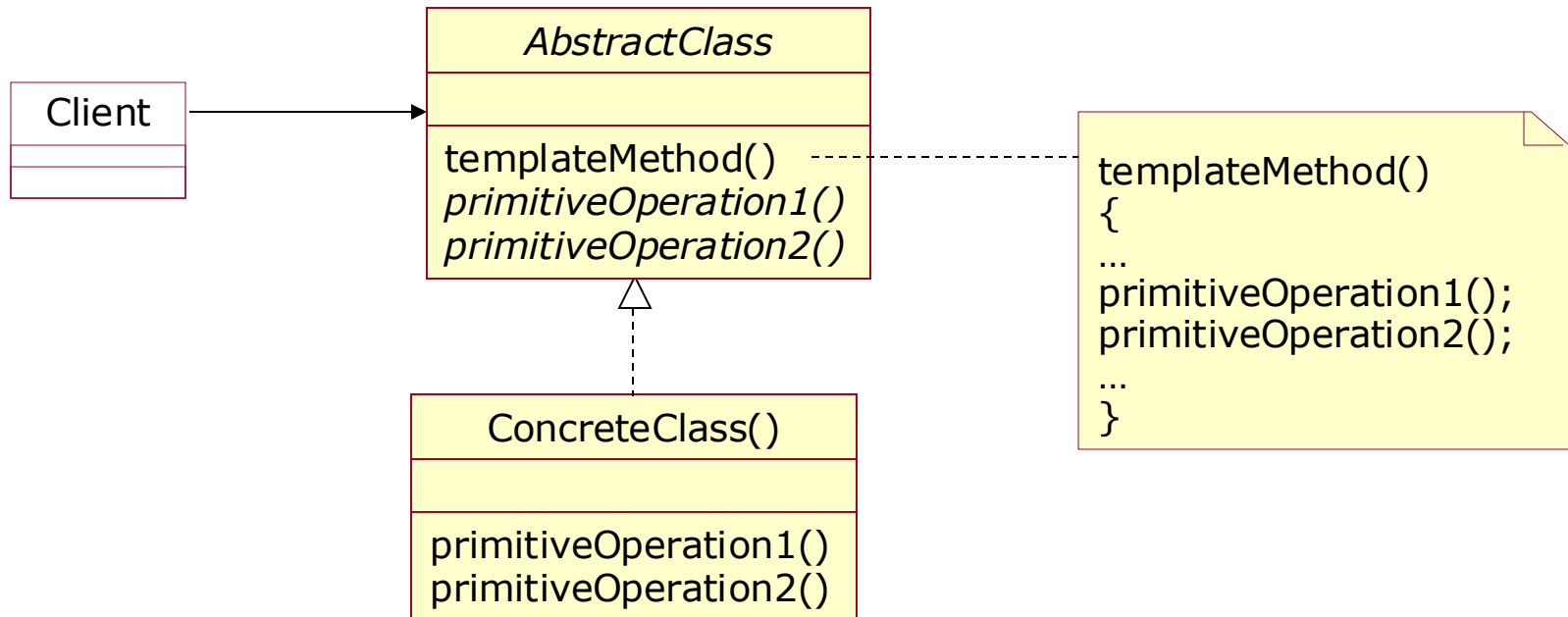
Classificazione: comportamentale basato su classi

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**

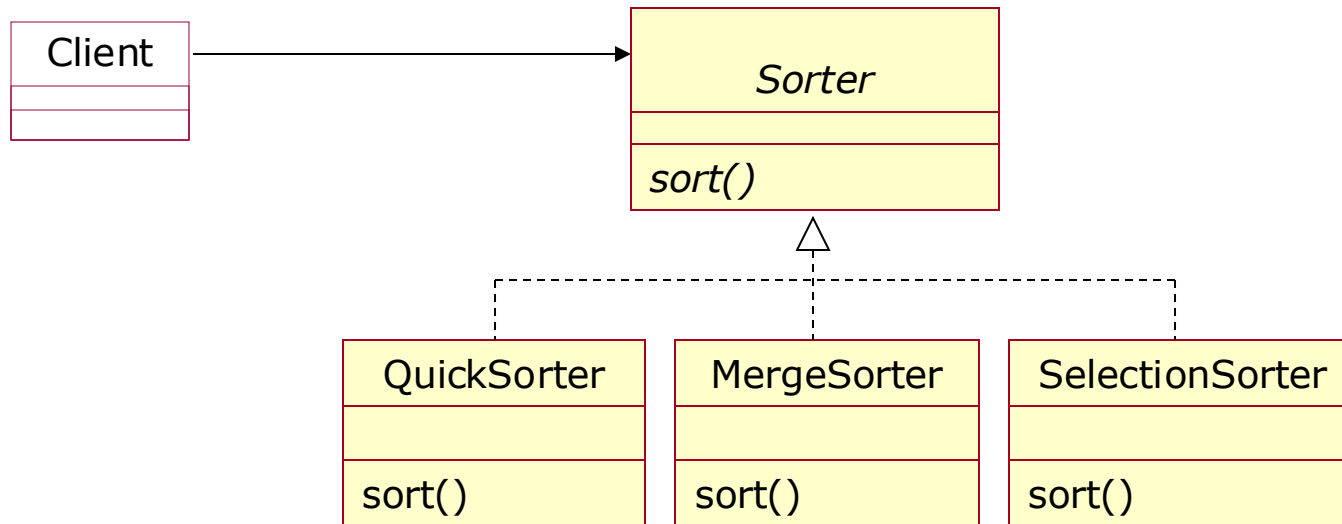
Strategy

- ♦ **Scopo:** Definire ed incapsulare 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

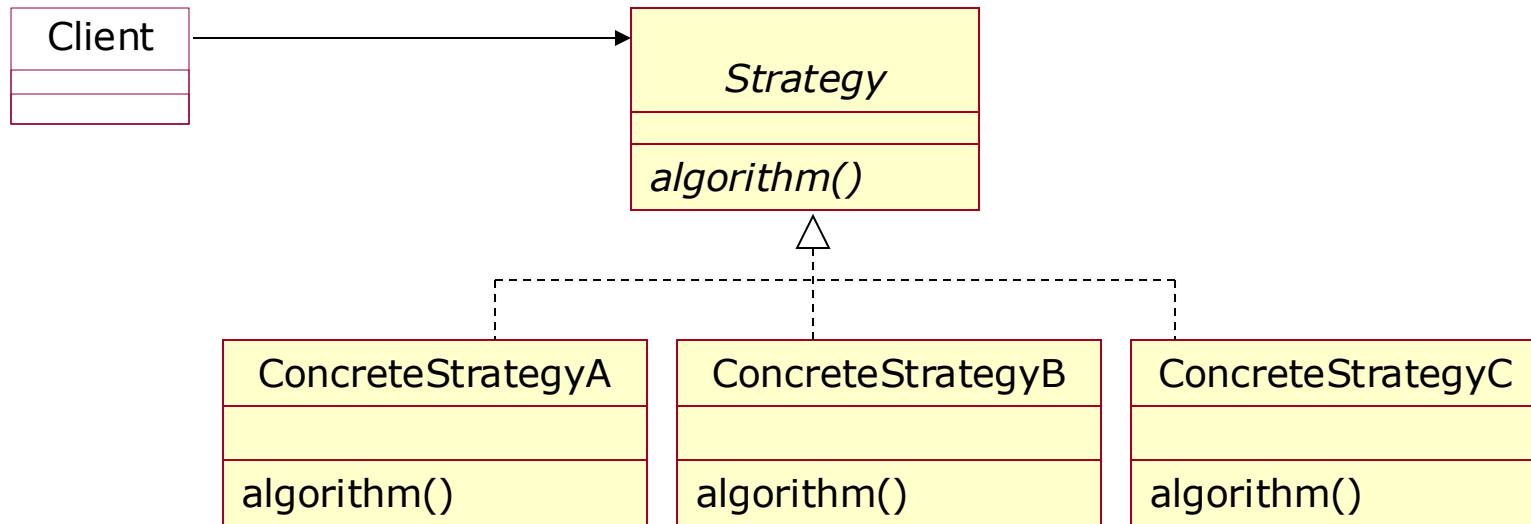
Classificazione: comportamentale basato su oggetti

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

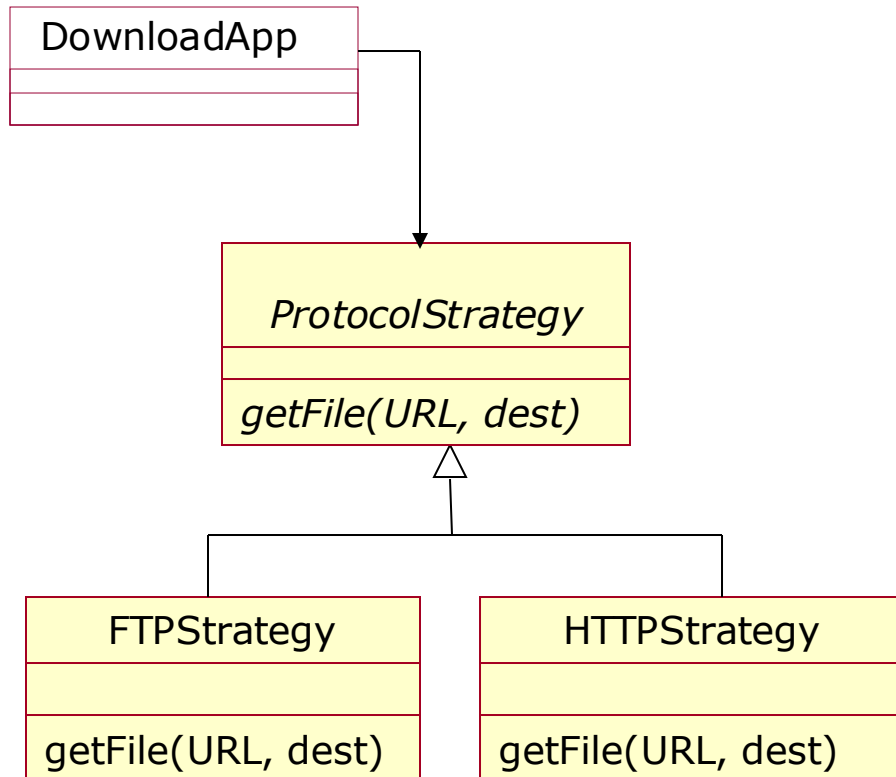
- ♦ Realizzare una applicazione per gestire il download da siti http e ftp. La selezione avviene in base all'inizio dell'url (ftp o http)

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

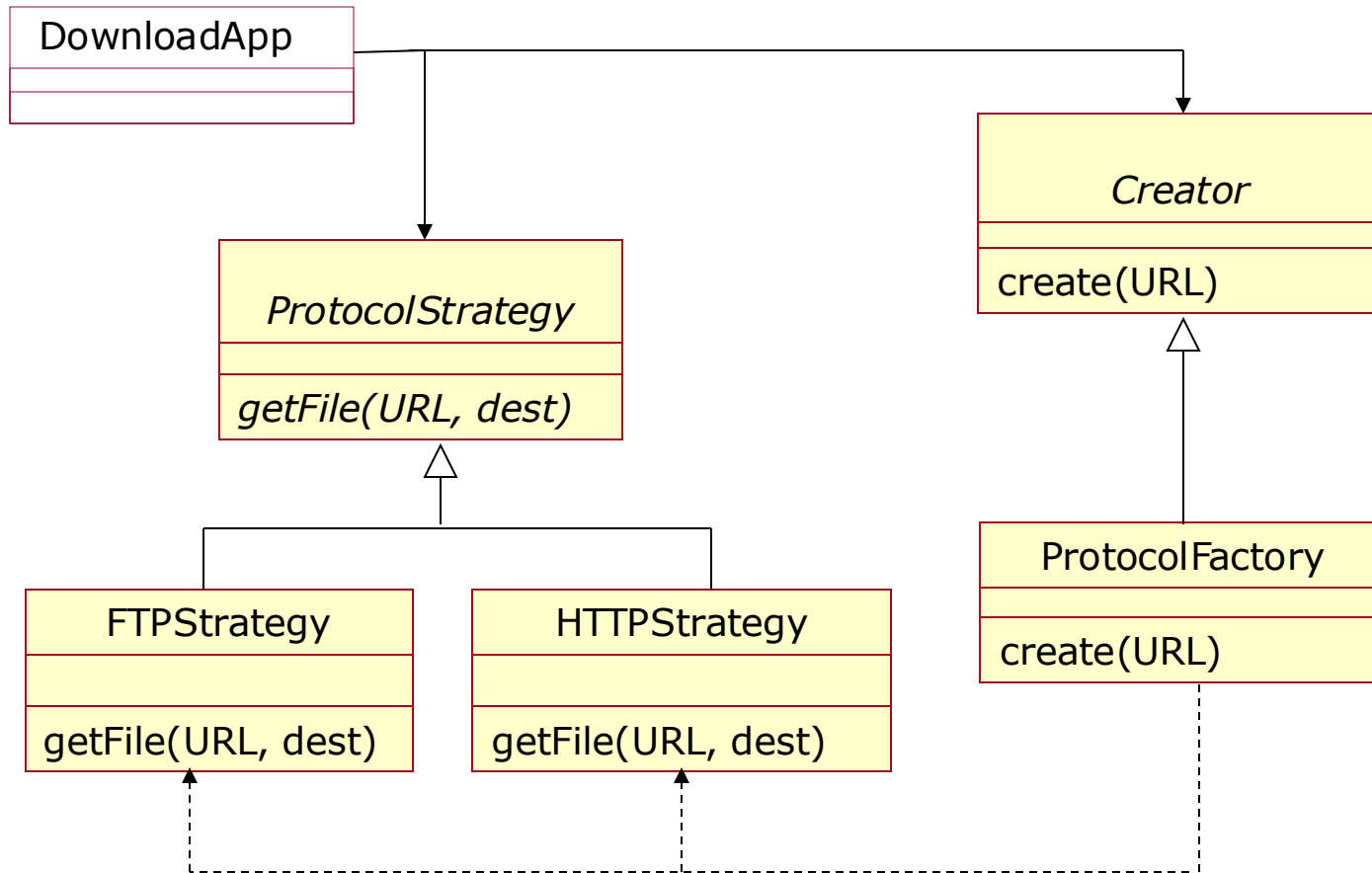


```
download( String url, String dest ) {  
    ProtocolStrategy ps=null;  
    url = url.toLowerCase();  
    if( url.startsWith( "ftp" ) ) {  
        ps = new FtpStrategy();  
    }  
    else if( url.startsWith( "http" ) ) {  
        ps = new HttpStrategy();  
    }  
    else {  
        //No operation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

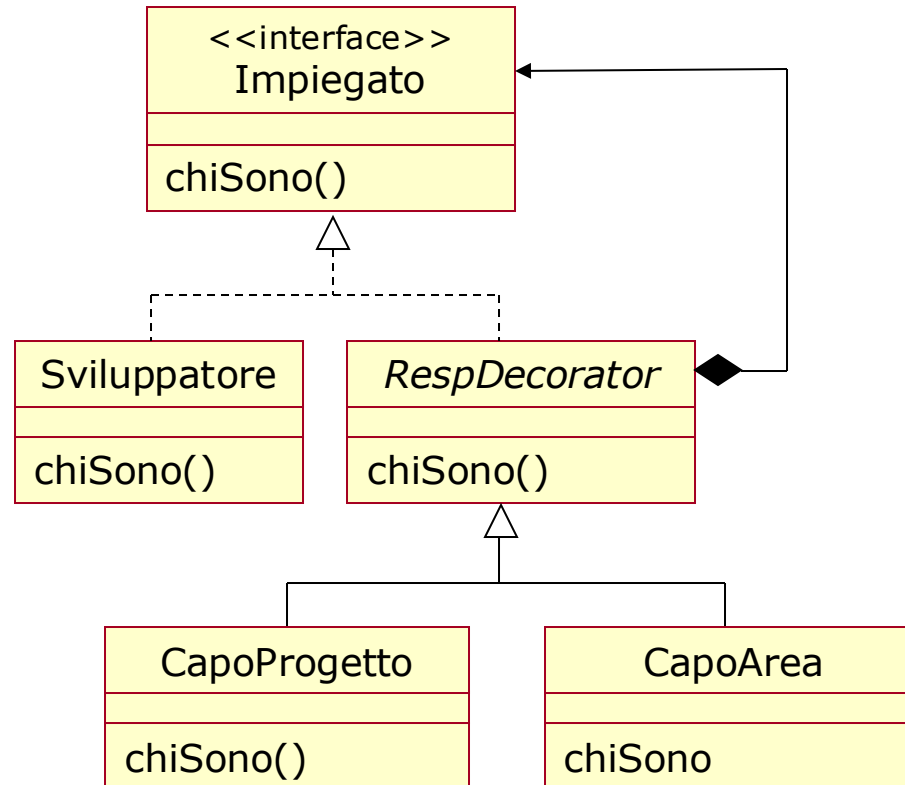
- ◆ Gli impiegati possono avere delle responsabilità aggiuntive: possono essere *CapoArea* o *CapoProgetto*, in modo non esclusivo.
- ◆ Una particolare categoria di impiegati che ci interessa sono gli *Sviluppatori*

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

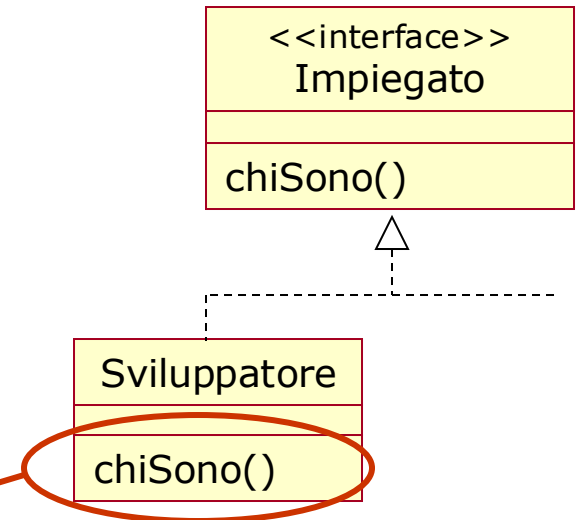
Soluzione 2/2



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

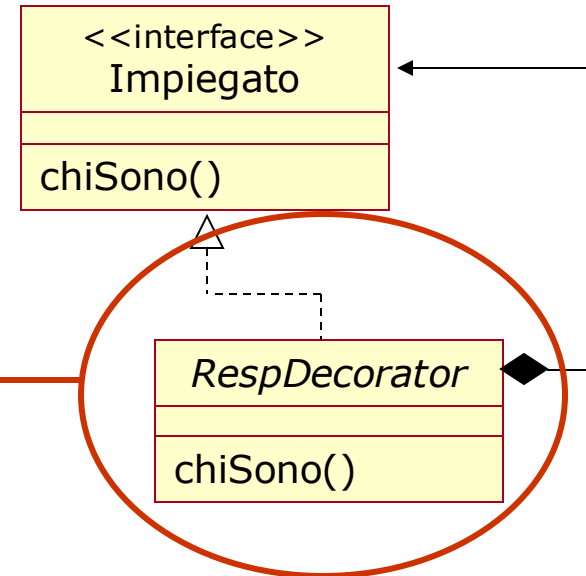
abstract class RespDecorator implements Impiegato {

protected Impiegato responsabile; 

public RespDecorator(Impiegato imp) {
 responsabile = imp;
}

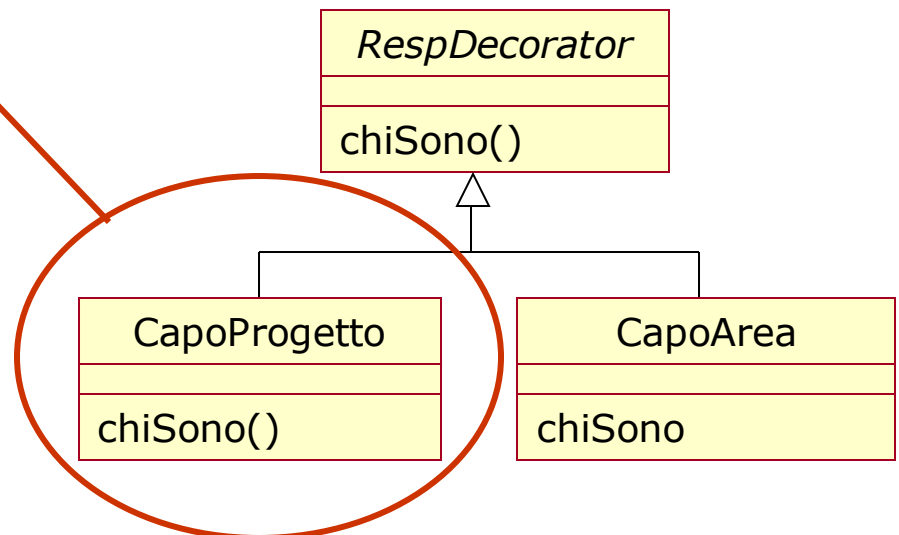
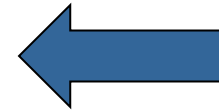
public String getName() {
 return responsabile.getName();
}

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();
```

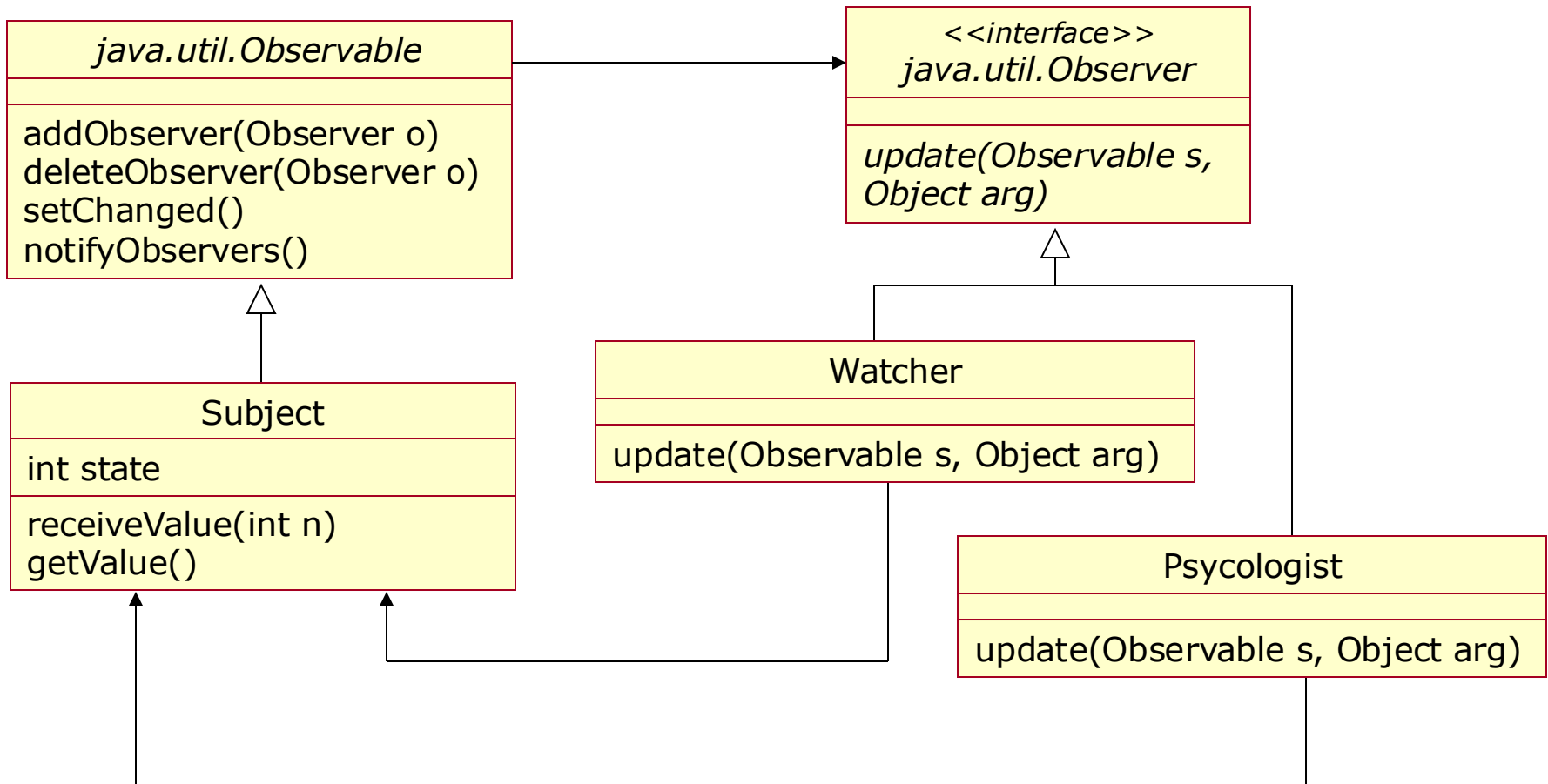
Salve, sono lo sviluppatore Pluto e sono anche un CapoProgetto!

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();  
        ...  
    }  
}
```

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

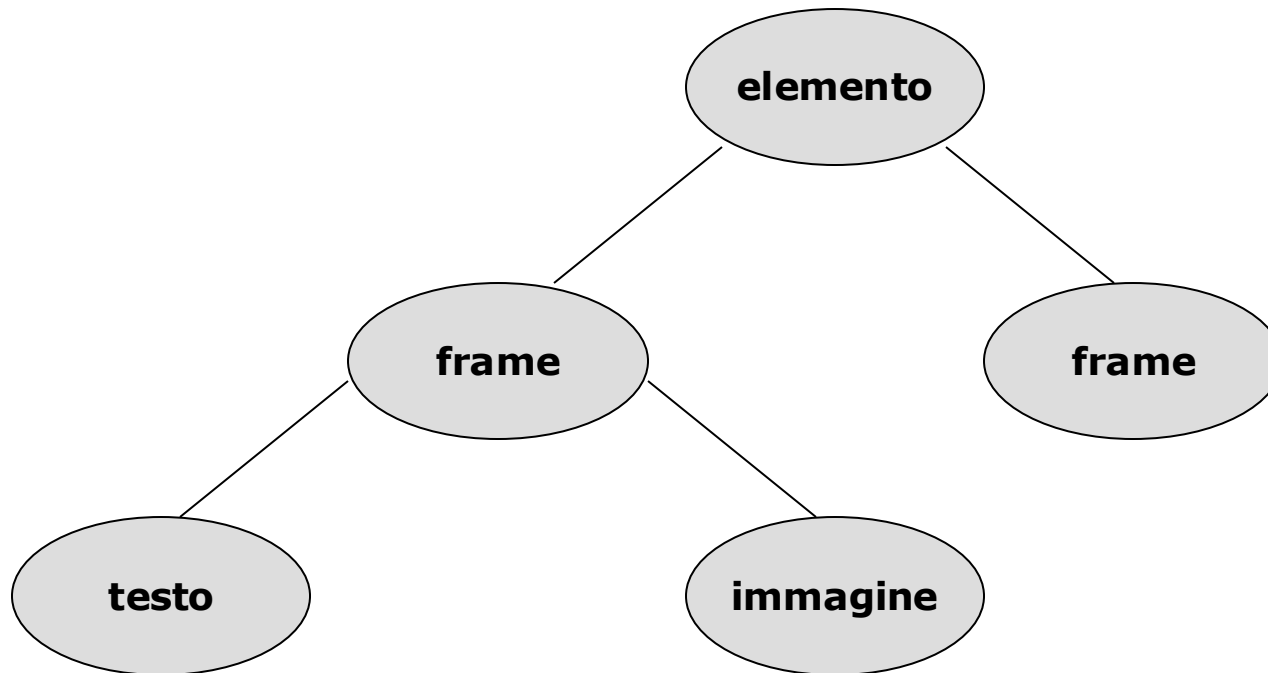
suggerimento: usare i pattern Composite, Strategy, Decorator....

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

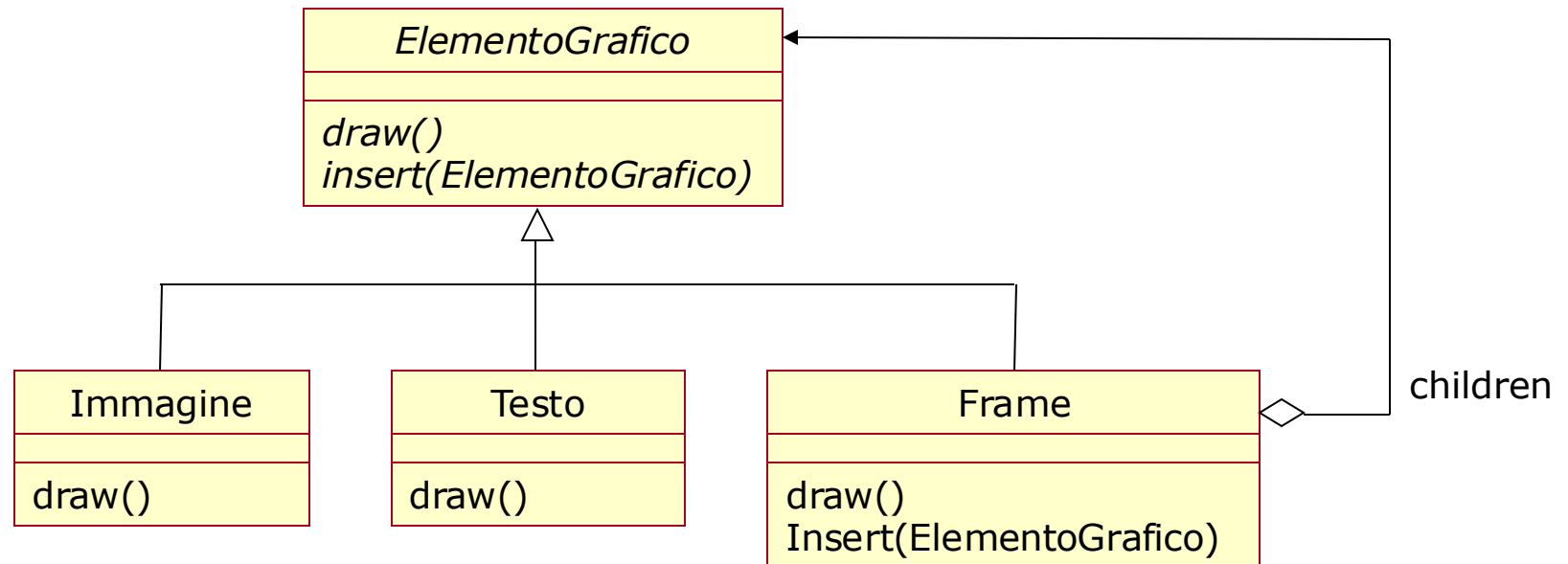
Soluzione 4/2

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



Soluzione 4/3

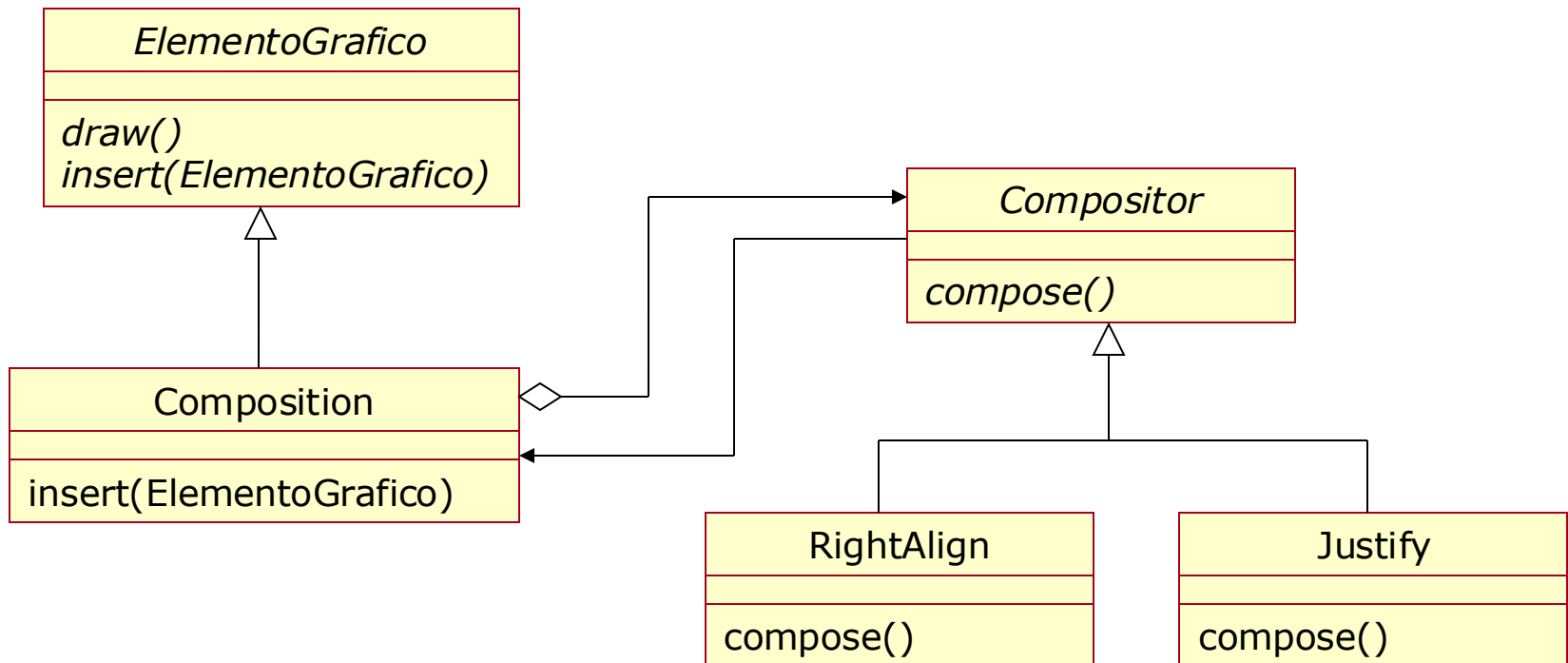
Rappresentazione degli elementi grafici: il pattern Composite



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

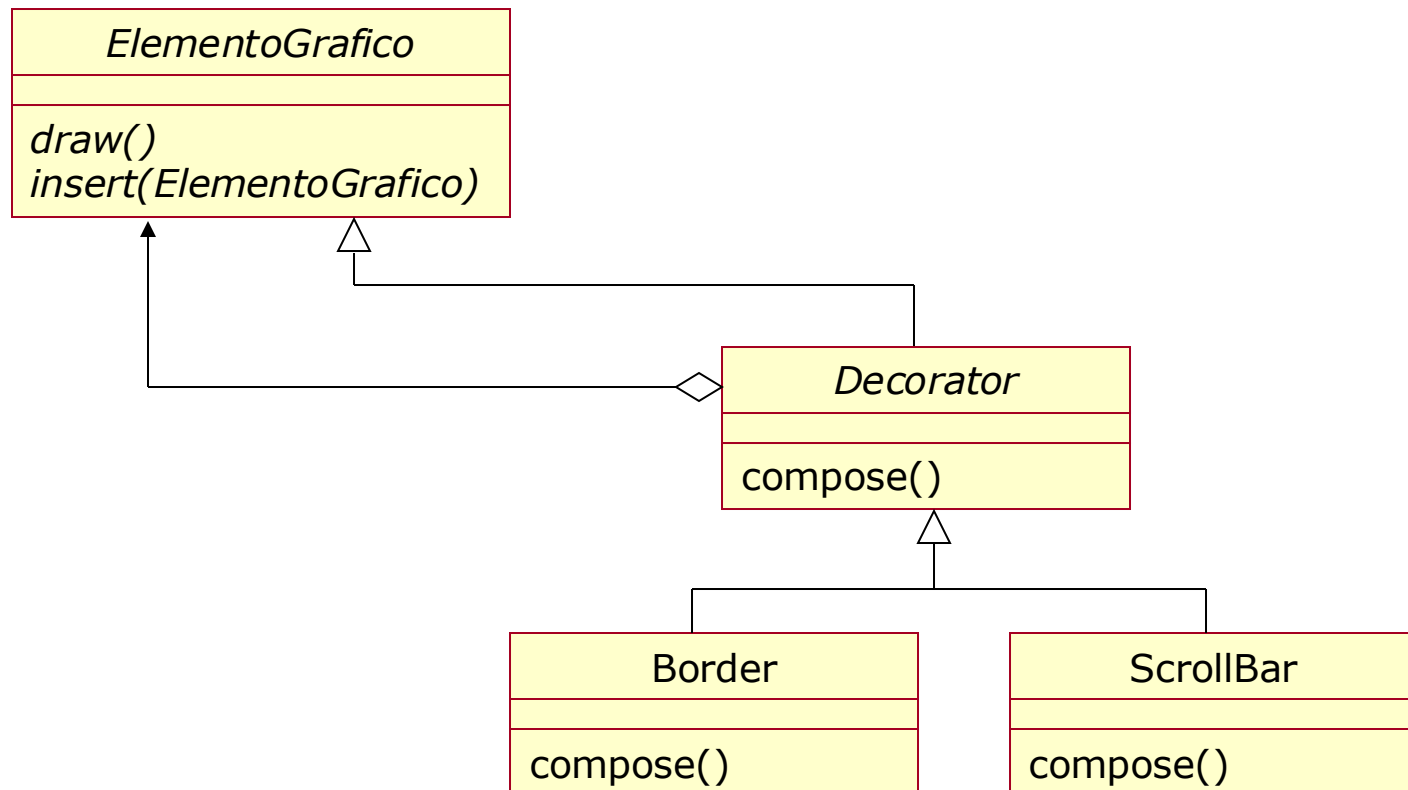
Soluzione 4/4

Rappresentazione degli algoritmi di formattazione:
il pattern Strategy



Soluzione 4/5

Rappresentazione delle proprietà grafiche: il pattern Decorator



Soluzione 4/6

Mettendo tutto insieme...

