Object-Oriented Design Patterns:

Decorator

Francesco De Rose,
matr. 60180

Università degli Studi di Brescia

Corso di Laurea Specialistica
in Ingegneria Informatica

Insegnamento di Ingegneria del Software B

Sommario

1) Partiremo con un semplice problema progettuale d'esempio...
2) al quale daremo una risposta semplice ma sbagliata, analizzando perché non funziona
3) poi vedremo una soluzione migliore
4) che generalizzeremo, descrivendo il pattern *Decorator*
5) infine vedremo due utilizzi noti del pattern

Il problema

- Consideriamo un dominio applicativo a noi *completamente* nuovo...

Da Ciro si possono mangiare due ottime pizze: la *classica* Margherita e l'*ottima* Napoletana.

Oggigiorno i clienti sono esigenti, quindi Ciro consente loro di personalizzare l'ordine aggiungendo alcuni ingredienti, per esempio: olive, peperoni, salsiccia, doppia mozzarella...

Ciro vi ha proposto di sviluppare per lui un sistema informativo, in cambio di un gustoso vitalizio in pizze.

Il menu

Pizzeria Da Ciro – MENU

PIZZE

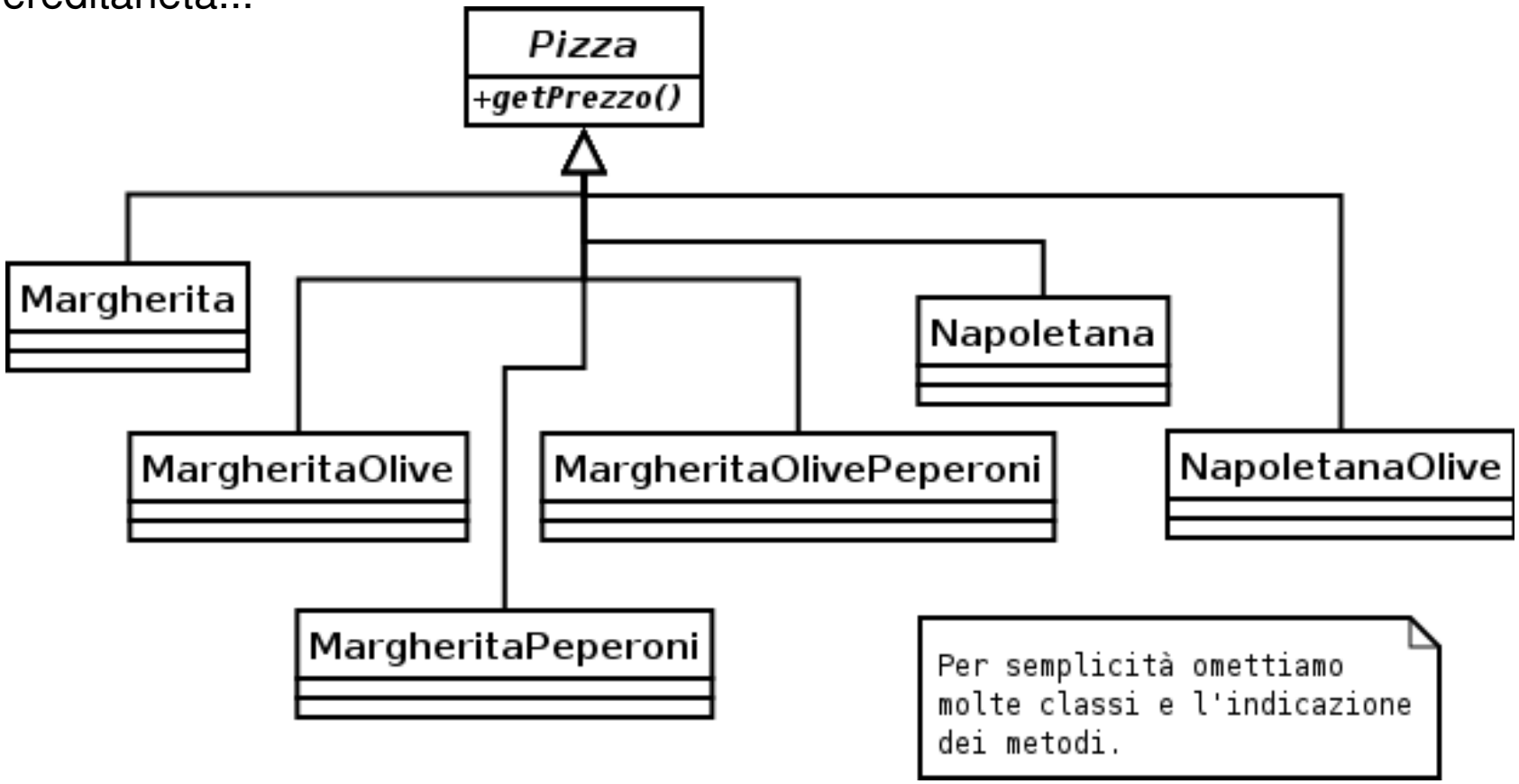
Margherita	€ 2,50
Napoletana	€ 2,70
...	

SUPPLEMENTI

Olive	€ 0,20
Peperoni	€ 0,50
...	

Una soluzione semplice (ed errata)

Il primo istinto potrebbe essere quello di lanciarsi sull'ereditarietà...



in realtà servono almeno $2^5 = 32$ sottoclassi!

Una soluzione semplice (ed errata)

```
public class Margherita {
    //...
    public float getPrezzo(){
        return 2.5;
    }
}
```

```
public class MargheritaOlive
   {
    //...
    public float getPrezzo(){
        return 2.5 + 0.20;
    }
}
```

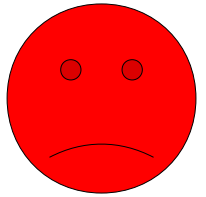
Il prezzo base...

Più il supplemento

Da implementare in ognuna delle 32 sottoclassi!

Analisi

- Questa soluzione non è molto buona...

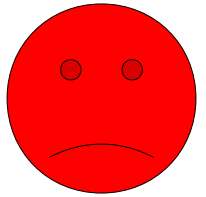


  - **Proliferazione di sottoclassi**: infatti servono 2^5 sottoclassi di Pizza, pur avendo semplificato di molto la realtà: per una pizzeria vera ne servirebbero molte di più!

  - Il sistema è di **difficile manutenzione**: se cambia il prezzo del supplemento Olive bisogna modificare il metodo getPrezzo di metà di tutte queste classi!

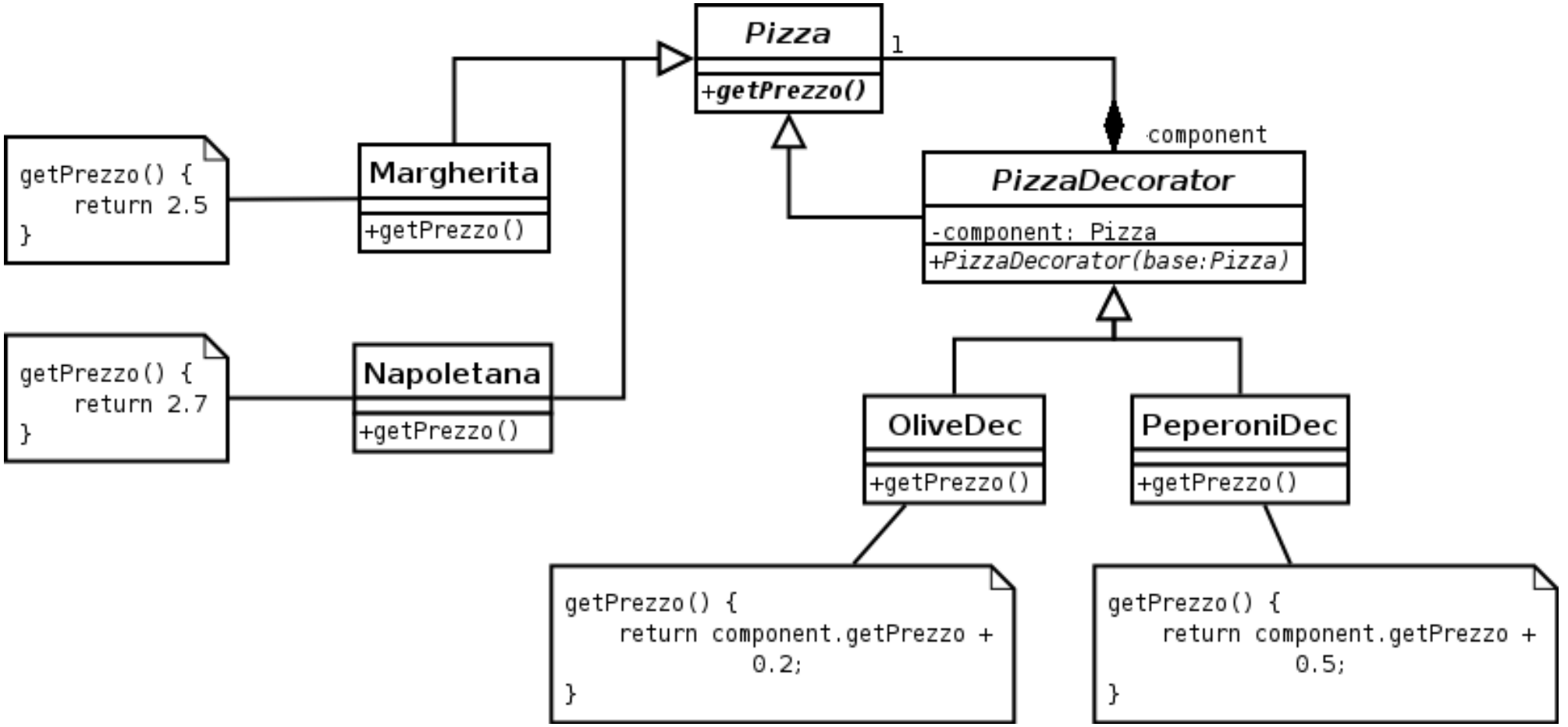
Analisi

- Questa soluzione non è molto buona...



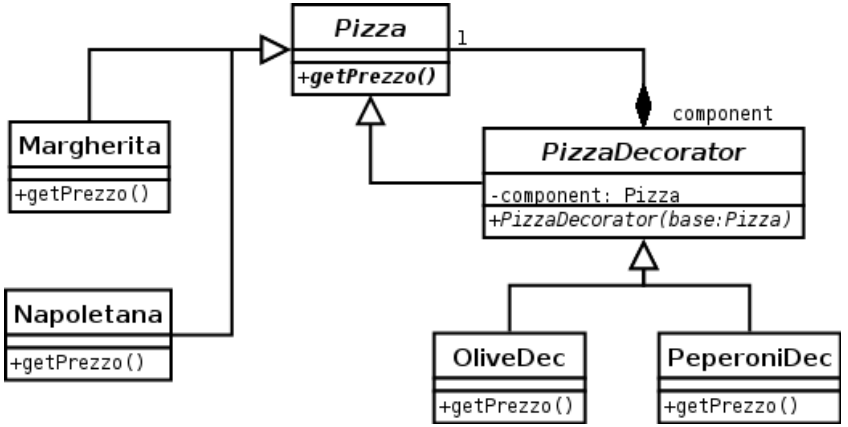
  - Il sistema è **statico**: non è possibile ad esempio istanziare una Margherita e, in un secondo momento, aggiungere le Olive.

Una soluzione più furba
(e da manuale...)



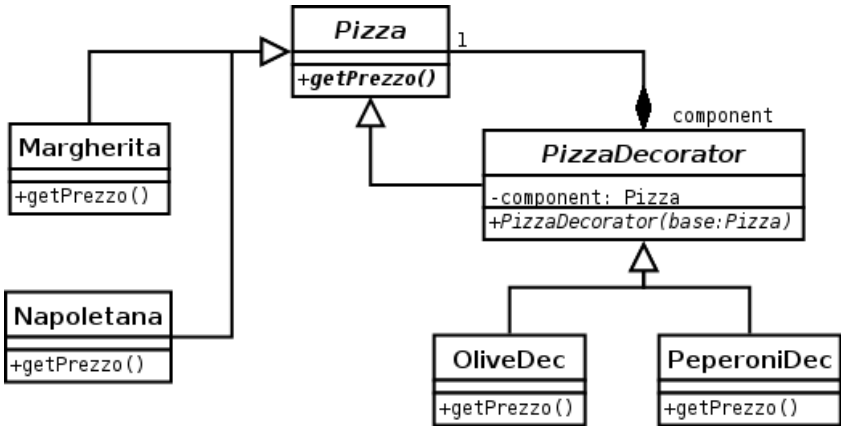
Struttura

- Sia Margherita (o Napoletana) sia OliveDec (o PeperoniDec) implementano la classe Pizza, che quindi è il supertipo comune.
- Quindi posso usare il tipo Pizza laddove non mi interessa se si tratta di una pizza con o senza supplementi.



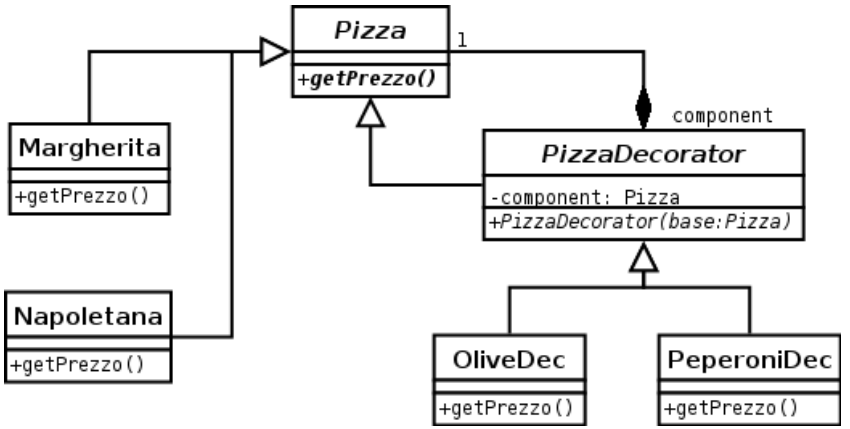
Struttura

- Un oggetto di (sotto)tipo PizzaDecorator ha un component di (sotto)tipo Pizza.
- Vedremo che di solito a questo ultimo è delegata parte del comportamento.



Il costruttore di PizzaDecorator

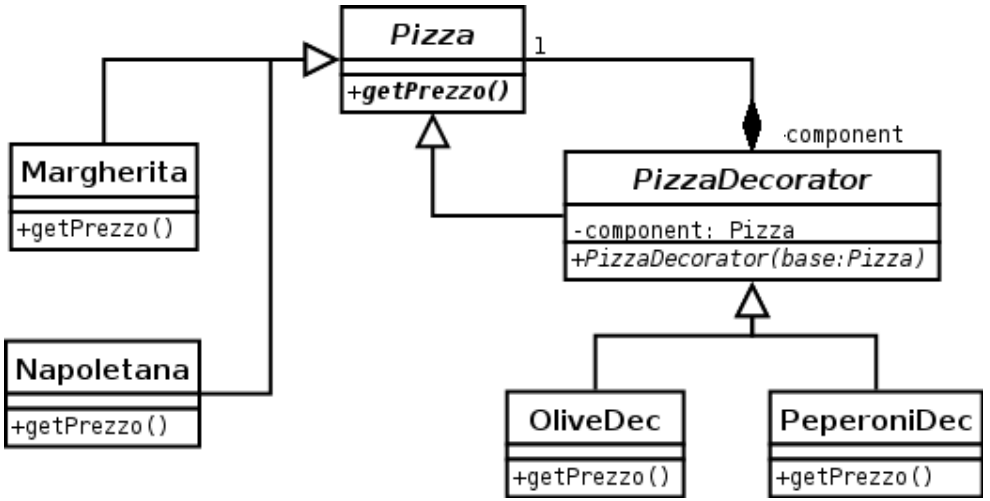
```
abstract class PizzaDecorator {
    private
        Pizza component;
        // ...
    public
        PizzaDecorator(Pizza base) {
            component = base;
        }
        // ...
}
```



Al momento dell'istanziazione di una sottoclasse di PizzaDecorator ci preoccupiamo della composizione.

Questo costruttore viene chiamato, per ereditarietà, quando si istanzia un oggetto della classe OliveDec o PeperoniDec.

Il costruttore in azione



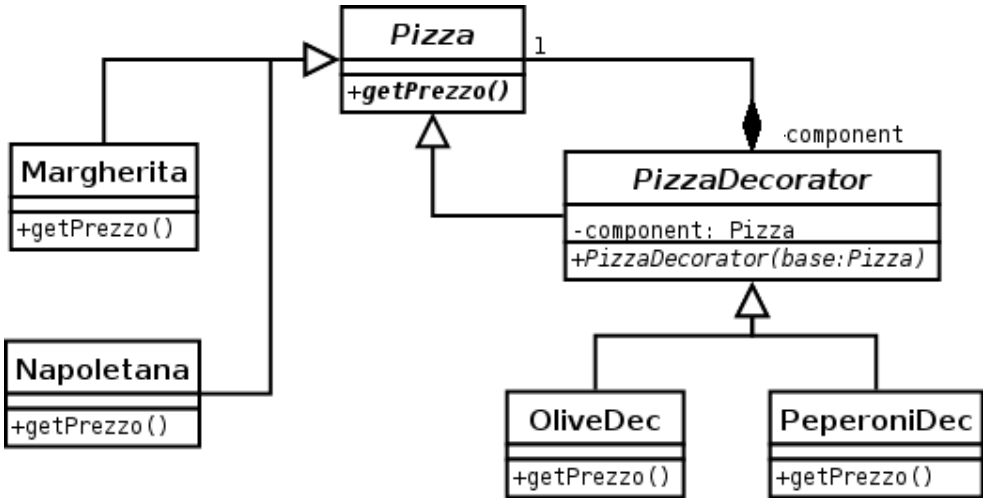
Istanziamo un paio di pizze:

```
Pizza p1 =
    new OliveDec(new Margherita());
 // Una margherita con olive
```

```
Pizza p2 = new PeperoniDec(
    new OliveDec(new Napoletana()));
 // Una napoletana con olive e peperoni
```

Nel caso di p2 stiamo inscatolando un oggetto Napoletana dentro un OliveDec che è a sua volta dentro PeperoniDec

Il costruttore in azione



La costruzione di un oggetto con due Decorator è resa possibile dal fatto che sia *component* sia *PizzaDecorator* sono di (sotto)tipo Pizza

Delega del comportamento

```
public class Margherita {
    //...
    public float getPrezzo(){
        return 2.5;
    }
}
```

```
public class OliveDec {
    //...
    public float getPrezzo(){
        return
          component.getPrezzo()+
          0.20;
    }
}
```

OliveDec **trasferisce** le richieste ricevute al suo oggetto Component...

...svolgendo **operazioni ulteriori** dopo il trasferimento della richiesta.

Caratterizzazione di Decorator

Alias:

è detto anche wrapper

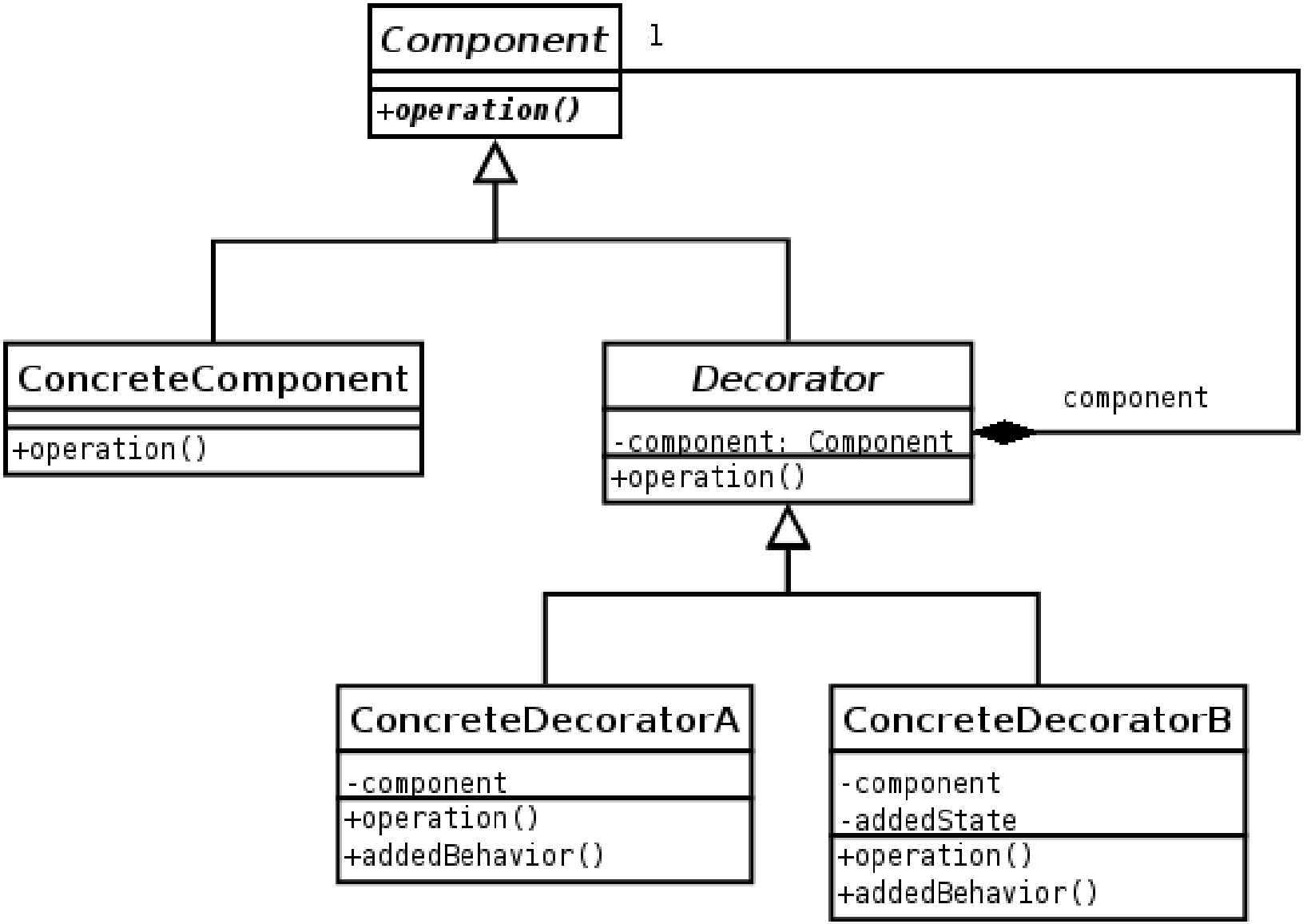
Scopo:

Aggiungere *dinamicamente* responsabilità ad un oggetto.

Classificazione:

- strutturale
- basato su oggetti

Il pattern *Decorator* in teoria



Partecipanti

- **Component**
  - Definisce l'interfaccia comune per gli oggetti ai quali possono essere aggiunte responsabilità dinamicamente
- **ConcreteComponent**
  - Definisce un oggetto al quale possono essere aggiunte responsabilità ulteriori (attraverso i Decorator)
- **Decorator**
  - Mantiene un riferimento a un oggetto Component e definisce un'interfaccia conforme all'interfaccia di Component
- **ConcreteDecorator**
  - Aggiunge responsabilità al componente

Collaborazioni

- Un Decorator **trasferisce** le richieste ricevute al suo oggetto Component.

- Può opzionalmente svolgere **operazioni ulteriori** prima e dopo il trasferimento della richiesta.

Applicabilità

- La GoF consiglia di applicare il pattern Decorator quando:
  - vogliamo poter aggiungere responsabilità a *singoli* oggetti *dinamicamente*
  - vogliamo poter togliere responsabilità agli oggetti
  - non è praticabile l'estensione attraverso *sottoclassi*

Conseguenze

- Maggiore **flessibilità** rispetto all'ereditarietà statica.

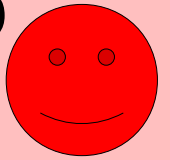
  - le responsabilità sono aggiunte e rimosse durante l'**esecuzione** (e non la compilazione)
  - Posso perfino aggiungere lo stesso decoratore più volte! (Con la derivazione non sarebbe possibile)

Conseguenze

- Maggiore **semplicità** delle classi definite

  - Si definisce una classe semplice, alla quale si aggiungono in modo incrementale le funzionalità particolari necessarie.

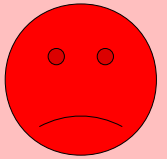
- Un decoratore e il suo Component **non** sono **identici**



  - Dal punto di vista dell'identità di un oggetto sono differenti. Pertanto non si può scrivere codice che dipenda da questa identità.

Conseguenze

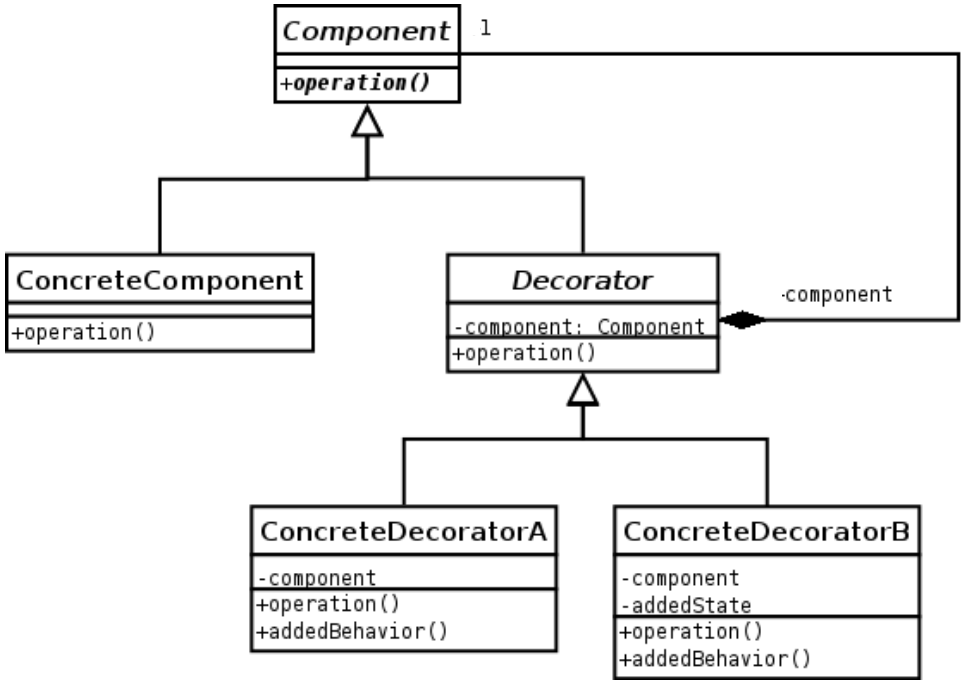
- Una grande **quantità** di piccoli oggetti.



  - Usare Decorator in un progetto spesso porta a sistemi composti da un gran numero di piccoli oggetti molto simili, differenti solo nel modo in cui sono interconnessi.

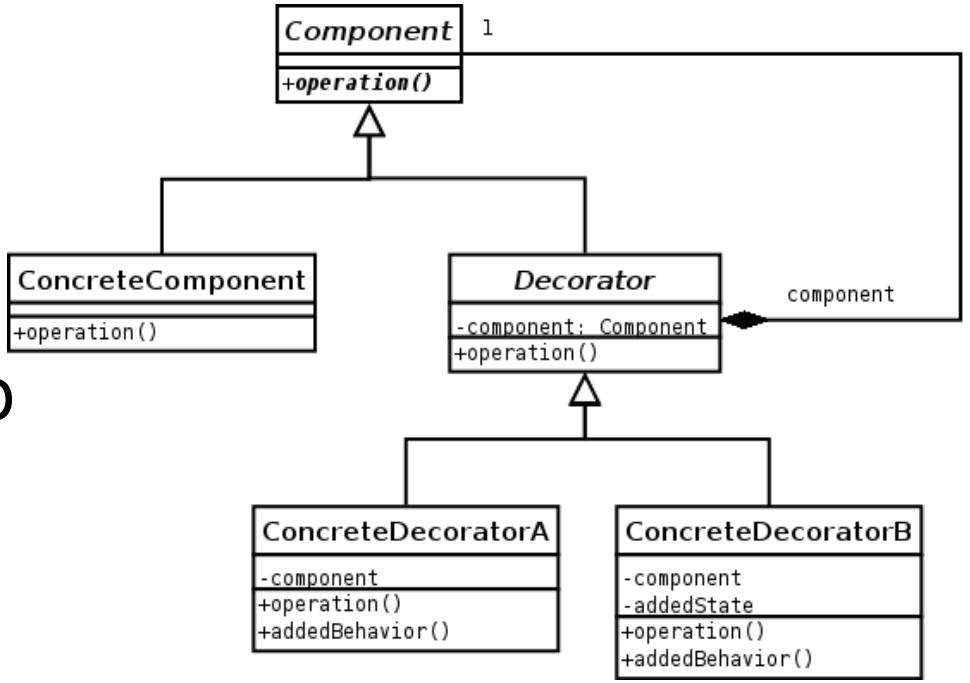
Implementazione

- Conformità delle interfacce
  - ConcreteComponent e Decorator devono ereditare da una superclasse comune, in modo che un oggetto “decorato” ed uno semplice abbiano lo stesso supertipo.



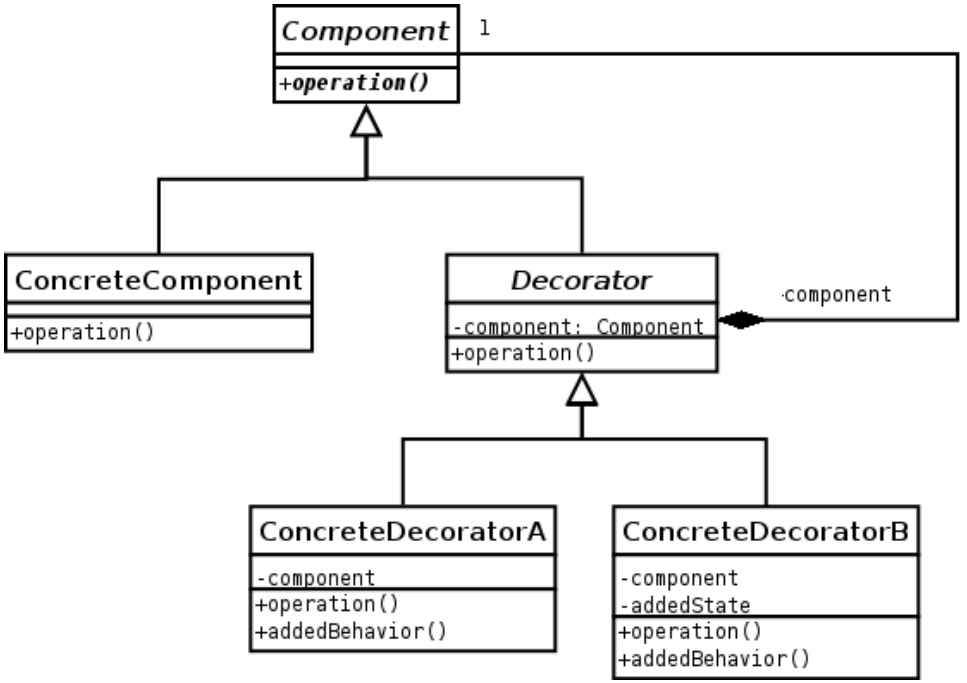
Implementazione

- Omissione della classe astratta Decorator
  - Se esiste un solo tipo di Decorator, oppure stiamo lavorando su una gerarchia di classi preesistente, è possibile omettere la classe astratta Decorator e semplificare il pattern.



Implementazione

- Mantenere Component leggere
  - Tanto i ConcreteComponent che ConcreteDecorator derivano da Component, i secondi saranno probabilmente usati in quantità.
    È quindi opportuno che Component sia più la più leggera possibile.



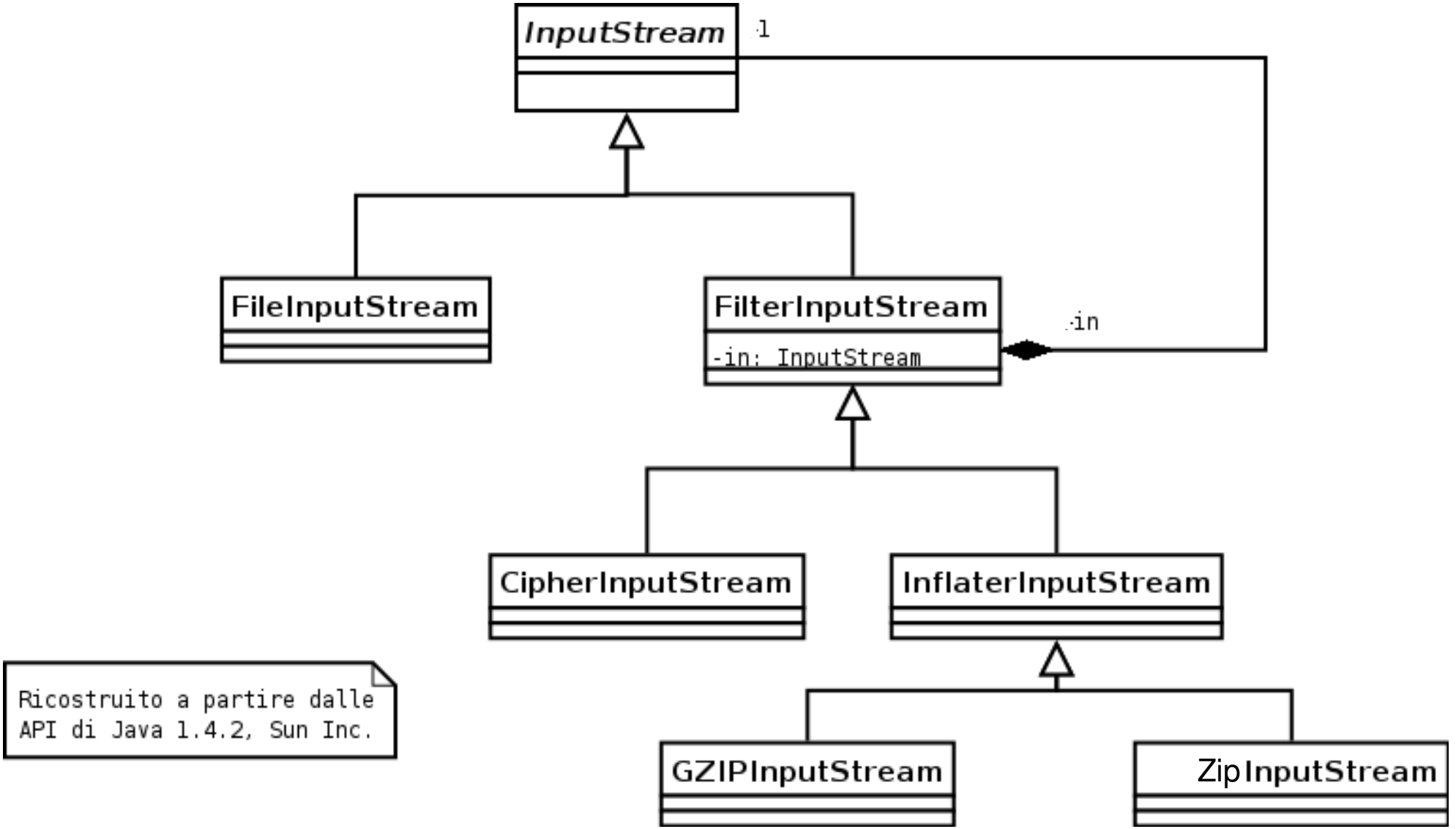
Pattern Correlati

Adapter:
Un Decorator modifica le responsabilità di un oggetto senza l'interfaccia, mentre un Adapter gli fornisce un'interfaccia completamente nuova.

Composite:
Un decoratore può essere visto come un oggetto composito degenere costituito da un singolo componente. Ad ogni modo, un decoratore è usato per aggiungere responsabilità ad un oggetto e non è pensato per gestire l'aggregazione tra oggetti.

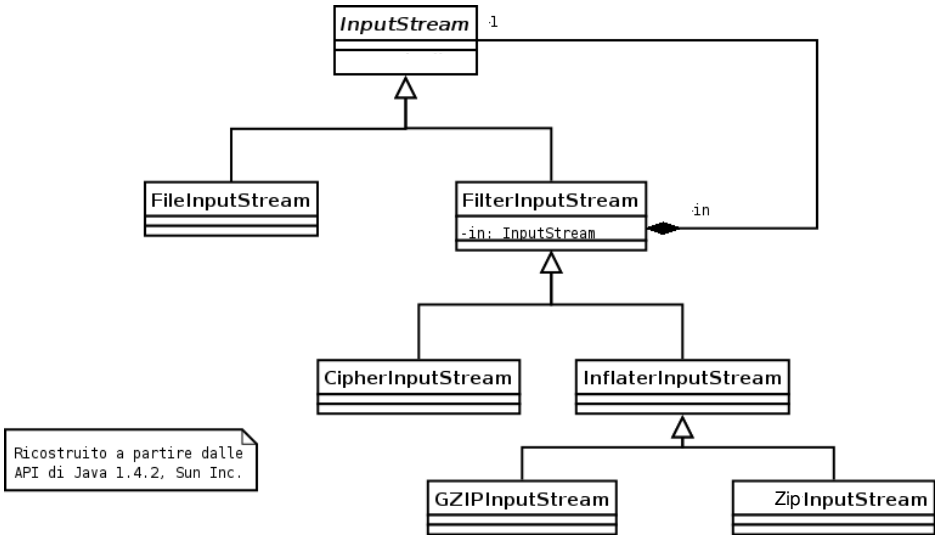
Strategy:
Un decoratore consente di cambiare la “pelle” di un oggetto, mentre un oggetto strategy consente di cambiarne “gli organi interni”. Si tratta di due alternative utilizzabili per modificare il comportamento di un oggetto.

Utilizzi noti



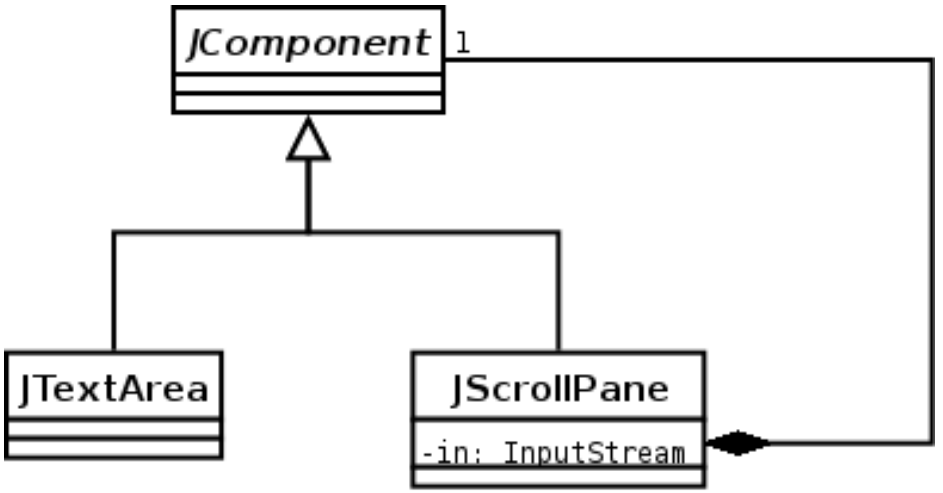
Utilizzi noti

- Un FilterInputStream decora un oggetto di tipo InputStream, per esempio aggiungendo capacità crittografiche oppure di compressione.



Utilizzi noti

- JScrollPane decora un Jcomponent (ad esempio una JTextArea che rappresenta un elemento GUI per la visualizzazione di testo) aggiungendo delle barre di scorrimento.



Bibliografia

- E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software

- E. Freeman et al, Head First Design Patterns, O'Reilly

  Il capitolo 3 relativo al pattern Decorator è disponibile gratuitamente sul sito dell'editore: http://www.oreilly.com/catalog/hfdesignpat/chapter/ch03.pdf

Domande ?