PATTERN DECORATOR

Corso di Laurea Specialistica in Ingegneria Informatica
Insegnamento di “Ingegneria del Software B”

Ex presentazione realizzata dallo studente Alberto Feriotti
nell’a.a. 2008/2009

CLASSIFICAZIONE

		Purpose		
		Creational	Structural	Behavioral
Scope	**Class**	Factory Method (107)	Adapater (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapater (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

- Necessità di aggiungere dinamicamente responsabilità a singoli oggetti e non all’intera classe.

MOTIVAZIONE

- Esempio: durante l'uso di un'interfaccia grafica, dovrebbe essere possibile aggiungere (e togliere) ai componenti grafici proprietà quali bordi o barre di scorrimento.

  Per ottenere l'effetto desiderato si può procedere seguendo due vie:

METODO 1

- <u>Usando l’ereditarietà</u>.
  Ereditare una proprietà da una classe permette di utilizzare tale proprietà in ognuna delle sue sottoclassi.

METODO 1: Considerazioni

- L'ereditarietà non è un approccio flessibile in quanto la proprietà viene aggiunta staticamente a tutte le sottoclassi; il client NON può controllare come e quando "decorare" un componente.

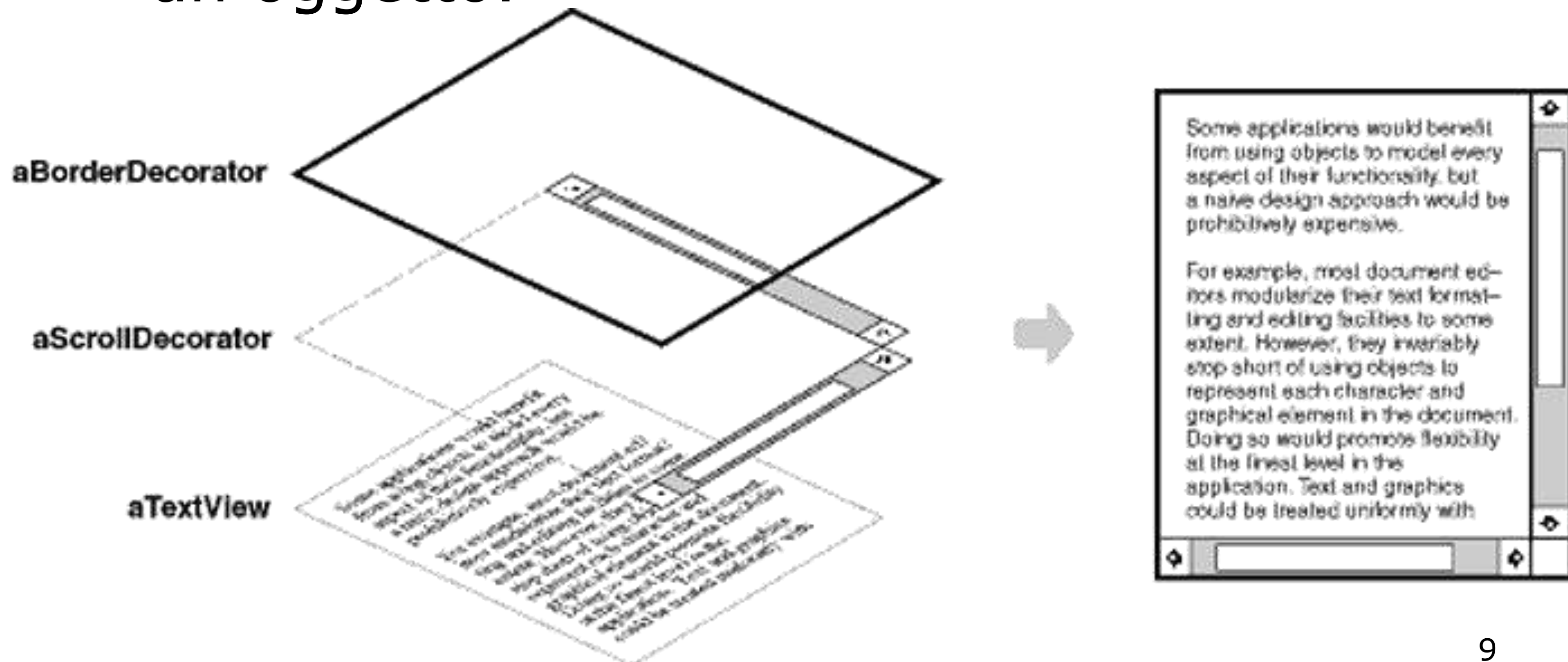
- Racchiudendo il componente da decorare in un altro responsabile dell’aggiunta della proprietà.
- L’oggetto “contenitore” si chiama DECORATOR.
- Approccio più flessibile.

DECORATOR: CARATTERISTICHE

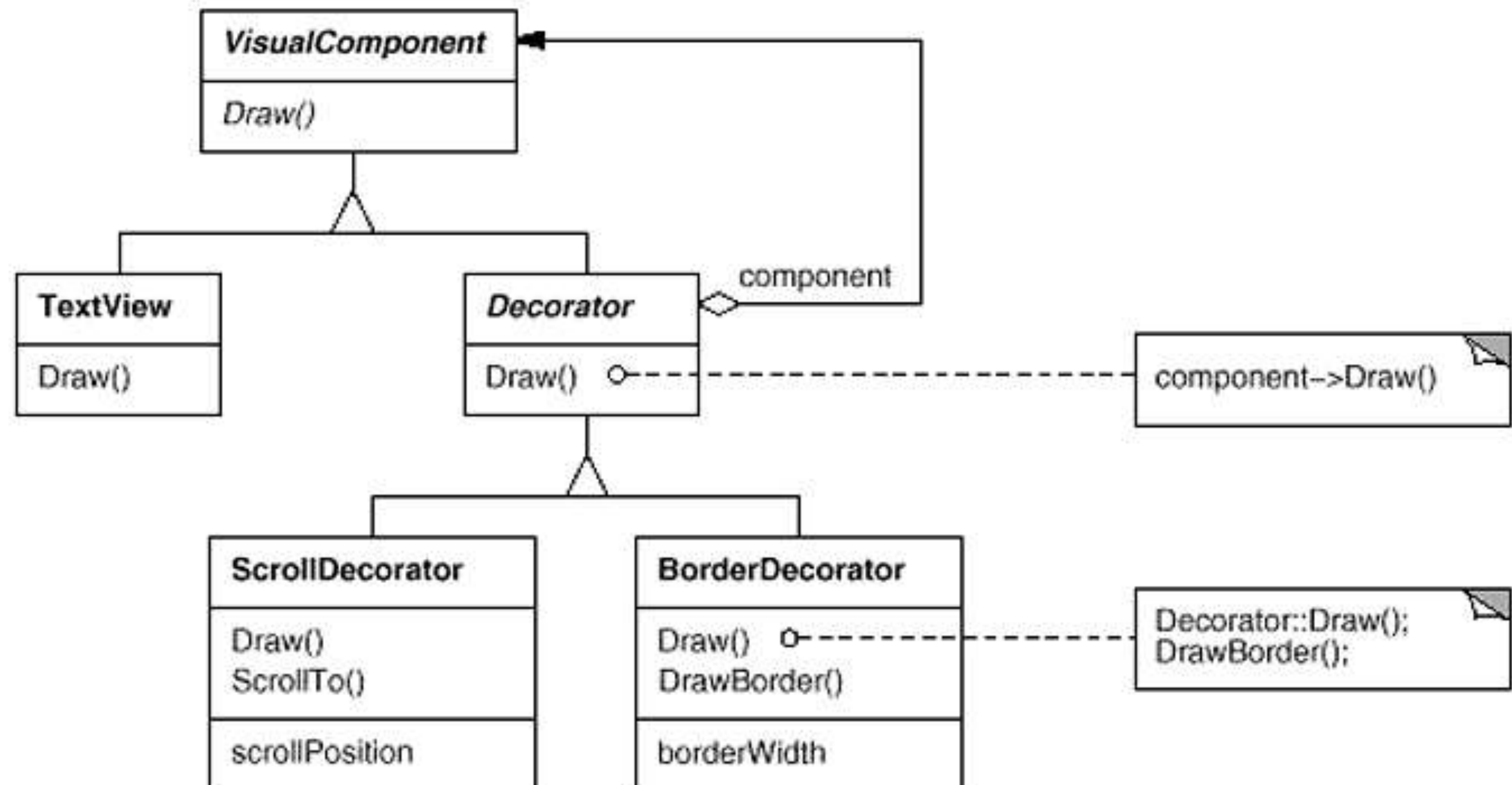
- Interfaccia TRASPARENTE e CONFORME a quella dell'oggetto da decorare per non rivelare la sua presenza ai client.
  Il Decorator delega le richieste al componente decorato, svolgendo anche <u>azioni aggiuntive.</u>

DECORATOR: TRASPARENZA

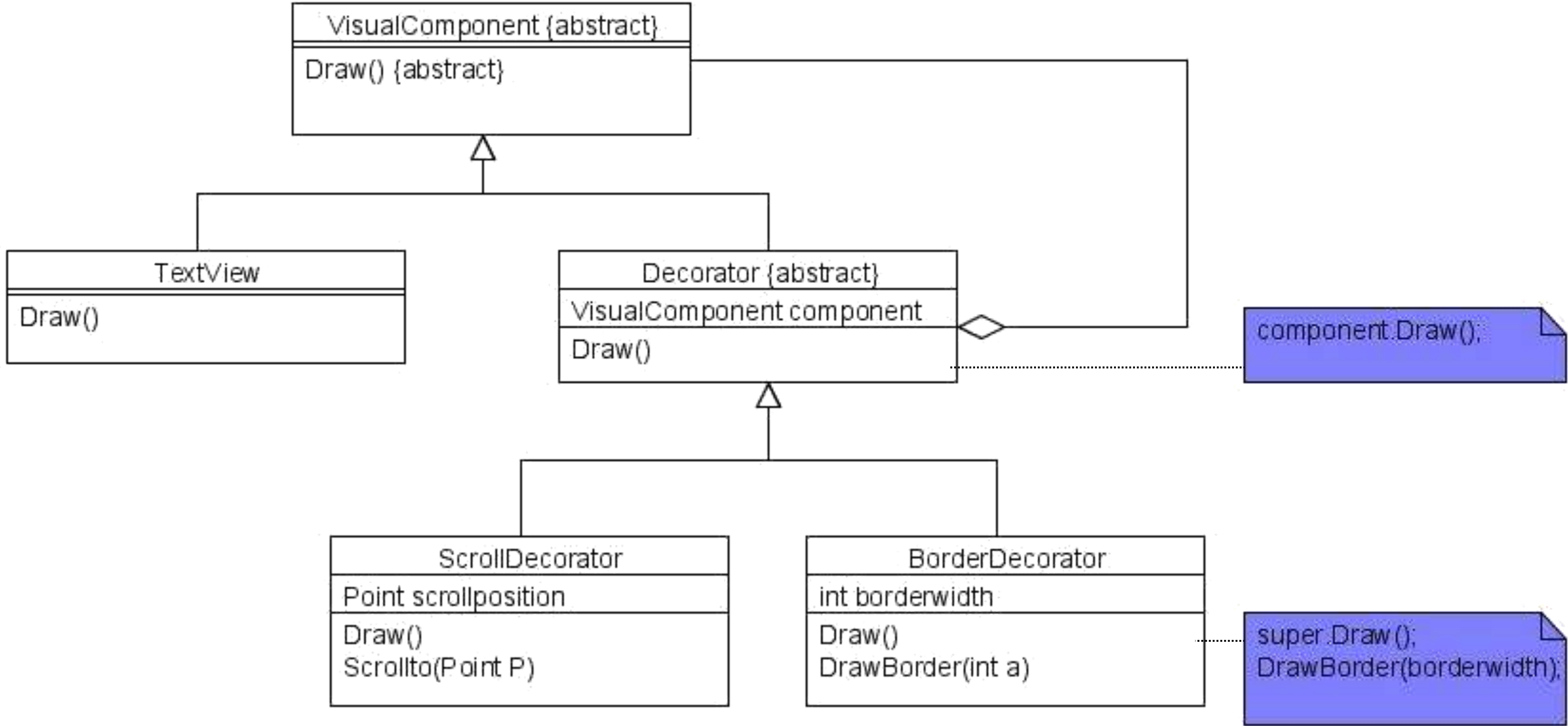
- Grazie alla trasparenza è possibile annidare i decoratori in maniera ricorsiva, con conseguente possibilità di aggiungere un numero illimitato di responsabilità ad un oggetto.



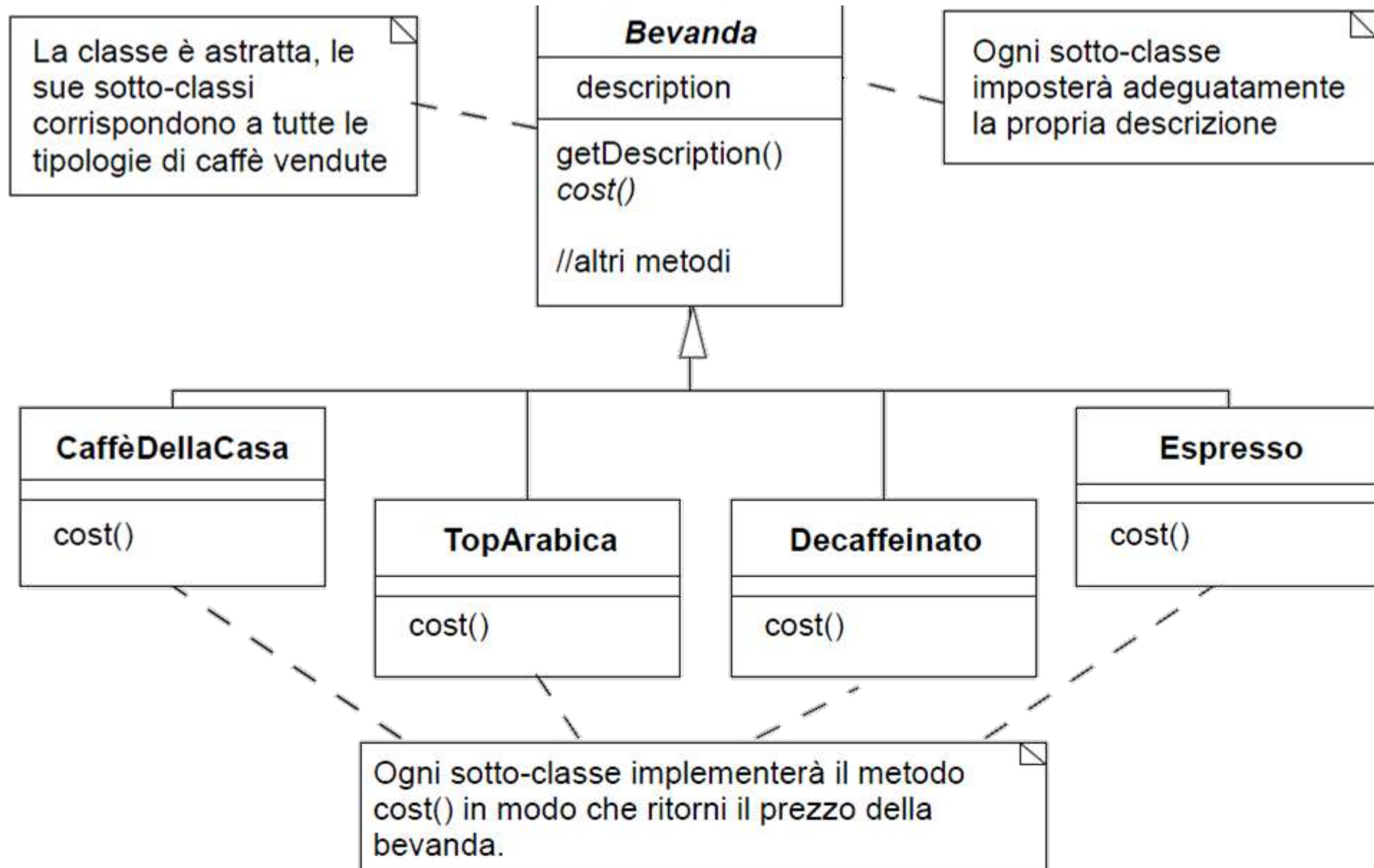
STRUTTURA: Diagramma OMT



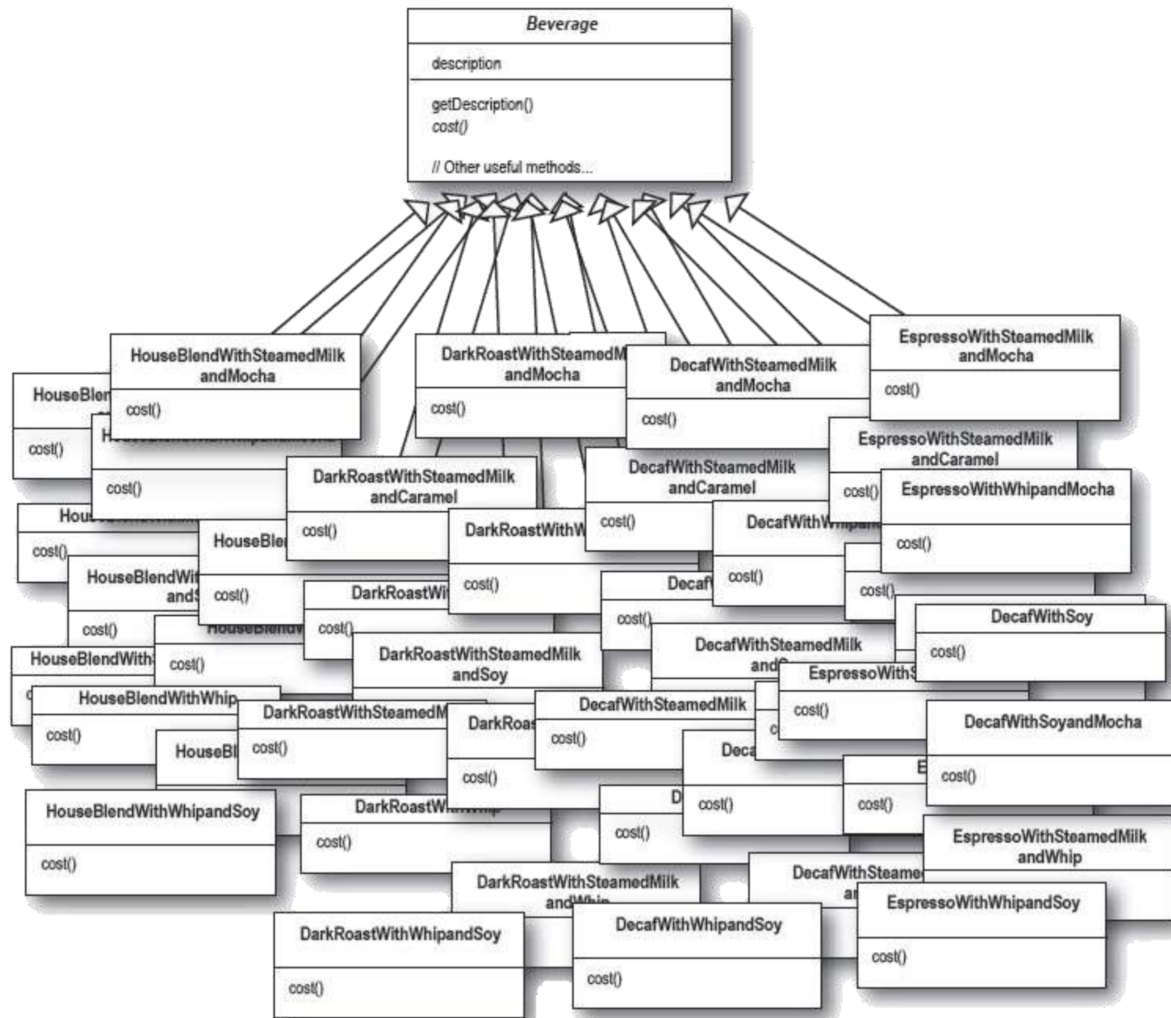
STRUTTURA: Diagramma UML



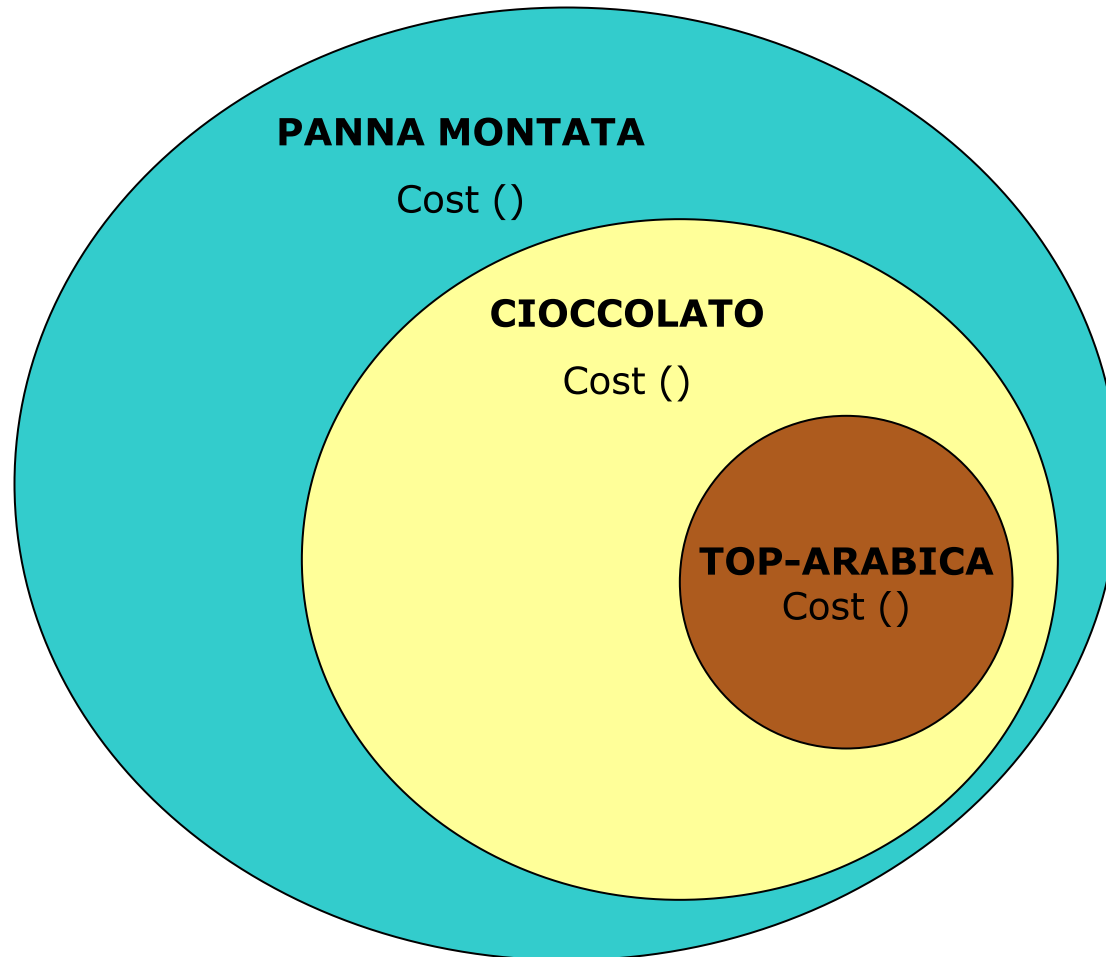
ESEMPIO STARBUZZ CAFFÈ (1)



ESEMPIO STARBUZZ CAFFÈ (2)



ESEMPIO STARBUZZ CAFFÈ (3)



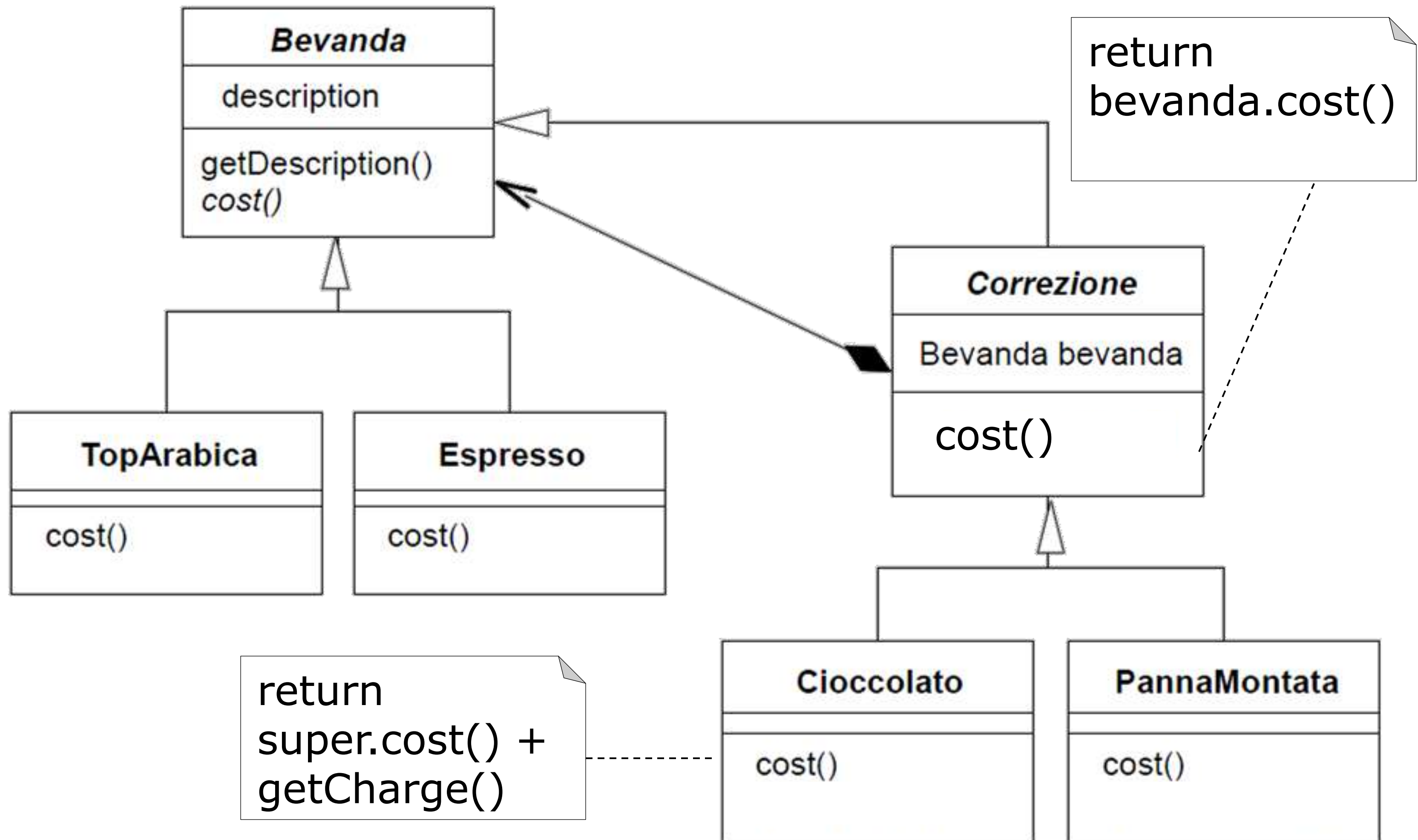
Detto anche: Wrapper

APPLICABILITÀ

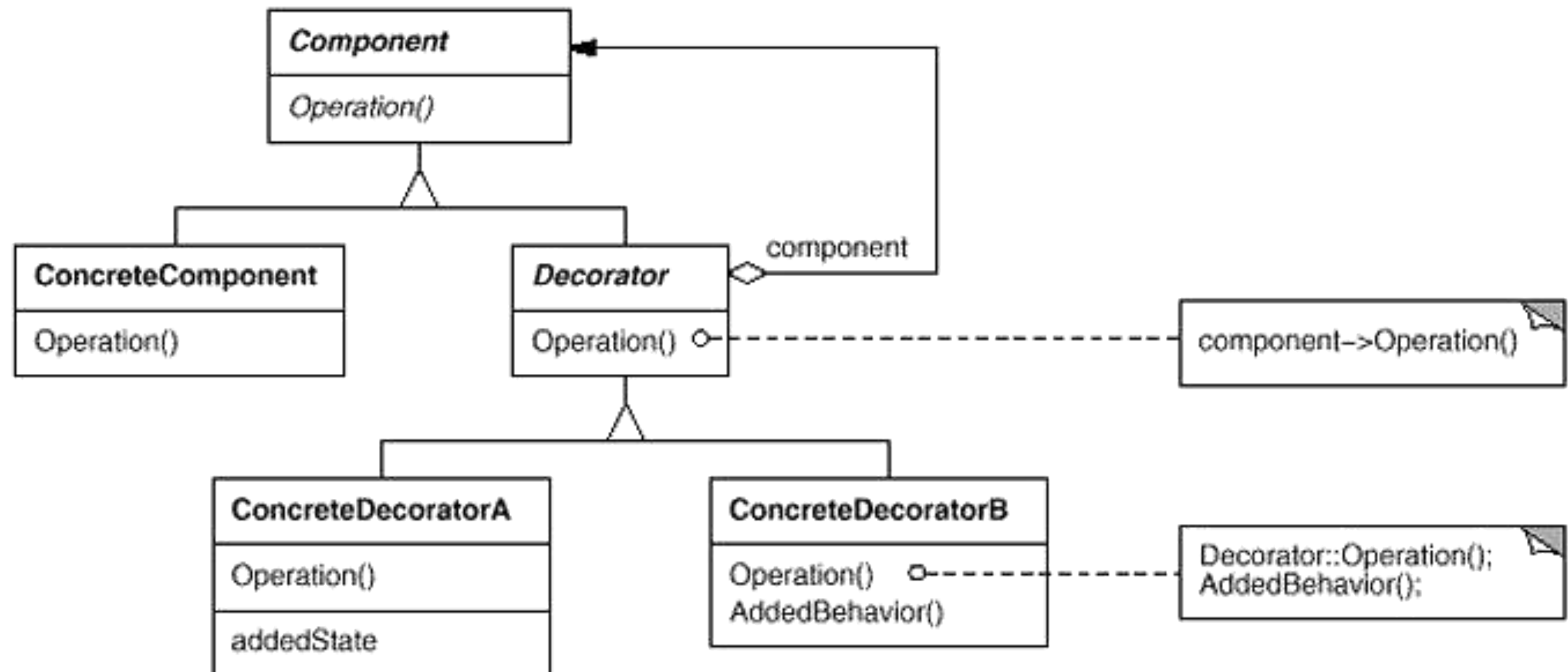
Quando usare Decorator:

- Si vuole aggiungere responsabilità a singoli oggetti.
- Si vuole togliere responsabilità agli oggetti
- L’estensione attraverso la definizione di classi specifiche non è praticabile in quanto porterebbe ad un’esplosione del numero di sottoclassi.

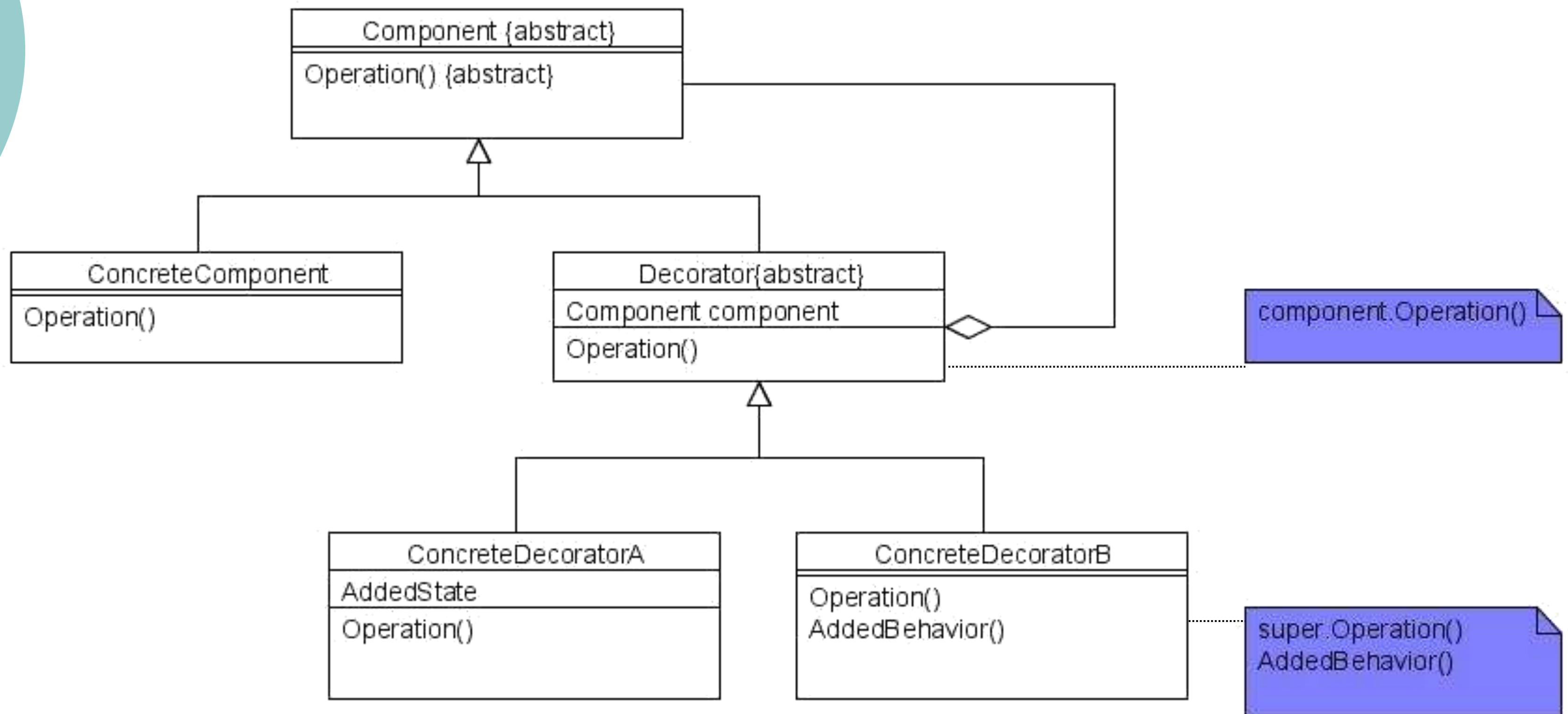
APPLICAZIONE DECORATOR



STRUTTURA: Diagramma OMT



STRUTTURA: Diagramma UML



PARTECIPANTI

- **Component:** definisce l’interfaccia comune degli oggetti decorati e non
- **ConcreteComponent:** definisce l’oggetto da decorare
- **Decorator:** mantiene un riferimento all’oggetto Component e definisce un’interfaccia conforme ad esso
- **ConcreteDecorator:** aggiunge responsabilità

PRO e CONTRO

PRO	CONTRO
○Maggiore flessibilità rispetto all'utilizzo dell'ereditarietà statica: Decorator permette di aggiungere responsabilità in modo dinamico (cioè in esecuzione) semplicemente collegando o scollegando i decoratori. Consente di "combinare" le responsabilità in maniera desiderata (si può anche applicare la stessa proprietà 2 volte. Es: doppio bordo). ○Evita di definire classi troppo complesse nella gerarchia: Decorator adotta un approccio di tipo "pagamento a consumo"; invece di supportare tutte le possibili caratteristiche in una classe complessa, le funzionalità vengono aggiunte partendo da elementi semplici e aggiungendo in maniera incrementale gli oggetti decoratori desiderati. In questo modo un'applicazione "paga solamente per le caratteristiche effettivamente utilizzate. Inoltre diventa più semplice aggiungere e togliere funzionalità a un oggetto decorato.	○Un decoratore e il suo componente NON sono IDENTICI: Un Decorator si comporta come un allegato trasparente, ma dal punto di vista dell'identità componente e decoratore non sono uguali. Non si dovrebbero avere dipendenze dall'identità di un oggetto. ○Moltitudine di piccoli oggetti: L'utilizzo di pattern Decorator porta spesso alla creazione di sistemi composti da molti piccoli oggetti. Questi oggetti si differenziano gli uni dagli altri solo per le reciproche interconnessioni. Il sistema risultante è pertanto semplice da personalizzare da parte di chi lo ha progettato ma risulta difficile da studiare e correggere da parte di esterni.

IMPLEMENTAZIONE:
ASPETTI DA CONSIDERARE DURANTE LA PROGETTAZIONE

1. **Conformità delle interfacce:** l'interfaccia del decoratore deve essere conforme a quella dell'oggetto decorato; le classi ConcreteDecorator devono quindi ereditare da una superclasse comune.

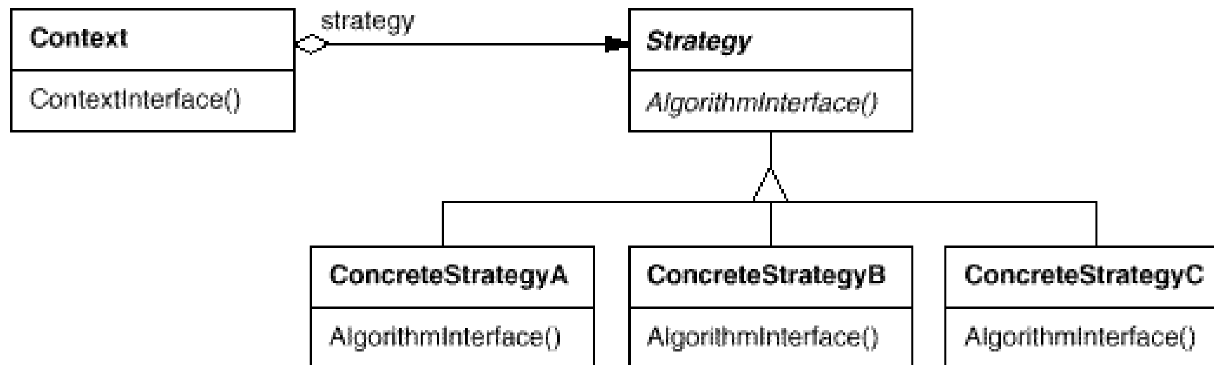
2. **Omissione classe astratta Decorator:** se esiste un solo tipo di decorazione o se stiamo lavorando su una gerarchia di classi preesistente è possibile omettere la classe astratta Decorator e semplificare il pattern; in questo caso le responsabilità di trasferimento delle richieste da Decorator a Component viene spostato nella classe ConcreteDecorator.

IMPLEMENTAZIONE:
ASPETTI DA CONSIDERARE DURANTE LA PROGETTAZIONE

3. **Classi Component leggere:** per semplificare la conformità ConcreteComponent-Decorator si fanno discendere entrambe le classi da una classe comune Component. È importante mantenere il più possibile leggera questa classe, focalizzandosi sulla definizione dell'interfaccia e rinviando la memorizzazione dei dati alle sottoclassi; in questo modo si evita di appesantire i componenti Decorator con funzionalità non necessarie.
4. **Cambiamento di "pelle" vs Cambiamento di "organi interni":** un decoratore può essere visto come un "rivestimento" che viene applicato su un oggetto in grado di modificarne il comportamento. Un'alternativa è fornita dal pattern Strategy: in questo caso il cambiamento avviene modificando gli "organi interni" (funzionalità interne dell'oggetto).

ESEMPIO PATTERN STRATEGY

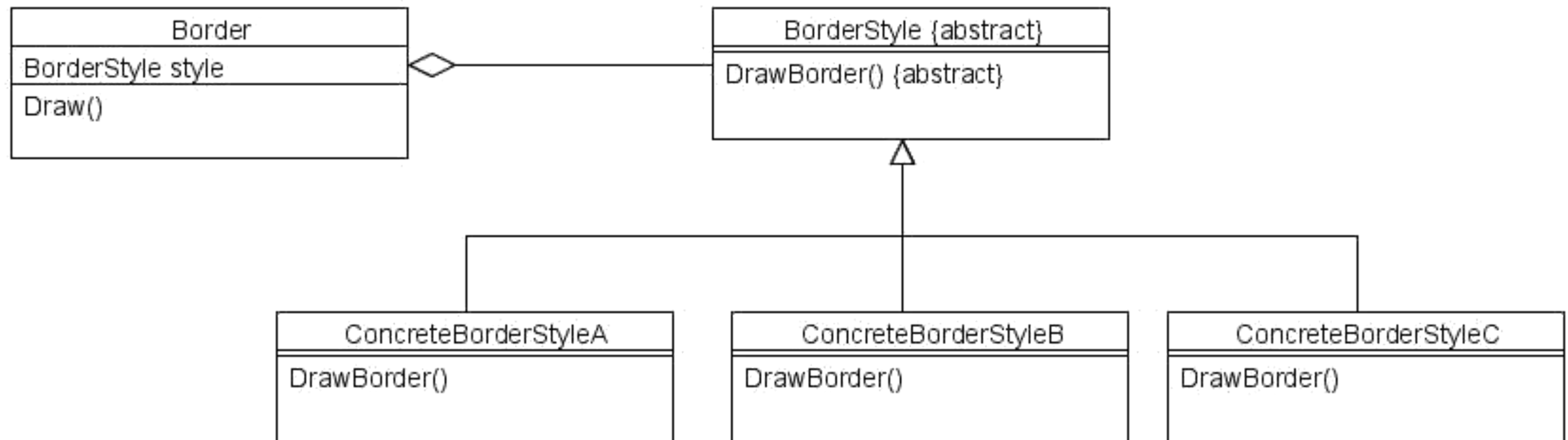
Nel Pattern Strategy il componente trasferisce parte del suo comportamento a un oggetto Strategy separato.



- L'obiettivo di questa architettura è **isolare** un algoritmo all'interno di un oggetto.
- Il pattern *Strategy* è utile in quelle situazioni dove è necessario <u>modificare dinamicamente</u> gli algoritmi utilizzati da un'applicazione.

ESEMPIO PATTERN STRATEGY

Il pattern Strategy permette di alterare o estendere le funzionalità di un oggetto modificando l'oggetto Strategy associato.

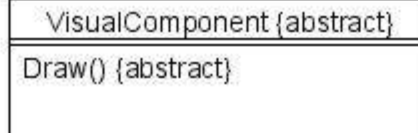


Borderstyle è un oggetto Strategy che incapsula una strategia per il disegno del bordo. Incrementando il numero di strategie implementate con oggetti esterni è possibile ottenere un effetto simile all'annidamento ricorsivo degli oggetti Decorator.

PATTERN DECORATOR:
PASSI PER LA CREAZIONE DEL CODICE

1. **Definizione dell'interfaccia comune**

```
abstract class VisualComponent { …

                                        void Draw(); }
```



PATTERN DECORATOR:

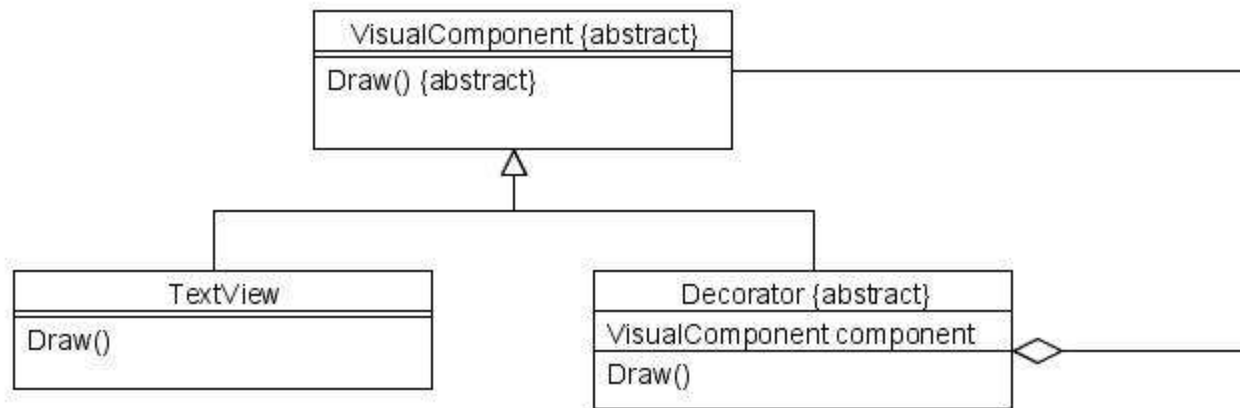
PASSI PER LA CREAZIONE DEL CODICE

2. Creare il secondo livello di classi “Core”(TextView) e Decorator, entrambe in relazione IS-A con l’interfaccia VisualComponent. Decorator presenta una relazione HAS-A con VisualComponent.

```
class TextView extends VisualComponent {
  private typeAtt attribute1,attribute2;


  public TextView(typeAtt a1, typeAtt a2) {
                                attribute1=a1;
                                attribute2=a2
                                }
  public void Draw() {…}
  }
```

```
abstract class Decorator extends VisualComponent {
  private VisualComponent component;
  public Decorator( VisualComponent c ) { component = c; }

  public void Draw() { component.Draw(); }
}
```

PATTERN DECORATOR:
PASSI PER LA CREAZIONE DEL CODICE



PATTERN DECORATOR:

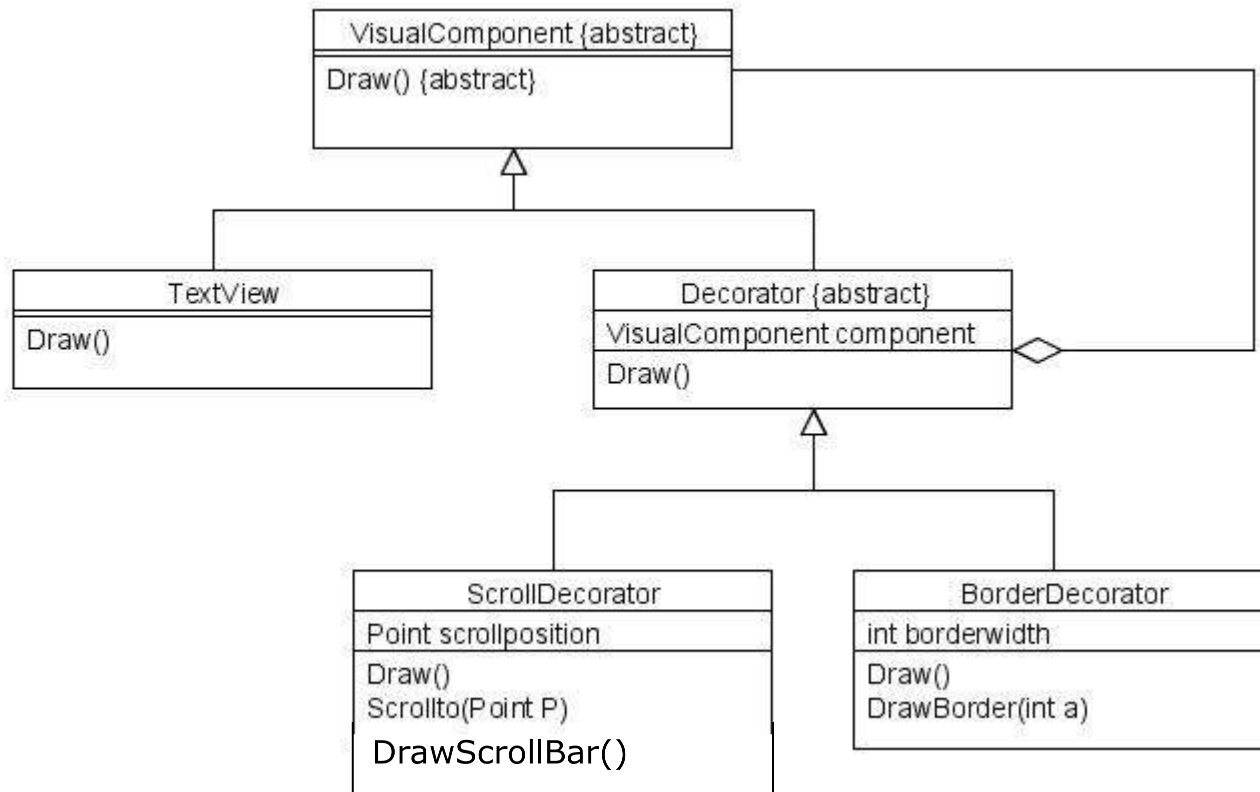
PASSI PER LA CREAZIONE DEL CODICE

3. Creazione dei ConcreteDecorator (BorderDecorator, ScrollDecorator) sottoclassi di Decorator.

```
class BorderDecorator extends Decorator {
  private int borderwidth;
  public BorderDecorator( VisualComponent c, int w ) {  super( c );
                                                         borderwidth = w; }
  public void Draw() {   super.Draw();
                         DrawBorder(borderwidth);  }
  private void DrawBorder(int a) {…}
}
```

```
class ScrollDecorator extends Decorator {
  private Point scrollPosition;
  public ScrollDecorator( VisualComponent c, Point p ) {   super( c ); scrollPosition = p;}

  public void Draw() {super.Draw(); DrawScrollBar();}
  private DrawScrollBar(){…}
  private void ScrollTo(Point p) {…}
}
```

PATTERN DECORATOR:
PASSI PER LA CREAZIONE DEL CODICE



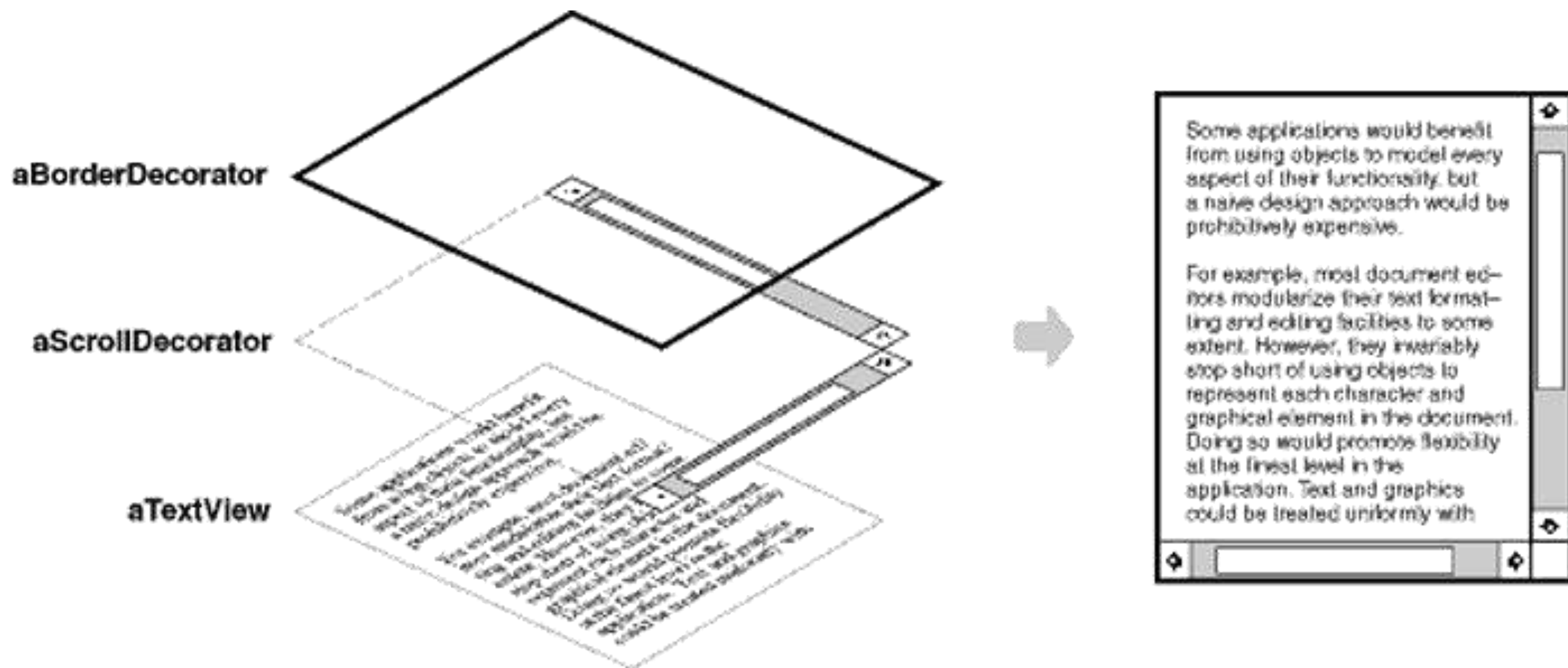
PATTERN DECORATOR:
PASSI PER LA CREAZIONE DEL CODICE

4. Il client può ora realizzare le composizioni desiderate.

```
public class DecoratorDemo {

  public static void main( String[] args ) {
     …

     VisualComponent componentеProva =
                new BorderDecorator(
                           new ScrollDecorator(new TextView( att1, att2 ), PointProva) ,
                            widthProva);

     componenteProva.Draw();

}
}
```

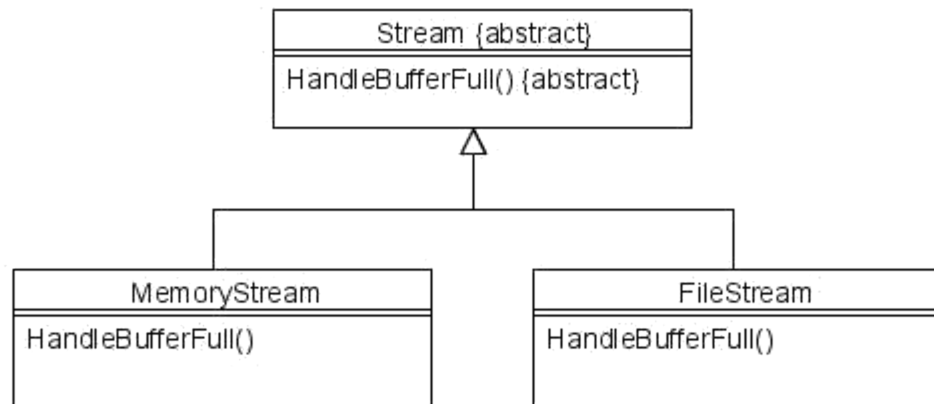
PATTERN DECORATOR:
PASSI PER LA CREAZIONE DEL CODICE

Risultato:



UTILIZZI NOTI: STREAM

- Gli stream sono astrazioni fondamentali per capire correttamente i processi di I/O; uno stream fornisce un'interfaccia in grado di convertire un oggetto in una sequenza di byte o caratteri, rendendo così possibile il salvataggio/caricamento degli oggetti in/da file. Una semplice implementazione di ciò potrebbe essere la seguente:



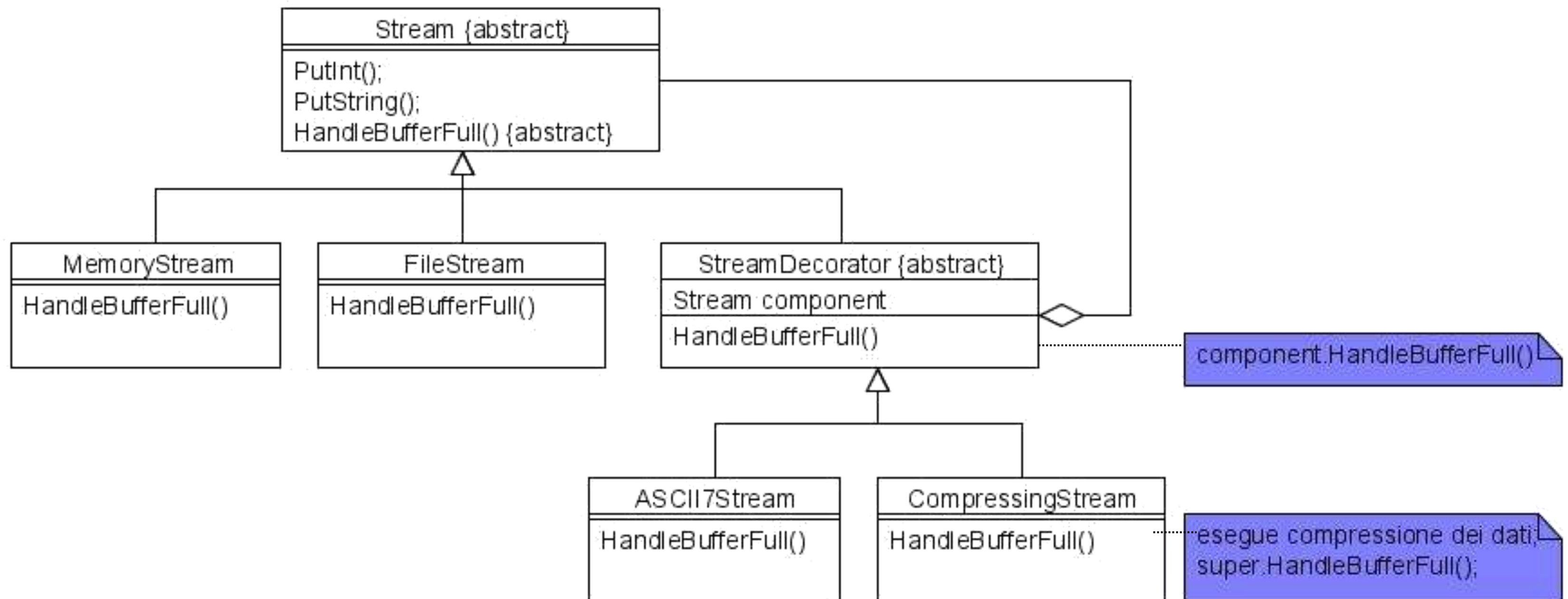
UTILIZZI NOTI: STREAM

Ma supponiamo di voler poter gestire anche:

- Compressione dello stream di dati con diversi algoritmi
- Convertire lo stream dei dati utilizzando una codifica a 7-bit ASCII, in modo da poter inviare il flusso attraverso canali di comunicazione ASCII

Il pattern Decorator fornisce una soluzione veloce:

UTILIZZI NOTI: STREAM



- La classe astratta Stream mantiene un Buffer interno e fornisce le operazioni di immissione dei dati nello stream (PutInt(), PutString()) e la funzione astratta per la gestione del buffer pieno HandleBufferFull().
- La versione della funzione presente in FileStream sovrascrive la funzione astratta e trasferisce il buffer all'interno di un file.
- La classe StreamDecorator mantiene riferimento a un oggetto Stream e inoltra le richieste a esso.
- Le sottoclassi di StreamDecorator sovrascrivono il metodo HandleBufferFull e svolgono altre operazioni prima di chiamare HandleBufferFull della classe genitrice.

UTILIZZI NOTI: STREAM

Creare un FileStream che comprime i dati binari e quindi li converte in codifica 7-bit ASCII:

```
Stream prova1 = new CompressingStream(
                                            new ASCII7Stream(
                                                           new FileStream(“FileName”)
                                            )
                                 );
prova1.PutInt(12);
prova1.PutString(“stringa”);
```

Bibliografia

- E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design patterns - Elements of reusable object oriented software, Addison-Wesley, 1995
- http://sourcemaking.com/design_patterns/decorator/java/3
- Steven John Metsker, Design Patterns Java Workbook, Addison Wesley, 2002
- Elaborato C. Tosoni A.A. 2007/2008