Object-Oriented Design Pattern:

Decorator

Corso di Laurea Specialistica in Ingegneria Informatica
Insegnamento di Ingegneria del Software B
a.a. 2007/2008

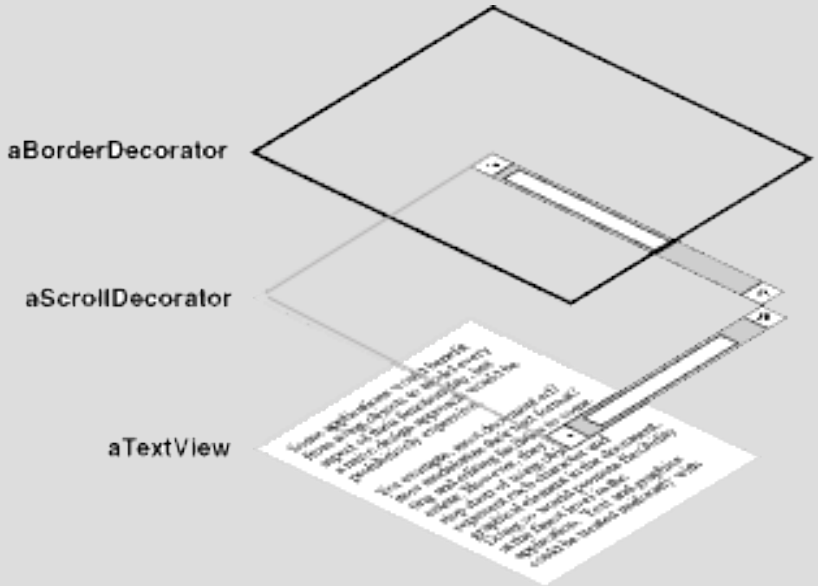
Tosoni Cristian mat. 65701

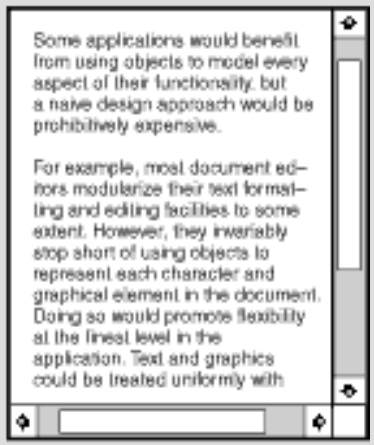
Decorator è:

- Un pattern Strutturale
  - Si concentra su come è possibile comporre classi e oggetti per creare grandi strutture con funzionalità complesse.
- Un pattern basato sugli oggetti
  - Sfrutta le potenzialità della delega per superare i limiti dell'ereditarietà.

Decorator

- Descrive come aggiungere dinamicamente funzionalità ad un oggetto.
- Compone gli oggetti in modo ricorsivo per permettere di aggiungere ad un oggetto un numero variabile di responsabilità
- Ad esempio, un oggetto Decorator che contiene un componente di un interfaccia utente può aggiungere ad esso bordi, ombreggiatura e barre di scorrimento.





Wellcome to Starbuzz Coffee

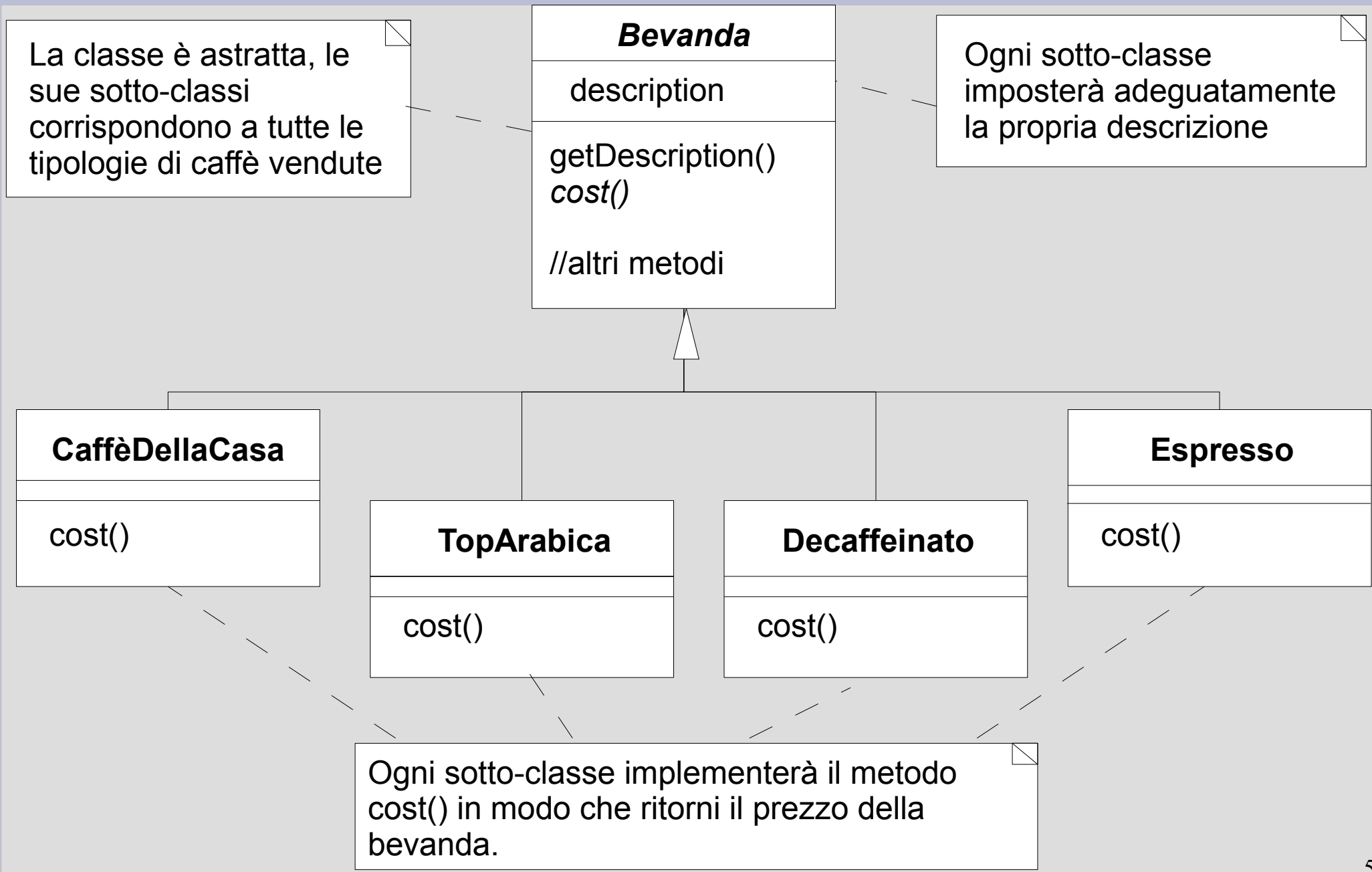
A Starbuzz Coffee è stato attribuito il titolo di “negozio di caffè con la maggior velocità di diffusione”.

“Vedi uno Starbuzz Coffee? Guarda dall'altro lato della strada, ne potrai vedere un altro!”

Il motivo della sua veloce diffusione consiste nello sforzo che viene continuamente fatto per aggiornare il proprio menù in modo che soddisfi ogni tipologia di cliente.

Quando il primo negozio venne inaugurato il diagramma delle classi dei loro prodotti era il seguente...

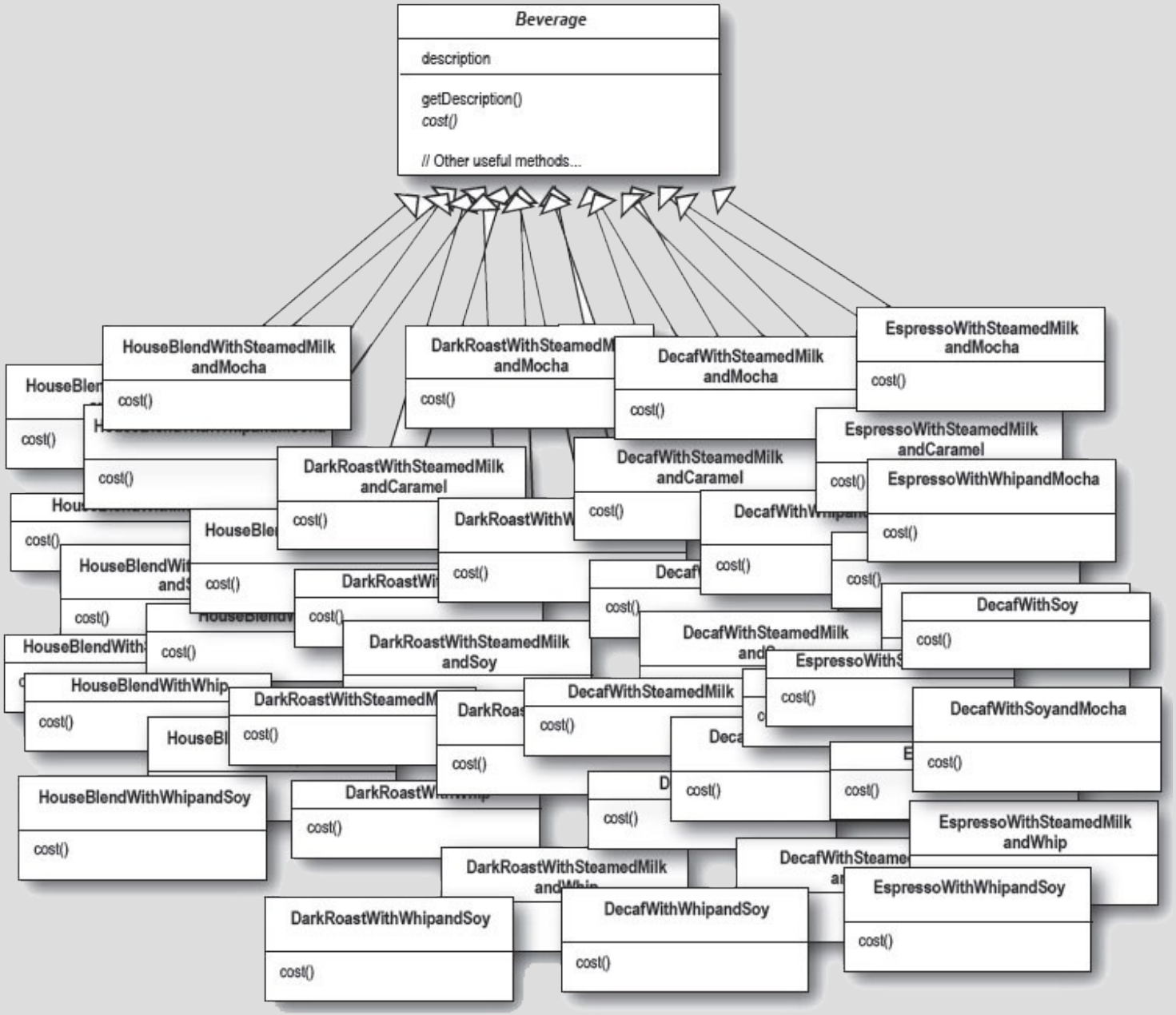
Starbuzz Coffee



Dopo un mese dall'apertura...

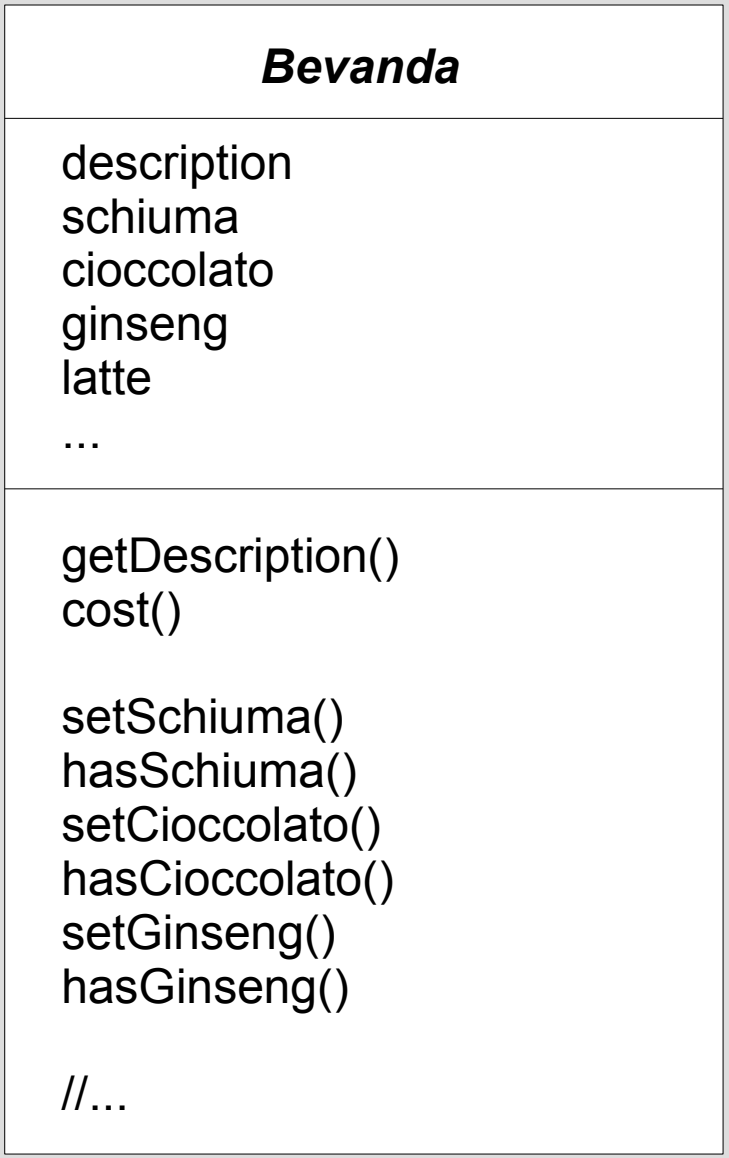
In aggiunta al tuo caffè puoi anche chiedere una o più variazioni alla ricetta base, come l'aggiunta di schiuma, cioccolato, ginseng, la tazza grande, ed ovviamente anche di latte.

“Macchiato caldo o freddo?”



Una possibile soluzione

- Perché costruire tutte quelle classi?
- Potremmo inserire nella classe base alcune variabili per tenere traccia della presenza o meno di ogni correzione.

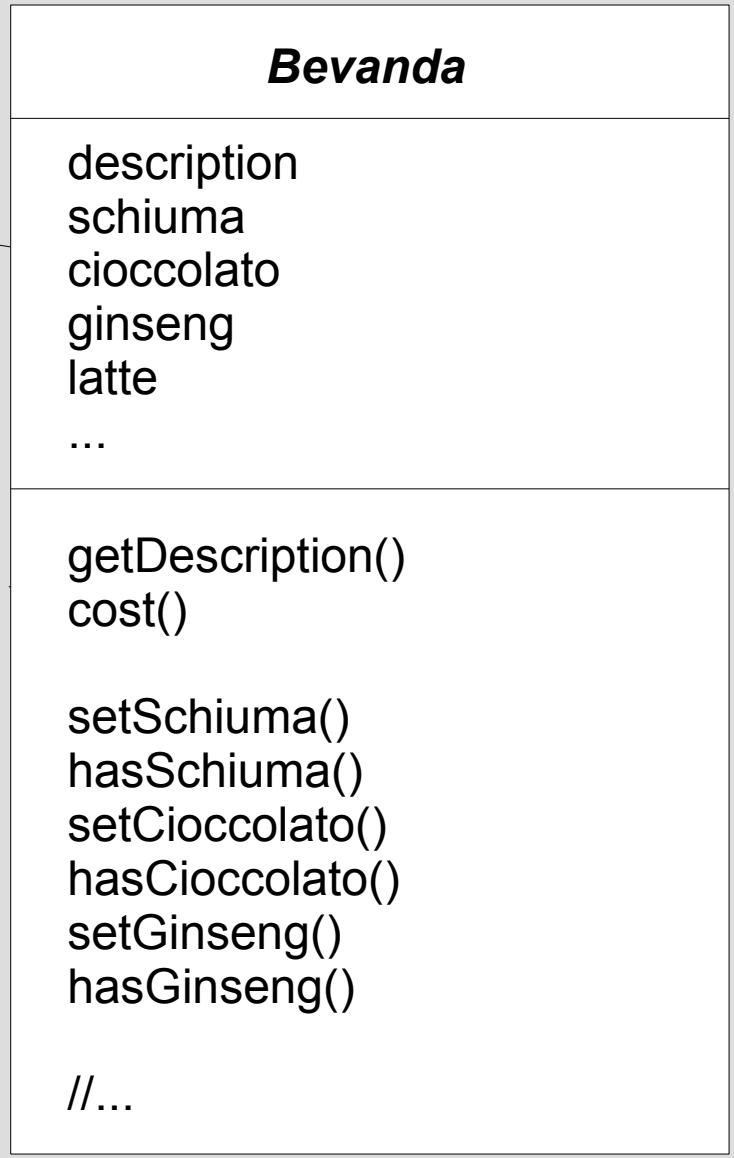


La nuova classe *Bevanda*

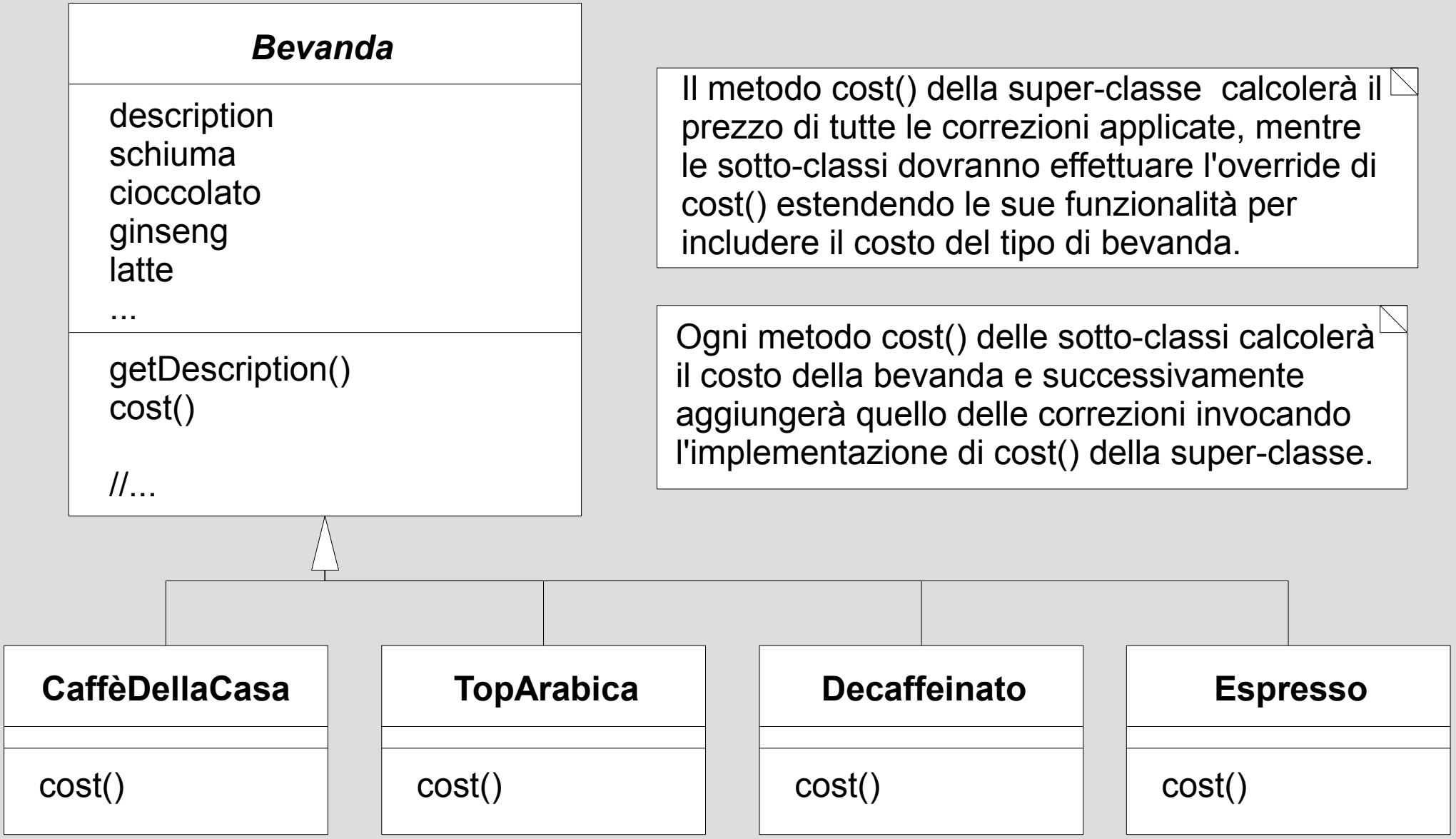
Nuove variabili booleane, una per ogni possibile correzione.

Ora sarà necessario creare un'implementazione di cost(), che non sarà più un metodo astratto. In modo che possa calcolare il costo del caffè tenendo conto delle correzioni.

I nuovi metodi serviranno per accedere in lettura o in scrittura alle variabili private della classe indicando quali correzioni sono state apportate.



La nuova gerarchia



Risultati

- Bene, il problema sembra essere risolto! Ora abbiamo un totale di 5 classi contro le innumerevoli di prima.
- É possibile ereditare funzionalità dalle super-classi .
  - Tutte le sotto-classi ereditano le stesse funzionalità, ma è possibile estenderle facendo l'override dei metodi e richiamando al loro interno il metodo originale della super-classe.
  - Si può variare dinamicamente la funzionalità, in correlazione con la sotto-classe istanziata.

Problematiche (1)

- Abbiamo veramente trovato una soluzione valida?
- Sono immaginabili alcune potenziali complicazioni utilizzando questo approccio, basta pensare a come si dovrà operare a fronte di modifiche al menù dello Starbuzz Coffee.

Problematiche (2)

- Un cambiamento nel prezzo delle correzioni ci obbliga a modificare il codice esistente.
- L'inserimento di nuove correzioni ci obbliga ad aggiungere nuovi attributi, nuove funzioni e a modificare il metodo cost() della super-classe.

Problematiche (3)

- Potrebbero essere inserite nel menù nuove tipologie di bevande (ad esempio il tè freddo), le correzioni non sono più adatte per tutte le bevande, ma la nuova sotto-classe erediterà comunque metodi come hasSchiuma()!
- E se il cliente vuole il doppio ginseng?

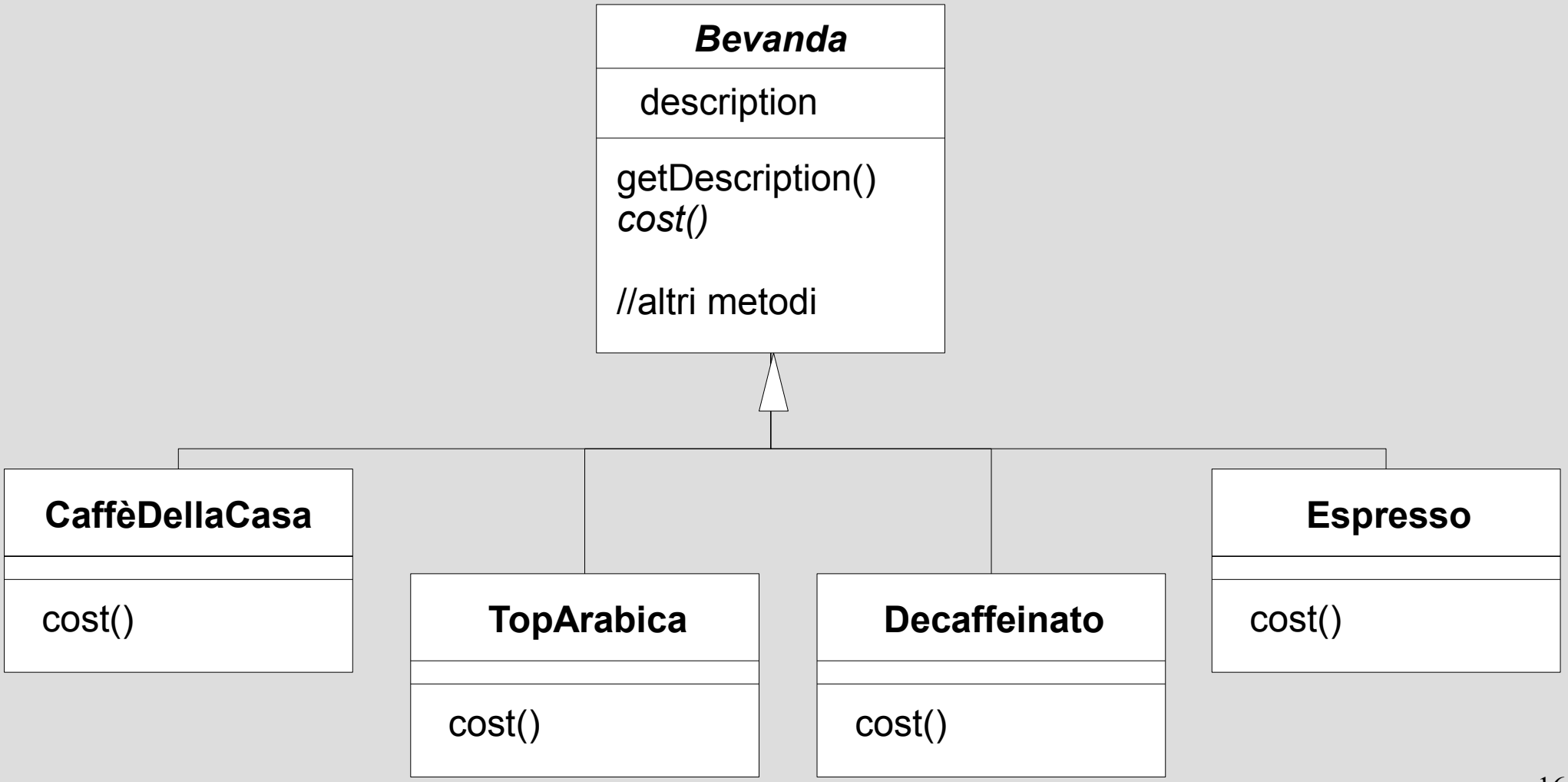
Conosciamo il Pattern Decorator

- Il punto di partenza è un'istanza di bevanda, quindi la “decoriamo” con le correzioni richieste dinamicamente.
- Per esempio: un cliente ordina un caffè TopArabica con cioccolato e panna montata.

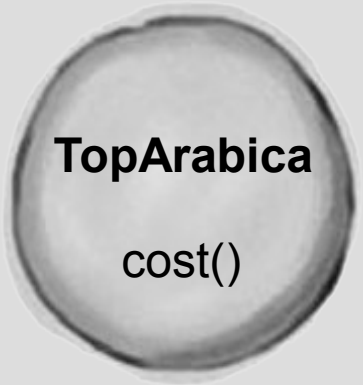
1. Realizziamo il caffè TopArabica
2. Lo decoriamo con la correzione al cioccolato
3. Lo decoriamo con la panna montata
4. Utilizziamo la funzione cost() sull'oggetto risultante, affidando alla delega l'aggiunta dei costi delle correzioni.

Il Pattern Decorator

- Riprendiamo la prima gerarchia delle classi dei prodotti di Starbuzz Coffee.

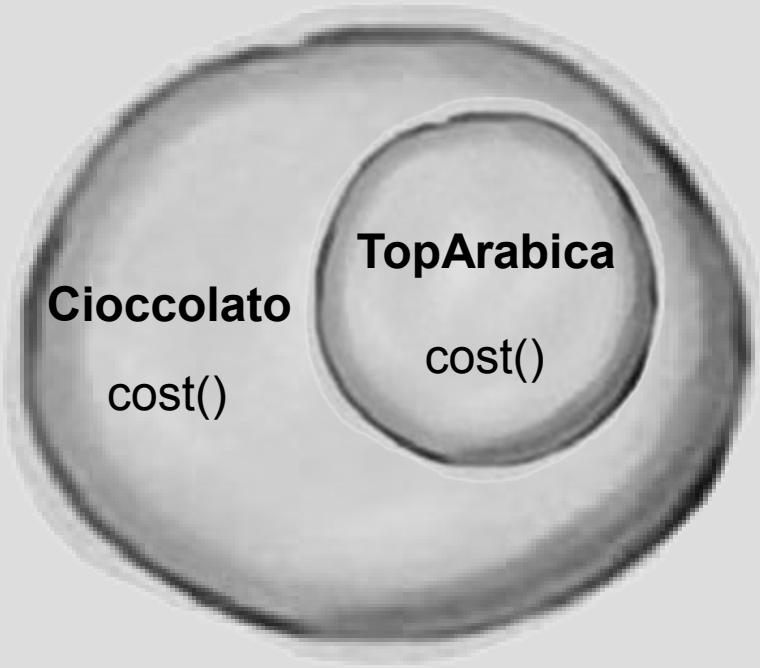


1. Creare il caffè TopArabica



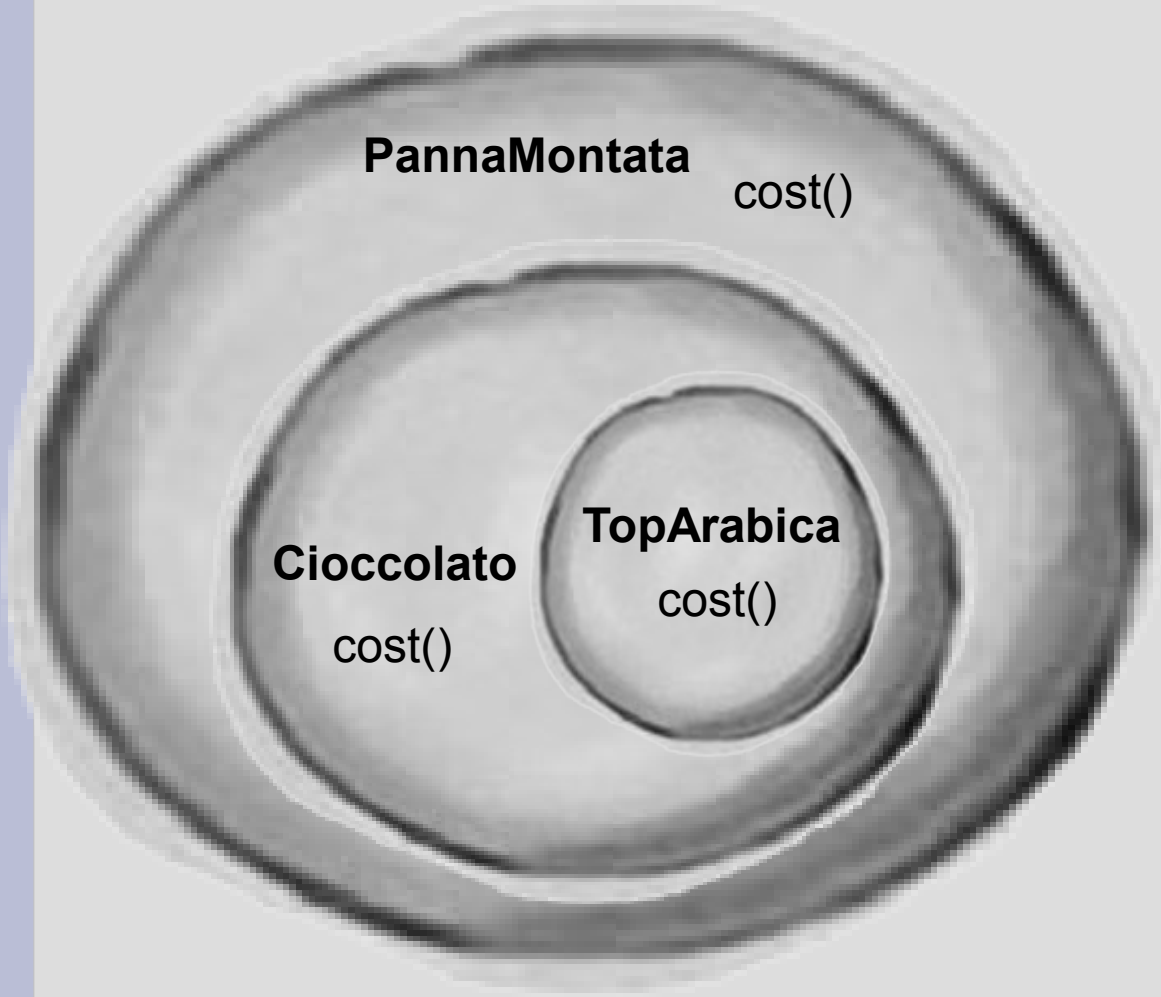
•L'istanza di **TopArabica** eredita da ***Bevanda*** e possiede un metodo cost() che calcola il prezzo della bevanda.

2. Il cliente vuole il cioccolato



- Creiamo un oggetto **Cioccolato** e lo avvolgiamo attorno a **TopArabica**.
- **Cioccolato** è un oggetto decorativo.
- Anche **Cioccolato** possiede un metodo cost() e quindi possiamo trattarlo come un caso particolare di bevanda.

3. Il cliente vuole la panna montata



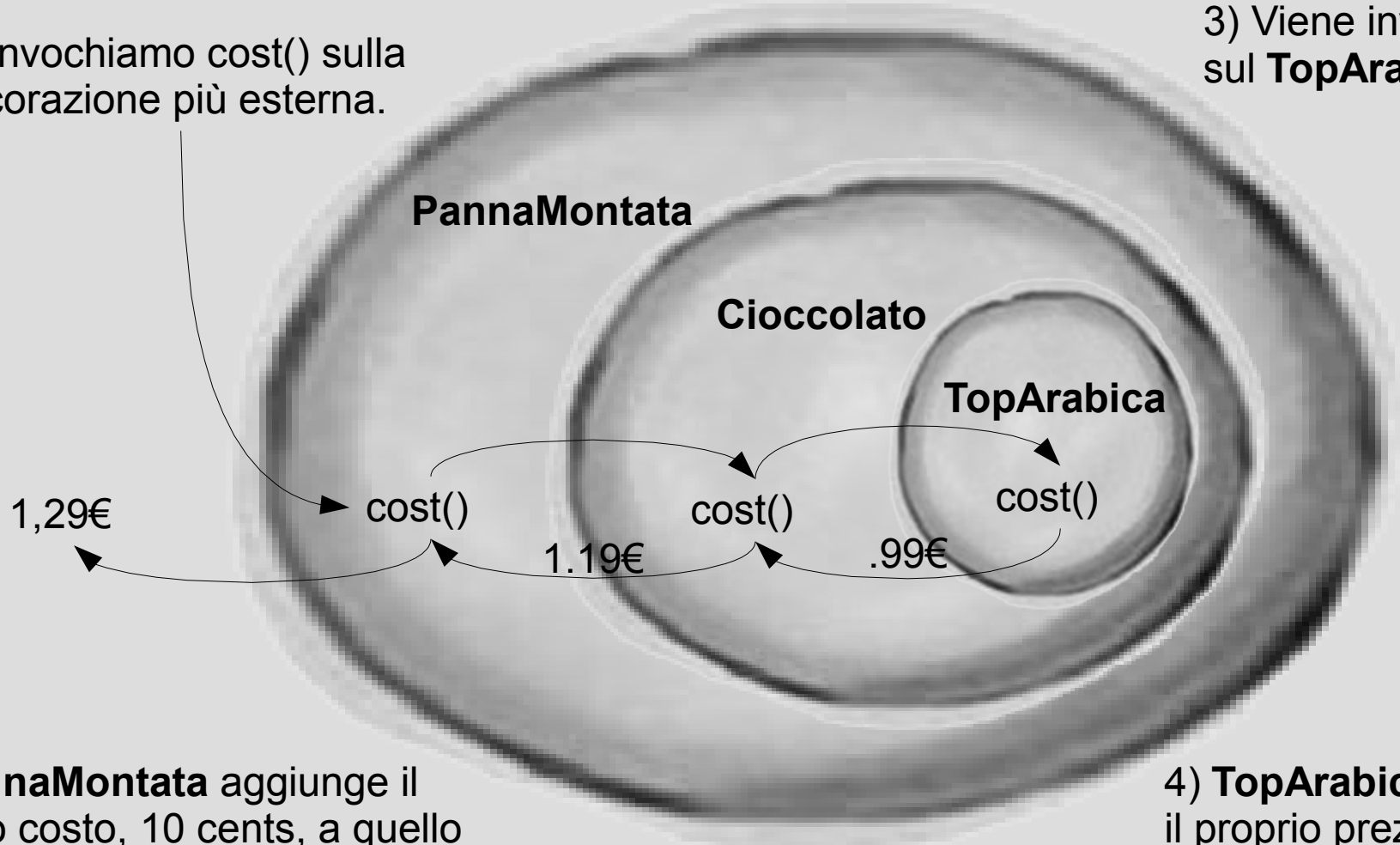
- Creiamo un oggetto **PannaMontata** e lo avvolgiamo attorno a **Cioccolato**.
- Anche **PannaMontata** è un oggetto decorativo.

4. Calcoliamo il prezzo

2) Viene invocato cost() sul **Cioccolato**.

3) Viene invocato cost() sul **TopArabica**.

1) Invochiamo cost() sulla decorazione più esterna.

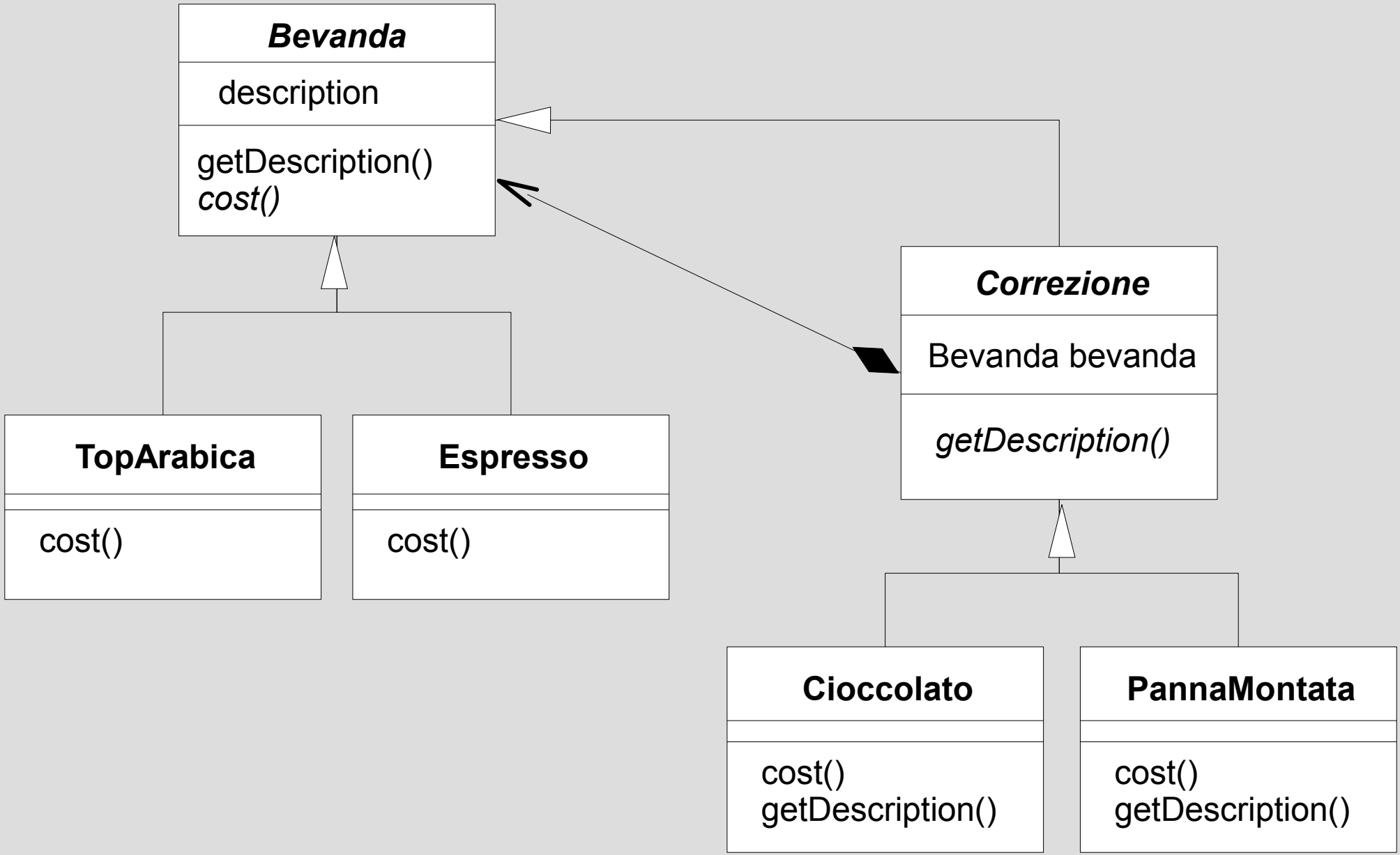


6) **PannaMontata** aggiunge il proprio costo, 10 cents, a quello restituito da **Cioccolato** e restituisce un totale di 1,29€.

5) **Cioccolato** aggiunge il proprio costo, 20 cents, a quello di **TopArabica** e restituisce 1,19€.

4) **TopArabica** restituisce il proprio prezzo, 99 cents.

La gerarchia delle Classi



Conseguenze

- É importante che una correzione sia dello stesso tipo dell'oggetto che vuole decorare. In modo da poter trattare entrambi attraverso le stesse modalità.
- Attraverso l'ereditarietà si garantisce il type matching, mentre attraverso la composizione si acquistano nuove funzionalità.
- Attraverso questo pattern si aumenta la flessibilità della struttura e si possono abbinare facilmente bevande e correzioni.

Il codice di Starbuzz (1)

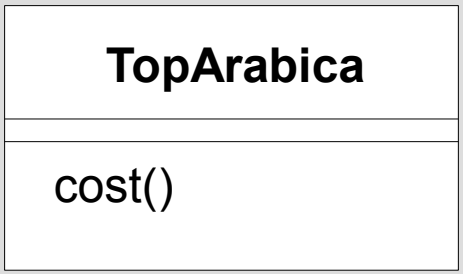
Bevanda
description
getDescription() *cost()*

```
public abstract class Bevanda {
        String description = “Bevanda Generica”;

        public String getDescription() {
                return description;
        }

        public abstract double cost();
}
```

Correzione
Bevanda bevanda
getDescription()

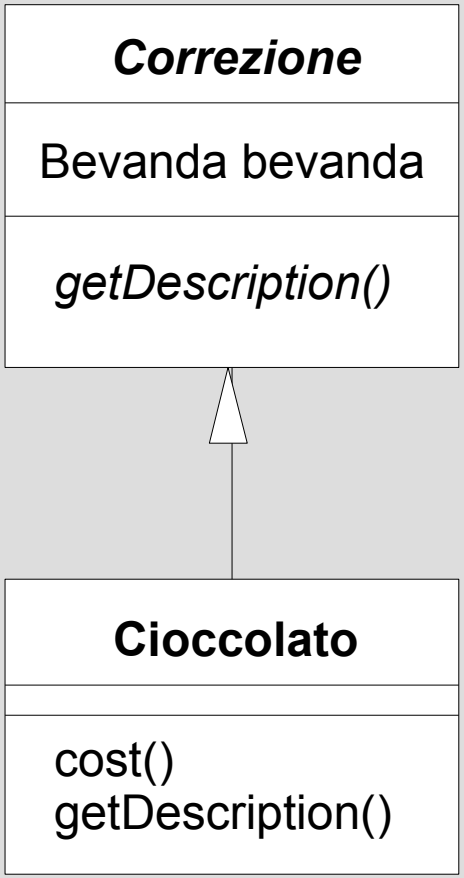
```
public abstract class Correzione extends Bevanda {

        Bevanda bevanda;

        public String getDescription() {
                return bevanda.getDescription() + description;
        }
}
```

Il codice di Starbuzz (2)




```
public class Espresso extends Bevanda {

    public Espresso() {
        description = “Espresso”;
    }

    public double cost() {
        return 1.99;
    }
}
public class TopArabica extends Bevanda {

    public TopArabica() {
        description = “TopArabica”;
    }

    public double cost() {
        return .89;
    }
}
```

Il codice di Starbuzz (3)




```
public class Cioccolato extends Correzione {

    public Cioccolato(Bevanda b) {
        bevanda = b;
        description = “, Cioccolato”;
    }

    public double cost() {
        return .20 + bevanda.cost();
    }
}
```

Serviamo del caffè!

```
public class StarbuzzCoffee {

    public static void main(String args[]) {

        Bevanda bevanda = new Espresso();
        System.out.println(bevanda.getDescription()+ “, € ” +bevanda.cost());

        Bevanda bevanda2 = new TopArabica();
        bevanda2 = new Cioccolato(bevanda2);
        bevanda2 = new Cioccolato(bevanda2);
        bevanda2 = new PannaMontata(bevanda2);
        System.out.println(bevanda2.getDescription() + “, € ” + bevanda2.cost());

        Bevanda bevanda3 = new Decaffeinato();
        bevanda3 = new Ginseng(bevanda3);
        bevanda3 = new Cioccolato(bevanda3);
        bevanda3 = new PannaMontata(bevanda3);
        System.out.println(bevanda3.getDescription() + “, € ” + bevanda3.cost());
    }
}
```

Risultato del test

```
% Java StarbuzzCoffee

Espresso, € 1.99
TopArabica, Cioccolato, Cioccolato, PannaMontata, € 1.49
Decaffeinato, Ginseng, Cioccolato, PannaMontata, € 1.34

%
```

Pattern Decorator

Scopo:

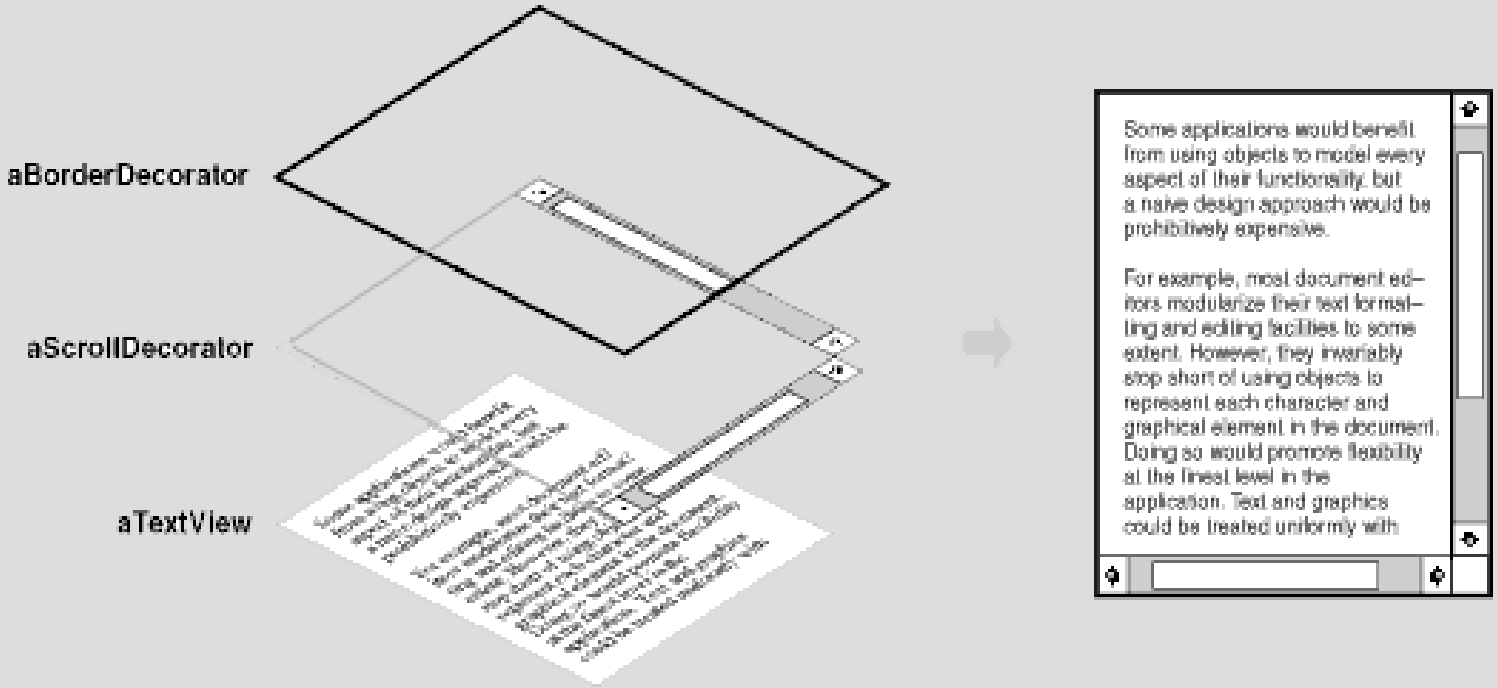
- Aggiungere dinamicamente responsabilità ad un oggetto.
- Alternativa flessibile all'utilizzo di sottoclassi come strumento per l'estensione delle funzionalità.

Detto anche:

- Wrapper: da “to wrap”: avvolgere, imballare.

Motivazioni (1):

- Alcune volte si vuole aggiungere funzionalità a un particolare oggetto, ma non all'intera classe a cui appartiene. Un toolkit di oggetti per generare un'interfaccia grafica dovrebbe permettere di aggiungere proprietà come i bordi e funzionalità come lo scrolling a qualsiasi componente grafico.
- Un metodo per aggiungere funzionalità è utilizzare l'ereditarietà. Ereditare un bordo da un'altra classe implica la presenza di un bordo attorno ad ogni sotto-classe.



Motivazioni (2):

•Un approccio più flessibile consiste nel racchiudere il componente in un altro oggetto che aggiungerà il bordo.

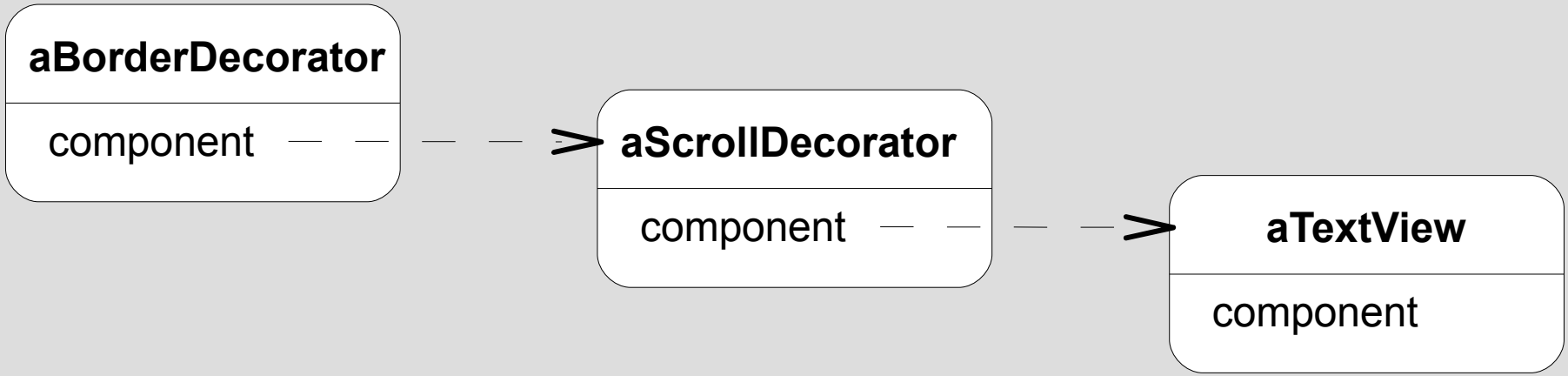
•L'oggetto in questione è chiamato **decorator.** Il decorator è conforme all'interfaccia dell'oggetto che contiene, in questo modo risulta trasparente rispetto al client.

•Il decorator redirige le richieste verso il suo contenuto e, prima o dopo il passaggio, può aggiungere funzionalità accessorie (come disegnare il bordo).

•La trasparenza permette inoltre di utilizzare questa tecnica ricorsivamente un numero illimitato di volte.

Motivazioni (3):

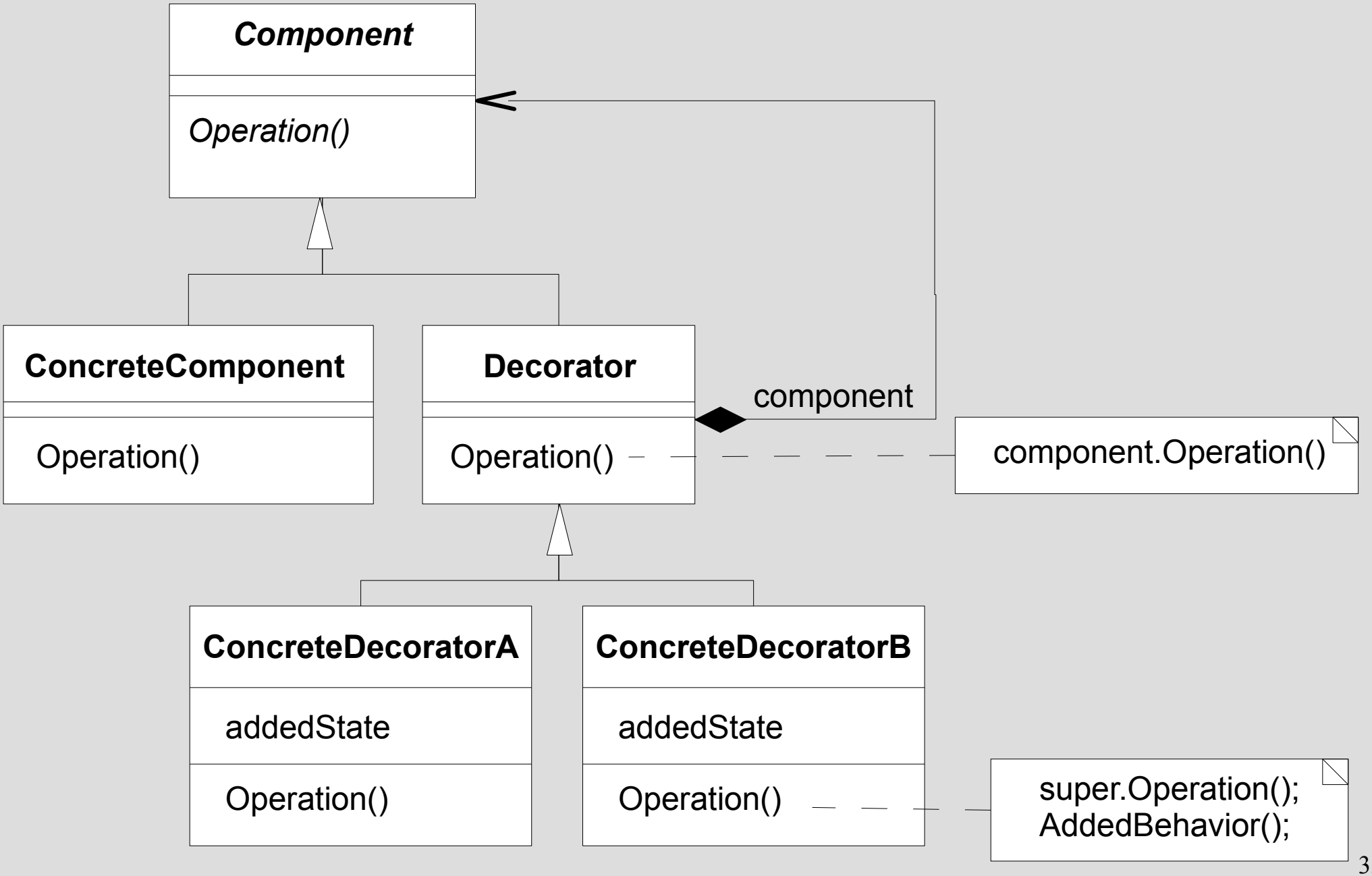
- Il seguente diagramma degli oggetti mostra come comporre una TextView con un BorderDecorator e uno ScrollDecorator per ottenere una TextView con funzionalità di scrolling e bordo.



Applicabilità:

- Si utilizza Decorator:
  - Per aggiungere responsabilità a singoli oggetti dinamicamente e in modo trasparente senza influenzare altri oggetti.
  - Per responsabilità che possono essere “ritirate”.
  - Quando l'estensione attraverso le sotto-classi non è effettuabile.
    - In alcuni casi è necessario un gran numero di estensioni che potrebbe produrre un esplosione del numero delle sotto-classi, per generare ogni tipologia di combinazione.
    - Alcune caratteristiche dovrebbero essere nascoste o non utilizzabili per alcune sotto-classi.

Struttura:


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component.Operation()
ConcreteDecoratorA
addedState
Operation()
ConcreteDecoratorB
addedState
Operation()
super.Operation();
AddedBehavior();

Partecipanti:

- Component (Bevanda):
  - Definisce l'interfaccia per quegli oggetti ai quali bisogna aggiungere dinamicamente responsabilità.
- ConcreteComponent (TopArabica):
  - Definisce un oggetto al quale è possibile aggiungere responsabilità.
- Decorator (Correzione):
  - Mantiene un riferimento ad un oggetto di tipo Component e definisce un'interfaccia conforme a quella di Component.
- ConcreteDecorator (Cioccolato):
  - Aggiunge responsabilità ad un ConcreteComponent.

Collaborazioni:

- Un Decorator trasferisce le richieste ricevute al suo oggetto Component.
- Può opzionalmente svolgere operazioni ulteriori prima e dopo il trasferimento della richiesta.

Conseguenze (1):

- Maggiore flessibilità rispetto all'ereditarietà statica:
  - Nuove responsabilità possono essere facilmente aggiunte e rimosse dinamicamente.
- Evita classi di grosse dimensioni nella parte alta della gerarchia:
  - Le funzionalità opzionali derivano dalla composizione di piccoli blocchi e come risultato un'applicazione non deve sovraccaricarsi per funzionalità che non utilizza.

Conseguenze (2):

- Un Decorator e un suo Componet non sono identici:
  - Un componente non si presenta in modo identico ad un componente decorato, quindi non sono intercambiabili in ogni situazione.
- Molti piccoli oggetti:
  - Un sistema, creato secondo il pattern Decorator, durante l'esecuzione presenterà molti piccoli oggetti, che si differenzieranno gli uni dagli altri solo per le reciproche interconnessioni.
  - Questi sistemi saranno personalizzabili in modo semplice da chi li conosce, ma sono difficilmente comprensibili o analizzabili dagli altri.

Implementazione (1):

- Conformità delle interfacce:
  - L'interfaccia di un oggetto decorazione deve essere conforme a quella dell'oggetto che deve decorare.
  - Le classi ConcreteDecorator devono derivare tutte dalla stessa classe genitrice.
- Omissione della classe astratta Decorator:
  - Se esiste un solo tipo di decorazione, oppure stiamo lavorando su una gerarchia di classi preesistente, è possibile omettere la classe astratta Decorator e semplificare il pattern.

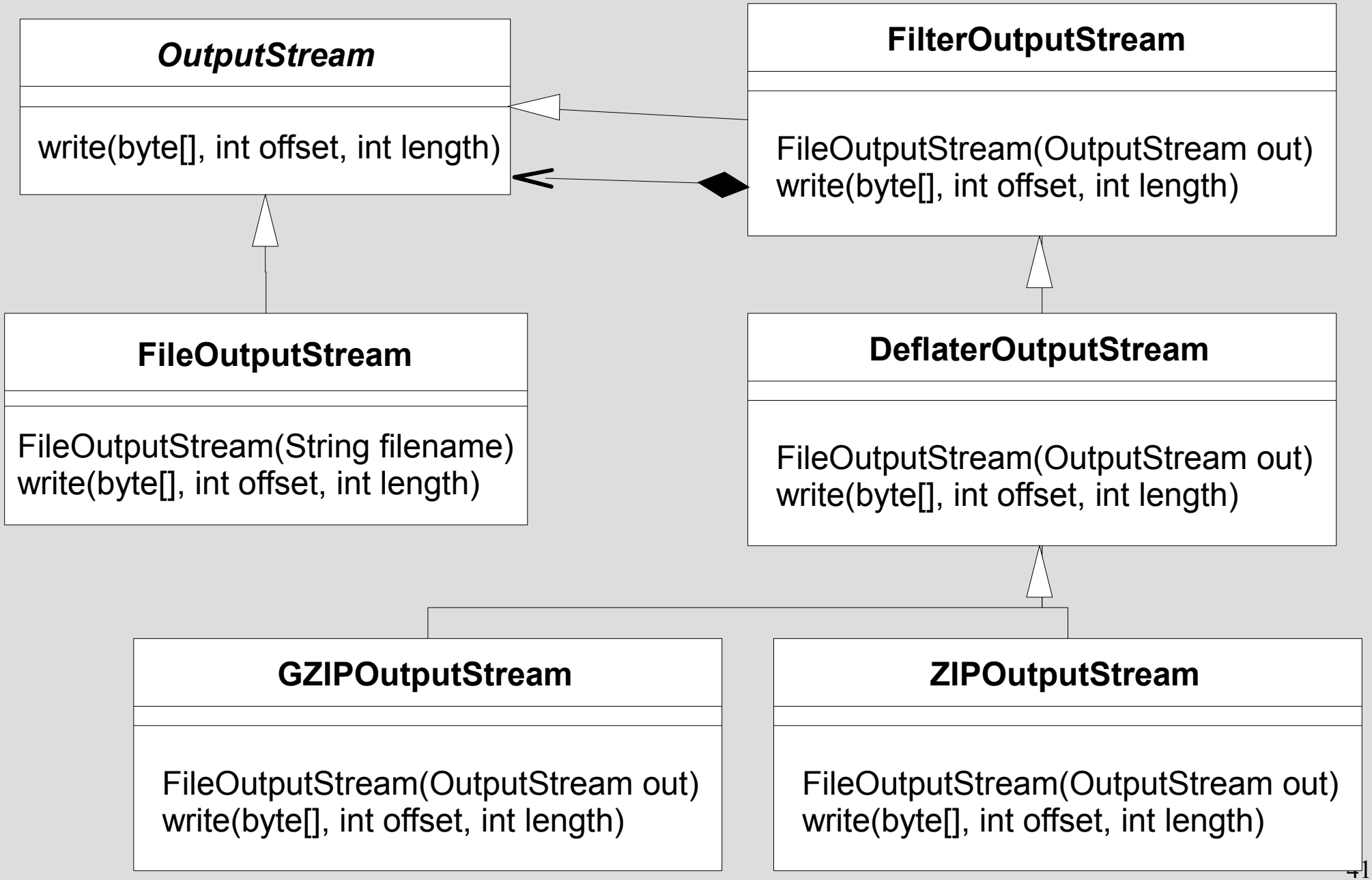
Implementazione (2):

- Mantenere la classe Component leggera:
  - È importante mantenere la classe comune Component leggera, dovrebbe servire solo per definire un'interfaccia.
  - Inserendo molte funzionalità nella classe Component aumenta la probabilità che le classi concrete ereditino funzioni di cui non hanno bisogno.
- Modificare la pelle di un oggetto oppure i suoi “organi interni”:
  - Il pattern Decoration fornisce un metodo per cambiare pelle ad un oggetto, o meglio per rivestirlo.
  - Il pattern Strategy modifica il comportamento di un oggetto andando a sostituire una sua parte interna che effettivamente implementa i diversi comportamenti.

Utilizzi noti:

- Un classico esempio di Decorator:
  - Gli Stream, ossia una collezione seriale di byte o caratteri.
  - In Java ogni oggetto Stream normalmente ne contiene un altro a cui reindirizza le sue operazioni.

La gerarchia OutputStream (1)



La gerarchia OutputStream (2)

- La classe GZIPOutputStream:
  - Implementa le sue operazioni write() comprimendo i byte ricevuti e scrivendoli sull'oggetto OutputStream richiesto dal costruttore.
  - Aggiunge la compressione al comportamento dell'oggetto OutputStream preso come parametro.
- Ogni costruttore di un oggetto OutputStream richiede un oggetto OutputStream:
  - È necessaria una classe che agisca come foglia, ad esempio FileOutputStream

La gerarchia OutputStream (3)

- La maggior parte delle classi stream Java appartengono al package java.io
  - DeflaterOutputStream, GZIPOutputStream e ZipOutputStream fanno parte del package java.util.zip.
    - ZipOutputStream permette di creare un file compresso che racchiude più di un file.
    - GZIPOutputStream comprime un singolo file in input.
- È quindi possibile avvolgere un oggetto GZIPOutputStream attorno ad uno FileOutputStream.

La gerarchia OutputStream (4)

```
import java.io.*;
import java.util.zip.*;
public class ShowGzip
{
      public static void main(String args[]) throws IOException
      {
            java.net.URL url = ClassLoader.getSystemResource(“demo.doc”);
            InputStream in = url.openStream();

            GZIPOutputStream out = new GZIPOutputStream(
                                                new FileOutputStream(url.getFile()+ “.gz”));

            byte[] data = new byte[100];
            while (true)
            {
                  int n = in.read(data);
                  if (n == -1) break;
                  out.write(data, 0, n);
            }

            out.close();
            in.close();
      }
}
```

Pattern Correlati:

- Adapter:
  - Con il pattern Decorator si modificano le responsabilità di un oggetto senza toccarne l'interfaccia, mentre con ilpattern Adapter gli si fornisce un'interfaccia completamente nuova.
- Composite:
  - Un decoratore può essere visto come un oggetto composito degenere, costituito da un singolo componente.
- Strategy:
  - Decorator consente di cambiare la “pelle” di un oggetto, mentre Strategy consente di cambiarne “gli organi interni”.

Riepilogo

- Quando un oggetto deve supportare una varietà di comportamenti che si desidera inoltre combinare tra loro, Decorator offre un'alternativa flessibile alla creazione di una nuova classe per ogni possibile combinazione di comportamenti.
- È possibile applicare Decorator nel proprio codice per costruire grandi gerarchie a partire da un insieme prefissato di classi.
- L'intento di Decorator è di permettere la definizione del comportamento di un oggetto attraverso la combinazione di classi cooperanti.

Fine.

Bibliografia

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