

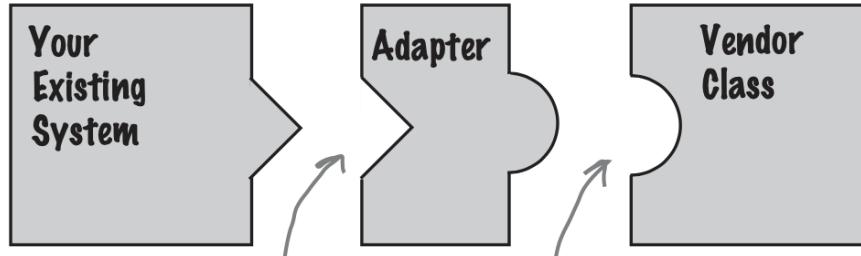


UNIVERSITÀ DEGLI STUDI
DI MILANO

Ingegneria del software

Progettazione

ADAPTER pattern

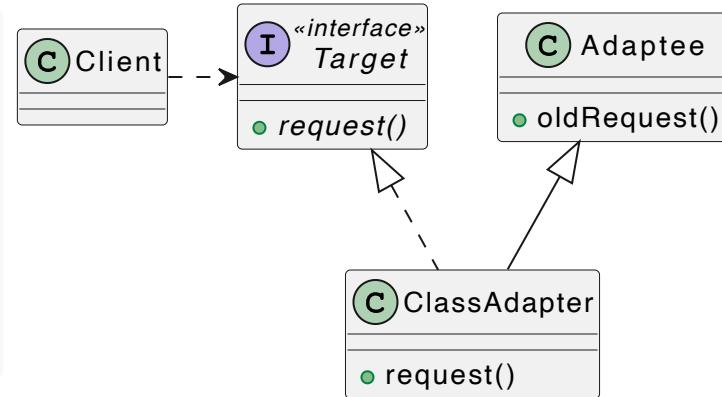


Due esempi di scenario:

- voglio usare uno o più componenti/librerie che ho trovato in giro, ma non sono direttamente compatibili
- sto facendo evolvere incrementalmente un sistema (legacy?) per cui mi trovo nella situazione di avere alcuni componenti vecchi che "provvisoriamente"?) devono collaborare con componenti nuovi

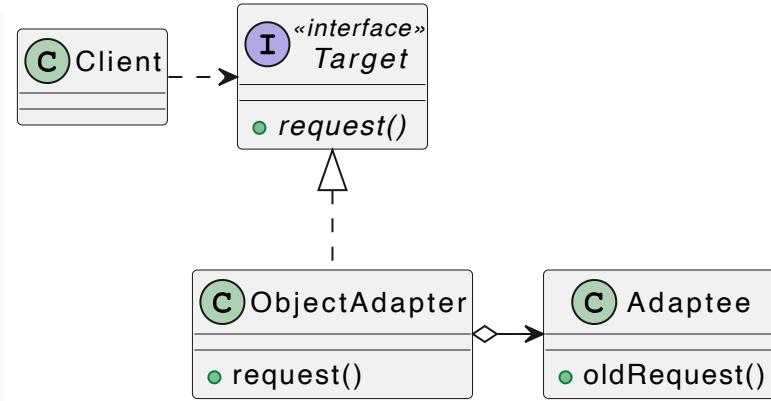
Class Adapter

```
public class Adapter extends Adaptee  
    implements Target {  
  
    @Override  
    public void request() {  
        this.oldRequest();  
    }  
}
```



Object Adapter

```
public class Adapter implements Target{  
    private final Adaptee adaptee;  
  
    public Adapter(Adaptee adaptee) {  
        assert adaptee != null;  
        this.adaptee = adaptee;  
    }  
  
    @Override  
    public void request() {  
        adaptee.oldRequest();  
    }  
}
```



PROs e CONs

Class Adapter

- potrebbe avere problemi con ereditarietà multipla
- è un unico oggetto che può essere usato contemporaneamente con le due interfacce diverse (vecchia e nuova)
- se un metodo non cambia, non devo fare nulla

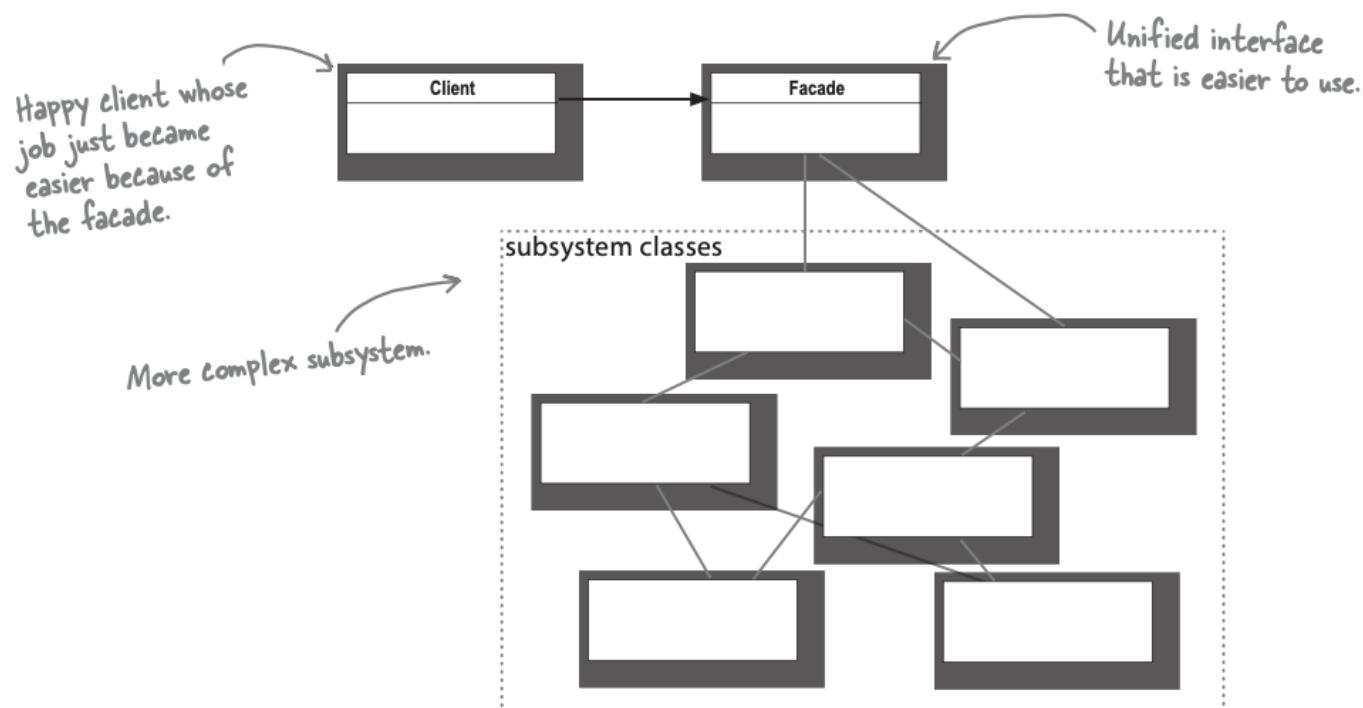
Object Adapter

- sono due oggetti distinti
- il nuovo non può più essere usato con interfaccia vecchia
- adatta un oggetto aderente a una interfaccia e non un classe
 - quindi adatta in realtà tutta una gerarchia di classi



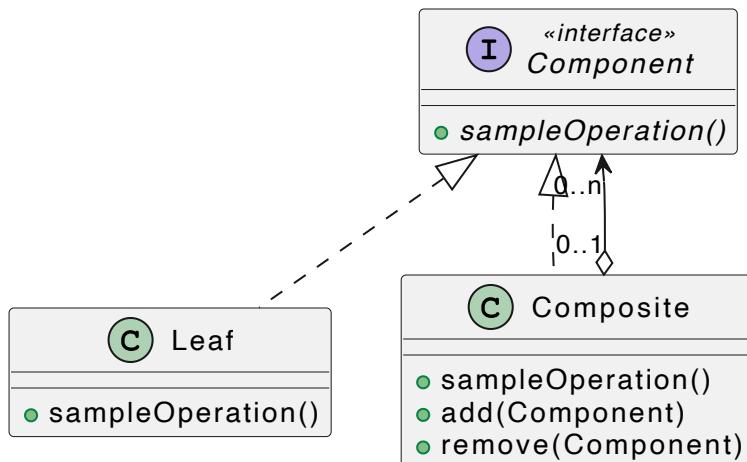
FACADE pattern

Fornisce una interfaccia unificata e semplificata a un insieme di interfacce separate.



COMPOSITE pattern

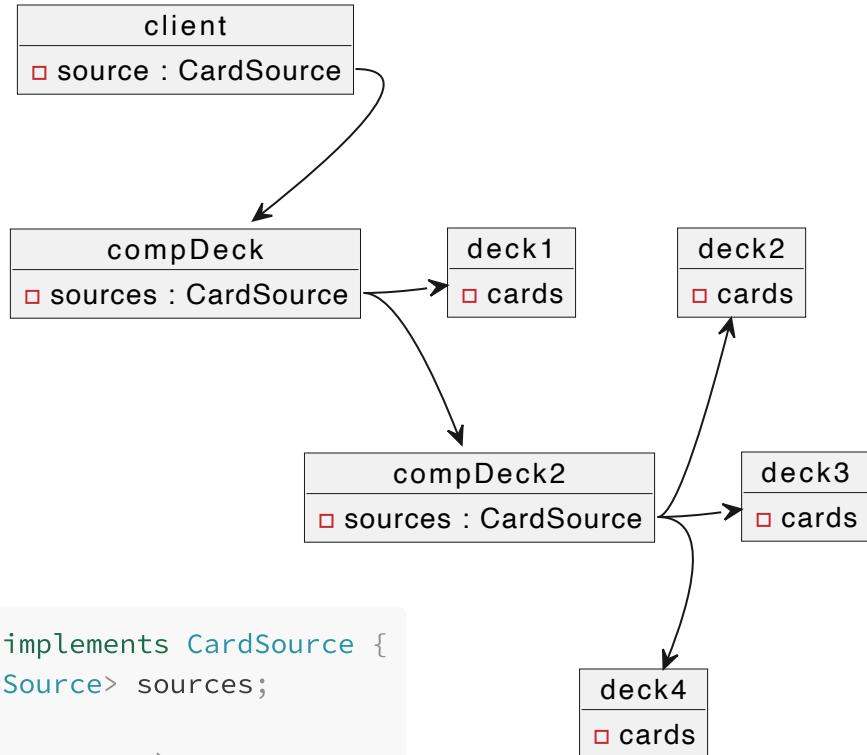
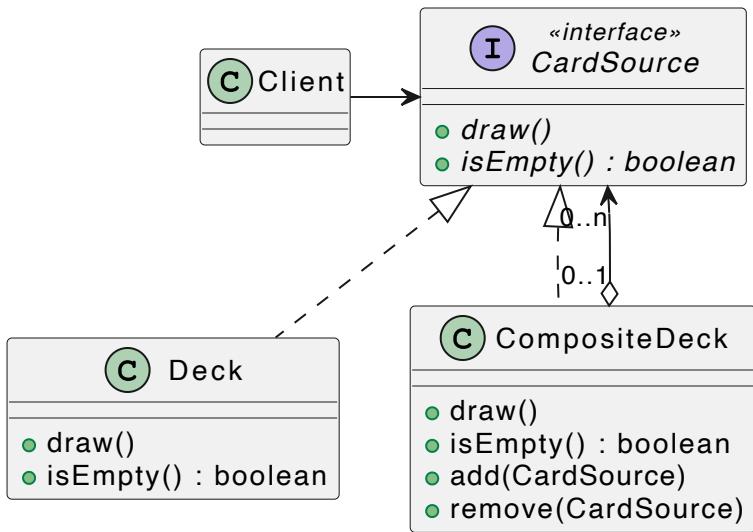
Gestire strutture ad albero per rappresentare gerarchie di parti e insiemi uniformemente



Il cliente interagisce esclusivamente tramite l'interfaccia Component

- Risulta semplice perché non si deve preoccupare se sta interagendo con elemento singolo o composito
- Minore possibilità di controllo su che tipo di oggetti possono essere dentro a particolari Composite

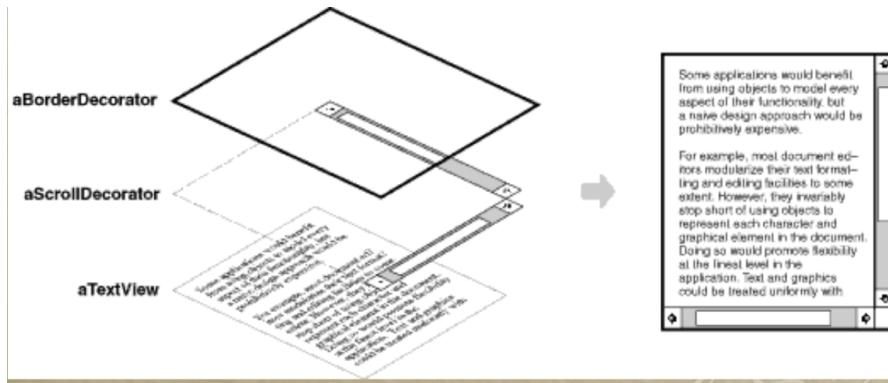
Composite Card Source



```
public class CardSourceComp implements CardSource {
    @NotNull private List<CardSource> sources;
    public boolean isEmpty() {
        for( CardSource source : sources )
            if( !source.isEmpty() ) return false;
        return true;
    }
}
```

DECORATOR pattern

Aggiungere nuove funzionalità o caratteristiche dinamicamente



FORMAGGI	SALUMI	VEGETARIANO	VARI
Stracchino, Fontina, Scamorza, Emmental, Pecorino, Mascarpone	Salamino Picc. Mortadella, Salsiccia, Wurstel	Lattuga ,Capperi Funghi Champ. Melanz. Fresche Zucchine Fresche Pomodori, Peperoni Cipolla, Olive nere Olive Verdi, Carciofi Mais, Limoni, Patate	Uova Panna, Crema di Carciofo, Crema di Radicchio
Mozzarella Grana,Ricotta Brie,Scamorza Gorgonzola	Prosc.Cotto Panc.Piccante	Spinaci(*),Rucola Pomodorini Funghi Misti Melanz. Grigliate Peperoni Filetti	Nutella Patatine Tonno Acciughe
	Prosc.Crudo Speck	Noci,FunghiPorcini	Pollo
Mozzarella Bufala (solo a Crudo)	Bresaola		Gamberetti

Card Decorator

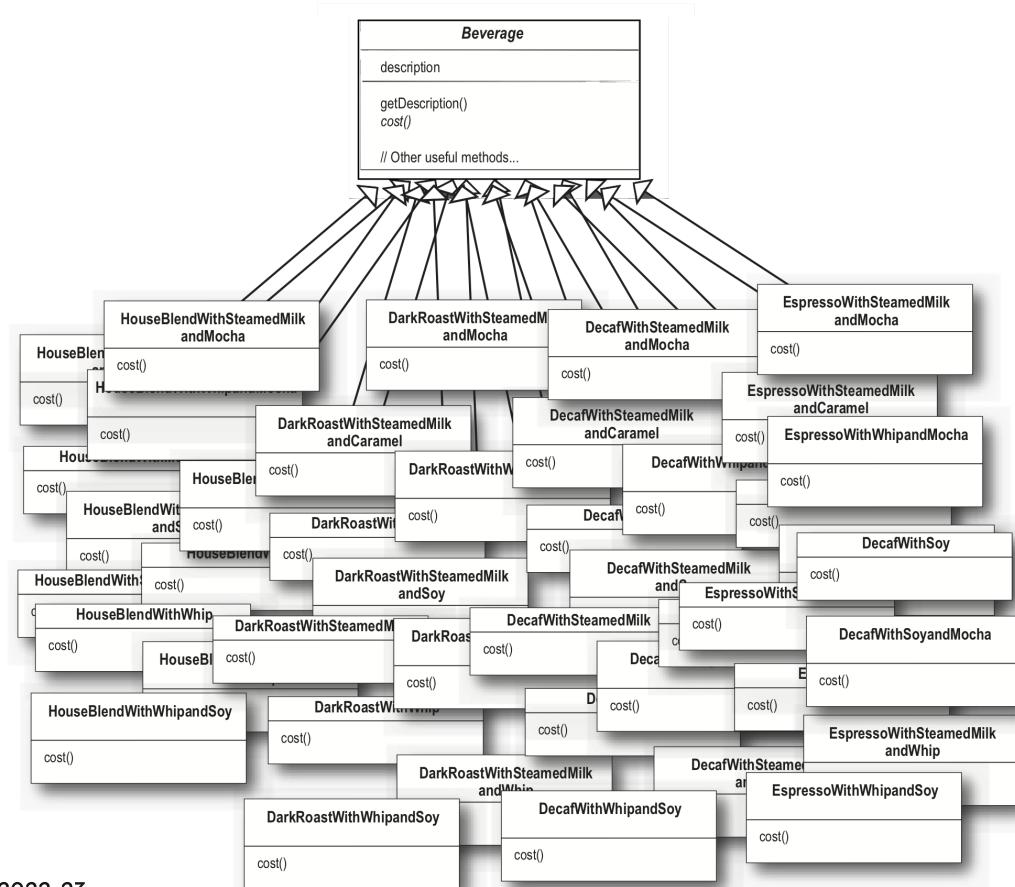
Ad esempio vogliamo aggiungere la funzionalità che ad ogni invocazione di `draw` venga scritta su console la carta estratta

```
public class LoggingDeck implements CardSource {  
    ...  
    public Card draw() {  
        Card card = cards.pop();  
        System.out.println(card);  
        return card;  
    }  
    ...  
}
```

Soluzione semplice ma da evitare

Gerarchia di classi

- creo una classe per ogni possibile combinazione di decorazioni



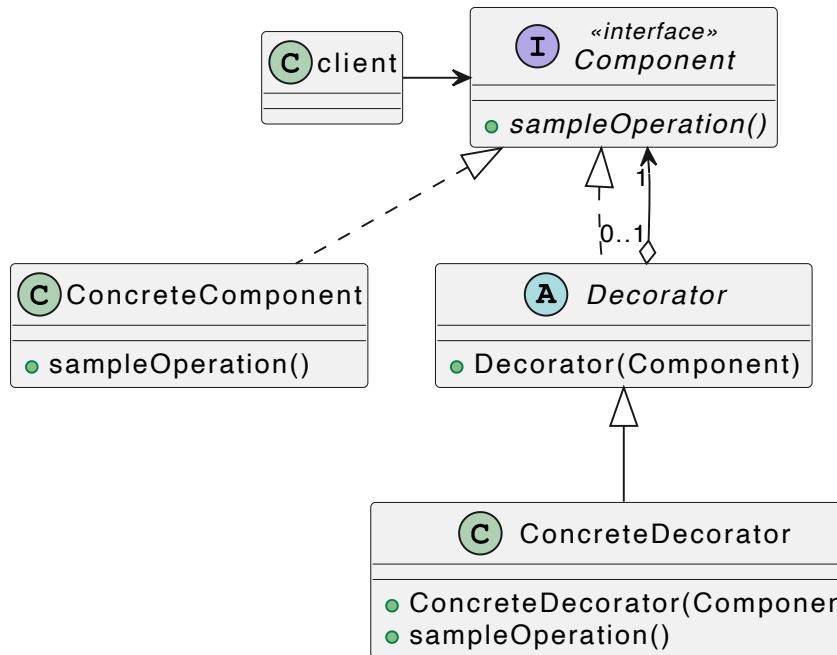
Soluzione semplice ma da evitare (2)

God Class[†]

Creo una classe unica in cui tramite degli attributi booleani e switch attualizzo le varie decorazioni

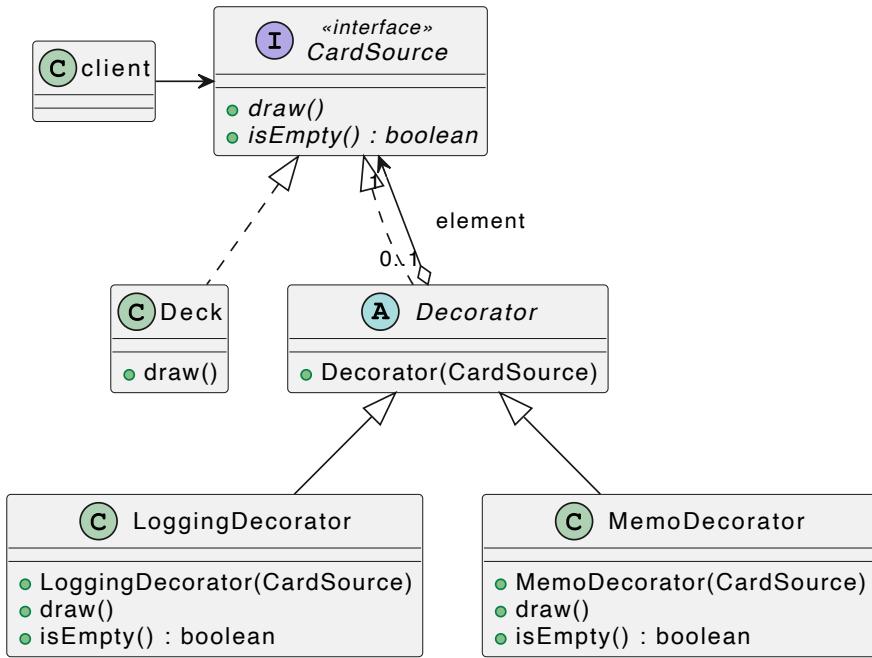
```
public class MultiModeDeck implements CardSource {  
    boolean logging = false;  
    boolean memo = false;  
  
    public void setLogging(boolean status) { logging = status; }  
    public void setMemo(boolean status) { memo = status; }  
  
    public Card draw() {  
        Card card = cards.pop();  
        if (logging) System.out.println(card);  
        if (memo) ...  
        return card;  
    }  
}
```

DECORATOR pattern



- Attacca le nuove responsabilità tramite l'aggiunta di nuovi oggetti

DECORATOR pattern e Decks



object diagram

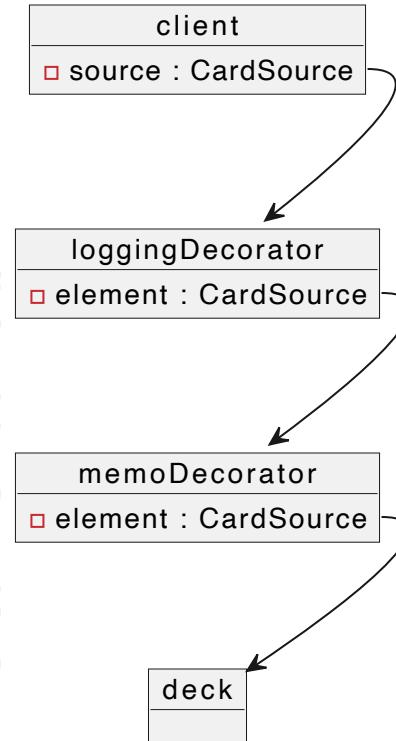
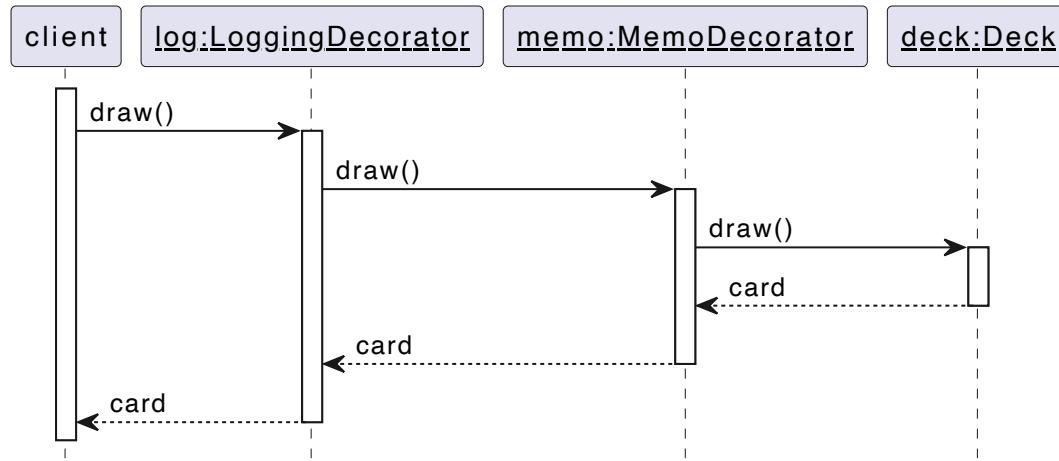


Diagramma di sequenza del Decorator



La classe astratta ?

Per capire cosa mettiamo nella classe astratta, partiamo da due *Decoratori* diretti di CardSource

```
public class LoggingDecorator implements CardSource {  
    private final CardSource element;  
  
    public LoggingDecorator(CardSource cardSource) {  
        element = cardSource;  
    }  
  
    public boolean isEmpty() {  
        return element.isEmpty();  
    }  
  
    public Card draw() {  
        Card card = element.draw();  
        System.out.println(card);  
        return card;  
    }  
}
```

```
public class MemoDecorator implements CardSource {  
    private final CardSource element;  
    private final List<Card> drawnCards;  
  
    public MemoDecorator(CardSource cardSource) {  
        element = cardSource;  
        drawCards = new ArrayList<>();  
    }  
  
    public boolean isEmpty() {  
        return element.isEmpty();  
    }  
  
    public Card draw() {  
        Card card = element.draw();  
        drawnCards.add(card);  
        return card;  
    }  
}
```

Prima versione

```
public abstract class Decorator
    implements CardSource {
    private final CardSource element;

    public Decorator(CardSource element) {
        assert element != null;
        this.element = element;
    }

    @Override
    public Card draw() {
        return element.draw();
    }

    @Override
    public boolean isEmpty() {
        return element.isEmpty();
    }
}
```

```
public class LoggingDecorator extends Decorator {
    public LoggingDecorator(CardSource element) {
        super(element);
    }

    @Override
    public Card draw() {
        Card card = super.draw();
        System.out.println(card);
        return card;
    }
}
```

```
public class MemoDecorator extends Decorator {
    private final List<Card> drawnCards;
    public MemoDecorator(CardSource element) {
        super(element);
        drawnCards = new ArrayList<>();
    }

    @Override
    public Card draw() {
        Card card = super.draw();
        drawnCards.add(card);
        return card;
    }
}
```

Seconda versione

```
public abstract class Decorator
    implements CardSource {
    @NotNull private final CardSource element;

    public Decorator(@NotNull CardSource element) {
        this.element = element;
    }
    @Override @NotNull
    public Card draw() {
        Card card = element.draw();
        decorationAction(card);
        return card;
    }
    @Override
    public boolean isEmpty() {
        return element.isEmpty();
    }
    protected void decorationAction(
        @NotNull Card card) {};
}
```

```
public class LoggingDecorator extends Decorator {
    public LoggingDecorator(@NotNull CardSource element) {
        super(element);
    }
    @Override
    protected void decorationAction(@NotNull Card card) {
        System.out.println(card);
    }
}
```

```
public class MemoDecorator extends Decorator {
    @NotNull private final List<Card> drawnCards;
    public MemoDecorator(@NotNull CardSource element) {
        super(element);
        drawnCards = new ArrayList<>();
    }
    @Override
    protected void decorationAction(@NotNull Card card) {
        drawnCards.add(card);
    }
}
```



Sul libro HeadFirst

```
public class Espresso extends Beverage {  
  
    public Espresso() {  
        description = "Espresso"; ←  
    }  
  
    public double cost() {  
        return 1.99;  
    }  
}
```

Finally, we need to... about

```
public abstract class Beverage {  
    String description = "Unknown Beverage";  
  
    public String getDescription() { ←  
        return description;  
    }  
  
    public abstract double cost();  
}
```

```
public abstract class CondimentDecorator extends Beverage {  
    Beverage beverage;  
    public abstract String getDescription(); ←  
}
```

```
public class Mocha extends CondimentDecorator {  
  
    public Mocha(Beverage beverage) { ←  
        this.beverage = beverage;  
    }  
  
    public String getDescription() {  
        return beverage.getDescription() + ", Mocha";  
    }  
  
    public double cost() {  
        return beverage.cost() + .20; ←  
    }  
}
```

Dove lo troviamo in Java?

Ad esempio nelle classi di *InputStreams*

