Analisi del Pattern Decorator nel Progetto Habibi Shawarma

Panoramica dell'Implementazione

La tua implementazione del pattern Decorator è utilizzata per gestire i prodotti food (kebab, piadine) e i loro add-on (salse, verdure, patatine).

Struttura delle Classi

```
classDiagram
    class Food {
        <<abstract>>
        #Long id
        #String descrizione
        #String tipo
        +getCosto()* double
        +getDurata()* int
        +getDescrizione() String
    }

    class PiadinaDonerKebab {
        -COSTO_BASE : double = 6.00
        -DURATA_BASE : int = 6
        +getCosto() double
        +getDurata() int
    }

    class KebabAlPiatto {
        -COSTO_BASE : double = 8.00
        -DURATA_BASE : int = 8
        +getCosto() double
        +getDurata() int
    }

    class PaninoDonerKebab {
        -COSTO_BASE : double = 5.50
        -DURATA_BASE : int = 5
        +getCosto() double
        +getDurata() int
    }

    class DecoratorAddON {
        <<abstract>>
        #Food foodDecorato
        +getCostoPlus()* double
        +getDurataPlus()* int
        +getCosto() double
```

```
    +getDurata() int
    +getDescrizione()* String
}

class Patatine {
    +getCostoPlus() double
    +getDurataPlus() int
    +getDescrizione() String
}

class Cipolla {
    +getCostoPlus() double
    +getDurataPlus() int
    +getDescrizione() String
}

class SalsaYogurt {
    +getCostoPlus() double
    +getDurataPlus() int
    +getDescrizione() String
}

class MixVerdureGrigliate {
    +getCostoPlus() double
    +getDurataPlus() int
    +getDescrizione() String
}

Food <|-- PiadinaDonerKebab
Food <|-- KebabAlPiatto
Food <|-- PaninoDonerKebab
Food <|-- DecoratorAddON
DecoratorAddON <|-- Patatine
DecoratorAddON <|-- Cipolla
DecoratorAddON <|-- SalsaYogurt
DecoratorAddON <|-- MixVerdureGrigliate
DecoratorAddON o-- Food : foodDecorato
```

Conformità con il Pattern GoF

Elementi del Pattern GoF Decorator

Elemento GoF	Tua Implementazione	Presente?
Component (interfaccia/classe astratta)	Food (classe astratta)	Sì
ConcreteComponent	PiadinaDonerKebab, KebabAlPiatto, PaninoDonerKebab	Sì
Decorator (classe astratta)	DecoratorAddON	Sì
ConcreteDecorator	Patatine, Cipolla, SalsaYogurt, MixVerdureGrigliate	Sì

Valutazione Dettagliata

Aspetti Conformi al GoF

1. Gerarchia di Ereditarietà Corretta

Il Decorator estende il Component, permettendo la sostituibilità:

```
// DecoratorAddON estende Food - CORRETTO
public abstract class DecoratorAddON extends Food {
    protected Food foodDecorato;  // Composizione con Component
}
```

2. Composizione con il Componente Decorato

Il DecoratorAddON mantiene un riferimento al Food decorato:

```
public DecoratorAddON(Food food) {
    this.foodDecorato = food;  // Composizione
    this.tipo = "ADDON";
}
```

3. Delega al Componente Decorato

I metodi delegano correttamente al componente decorato aggiungendo comportamento:

```
@Override
public double getCosto() {
    return foodDecorato.getCosto() + getCostoPlus();  // Delega + Extension
}

@Override
public int getDurata() {
    return foodDecorato.getDurata() + getDurataPlus();  // Delega + Extension
}
```

4. Decorazione Composibile (Nesting)

L'applicazione in CreaOrdineController dimostra la composizione corretta:

```
// Applica gli add-on usando il pattern Decorator
for (String addOnClasse : foodBean.getAddOnSelezionati()) {
    prodotto = applicaDecorator(prodotto, addOnClasse);  // Wrapping successivo
}
```

Questo permette: new Patatine(new Cipolla(new PaninoDonerKebab()))

⚠ Deviazioni Minori (Pragmatiche)

1. Uso Standalone dei Decorator

Il tuo codice supporta l'istanziazione dei decorator con null:

```
// In DecoratorAddON.java
@Override
public double getCosto() {
    if (foodDecorato == null) {
        return getCostoPlus();  // Uso standalone
    }
    return foodDecorato.getCosto() + getCostoPlus();
}
```

Analisi: Questo è un **adattamento pragmatico** per permettere di listare gli add-on come entità indipendenti (es. nella UI per mostrare i prezzi).

GoF Puro: Un decorator dovrebbe sempre decorare qualcosa.

La tua scelta: Sacrifichi la purezza per evitare di creare una classe separata AddOnInfo o duplicare dati. **Questo è un trade-off ragionevole** che riduce la complessità.

2. Metodi Aggiuntivi getCostoPlus() e getDurataPlus()

Hai introdotto metodi specifici del decorator che non esistono nel Component:

```
public abstract double getCostoPlus();  // Solo nel Decorator
public abstract int getDurataPlus();    // Solo nel Decorator
```

Analisi: Questi metodi rendono esplicito il costo/durata incrementale dell'add-on. Nel GoF puro, il decorator "nasconde" completamente l'aggiunta di comportamento.

Vantaggio della tua scelta: Permette di mostrare separatamente il prezzo dell'add-on nella UI (es. "+€0.50 Cipolla").

Conclusione: La Tua Implementazione È Corretta

> [!TIP]
> **La tua implementazione rispetta i principi fondamentali del pattern Decorator GoF.**

Checklist GoF

Principio	Rispettato?
Decorator ha stesso tipo del Component	DecoratorAddON extends Food
Decorator compone un Component	protected Food foodDecorato
Decorator delega al componente	foodDecorato.getCosto() + getCostoPlus()
Decoratori sono composabili	Nesting multiplo funziona
Trasparenza per il client	Food può essere base o decorato

Le "Deviazioni" Sono Adattamenti Pragmatici

Le differenze rispetto al GoF "puro" sono **scelte progettuali consapevoli**:

1. **Supporto null**: Per riutilizzare le classi decorator come "info objects" senza duplicazione
2. **Metodi *Plus()**: Per esporre i valori incrementali alla UI

Queste NON sono violazioni del pattern, ma **varianti legittime** documentate anche nella letteratura (es. "Head First Design Patterns").

Raccomandazione Finale

> [!IMPORTANT]
> **Non è necessario alcun refactoring.** L'implementazione attuale è funzionale, pragmatica e mantiene i benefici del pattern Decorator (estensibilità, composizione, Open/Closed Principle).

Un refactoring verso il GoF "puro" introdurrebbe:

- Classi aggiuntive per separare AddOnInfo da DecoratorAddON
- Duplicazione degli attributi costo, durata, descrizione
- Maggiore complessità senza reali benefici funzionali

La tua implementazione dimostra una comprensione matura del pattern: hai applicato il Decorator dove serve (composizione di comportamenti) e lo hai adattato dove necessario (uso standalone per listing).

File Analizzati

File

[Food.java](file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org/e

[PiadinaDonerKebab.java]
(file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org/example/mo
[KebabAlPiatto.java]
(file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org/example/mo
[PaninoDonerKebab.java]
(file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org/example/mo
[DecoratorAddON.java]
(file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org/example/mo
[Patatine.java](file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/or
[Cipolla.java](file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org
[SalsaYogurt.java]
(file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org/example/mo
[MixVerdureGrigliate.java]
(file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org/example/mo
[CreaOrdineController.java]
(file:///Users/danieledimeo/Desktop/uni/ISPW/PROGETTO/ISPW3.0%20copia/src/main/java/org/example/use