

TÓPICO 11: DECORATOR

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

Decorator é um padrão de projeto **estrutural** que permite **adicionar responsabilidades a um objeto** de forma dinâmica como uma alternativa flexível à herança.



FRADEL AND SPIES CO

MENU

OPEN DAILY

07.00AM

- 19.00PM

Coffee

Espresso	\$2.50
Americano	\$3.00
Iced Latte	\$5.00
Iced Mocha	\$5.50
Cold Brew	\$4.50

Non Coffee

Hot Chocolate	\$3.75
Matcha Latte	\$5.00
Thai Tea	\$4.50
Chai Latte	\$4.50
Herbal Tea	\$2.50

Pastries

Butter Croissant	\$3.25
Almond Croissant	\$4.00
Almond Danish	\$4.25
Cinnamon Roll	\$4.25

Add-ons

Extra Shot	\$1.00
Oat/Soy Milk	\$0.70
Flavored Syrups	\$0.50

SIMPLE MOMENTS.
PERFECT BREWS.

+123-456-7890
123 ANYWHERE ST., ANY CITY



Problema

Uma cafeteria vende bebidas e **complementos combináveis**. Por herança, cada **combinação** viraria uma subclasse nova:

- Cafe , CafeComLeite ,
CafeComChantilly ,
CafeComLeiteComChantilly ...

Com N complementos, o número de combinações **explode**:

```
public class Cafe {  
    public String getDescricao() { return "Café"; }  
    public double custo() { return 5.0; }  
}  
  
public class CafeComLeite extends Cafe {  
    public String getDescricao() { return "Café com leite"; }  
    public double custo() { return 6.5; }  
}  
  
public class CafeComLeiteComChantilly extends CafeComLeite {  
    public String getDescricao() { return "Café com leite e com chantilly"; }  
    public double custo() { return 7.0; }  
}
```

A herança aqui tem dois problemas sérios:

- **explosão de subclasses**: N complementos geram **2^N** combinações
- **estático**: a "decoreação" fica fixa em tempo de compilação
- mudar o preço base do café exige revisar **todas** as combinações
- não dá para montar a mesma bebida com **ordens diferentes**

Solução

Em vez de subclasses para cada combinação, vamos **envolver** a bebida em camadas: cada **complemento** é um objeto que **embrulha outro** e soma o próprio comportamento, mantendo a **mesma interface**.

Cada camada se conecta como uma **cadeia**:

- o objeto "externo" **delega** ao interno e **acrescenta** algo
- o cliente enxerga **sempre a mesma interface** (Bebida)
- combinações são montadas **em tempo de execução**
- trocar a ordem ou os complementos **não cria classes novas**



Implementação

1. Primeiro, definimos a **interface comum** `Bebida` :

```
public interface Bebida {  
    String getDescricao();  
    double custo();  
}
```

2. Criamos o **componente concreto**: o `Cafe`, a bebida base a ser decorada:

```
public class Cafe implements Bebida {  
    public String getDescricao() {  
        return "Café";  
    }  
  
    public double custo() {  
        return 5.0;  
    }  
}
```

3. Criamos o **decorator abstrato**: guarda a referência à bebida embrulhada:

```
public abstract class Complemento implements Bebida {  
    protected final Bebida bebida;  
  
    public Complemento(Bebida bebida) {  
        this.bebida = bebida;  
    }  
}
```

Ele **implementa a mesma interface** e **delega** ao objeto interno.

4. Criamos os **decorators concretos**, que somam descrição e custo:

```
public class Leite extends Complemento {
    public Leite(Bebida bebida) { super(bebida); }

    public String getDescricao() {
        return bebida.getDescricao() + " com leite";
    }

    public double custo() {
        return bebida.custo() + 1.5;
    }
}

public class Chantilly extends Complemento {
    public Chantilly(Bebida bebida) { super(bebida); }

    public String getDescricao() {
        return bebida.getDescricao() + " com chantilly";
    }

    public double custo() {
        return bebida.custo() + 2.0;
    }
}
```

Utilização

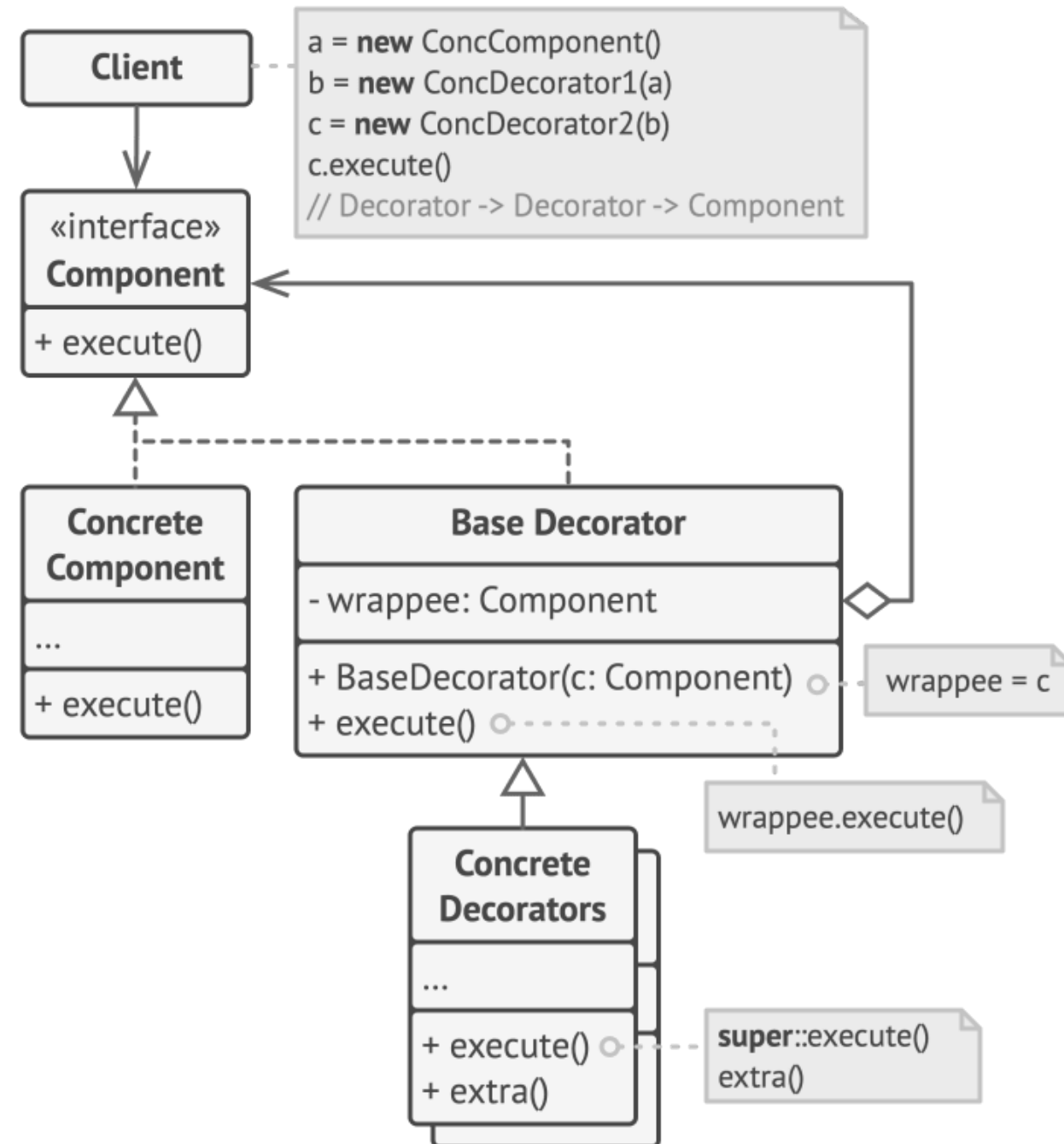
O cliente **monta a cadeia** envolvendo a bebida nos complementos desejados:

```
Bebida bebida = new Cafe();  
bebida = new Leite(bebida);  
bebida = new Chantilly(bebida);  
  
System.out.println(bebida.getDescricao()); // Café com leite com chantilly  
System.out.println(bebida.custo());        // 8.5
```

- **qualquer combinação** sem criar uma classe nova
- a **ordem** dos complementos fica sob controle do cliente
- `custo()` e `getDescricao()` **somam** camada por camada
- testar cada complemento é fácil: ele depende só de `Bebida`

Diagramas

- **Component:** interface comum do objeto e dos decoradores
- **ConcreteComponent:** objeto que recebe as responsabilidades
- **Decorator:** mantém referência ao **Component** e delega
- **ConcreteDecorator:** adiciona responsabilidades
- **Client:** envolve o componente com os decoradores que



Analogias

- **Pizza montada:**
 - a **massa** é a base; ingredientes são adicionados por cima
 - "queijo extra" ou "borda recheada" **envolvem** a pizza sem mudar o que ela é
 - no fim, continua sendo uma **pizza**, só que "decorada"

Aplicações

- Adicionar responsabilidades **individualmente e em combinações**
- Quando **herança** geraria explosão de subclasses ou for estática demais
- Adicionar comportamento **em tempo de execução** (e poder escolher a ordem)

Exemplos típicos:

- **streams do java.io**: `new BufferedInputStream(new FileInputStream(...))`
- **Collections**: `unmodifiableList`, `synchronizedList`
- **filtros/interceptadores**: filtros de servlets, middlewares
- adicionar **log, cache, compressão, cifragem** a um serviço

Prós

- **Combinações em runtime**, sem criar classes novas
- **OCP**: novos decoradores entram sem tocar no componente base
- **SRP**: cada responsabilidade fica isolada em um decorador
- As camadas podem ser **empilhadas** na ordem desejada

Contras

- Muitos **objetos pequenos** e camadas no sistema
- A **ordem** dos decoradores pode afetar o resultado
- Difícil **inspecionar** o objeto real sob as camadas
- Remover um complemento exige **reconstruir** a cadeia

Regra: use quando precisar somar responsabilidades **em combinações** sem multiplicar subclasses.

Resumo

- **Embrulha** um objeto com outro da **mesma interface**
- Cada camada **delega** e **soma** a própria responsabilidade
- Combinações são montadas em **runtime**, sem explosão de classes
- Muito usado para adicionar funcionalidades a **serviços e streams**