

Padrões de projeto estruturais

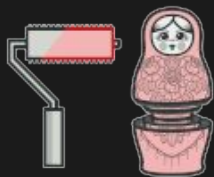
Matheus Barbosa
matheus.barbosa@dcx.ufpb.br

Explicam como montar
objetos e classes em
estruturas maiores, mas ainda
mantendo essas estruturas
flexíveis e eficientes.



Facade

Fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer outro conjunto complexo de classes.



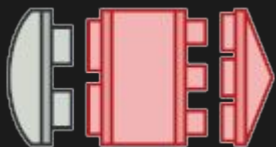
Decorator

Fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer outro conjunto complexo de classes.



Composite

Permite que você componha objetos em estrutura de árvores e então trabalhe com essas estruturas como se fossem objetos individuais.



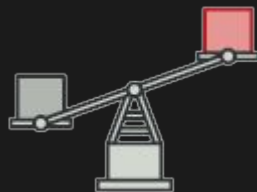
Adapter

Permite a colaboração de objetos de interfaces incompatíveis.



Bridge

Permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.



Flyweight

Permite que você coloque mais objetos na quantidade disponível de RAM ao compartilhar partes do estado entre múltiplos objetos ao invés de manter todos os dados em cada objeto.



Proxy

Permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original.

Facade

Oferecer uma interface única para um conjunto de interfaces de um subsistema. Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar

Problema

Imagina que numa cafeteria você tem vários subsistemas para processar um pedido:

Cozinha → prepara os sanduíches.

Barista → faz os cafés.

Caixa → calcula e cobra o total.

Entrega → leva o pedido até a mesa.

Se o cliente fosse falar com cada um desses diretamente, seria confuso...

```
// Subsistemas
class Cozinha {
    void prepararSanduiche() {
        System.out.println("Sanduíche preparado.");
    }
}

class Barista {
    void prepararCafe() {
        System.out.println("Café preparado.");
    }
}

class Caixa {
    void cobrar(double valor) {
        System.out.println("Pagamento de R$" + valor + " recebido.");
    }
}

class Entrega {
    void entregarPedido() {
        System.out.println("Pedido entregue na mesa.");
    }
}
```

```
Main.java

public class Main {
    public static void main(String[] args) {
        // Criando cada subsistema separadamente
        Cozinha cozinha = new Cozinha();
        Barista barista = new Barista();
        Caixa caixa = new Caixa();
        Entrega entrega = new Entrega();

        // Cliente precisa chamar cada passo manualmente
        cozinha.prepararSanduiche();
        barista.prepararCafe();
        caixa.cobrar(25.0);
        entrega.entregarPedido();
    }
}
```

Solução

O Facade entra aqui como Atendente:

- O cliente só fala com o atendente.
- O atendente sabe qual subsistema acionar (barista, cozinha, caixa, entrega).

```

// Subsistemas
class Cozinha {
    void prepararSanduiche() {
        System.out.println("Sanduíche preparado.");
    }
}

class Barista {
    void prepararCafe() {
        System.out.println("Café preparado.");
    }
}

class Caixa {
    void cobrar(double valor) {
        System.out.println("Pagamento de R$" + valor + " recebido.");
    }
}

class Entrega {
    void entregarPedido() {
        System.out.println("Pedido entregue na mesa.");
    }
}

```

AtendenteFacade.java

```

class AtendenteFacade {
    private Cozinha cozinha;
    private Barista barista;
    private Caixa caixa;
    private Entrega entrega;

    public AtendenteFacade() {
        this.cozinha = new Cozinha();
        this.barista = new Barista();
        this.caixa = new Caixa();
        this.entrega = new Entrega();
    }

    public void fazerPedido() {
        cozinha.prepararSanduiche();
        barista.prepararCafe();
        caixa.cobrar(25.0);
        entrega.entregarPedido();
    }
}

```

Main.java

```

// Cliente
public class Main {
    public static void main(String[] args) {
        AtendenteFacade atendente = new AtendenteFacade();
        atendente.fazerPedido();
        // Cliente só fala com o atendente
    }
}

```


Uma fachada pode se tornar um objeto deus acoplado a todas as classes de uma aplicação.

Decorator

Anexar responsabilidades adicionais a um objeto dinamicamente. Decorators oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade.

Problema

A cafeteria oferece cafés que podem ter diversos adicionais. O cliente pode escolher qualquer combinação de adicionais no café.

Exemplos:

- Café simples

- Café com leite

- Café com leite + chocolate

- Café com chantilly

Como funciona o pedido:

1. O cliente escolhe a bebida desejada.
2. O sistema precisa instanciar **uma classe específica** que já representa exatamente aquela combinação de adicionais.
3. Para cada nova combinação possível, uma nova classe precisa ser criada.

```
// Classe base
class CafeSimples {
    public String getDescricao() {
        return "Café simples";
    }

    public double preco() {
        return 5.0;
    }
}

// Combinações possíveis sem decorator
class CafeComLeite extends CafeSimples {
    public String getDescricao() {
        return "Café simples, com leite";
    }

    public double preco() {
        return 5.0 + 2.0;
    }
}

class CafeComLeiteChocolate extends CafeSimples {
    public String getDescricao() {
        return "Café simples, com leite, com chocolate";
    }

    public double preco() {
        return 5.0 + 2.0 + 3.0;
    }
}
```

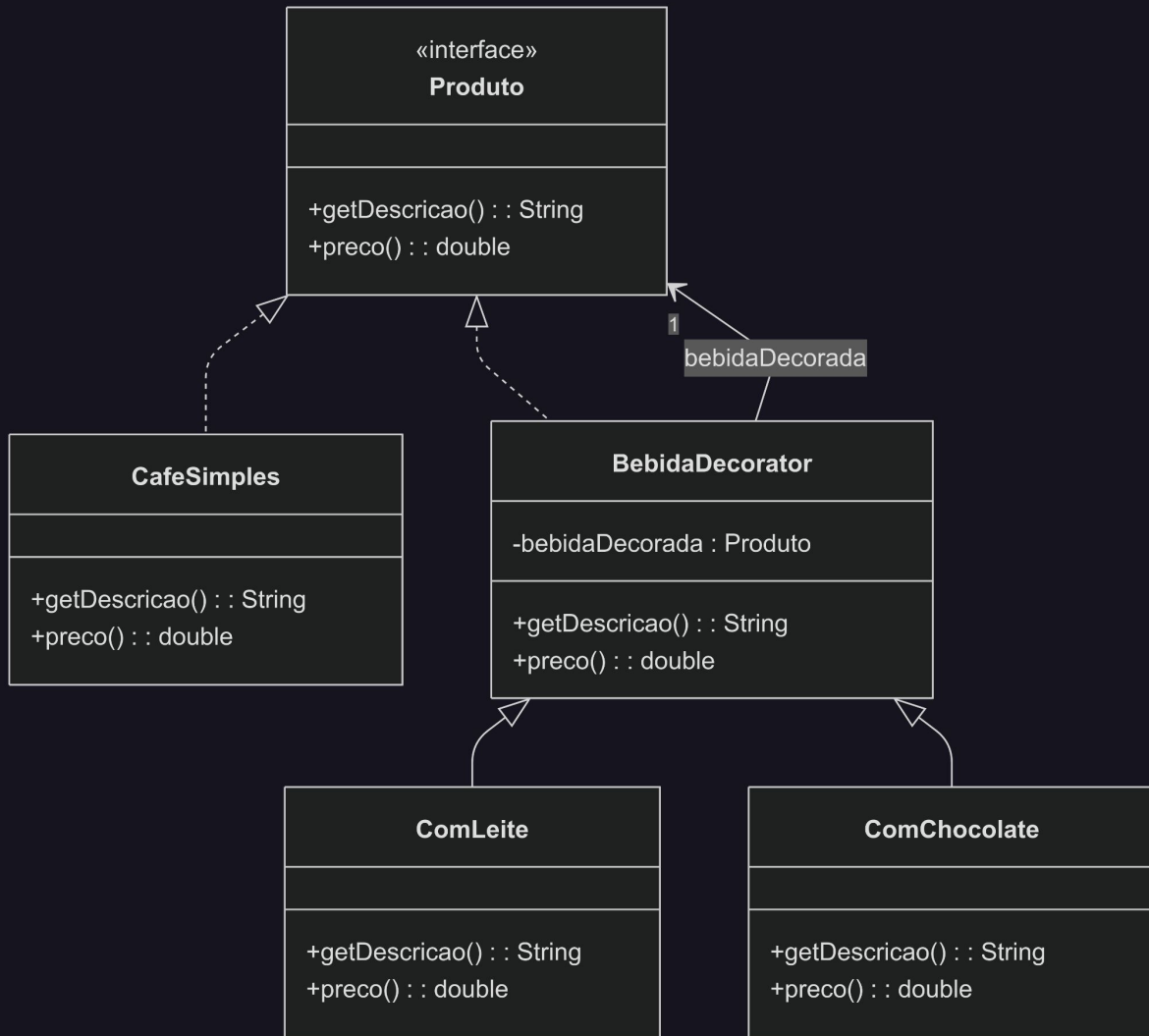
Solução

Define uma interface comum para o produto (bebida)

Uma implementação base inicial (café simples)

Decorators estendem funcionalidades (BebidaDecorator)

Funcionalidades podem ser combinadas dinamicamente



```

// Interface comum
interface Produto {
    String getDescricao();
    double preco();
}

```

```

// Bebida base

```

```

class CafeSimples implements Produto {
    public String getDescricao() {
        return "Café simples";
    }

    public double preco() {
        return 5.0;
    }
}

```

```

abstract class BebidaDecorator implements Produto {
    protected Bebida bebidaDecorada;

    public BebidaDecorator(Bebida bebidaDecorada) {
        this.bebidaDecorada = bebidaDecorada;
    }

    public String getDescricao() {
        return bebidaDecorada.getDescricao();
    }

    public double preco() {
        return bebidaDecorada.preco();
    }
}

```

```

// Decorators concretos
class ComLeite extends BebidaDecorator {
    public ComLeite(Bebida bebidaDecorada) {
        super(bebidaDecorada);
    }

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

    public double preco() {
        return bebidaDecorada.preco() + 2.0;
    }
}

class ComChocolate extends BebidaDecorator {
    public ComChocolate(Bebida bebidaDecorada) {
        super(bebidaDecorada);
    }

    public String getDescricao() {
        return bebidaDecorada.getDescricao() + ", com chocolate";
    }

    public double preco() {
        return bebidaDecorada.preco() + 3.0;
    }
}

```

Pode gerar muitas criações de objetos, se usado em excesso.

Composite

Compor objetos em estruturas de árvore para representar hierarquias todo-parte. Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme.

Problema

Na cafeteria, além de bebidas, você pode ter combos:

- Café + pão de queijo

- Café + sanduíche + bolo

- Combo especial com vários itens dentro de outros combos

Sem o Composite, você precisa tratar itens simples (bebida, bolo, sanduíche) e combos de forma diferente, escrevendo códigos separados para cada caso. Isso deixa o sistema pouco flexível e difícil de manter.

Como funciona o pedido:

Aqui o cliente precisa verificar se o pedido é um item único ou um combo e tratar manualmente:

```
public class Main {  
    public static void main(String[] args) {  
        Produto cafe = new Produto("Café simples", 5.0);  
        Produto sanduiche = new Produto("Sanduíche", 12.0);  
  
        Combo combo = new Combo("Combo Café da Manhã");  
        combo.adicionarItem(cafe);  
        combo.adicionarItem(sanduiche);  
  
        // Problema: cliente precisa saber se é combo ou item simples  
        System.out.println("Pedido: " + combo.nome + " - R$" + combo.precoTotal());  
        System.out.println("Pedido: " + cafe.descricao + " - R$" + cafe.preco);  
    }  
}
```

```
class Produto {  
    String descricao;  
    double preco;  
  
    Produto (String descricao, double preco) {  
        this.descricao = descricao;  
        this.preco = preco;  
    }  
}  
  
class Combo {  
    String nome;  
    List<Produto > itens = new ArrayList<>();  
  
    Combo(String nome) {  
        this.nome = nome;  
    }  
  
    void adicionarItem(Produto produto) {  
        itens.add(produto);  
    }  
  
    double precoTotal() {  
        double total = 0;  
        for (Produto b : itens) {  
            total += b.preco;  
        }  
        return total;  
    }  
}
```

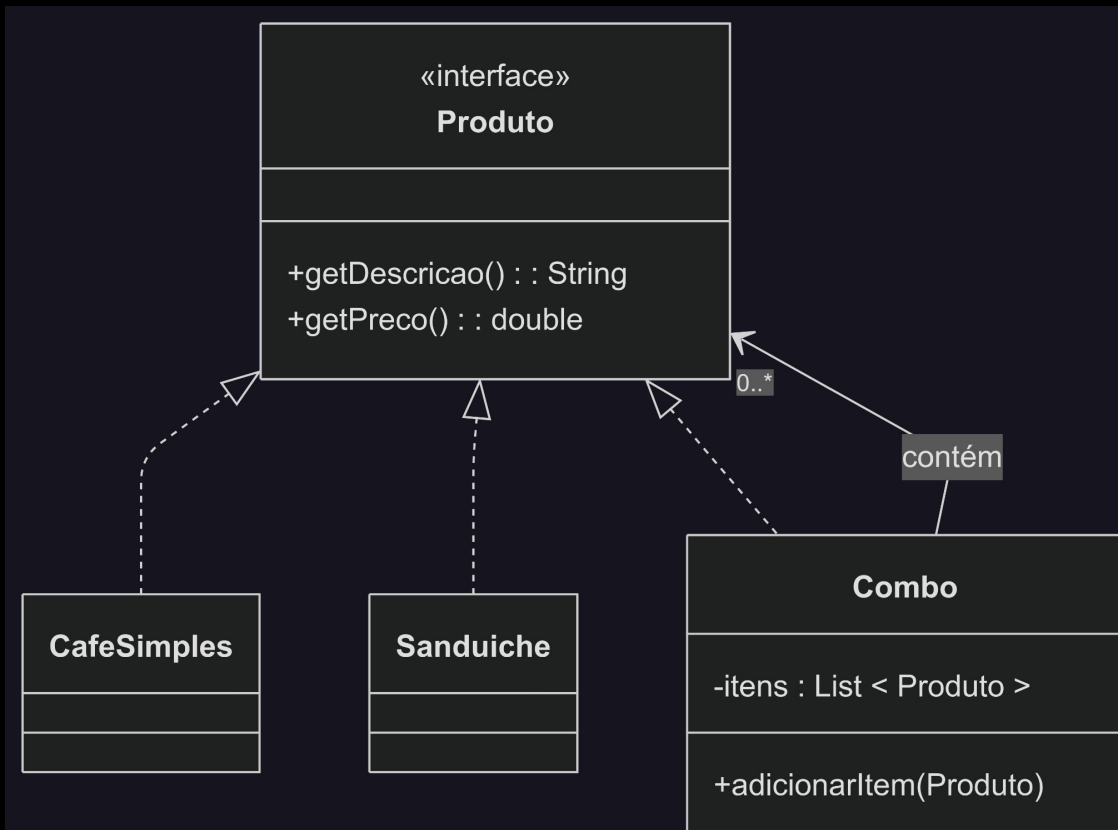
Solução

Cria uma interface única para itens do pedido (Produto).

Folhas: itens simples (café, bolo, sanduíche).

Compostos: combos que agrupam vários itens (inclusive outros combos).

Assim, tanto itens simples quanto combos são **tratados da mesma forma** pelo cliente. O cliente não precisa se preocupar se o pedido é um único item ou um combo.



```

interface Produto {
    String getDescricao();
    double getPreco();
}

// Folhas
class CafeSimples implements Produto {
    public String getDescricao() {
        return "Café Simples";
    }

    public double getPreco() {
        return 5.0;
    }
}

class Sanduiche implements Produto {
    public String getDescricao() {
        return "Sanduíche";
    }

    public double getPreco() {
        return 15.0;
    }
}

```

```

class Combo implements Produto {
    private Produto[] produtos;

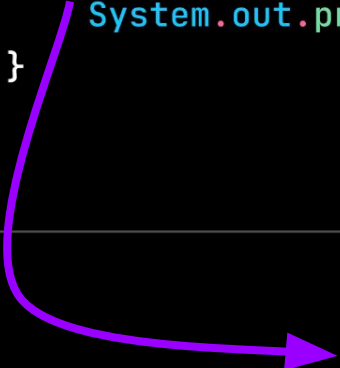
    public Combo(Produto... produtos) {
        this.produtos = produtos;
    }

    public String getDescricao() {
        StringBuilder descricao = new StringBuilder("Combo de: ");
        for (int i = 0; i < produtos.length; i++) {
            descricao.append(produtos[i].getDescricao());
            if (i < produtos.length - 1) {
                descricao.append(", ");
            }
        }
        return descricao.toString();
    }

    public double getPreco() {
        double total = 0.0;
        for (Produto produto : produtos) {
            total += produto.getPreco();
        }
        return total * 0.9; // 10% de desconto no combo
    }
}

```

```
public class App {  
    public static void main( String[] args ){  
        Produto cafe = new CafeSimples();  
        Produto sanduiche = new Sanduiche();  
  
        Combo comboCafe = new Combo(cafe, sanduiche);  
  
        System.out.println(cafe.getDescricao() + " - R$" + cafe.getPreco());  
        System.out.println(comboCafe.getDescricao() + " - R$" + comboCafe.getPreco());  
    }  
}
```



```
Café Simples - R$5.0  
Combo de: Café Simples, Sanduíche - R$18.0
```

Pode ser difícil providenciar uma interface comum para classes cuja funcionalidade difere muito. Em certos cenários, você precisaria generalizar muito a interface componente, fazendo dela uma interface de difícil compreensão.