

TÓPICO 12: STRATEGY

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

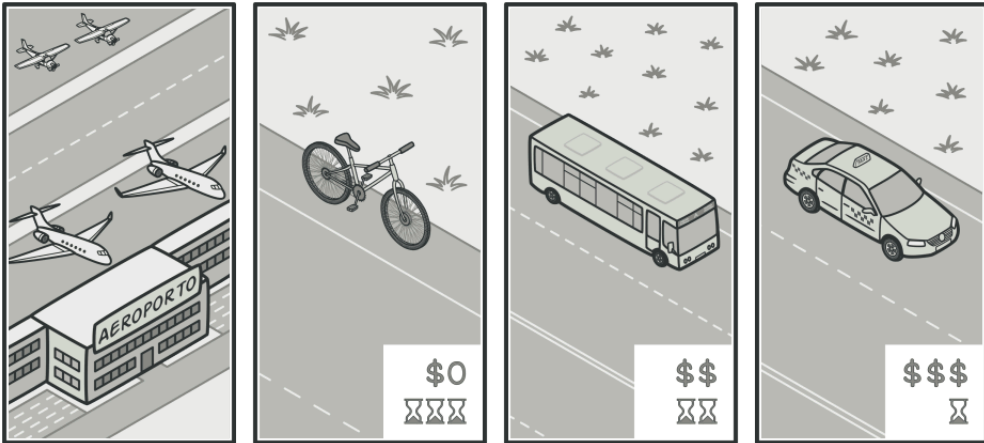
Strategy é um padrão de projeto **comportamental** que define uma **família de algoritmos**, encapsula cada um em uma classe própria e os torna **intercambiáveis** em tempo de execução.



Problema

Uma loja virtual calcula o **frete** de um pedido. Hoje a regra vive dentro de uma única classe, decidida por um `if/else` que cresce a cada nova modalidade:

- o **cliente** escolhe a modalidade (Sedex, PAC, Retirada...)
- cada modalidade tem uma **fórmula diferente**
- adicionar uma nova modalidade exige **editar** a classe existente



```
public class CalculadoraFrete {  
    public double calcular(String modalidade, double peso) {  
        if (modalidade.equals("sedex")) {  
            return 20.0 + peso * 3.0;  
        } else if (modalidade.equals("pac")) {  
            return 10.0 + peso * 1.5;  
        } else if (modalidade.equals("retirada")) {  
            return 0.0;  
        }  
        throw new IllegalArgumentException("Modalidade inválida");  
    }  
}
```

Essa solução tem problemas sérios:

- **viola o OCP**: cada nova modalidade **altera** o método `calcular`
- a classe acumula **todos** os algoritmos e vira um "monstro"
- as regras ficam **difíceis de testar** isoladamente
- o cliente precisa passar **strings mágicas** (`"sedex"` , `"pac"`)
- reusar uma fórmula em outro contexto vira **copiar e colar**

Solução

Em vez de decidir por `if/else`, vamos **extrair cada algoritmo** para uma classe que segue uma **interface comum** e entregá-la ao contexto:

- **Strategy**: interface que define o método do algoritmo
- **ConcreteStrategy**: cada algoritmo em uma classe própria
- **Context**: mantém uma referência à estratégia e **delega** a ela
- O contexto **não sabe** qual algoritmo está por baixo

A ideia central:

- cada algoritmo fica **isolado e testável**
- o contexto **delega** o cálculo à estratégia atual
- trocar de algoritmo = **trocar o objeto**, não o contexto
- novos algoritmos entram **sem tocar** nos antigos



Implementação

1. Primeiro, definimos a **interface comum** das estratégias:

```
public interface EstrategiaFrete {  
    double calcular(double peso);  
}
```

2. Criamos as **estratégias concretas**: cada modalidade é um algoritmo isolado:

```
public class FreteSedex implements EstrategiaFrete {  
    public double calcular(double peso) {  
        return 20.0 + peso * 3.0;  
    }  
}  
  
public class FretePac implements EstrategiaFrete {  
    public double calcular(double peso) {  
        return 10.0 + peso * 1.5;  
    }  
}
```

3. Criamos o **contexto**, que guarda a estratégia e **delega** a ela:

```
public class Pedido {  
    private EstrategiaFrete estrategia;  
  
    public Pedido(EstrategiaFrete estrategia) {  
        this.estrategia = estrategia;  
    }  
  
    public void setEstrategia(EstrategiaFrete estrategia) {  
        this.estrategia = estrategia;  
    }  
  
    public double totalFrete(double peso) {  
        return estrategia.calcular(peso);  
    }  
}
```

Utilização

O cliente escolhe a estratégia e pode **trocá-la em tempo de execução**:

```
Pedido pedido = new Pedido(new FreteSedex());  
System.out.println(pedido.totalFrete(2.0)); // 26.0  
  
pedido.setEstrategia(new FretePac());          // troca em runtime  
System.out.println(pedido.totalFrete(2.0)); // 13.0
```

- o Pedido **não conhece** as fórmulas concretas
- trocar a modalidade **não altera** a classe Pedido
- cada estratégia é **testável** de forma independente
- novas fórmulas entram **sem recompilar** o contexto

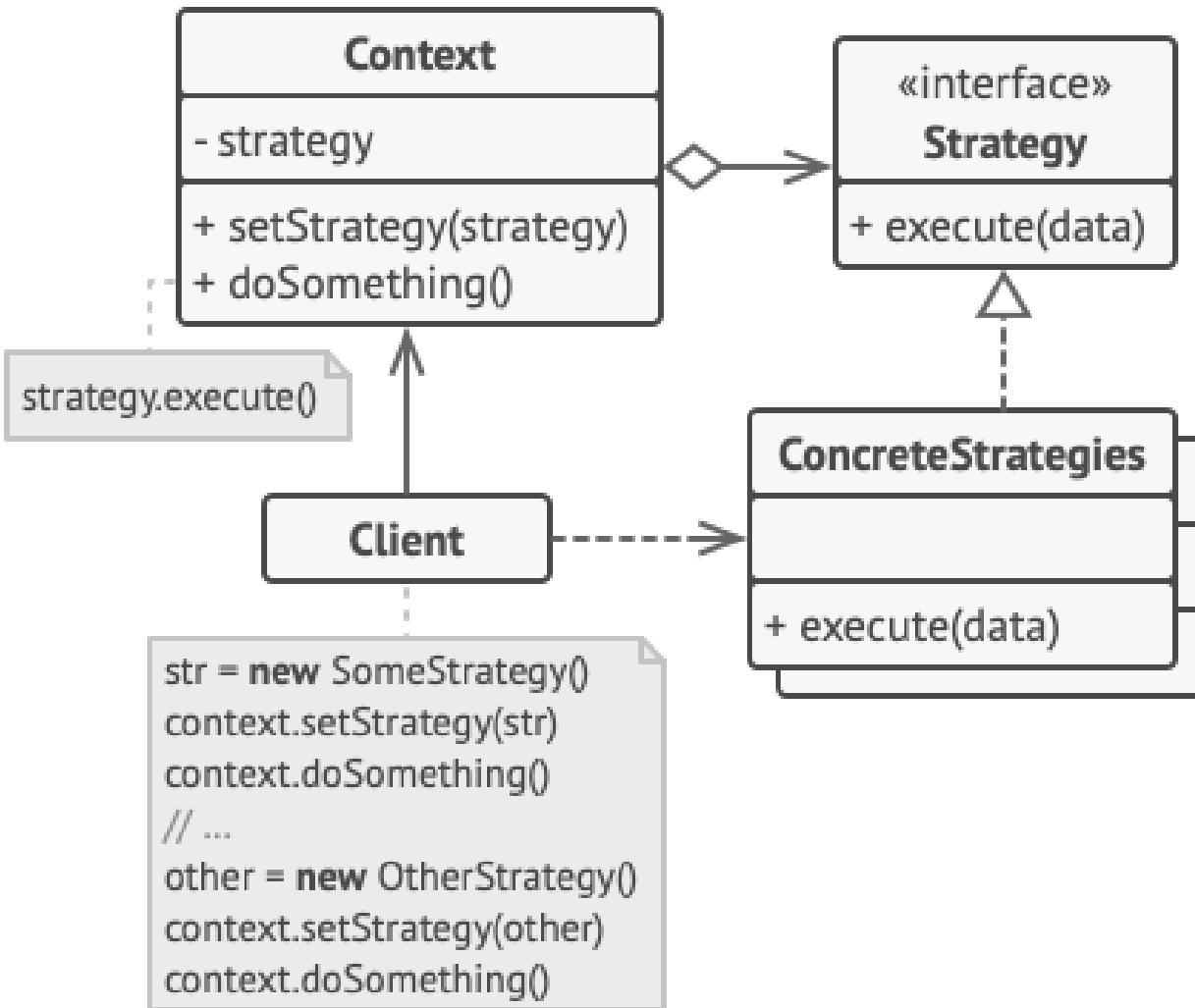
Uso em outros paradigmas

O Padrão Strategy pode ser resolvido de forma diferente em linguagens que suportam **funções de primeira classe** (como JavaScript, Python, Ruby):

```
pedido.setEstrategia((peso) => 20 + peso * 3); // Sedex  
pedido.setEstrategia((peso) => 10 + peso * 1.5); // PAC  
pedido.setEstrategia((peso) => 0); // Retirada
```

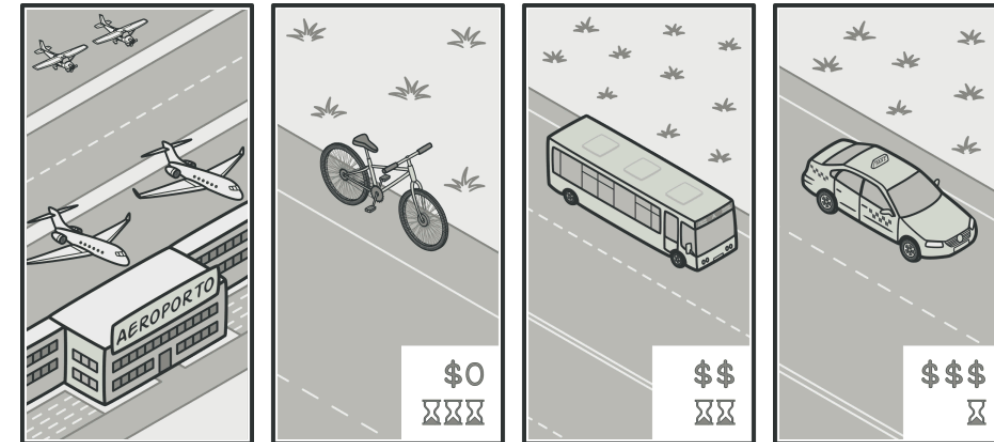
Diagramas

- **Context**: mantém a referência à estratégia e delega a ela
- **Strategy**: interface comum de todos os algoritmos
- **ConcreteStrategy**: implementação de cada algoritmo
- **Client**: escolhe a estratégia e a entrega ao contexto



Analogias

- **Planejar uma viagem:**
 - para o mesmo destino, dá para ir **a pé, de carro ou de transporte público**
 - o **objetivo é o mesmo** (chegar ao destino), o **algoritmo** de deslocamento muda
 - trocar o meio de transporte **não muda a viagem**, só como ela é feita



Aplicações

- Quando existem **vários algoritmos** para a mesma tarefa e o cliente escolhe qual usar
- Quando a classe está cheia de `if/else` / `switch` decidindo comportamento
- Quando os algoritmos precisam **variar independentemente** de quem os usa

Exemplos típicos:

- **cálculo de frete**, impostos e descontos
- **métodos de pagamento** (Pix, cartão, boleto)
- **ordenadores**: `Comparator` do Java
- **compressão, criptografia e formatos** de arquivo

Prós

- **OCP**: novos algoritmos entram sem alterar o contexto
- **SRP**: cada algoritmo fica na sua própria classe
- Troca de comportamento em **runtime**
- **Desacopla** o cliente das regras concretas
- Elimina `if/else` gigantes

Contras

- **Mais classes:** uma por algoritmo (pode virar muitas)
- O cliente precisa **conhecer** as estratégias disponíveis para escolher
- Se o algoritmo for **único e estável**, o contexto extra é **overengineering**
- A comunicação de dados entre contexto e estratégia exige cuidado

Regra: use quando houver **variação real de algoritmos** que o cliente escolhe.

Resumo

- **Strategy**: encapsula uma família de **algoritmos intercambiáveis**
- O **contexto delega**; o cliente **escolhe** e pode trocar em runtime
- Elimina `if/else` de decisão e respeita **OCP/SRP**
- Muito usado em **frete, pagamentos e comparações** (`Comparator`)
- Diferença do **Bridge**: Bridge separa **abstração × plataforma**; Strategy apenas muda o **algoritmo** de uma tarefa