2025-2 / Lorenzon : OO - Design Patterns - Strategy

Design Patterns – Strategy

O Strategy é mais um dos padrões **comportamentais** do catálogo GoF (Gang of Four). Ele é usado quando precisamos definir uma família de algoritmos, encapsulá-los e torná-los intercambiáveis em tempo de execução. O objetivo é permitir que o algoritmo varie independentemente dos clientes que o utilizam.

A ideia central é separar o comportamento (estratégia) do contexto (objeto que usa a estratégia). Isso favorece a flexibilidade e a manutenção, evitando códigos cheios de if/else ou switch para tratar diferentes variações de um mesmo comportamento.

Utilização

Sua utilização é adequada quando:

- Há várias formas de realizar uma mesma tarefa e queremos trocar essas formas dinamicamente;
- Para eliminar condicionais complexos (ex.: muitos if/else ou switch);
- Quando desejamos aplicar o princípio Open/Closed (OCP): aberto para extensão, fechado para modificação;
- Quando precisamos aplicar polimorfismo para comportamentos diferentes que compartilham uma interface comum.

Benefícios

Alguns benefícios claros de utilização do padrão

- Desacoplamento: o contexto não conhece detalhes das estratégias, apenas a interface;
- Extensibilidade: novas estratégias podem ser adicionadas sem modificar o código existente;
- Flexibilidade: a estratégia pode ser trocada em tempo de execução;
- Reuso de código: várias classes podem compartilhar a mesma estratégia.

Um exemplo de uso em Java

Primeiramente, vamos avaliar um código onde se dá origem a aplicação do padrão:

```
public class CarrinhoDeCompras {
    private String tipoPagamento; // "cartao", "pix", "boleto"

    public void setTipoPagamento(String tipoPagamento) {
        this.tipoPagamento = tipoPagamento;
    }

    public void finalizarCompra(double valor) {
        if (tipoPagamento == null) {
            System.out.println("Nenhuma forma de pagamento selecionada.");
            return;
        }

        if (tipoPagamento.equalsIgnoreCase("cartao")) {
            System.out.println("Pagamento de R$" + valor +
                               " realizado com Cartão de Crédito.");
        } else if (tipoPagamento.equalsIgnoreCase("pix")) {
            System.out.println("Pagamento de R$" + valor +
                               " realizado via Pix.");
        } else if (tipoPagamento.equalsIgnoreCase("boleto")) {
            System.out.println("Pagamento de R$" + valor +
                               " realizado com Boleto.");
        } else {
            System.out.println("Forma de pagamento inválida.");
        }
    }
}
```

Nesta versão de Código encontramos alguns problemas como baixa flexibilidade para adicionar uma nova forma de pagamento (ex.: PayPal), seria necessário **alterar a classe CarrinhoDeCompras**, quebrando o princípio **OCP (Open/Closed)** do SOLID. A cada novo pagamento, o número de if/else cresce, ficando sempre tudo na mesma classe. Uma alteração num meio de pagamento exige alterar o controle de todas.

Então, por este exemplo, temos uma aplicação de pagamentos que pode ser feita por diferentes meios: Cartão de Crédito, Pix ou Boleto.

O contrato das estratégias

```
public interface PagamentoStrategy {
    void pagar(double valor);
}
```

As estratégias concretas

```
public class PagamentoCartaoCredito implements PagamentoStrategy {
    @Override
    public void pagar(double valor) {
        System.out.println("Pagamento de R$" + valor +
                           " realizado com Cartão de Crédito.");
    }
}
```

```
public class PagamentoPix implements PagamentoStrategy {
    @Override
    public void pagar(double valor) {
        System.out.println("Pagamento de R$" + valor + " realizado via Pix.");
    }
}
```

```
public class PagamentoBoleto implements PagamentoStrategy {
    @Override
    public void pagar(double valor) {
        System.out.println("Pagamento de R$" + valor +
                           " realizado com Boleto.");
    }
}
```

Classe contexto que deriva as estratégias

```
public class CarrinhoDeCompras {
    private PagamentoStrategy estrategiaPagamento; //associação

    // Permite trocar a estratégia dinamicamente
    public void setEstrategiaPagamento(PagamentoStrategy estrategiaPagamento) {
        this.estrategiaPagamento = estrategiaPagamento;
    }

    public void finalizarCompra(double valor) {
        if (estrategiaPagamento == null) {
            System.out.println("Nenhuma forma de pagamento selecionada.");
        } else {
            estrategiaPagamento.pagar(valor);
        }
    }
}
```

Uma aplicação de exemplo

```
public class Strategy {
    public static void main(String[] args) {
        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();
        // Pagamento com Cartão
        carrinho.setEstrategiaPagamento(new PagamentoCartaoCredito());
        carrinho.finalizarCompra(250.0);
        // Troca dinâmica para Pix
        carrinho.setEstrategiaPagamento(new PagamentoPix());
        carrinho.finalizarCompra(100.0);
        // Troca dinâmica para Boleto
        carrinho.setEstrategiaPagamento(new PagamentoBoleto());
        carrinho.finalizarCompra(300.0);
    }
}
```

- Interface comum (Strategy): define o contrato de comportamento;
- Classes concretas: implementam diferentes formas de realizar a ação;
- Contexto: delega a execução do comportamento para a estratégia escolhida;
- Troca dinâmica: mostra a aplicação prática de polimorfismo em tempo de execução;
- Princípio "Composição sobre Herança": o comportamento é **injetado** no contexto em vez de herdado, por associação.

Injeção (Associação / Composição)

- Em vez de herdar, a classe **recebe uma referência** a um objeto que sabe como realizar o comportamento;
- Isso é uma **associação** (ou composição, dependendo da relação de dependência);
- No Strategy, o CarrinhoDeCompras não sabe como pagar, ele **recebe (injeta)** uma PagamentoStrategy.