

# STRATEGY




5by5 | Soluções em  
Sistemas



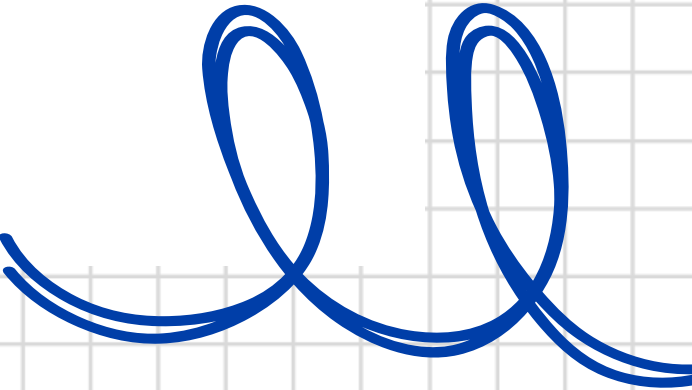
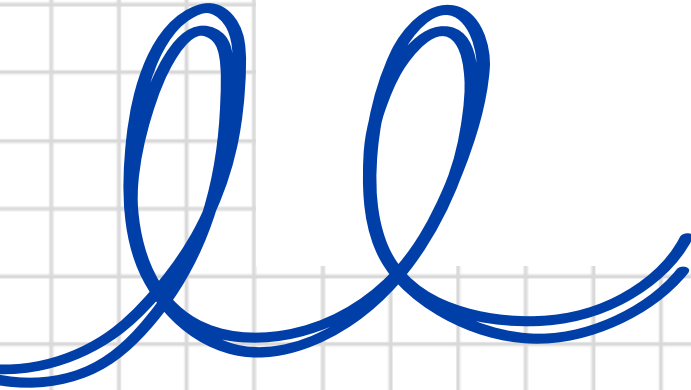
Edenilson Silva Garcia Junior



# SITUAÇÃO-PROBLEMA



**Imagine um sistema de e-commerce onde é necessário fazer cálculos diferentes à depender do tipo de frete escolhido pela empresa no momento, no qual podem ser: aéreo, terrestre, marítimo, express.**





# Isso é fácil!

**Basta criar uma classe pedido e fazer os calculos dos fretes ali mesmo!**

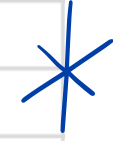
```
1  internal class Pedido
2  {
3      private string _tipoFrete;
4
5      public Pedido(string t) { _tipoFrete = t; }
6
7      public double CalcularFrete()
8      {
9          return _tipoFrete switch
10         {
11             "Terrestre" => CalcularFreteTerreste(),
12             "Aereo" => CalcularFreteAereo(),
13             "Maritmo" => CalcularFreteMaritmo(),
14             "Express" => CalcularFreteExpress(),
15             _ => throw new Exception("Tipo de frete não encontrado")
16         };
17     }
18
19     private double CalcularFreteTerreste() => 10.0f;
20
21     private double CalcularFreteAereo() => 20.0f;
22
23     private double CalcularFreteMaritmo() => 15.0f;
24
25     private double CalcularFreteExpress() => 25.0f;
26 }
```



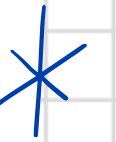
```
1 static void Main(string[] args)
2 {
3     Console.WriteLine("====|e-commerce|====");
4
5     Pedido p1 = new("Terrestre");
6     Console.WriteLine($"Frete terrestre: {p1.CalcularFrete()}");
7
8     Pedido p2 = new("Aereo");
9     Console.WriteLine($"Frete aéreo: {p2.CalcularFrete()}");
10
11    Pedido p3 = new("Maritimo");
12    Console.WriteLine($"Frete marítimo: {p3.CalcularFrete()}");
13
14    Pedido p4 = new("Express");
15    Console.WriteLine($"Frete express: {p4.CalcularFrete()}");
16 }
```

```
====|e-commerce|====
Frete terrestre: 10
Frete aéreo: 20
Frete marítimo: 15
Frete express: 25
```

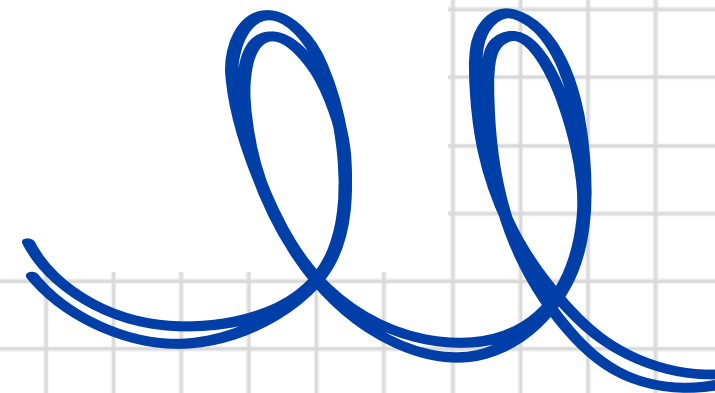
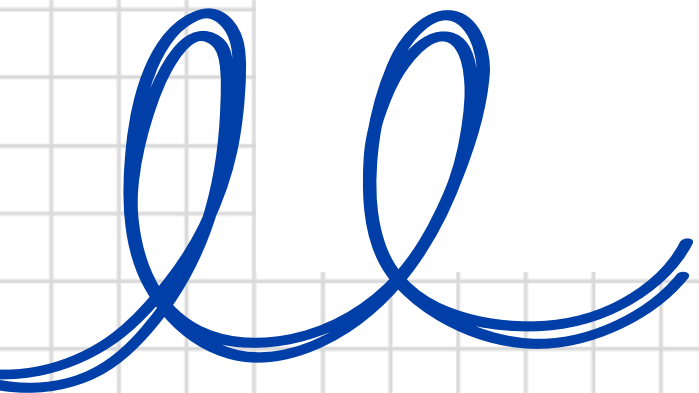


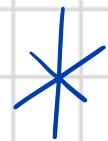


# Funcionou, e agora?



**A empresa gostou tanto da implementação do sistema que decidiu expandir seus negócios e adicionar fretes Ferroviários.**





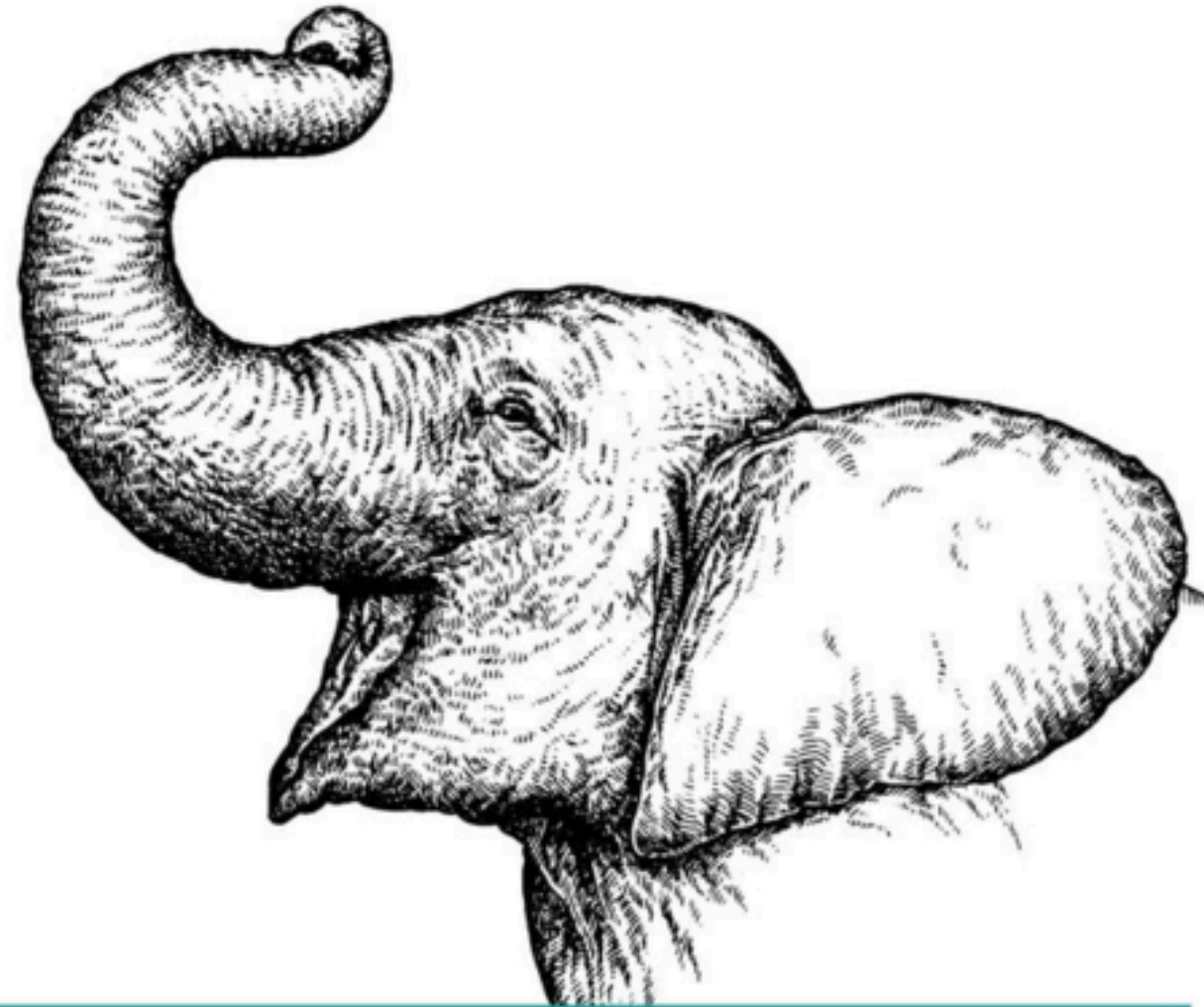
**Basta modificar a  
classe Pedido e  
adicionar o frete!**

```
1  internal class Pedido
2  {
3      private string _tipoFrete;
4
5      public Pedido(string t) { _tipoFrete = t; }
6
7      public double CalcularFrete()
8      {
9          return _tipoFrete switch
10         {
11             "Terrestre" => CalcularFreteTerrestre(),
12             "Aereo" => CalcularFreteAereo(),
13             "Maritmo" => CalcularFreteMaritmo(),
14             "Express" => CalcularFreteExpress(),
15             "Ferroviario" => CalcularFreteFerroviario(),
16             _ => throw new Exception("Tipo de frete não encontrado")
17         };
18     }
19
20     private double CalcularFreteTerrestre() => 10.0f;
21
22     private double CalcularFreteAereo() => 20.0f;
23
24     private double CalcularFreteMaritmo() => 15.0f;
25
26     private double CalcularFreteExpress() => 25.0f;
27
28     private double CalcularFreteFerroviario() => 12.0f;
29 }
```





*The answer to every programming question ever conceived*



# It Depends

*The Definitive Guide*


O RLY?

@ThePracticalDev



# SITUAÇÃO-PROBLEMA

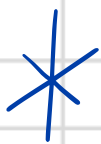
**Ao fazer essa adição, você, como programador, aumenta a complexidade do código, conseqüentemente aumentando o acoplamento de código entre as camadas.**



**Acoplamento: grau de dependência entre diferentes módulos ou componentes de um sistema de software.**





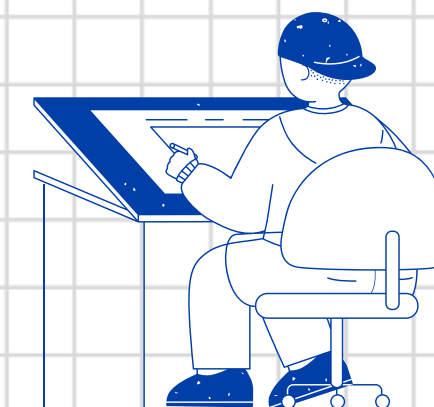


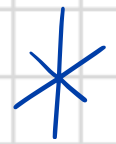
# STRATEGY PATTERN

Definição(refactoring.guru) : O Strategy é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

Definição de intercambiáveis: significa que diferentes algoritmos ou comportamentos podem ser trocados entre si de forma flexível e dinâmica. Em outras palavras, você pode substituir um comportamento por outro facilmente, sem precisar alterar a estrutura ou a lógica do restante do sistema.

Esse conceito do padrão está relacionado com o Princípio da Inversão de Dependência (DIP) do SOLID, resumidamente, devemos depender de abstrações ao invés de implementações concretas, isso diminui o acoplamento entre as camadas do sistema

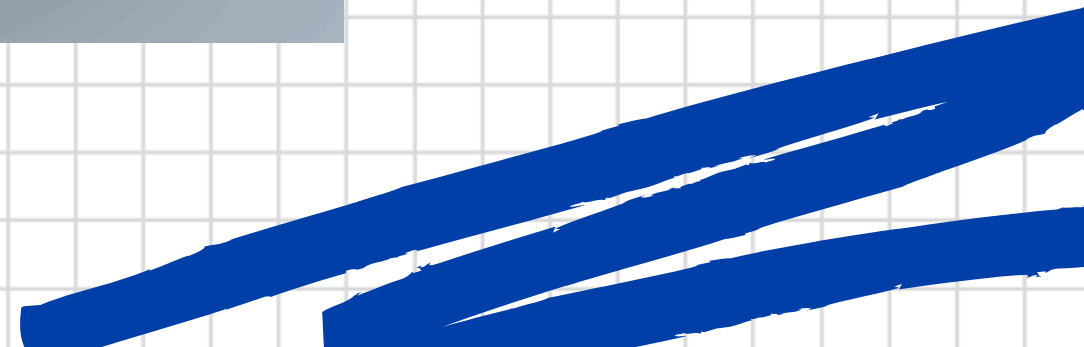




# Abstrair o método de calcular frete



```
1  public interface IFreteStrategy  
2  {  
3      double CalcularFrete();  
4  }
```



# Abstrair o método de calcular frete

```
1 public class FreteTerrestre : IFreteStrategy
2 {
3     public double CalcularFrete() => 10.0f;
4 }
5
6
7 public class FreteAereo : IFreteStrategy
8 {
9     public double CalcularFrete() => 20.0f;
10 }
11
12
13 public class FreteMaritimo : IFreteStrategy
14 {
15     public double CalcularFrete() => 15.0f;
16 }
17
18
19 public class FreteExpress : IFreteStrategy
20 {
21     public double CalcularFrete() => 25.0f;
22 }
23
24
25 public class FreteFerroviario : IFreteStrategy
26 {
27     public double CalcularFrete() => 12.0f;
28 }
```

```
1 public class Pedido
2 {
3     public IFreteStrategy TipoFrete { get; }
4
5     public Pedido(IFreteStrategy t)
6     {
7         TipoFrete = t;
8     }
9 }
```

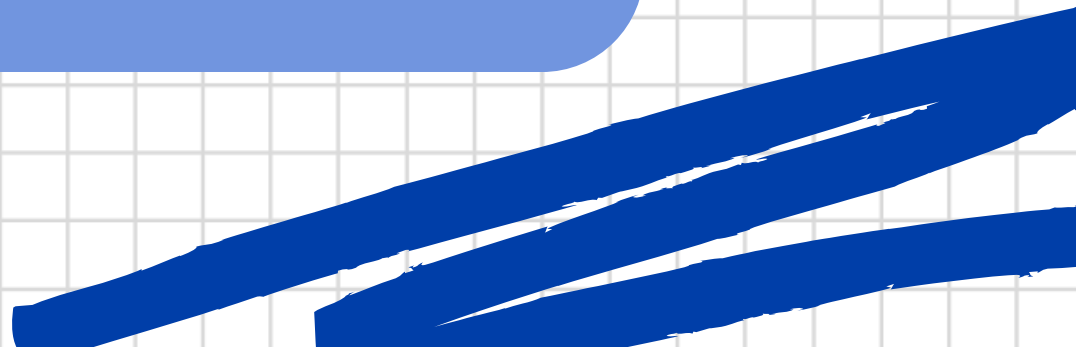
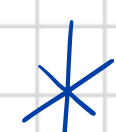
# Main()

```
1 internal class Program
2 {
3     static void Main(string[] args)
4     {
5         Console.WriteLine("====|e-commerce|====");
6
7         var p1 = new Pedido(new FreteTerrestre());
8         Console.WriteLine($"Frete terrestre: {p1.TipoFrete.CalcularFrete()}");
9
10        var p2 = new Pedido(new FreteAereo());
11        Console.WriteLine($"Frete aéreo: {p2.TipoFrete.CalcularFrete()}");
12
13        var p3 = new Pedido(new FreteMaritimo());
14        Console.WriteLine($"Frete marítimo: {p3.TipoFrete.CalcularFrete()}");
15
16        var p4 = new Pedido(new FreteExpress());
17        Console.WriteLine($"Frete express: {p4.TipoFrete.CalcularFrete()}");
18
19        var p5 = new Pedido(new FreteFerroviario());
20        Console.WriteLine($"Frete ferroviário: {p5.TipoFrete.CalcularFrete()}");
21    }
22 }
```



# Resumo

**O padrão Strategy ajuda a manter o código limpo e flexível, permitindo que diferentes algoritmos (como o cálculo do frete) sejam intercambiáveis sem modificar o cliente (neste caso, a classe Pedido). Isso é especialmente útil quando você precisa adicionar novos comportamentos ou alterar os existentes sem afetar o restante do sistema.**



# Fim!



## Fontes:

[refactoring.guru/pt-br/design-patterns/strategy](https://refactoring.guru/pt-br/design-patterns/strategy)  
[youtube.com/watch?v=WPdrnuSHAQs](https://youtube.com/watch?v=WPdrnuSHAQs)  
[youtube.com/watch?v=IDSMmBmE\\_lk&t=288s](https://youtube.com/watch?v=IDSMmBmE_lk&t=288s) ( Exemplo em java =D )

## Código-fonte

Disponível em:  
[github.com/edenilsonjunior/5by5-Strategy](https://github.com/edenilsonjunior/5by5-Strategy)