

Strategy

Padrões de Projeto
Comportamentais I

Prof. Me. Jefferson Passerini





– Strategy – Padrões Comportamentais



- 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.**
- O Strategy encapsula cada um dos algoritmos dentro da família e os torna intercambiáveis.**
- Permite que o algoritmo varie independentemente dos clientes que o utilizam.**





– Strategy – Padrões Comportamentais



- O padrão Strategy aprimora a comunicação entre os objetos, pois passa a existir uma distribuição de responsabilidades.**
- O objetivo é representar os comportamentos de um objeto por meio de uma família de algoritmos que os implementam.**

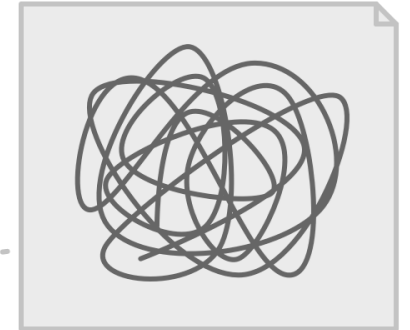
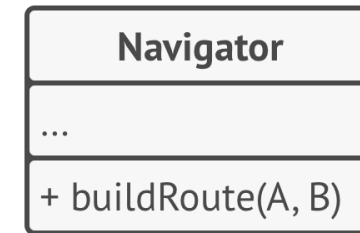


– Problema

- Um dia você decide criar **uma aplicação de navegação para viajantes casuais**. A aplicação estava centrada em um mapa bonito que ajudava os usuários a se orientarem rapidamente em uma cidade.
- **Uma das funcionalidades mais pedidas para a aplicação era o planejamento automático de rotas**. Um usuário deveria ser capaz de entrar com um endereço e ver a rota mais rápida no mapa.
- **A primeira versão da aplicação podia apenas construir rotas sobre rodovias**, e isso agradou muito quem viaja de carro;
- Porém aparentemente, nem todo mundo dirige em suas férias, então a **próxima atualização vocês adicionou uma opção para permitir que as pessoas usem o transporte público**.

– Problema

- Contudo isso foi apenas o começo. **Mais tarde você planejou adicionar um construtor de rotas para ciclistas**, e ainda outra opção para construir rotas até todas as atrações turísticas de uma cidade.
- Embora **da perspectiva de negócio a aplicação tenha sido sucesso**, a parte técnica causou a **você muitas dores de cabeça**. Cada vez que você adicionava um novo algoritmo de roteamento, a classe principal do navegador dobrava de tamanho.
- + Em determinado momento, **o monstro se tornou algo muito difícil de se manter**.





– Problema



- **Qualquer mudança a um dos algoritmos**, seja uma simples correção de bug ou um pequeno ajuste no valor das ruas, **afetava toda a classe aumentando a chance de criar um erro no código já existente.**
- Além disso, o trabalho em equipe se tornou ineficiente.
- Seus companheiros de equipe, que foram contratados após o bem sucedido lançamento do produto, se queixavam que gastavam muito tempo resolvendo conflitos de fusão.
- Implementar novas funcionalidades necessitava mudanças na classe gigantesca, conflitando com os código criados por outras pessoas.





– Solução



- O padrão Strategy sugere que você pegue uma classe que faz algo específico em diversas maneiras diferentes e extraia todos esses algoritmos para classes separadas chamadas estratégias.
- A classe original, chamada contexto, deve ter um campo para armazenar uma referência para uma dessas estratégias. O contexto delega o trabalho para um objeto estratégia ao invés de executá-lo por conta própria.
- O contexto não é responsável por selecionar um algoritmo apropriado para o trabalho, ao invés disso, o cliente para a estratégia desejada para o contexto.



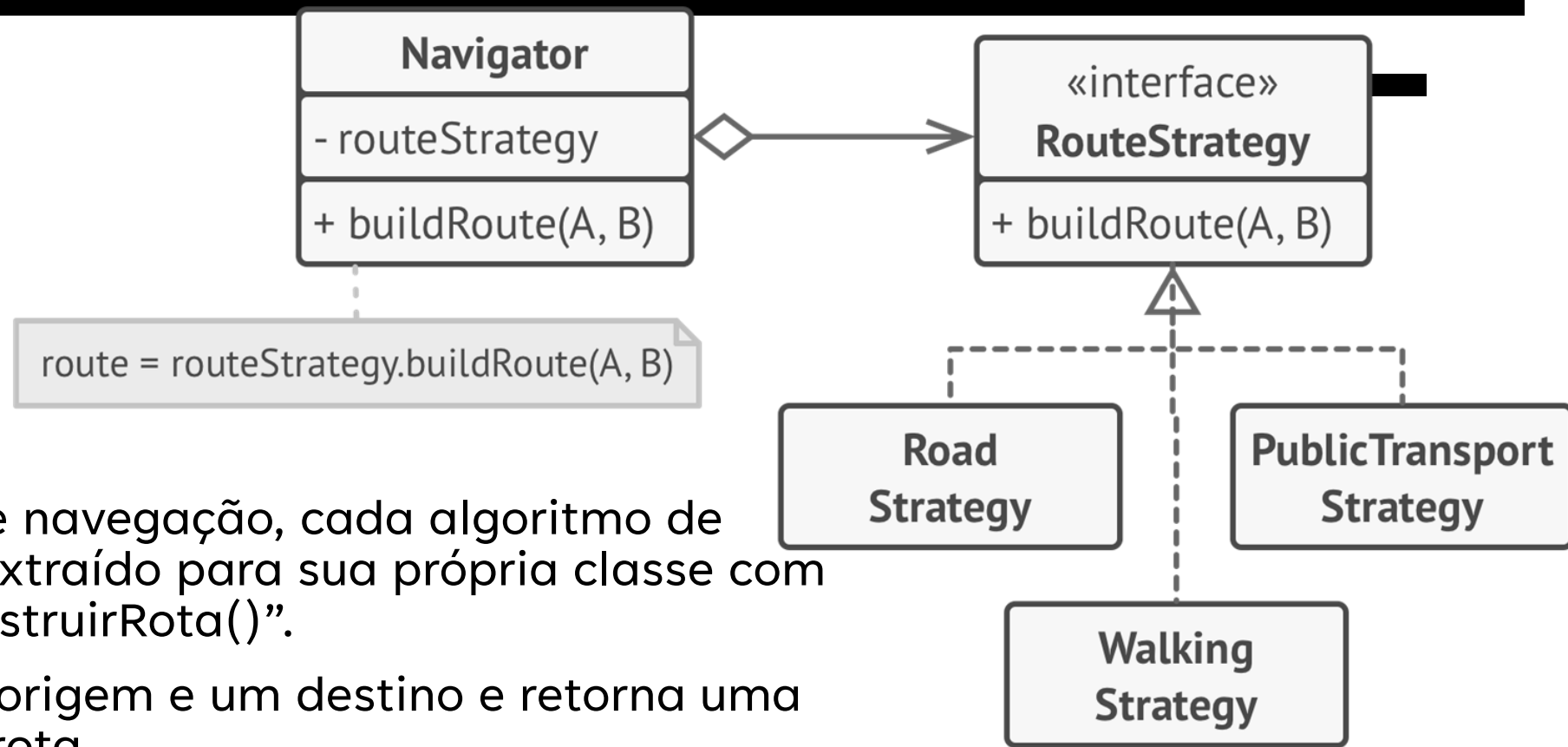


– Solução



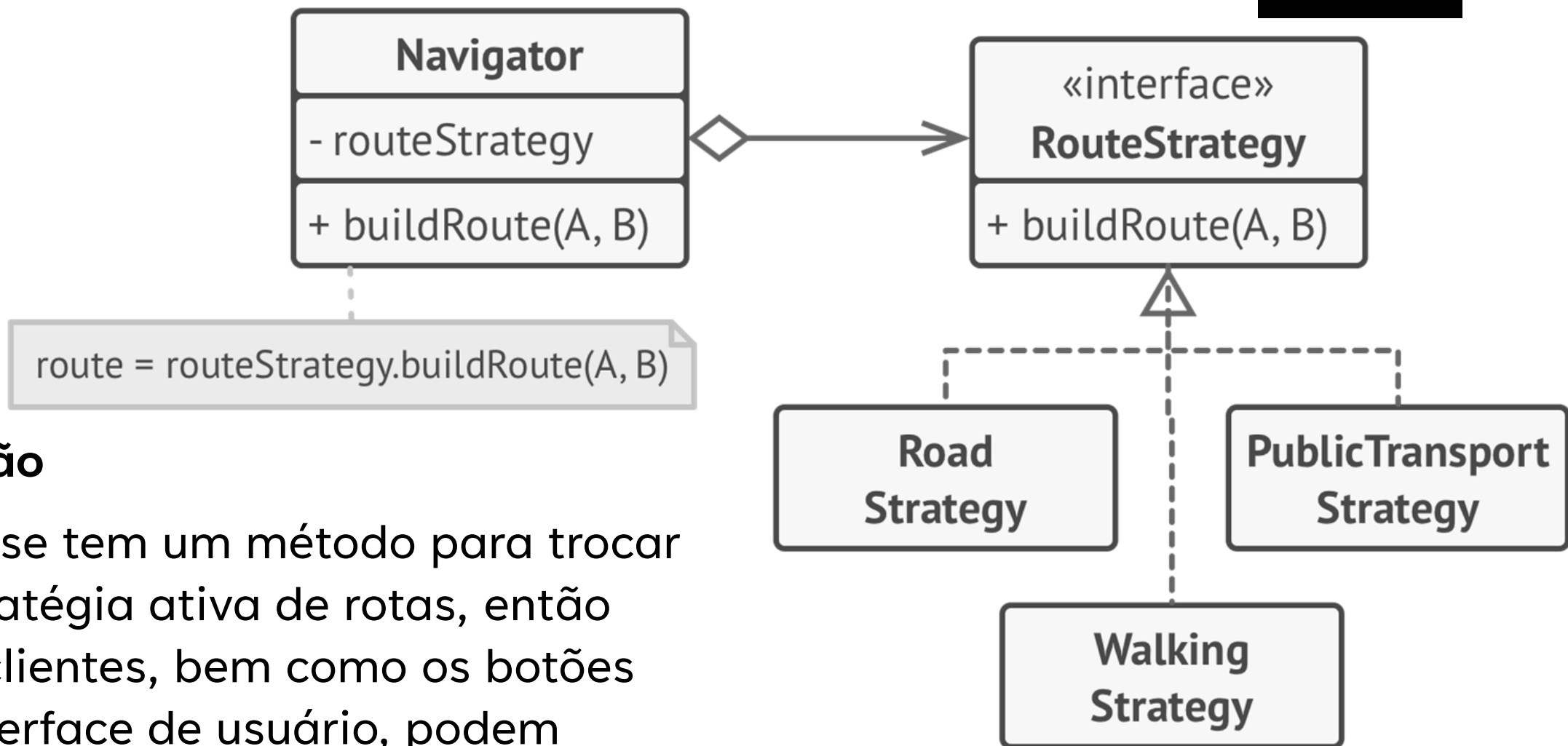
- Na verdade o contexto não sabe muito sobre as estratégias. Ele trabalha com todas elas através de uma interface genérica, que somente expõe um único método para acionar o algoritmo encapsulado dentro da estratégia selecionada.
- Desta forma o contexto se torna independente das estratégias concretas, então você pode adicionar novos algoritmos ou modificar os existentes sem modificar o código ou outras estratégias.





– Solução

- Em nossa aplicação de navegação, cada algoritmo de roteamento pode ser extraído para sua própria classe com um único método “construirRota()”.
- O método aceita uma origem e um destino e retorna uma coleção de pontas da rota.
- Mesmo dando os mesmos argumentos, cada classe de roteamento pode construir uma rota diferente, a classe navegadora principal não se importa com qual algoritmo está selecionado uma vez que seu trabalho é renderizar os pontos no mapa.

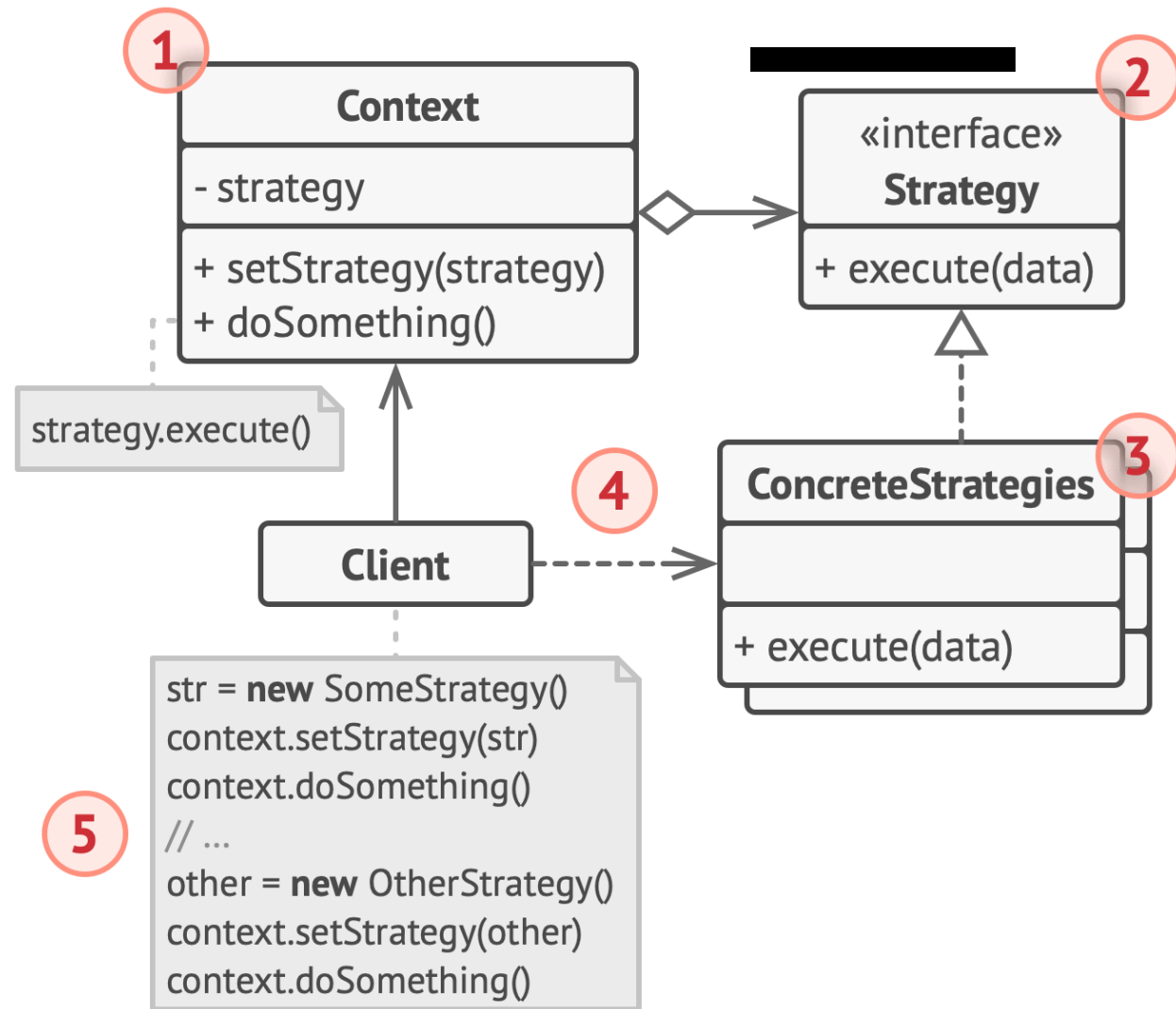


– Solução

- A classe tem um método para trocar a estratégia ativa de rotas, então seus clientes, bem como os botões da interface de usuário, podem substituir o comportamento de rotas selecionado por um outro.

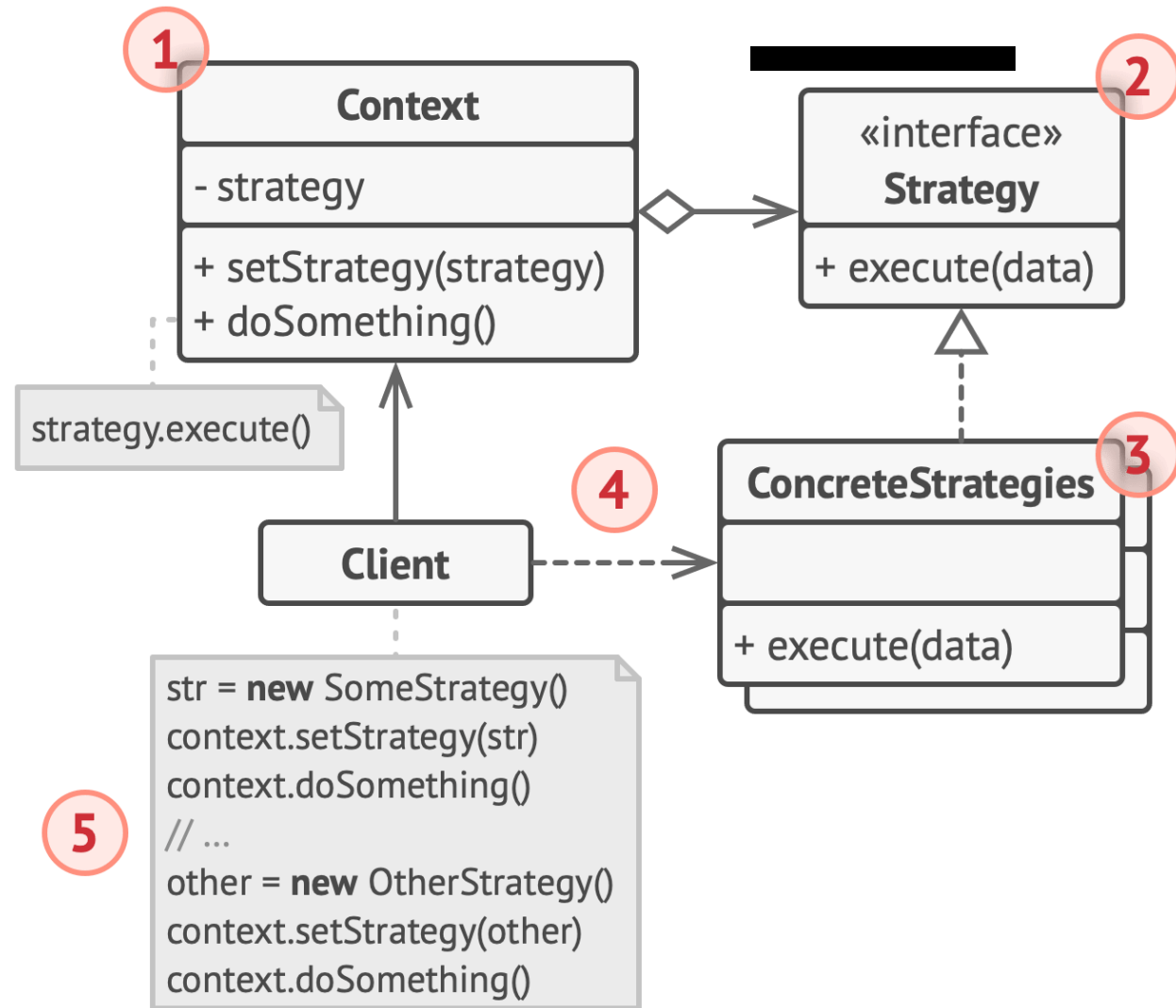
– Estrutura

- 1 – O **Contexto** mantém uma referência para uma das estratégias concretas e se comunica com esse objeto através da interface da estratégia.
- 2 – A interface **Estratégia** é comum à todas as estratégias concretas. Ela declara um método que o contexto usa para executar uma estratégia.
- 3 – **Estratégias Concretas**
+ implementam diferentes variações de um algoritmo que o contexto usa.



– Estrutura

- 4 – O **Contexto** chama o método de execução no objeto estratégia ligado cada vez que ele precisa rodar um algoritmo. O contexto não sabe qual o tipo de estratégia ele está trabalhando ou como o algoritmo é executado.
- 5 – O **Cliente** cria um objeto estratégia específico e passa ele para o contexto. O contexto expõe um setter que permite o cliente mudar a estratégia associada com contexto durante a execução.





– Aplicabilidade



- **Utilize o padrão Strategy quando você quer usar diferentes variantes de um algoritmo dentro de um objeto e ser capaz de trocar de um algoritmo para outro durante a execução.**
- O padrão Strategy permite que você altere indiretamente o comportamento de um objeto durante a execução ao associá-lo com diferentes sub-objetos que pode fazer sub-tarefas específicas em diferentes formas.





– Aplicabilidade



- **Utilize o Strategy quando você tem muitas classes parecidas que somente diferem na forma que elas executam algum comportamento.**
- O padrão Strategy permite que você extraia o comportamento variante para uma hierarquia de classe separada e combine as classes originais em uma, portando reduzindo código duplicado.





– Aplicabilidade



- **Utilize o padrão para isolar a lógica do negócio de uma classe dos detalhes de implementação de algoritmos que podem não ser tão importantes no contexto da lógica.**
- O padrão Strategy permite que você isole o código, dados internos, e dependências de vários algoritmos do restante do código. Vários clientes podem obter uma simples interface para executar os algoritmos e trocá-los durante a execução do programa.





– Aplicabilidade



- **Utilize o padrão quando sua classe tem um operador condicional muito grande que troca entre diferentes variantes do mesmo algoritmo.**
- O padrão Strategy permite que você se livre dessa condicional ao extrair todos os algoritmos para classes separadas, todos eles implementando a mesma interface. O objeto original delega a execução de um desses objetos, ao invés de implementar todas as variantes do algoritmo.





– Prós

- Você pode trocar algoritmos usados dentro de um objeto durante a execução.
- Você pode isolar os detalhes de implementação de um algoritmo do código que usa ele.
- Você pode substituir a herança por composição.
- Princípio aberto/fechado. Você pode introduzir novas estratégias sem mudar o contexto.





– Contrás



- Se você só tem um par de algoritmos e eles raramente mudam, não há motivo real para deixar o programa mais complicado com novas classes e interfaces que vêm junto com o padrão.
- Os Clientes devem estar cientes das diferenças entre as estratégias para serem capazes de selecionar a adequada.
- Muitas linguagens de programação modernas tem suporte do tipo funcional que permite que você implemente diferentes versões de um algoritmo dentro de um conjunto de funções anônimas. Então você poderia usar essas funções exatamente como se estivesse usando objetos estratégia, mas sem inchar seu código com classes e interfaces adicionais.



– Relação com outros padrões


- O **Bridge**, **State**, **Strategy** (e de certa forma o **Adapter**) têm estruturas muito parecidas. De fato, todos esses padrões estão baseados em composição, o que é delegar o trabalho para outros objetos. Contudo, eles todos resolvem problemas diferentes. Um padrão não é apenas uma receita para estruturar seu código de uma maneira específica. Ele também pode comunicar a outros desenvolvedores o problema que o padrão resolve.
- O **Decorator** permite que você mude a pele de um objeto, enquanto o **Strategy** permite que você mude suas entranhas.





– Relação com outros padrões



- O **Command** e o **Strategy** podem ser parecidos porque você pode usar ambos para parametrizar um objeto com alguma ação. Contudo, eles têm propósitos bem diferentes.
 - Você pode usar o Command para converter qualquer operação em um objeto. Os parâmetros da operação se transformam em campos daquele objeto. A conversão permite que você atrase a execução de uma operação, transforme-a em uma fila, armazene o histórico de comandos, envie comandos para serviços remotos, etc.
 - Por outro lado, o Strategy geralmente descreve diferentes maneiras de fazer a mesma coisa, permitindo que você troque esses algoritmos dentro de uma única classe contexto.
- 



– Relação com outros padrões



- O **Template Method** é baseado em herança: ele permite que você altere partes de um algoritmo ao estender essas partes em subclasses. O **Strategy** é baseado em composição: você pode alterar partes do comportamento de um objeto ao suprir ele como diferentes estratégias que correspondem a aquele comportamento. O Template Method funciona a nível de classe, então é estático. O Strategy trabalha a nível de objeto, permitindo que você troque os comportamentos durante a execução.





– Relação com outros padrões

- O **State** pode ser considerado como uma extensão do **Strategy**.
 - Ambos padrões são baseados em composição: eles mudam o comportamento do contexto ao delegar algum trabalho para objetos auxiliares.
 - O Strategy faz esses objetos serem completamente independentes e alheios entre si.]
 - Contudo, o State não restringe dependências entre estados concretos, permitindo que eles alterem o estado do contexto à vontade.



Exemplo Strategy C# e Java

Padrões de Projeto
Comportamentais I

Prof. Me. Jefferson Passerini



– Problema a ser implementado

• Cenário 1

- Imagine que você está desenvolvendo um software para um e-commerce;
- Neste software um pedido pode ser enviado através de 2 tipos de frete:
 - Frete Comum → 5% do valor do pedido
 - Frete Expresso → 10% do valor do pedido
- E os métodos **calculaFreteComum()** e **calculaFreteExpresso()** são responsáveis por esses cálculos.

Pedido
- valor: float
+ getValor(): float
+ setValor(): void
+ calculaFreteComum(): float
+ calculaFreteExpresso(): float

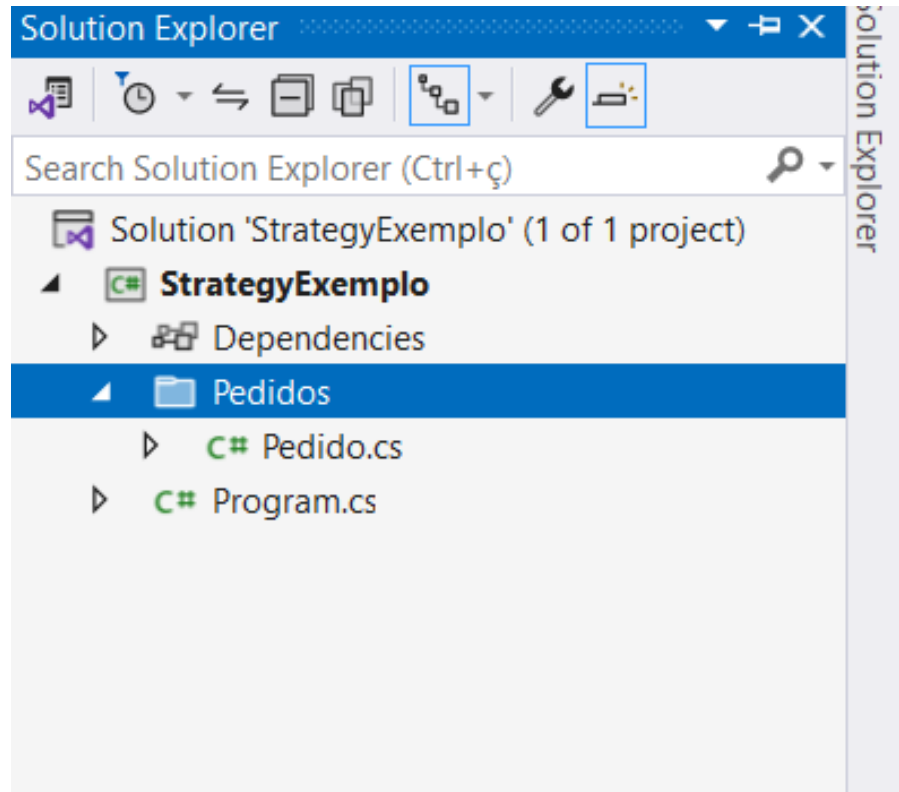


• Cenário 1

- Crie um novo projeto em Java ou C# (tipo console application)
- **Crie um Pacote (Pasta) denominada Pedidos e implemente dentro deste pacote a classe Pedido.**
- O main de nossa aplicação será nossa classe cliente.
- Desenvolva o código em Java ou C# de nossa classe Pedido conforme demonstrado no próximo slide.



• C# - Classe Pedido



```
6
7 namespace StrategyExemplo.Pedidos
8 {
9     0 references
10    public class Pedido
11    {
12        2 references
13        public double valor { get; set; }
14
15        0 references
16        public double calculaFreteComum()
17        {
18            return this.valor*0.05;
19        }
20
21        0 references
22        public double calculaFreteExpresso()
23        {
24            return this.valor * 0.1;
25        }
26    }
27 }
```

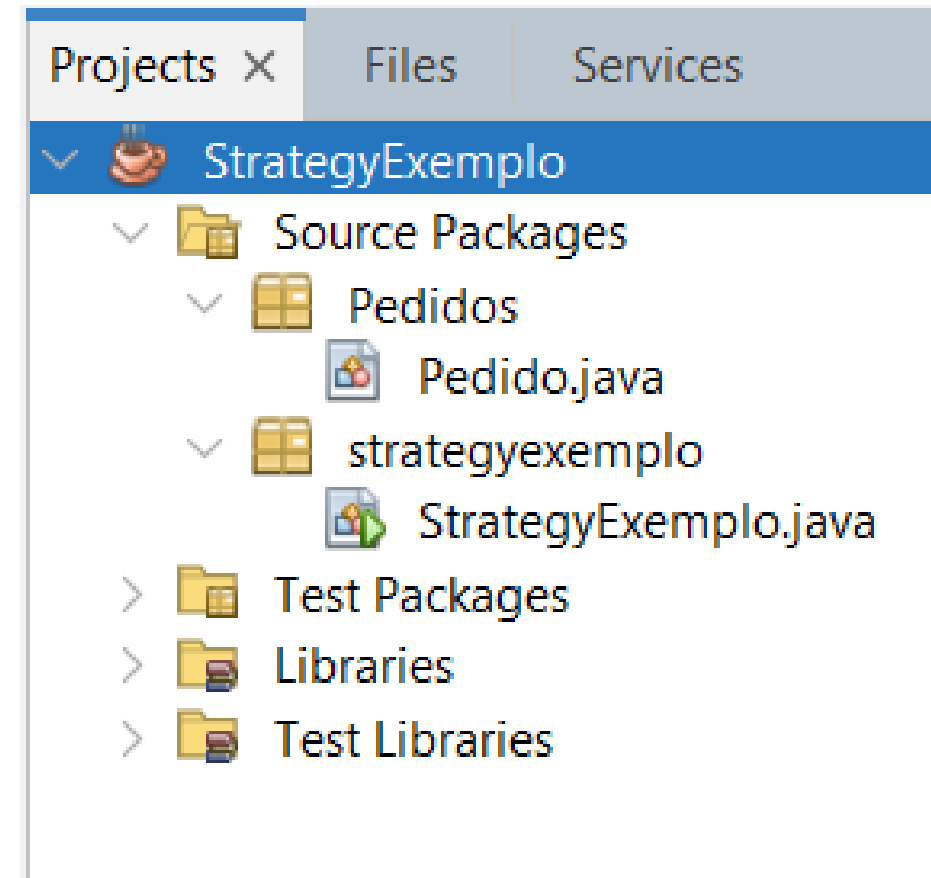
• C# - Main

```
1  using StrategyExemplo.Pedidos;
2
3  //criação do pedido
4  Pedido pedido = new Pedido();
5
6  //atribuição do valor
7  pedido.valor = 100;
8
9  //calcula para frete comum
10 Console.WriteLine("Frete Comum: R$"+pedido.calculaFreteComum());
11 //calcula para frete
12 Console.WriteLine("Frete Expresso: R$"+pedido.calculaFreteExpresso());
13
```



• Java – Classe Pedido

```
1  [+ ...4 lines
5  package Pedidos;
6
7  [+ /**...4 lines */
11 public class Pedido {
12
13     private double valor;
14
15     public double getValor() {
16         return valor;
17     }
18
19     public void setValor(double valor) {
20         this.valor = valor;
21     }
22
23     public double calculaFreteComum() {
24         return this.valor*0.05;
25     }
26
27     public double calculaFreteExpresso() {
28         return this.valor*0.1;
29     }
30 }
```



• Java – Main

```
5  package strategyexemplo;
6
7  [- import Pedidos.Pedido;
8
9  [+ /**...4 lines */
13 public class StrategyExemplo {
14
15  [+ /**...3 lines */
18  [- public static void main(String[] args) {
19
20      //criação do pedido
21      Pedido pedido = new Pedido();
22      //define o valor do pedido
23      pedido.setValor(100);
24      //calcula o frete comum
25      System.out.println("Frete Comum R$"+pedido.calculaFreteComum());
26      //calcula o frete Expresso
27      System.out.println("Frete Comum R$"+pedido.calculaFreteExpresso());
28
29  }
30
31 }
```

- **Cenário 1 – Resultado Esperado**
- Se você definiu o valor do pedido como R\$100,00
- Então você obteve:
- Frete Comum: R\$5
- Frete Expresso: R\$10



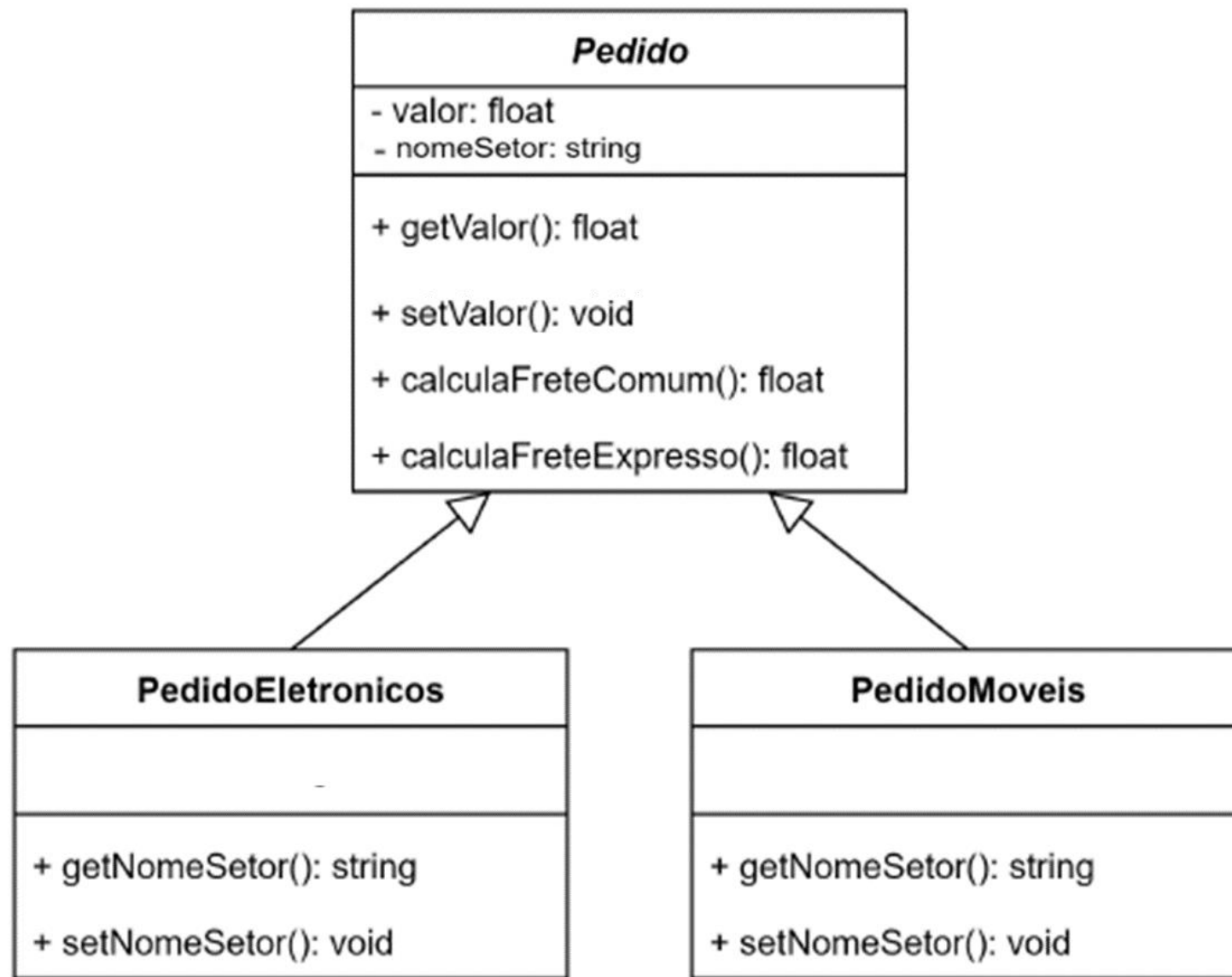
- **Cenário 1 – Resultado Esperado**
- Se você definiu o valor do pedido como R\$100,00
- Então você obteve:
- Frete Comum: R\$5
- Frete Expresso: R\$10



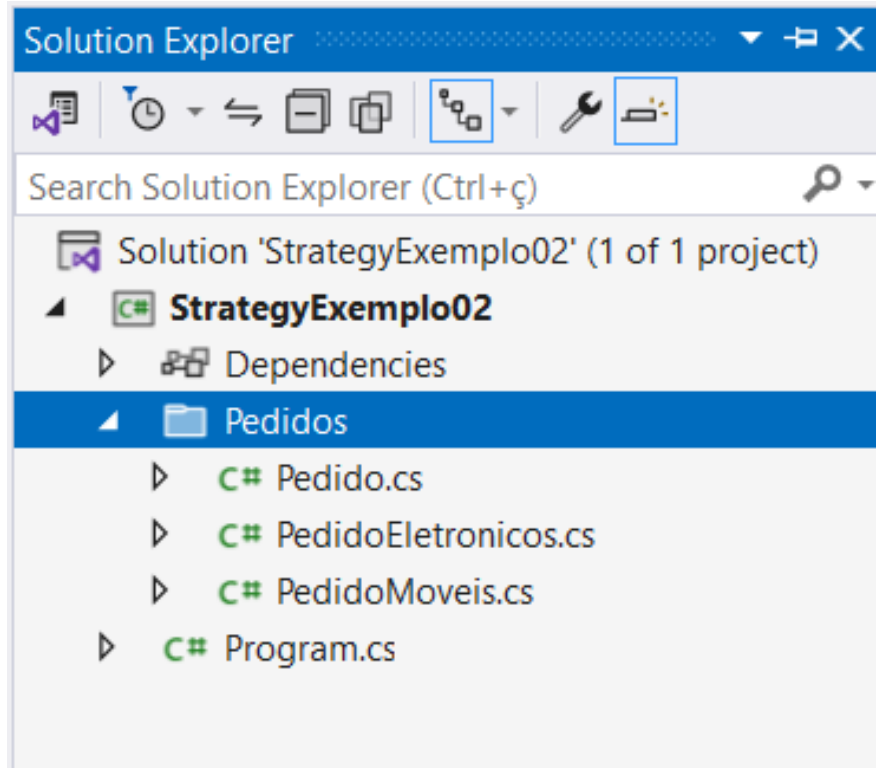
- **Cenário 2**
- Imagine agora que **o e-commerce cresceu e foi dividido em setores;**
- Os pedidos de cada setor possuem características diferentes, de modo a ser necessária a criação de uma classe de pedido para cada setor;
- **Inicialmente existirão 2 setores: móveis e eletrônicos;**
- Neste caso basta **transforma a classe Pedido em uma superclasse abstrata**, que por sua vez **implementa os métodos responsáveis por calcular os diferentes tipos de frete.**
- **Os pedidos de cada setor, que são subclasses, herdam as características da classe Pedido**, e graças a herança também passam a saber calcular os diferentes tipos de frete.



• Cenário 2

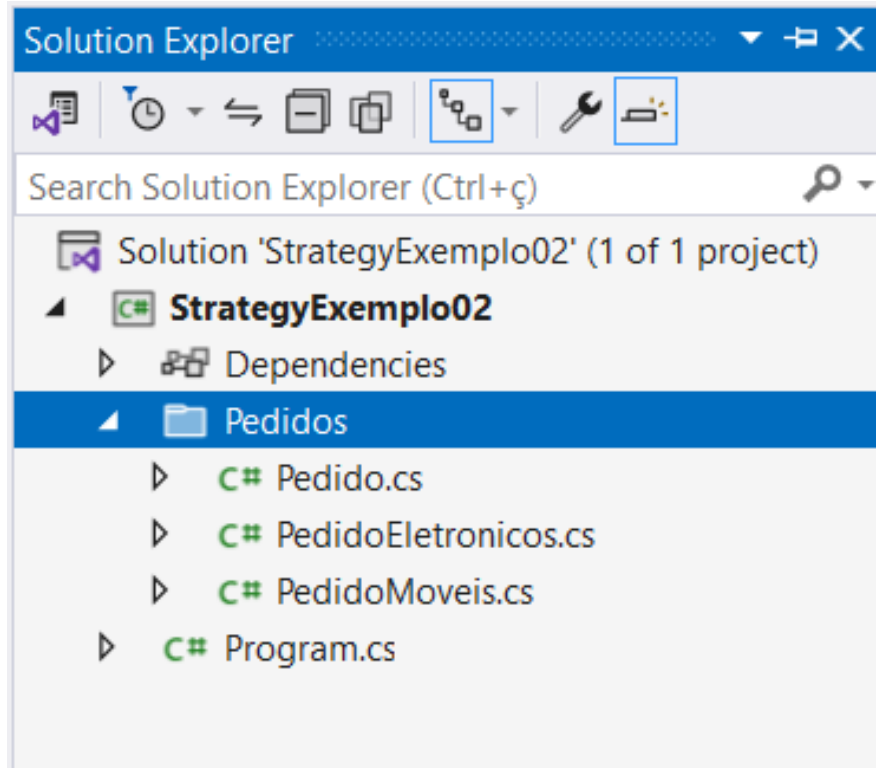


• C# - Classe Pedido



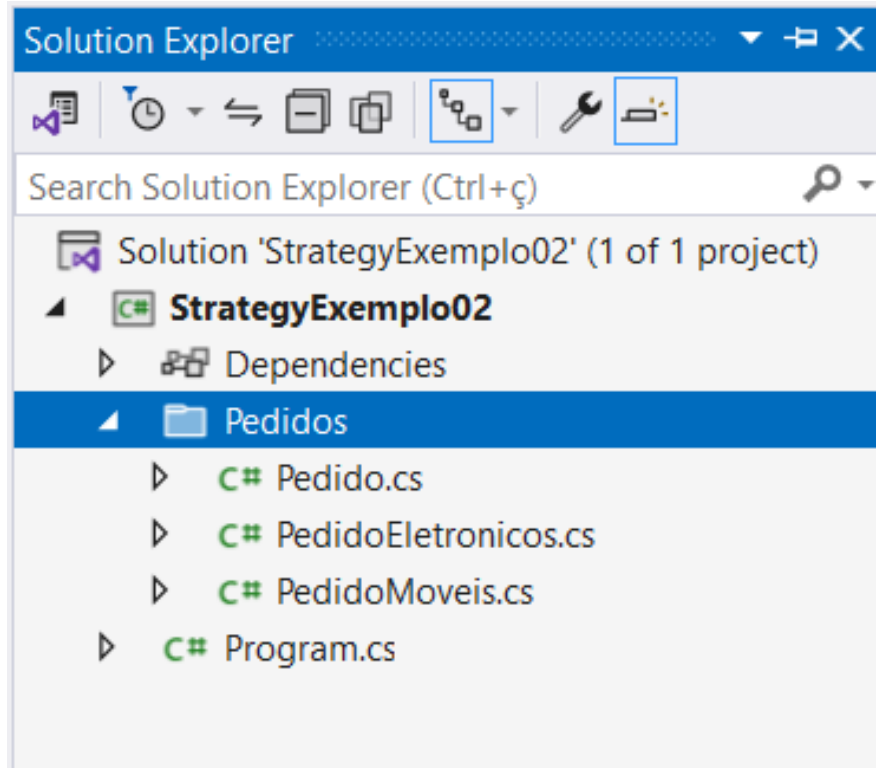
```
6
7 namespace StrategyExemplo02.Pedidos
8 {
9     3 references
10    public abstract class Pedido
11    {
12        3 references
13        public double valor { get; set; }
14        4 references
15        public string? nomeSetor { get; set; }
16
17        1 reference
18        public double calculaFreteComum()
19        {
20            return this.valor * 0.05;
21        }
22
23        1 reference
24        public double calculaFreteExpresso()
25        {
26            return this.valor * 0.1;
27        }
28    }
29 }
```

• C# - Classe PedidoMoveis



```
6
7 namespace StrategyExemplo02.Pedidos
8 {
9     1 reference
10     public class PedidoMoveis : Pedido
11     {
12         0 references
13         public PedidoMoveis()
14         {
15             this.nomeSetor = "Móveis";
16         }
17     }
18 }
```

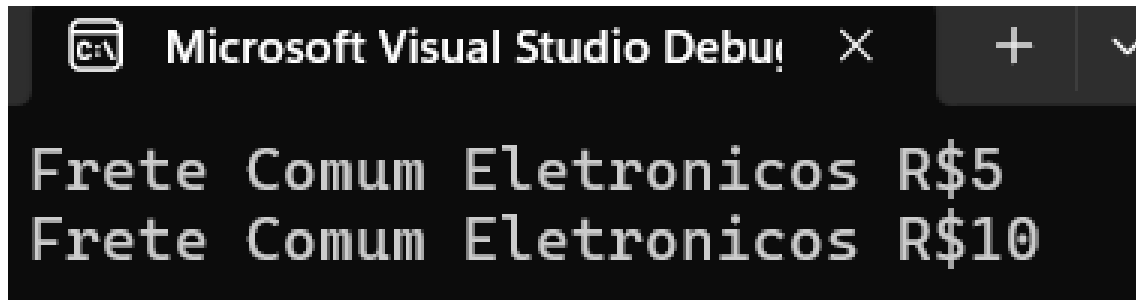
- C# - Classe PedidoEletronicos



```
7 namespace StrategyExemplo02.Pedidos
8 {
9     2 references
10    public class PedidoEletronicos : Pedido
11    {
12        1 reference
13        public PedidoEletronicos()
14        {
15            this.nomeSetor = "Eletronicos";
16        }
17    }
```

• C# - Main

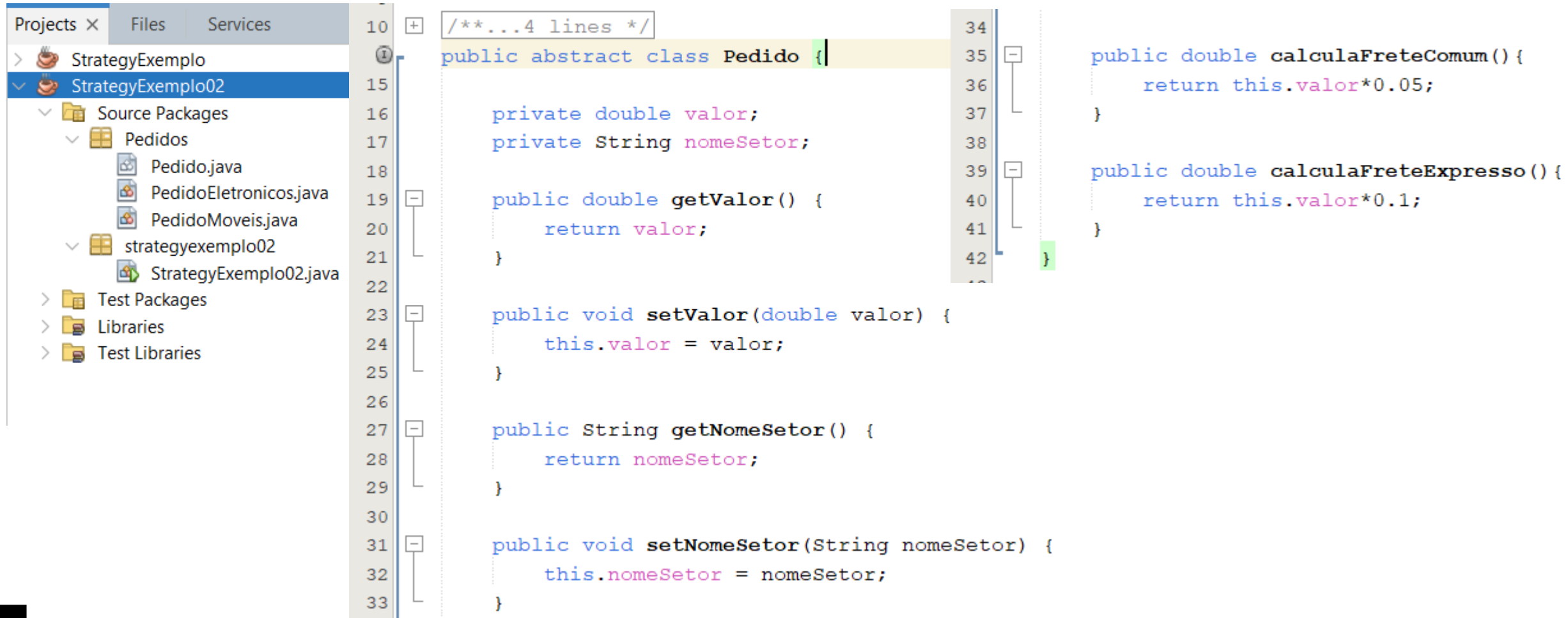
```
1 using StrategyExemplo02.Pedidos;
2
3
4 //Cria pedido de eletronicos
5 Pedido pedidoEletro = new PedidoEletronicos();
6 //define valor do pedido
7 pedidoEletro.valor = 100;
8 //Calcula frete comum
9 Console.WriteLine("Frete Comum "+pedidoEletro.nomeSetor+" R$"+pedidoEletro.calculaFreteComum());
10 //calcula frete expresso
11 Console.WriteLine("Frete Comum " + pedidoEletro.nomeSetor + " R$" + pedidoEletro.calculaFreteExpresso());
12
```



Microsoft Visual Studio Debug Console window showing the output of the C# code. The window title is "Microsoft Visual Studio Debug Console" with a close button (X) and a dropdown arrow. The output text is:

```
Frete Comum Eletronicos R$5
Frete Comum Eletronicos R$10
```

• Java - Classe Pedido



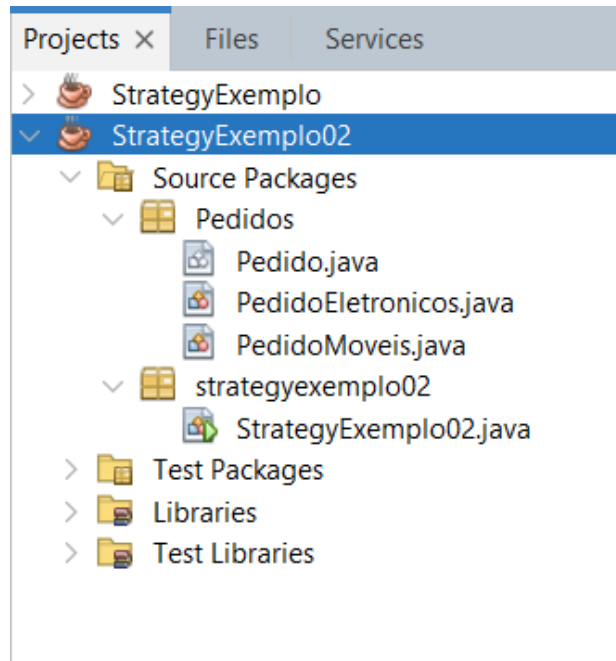
The screenshot displays an IDE with the following components:

- Project Explorer (Left):** Shows a project named 'StrategyExemplo' with a sub-project 'StrategyExemplo02'. Under 'Source Packages', there is a 'Pedidos' package containing 'Pedido.java', 'PedidoEletronicos.java', and 'PedidoMoveis.java'. There is also a 'strategyexemplo02' package and 'StrategyExemplo02.java'.
- Code Editor (Center):** Displays the implementation of the 'Pedido' class. The code is as follows:

```
10  /**...4 lines */
11  public abstract class Pedido {
12
13      private double valor;
14      private String nomeSetor;
15
16      public double getValor() {
17          return valor;
18      }
19
20      public void setValor(double valor) {
21          this.valor = valor;
22      }
23
24      public String getNomeSetor() {
25          return nomeSetor;
26      }
27
28      public void setNomeSetor(String nomeSetor) {
29          this.nomeSetor = nomeSetor;
30      }
31
32  }
```
- Code Editor (Right):** Displays the implementation of the 'Calculadora' class, showing two methods:

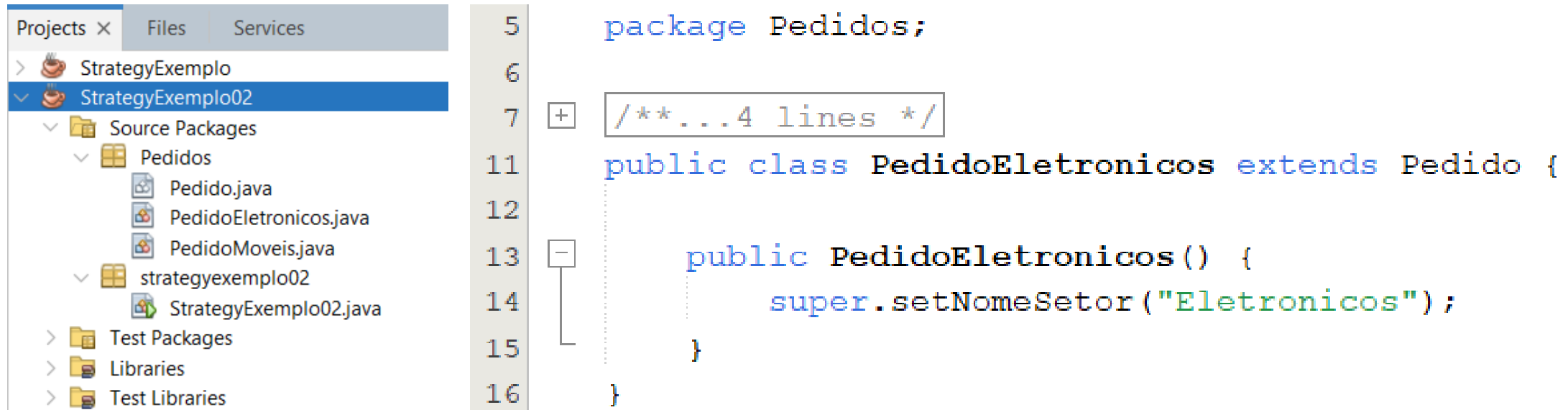
```
34
35  public double calculaFreteComum() {
36      return this.valor*0.05;
37  }
38
39  public double calculaFreteExpresso() {
40      return this.valor*0.1;
41  }
42  }
```

• Java - Classe PedidoMoveis



```
5 package Pedidos;
6
7 /**...4 lines */
11 public class PedidoMoveis extends Pedido
12
13     public PedidoMoveis() {
14         super.setNomeSetor("Móveis");
15     }
16 }
```

• Java - Classe PedidoMoveis



The screenshot shows an IDE with the following project structure in the left sidebar:

- Projects ×
 - Files
 - Services
- > StrategyExemplo
- ▼ StrategyExemplo02
 - ▼ Source Packages
 - ▼ Pedidos
 - Pedido.java
 - PedidoEletronicos.java
 - PedidoMoveis.java
 - ▼ strategyexemplo02
 - StrategyExemplo02.java
 - > Test Packages
 - > Libraries
 - > Test Libraries

The main editor displays the code for the `PedidoEletronicos` class, with line numbers 5 through 16 on the left:

```
5 package Pedidos;
6
7 /** ... 4 lines */
11 public class PedidoEletronicos extends Pedido {
12
13     public PedidoEletronicos() {
14         super.setNomeSetor("Eletronicos");
15     }
16 }
```

A large black plus sign is visible in the bottom-left corner of the slide.

• Java - Main

```
14 public class StrategyExemplo02 {
15
16     /**...3 lines */
19     public static void main(String[] args) {
20         //cria pedido
21         Pedido pedidoEleetro = new PedidoEletronicos();
22         //define valor do pedido
23         pedidoEleetro.setValor(100);
24         //calcula frete comum
25         System.out.println("Frete Comum - "+pedidoEleetro.getNomeSetor()+
26             " R$"+pedidoEleetro.calculaFreteComum());
27         //calcula frete expresso
28         System.out.println("Frete Comum - "+pedidoEleetro.getNomeSetor()+
29             " R$"+pedidoEleetro.calculaFreteExpresso());
30     }
31 }
```

Output - StrategyExemplo02 (run) ×



run:

Frete Comum - Eletronicos R\$5.0

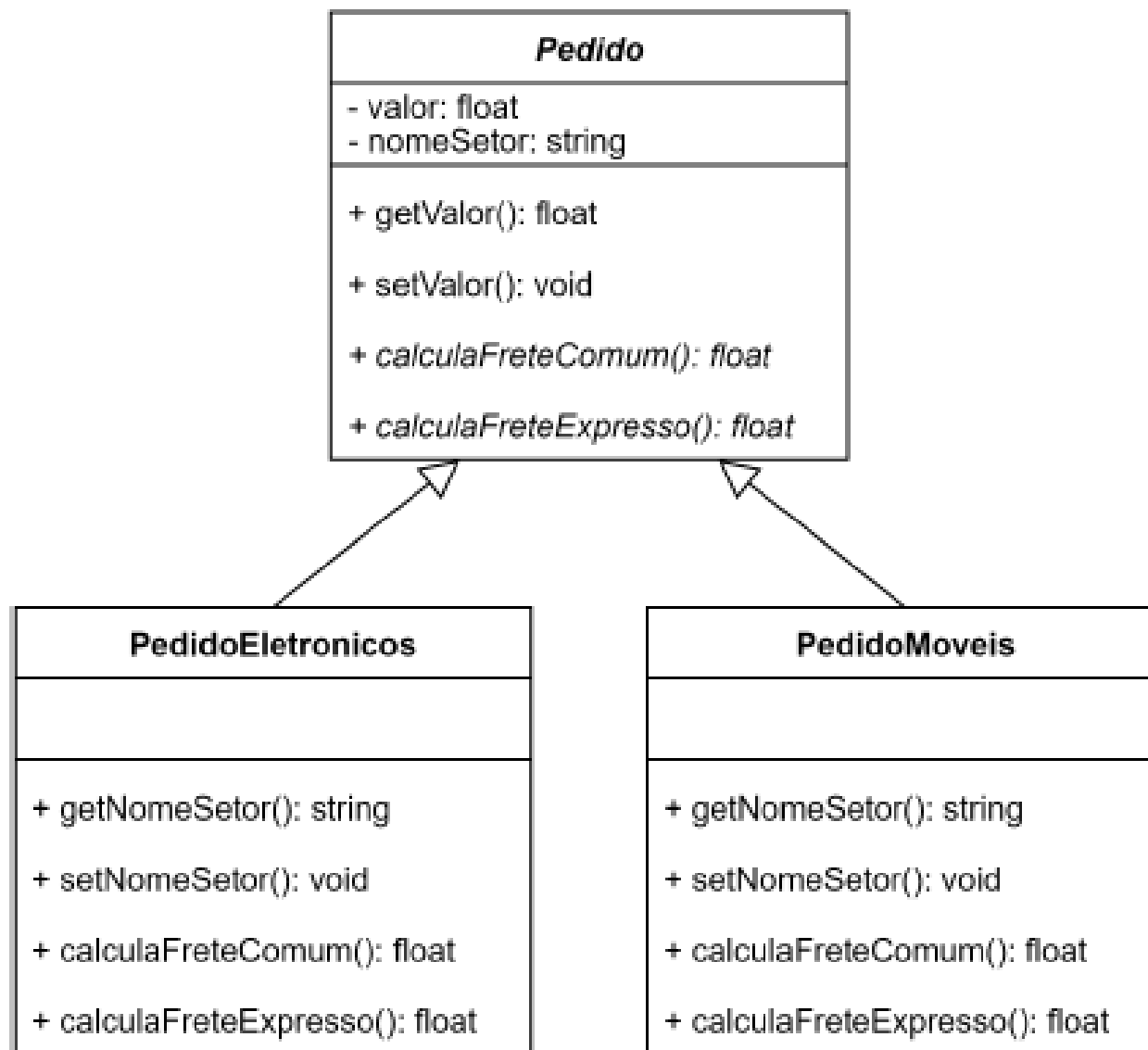
Frete Comum - Eletronicos R\$10.0

BUILD SUCCESSFUL (total time: 0 seconds)

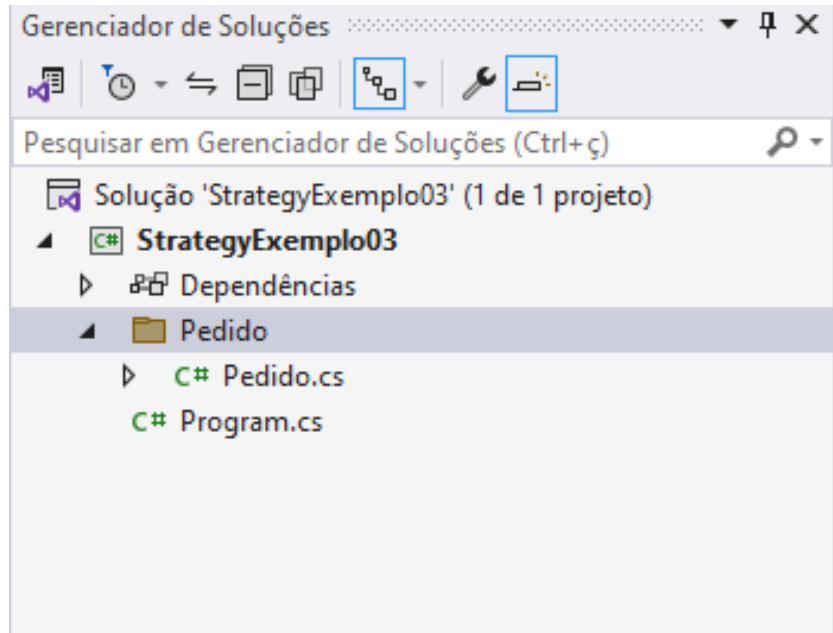
- Cenário 3
- Tudo certo até agora !!
- Mas considere que o **setor de móveis fica em um estado do Brasil onde o frete comum é o único disponível.**
- Temos um **problema**, pois devido a **herança todas as subclasses de Pedido podem calcular todos os tipos de frete;**
- Porém a subclasse **PedidoMoveis** só deveria **aceitar apenas o Frete Comum**



- **Cenário 3**
- É possível contornar isso tornando abstratos os métodos de cálculo de frete na classe abstrata Pedido;
- Assim, todos os subtipos que herdam de Pedido serão obrigados a implementar seus próprios cálculos.

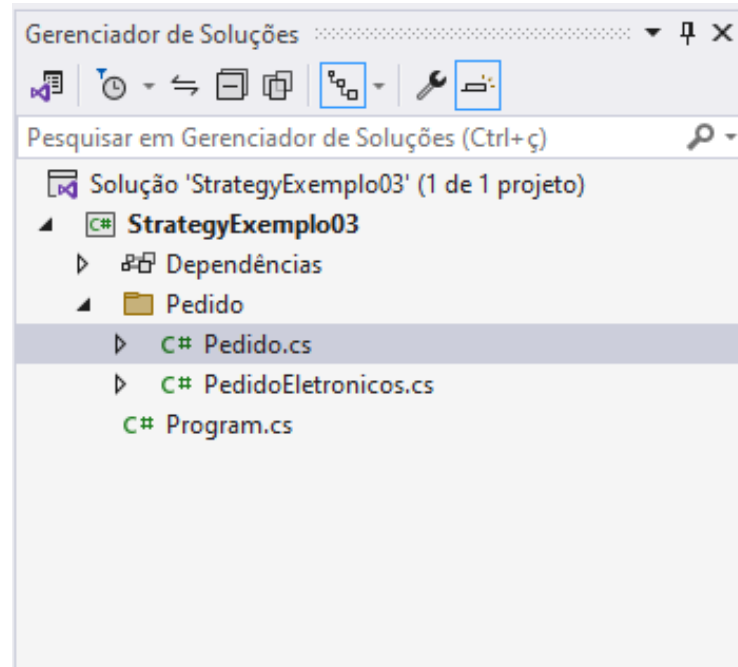


• C# - Classe Pedido



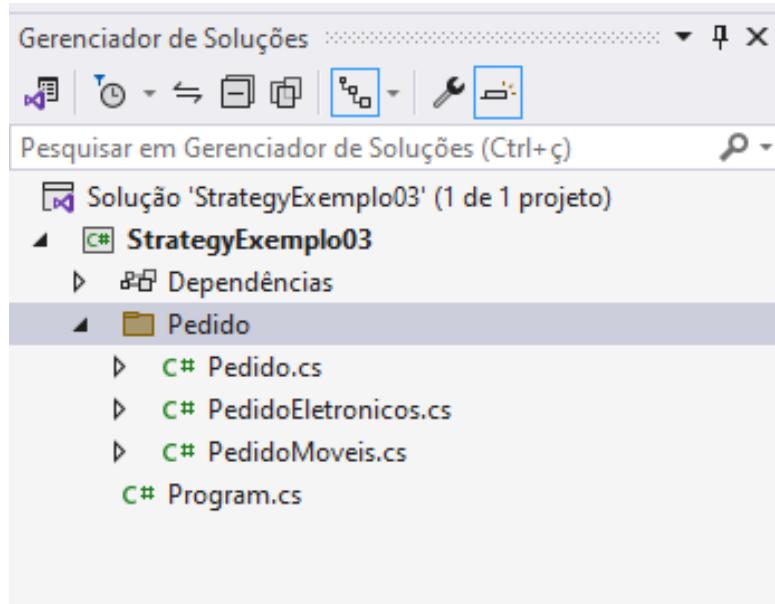
```
7 namespace StrategyExemplo03.Pedido
8 {
9     1 referência
10    public abstract class Pedido
11    {
12        2 referências
13        public double valor { get; set; }
14        1 referência
15        public string? nomeSetor { get; set; }
16
17        1 referência
18        public abstract double calculaFreteComum();
19        1 referência
20        public abstract double calculaFreteExpresso();
21    }
22 }
```

• C# - Classe PedidoEletronicos



```
7 namespace StrategyExemplo03.Pedido
8 {
9     1 referência
10    internal class PedidoEletronicos : Pedido
11    {
12        0 referências
13        public PedidoEletronicos()
14        {
15            this.nomeSetor = "Eletronicos";
16        }
17
18        1 referência
19        public override double calculaFreteComum()
20        {
21            return this.valor * 0.05;
22        }
23
24        1 referência
25        public override double calculaFreteExpresso()
26        {
27            return this.valor * 0.1;
28        }
29    }
30 }
```

• C# - Classe PedidoMoveis



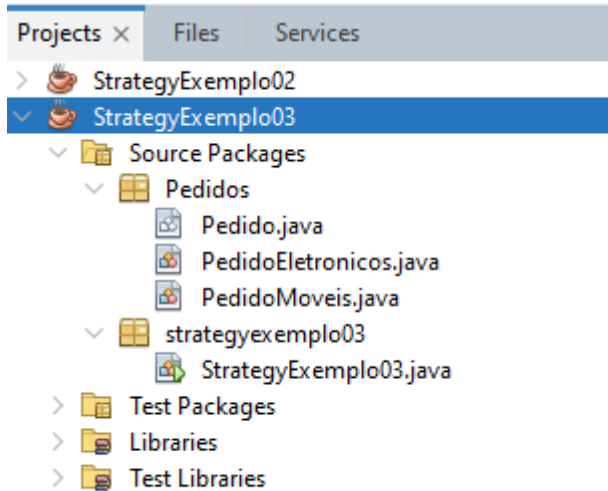
```
7 namespace StrategyExemplo03.Pedido
8 {
9
10     1 referência
11     public class PedidoMoveis : Pedido
12     {
13         0 referências
14         public PedidoMoveis()
15         {
16             this.nomeSetor = "Moveis";
17         }
18
19         1 referência
20         public override double calculaFreteComum()
21         {
22             return this.valor * 0.05;
23         }
24
25         1 referência
26         public override double calculaFreteExpresso()
27         {
28             throw new Exception("Não é permitido frete expresso");
29         }
30     }
31 }
```

• C# - Main

```
1  using StrategyExemplo03.Pedido;
2
3  try
4  {
5      //Cria pedido de eletronicos
6      Pedido pedidoEleetro = new PedidoEletronicos();
7      //define valor do pedido
8      pedidoEleetro.valor = 100;
9      //Calcula frete comum
10     Console.WriteLine("Frete Comum " + pedidoEleetro.nomeSetor + " R$" + pedidoEleetro.calculaFreteComum());
11     //calcula frete expresso
12     Console.WriteLine("Frete Comum " + pedidoEleetro.nomeSetor + " R$" + pedidoEleetro.calculaFreteExpresso());
13
14     //Cria pedido de moveis
15     Pedido pedidoMoveis = new PedidoMoveis();
16     //define valor do pedido
17     pedidoMoveis.valor = 100;
18     //Calcula frete comum
19     Console.WriteLine("Frete Comum " + pedidoMoveis.nomeSetor + " R$" + pedidoMoveis.calculaFreteComum());
20     //calcula frete expresso
21     Console.WriteLine("Frete Expresso " + pedidoMoveis.nomeSetor + " R$" + pedidoMoveis.calculaFreteExpresso());
22 }
23
24 catch (Exception e)
25 {
26     Console.WriteLine(e.ToString());
27 }
```

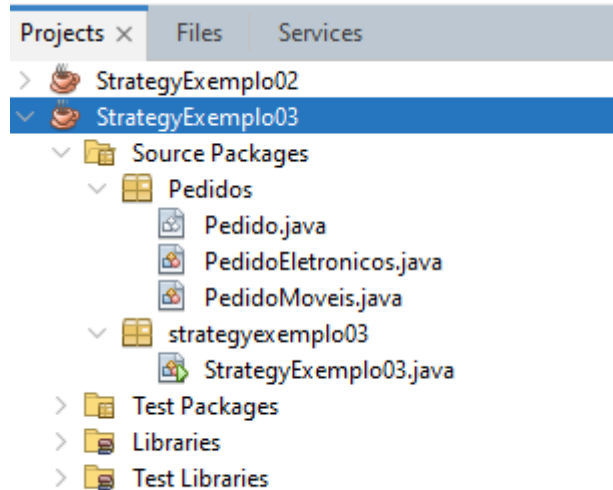
```
Frete Comum Eletronicos R$5
Frete Comum Eletronicos R$10
Frete Comum Moveis R$5
System.Exception: Não é permitido frete expresso
```

• Java - Classe Pedido



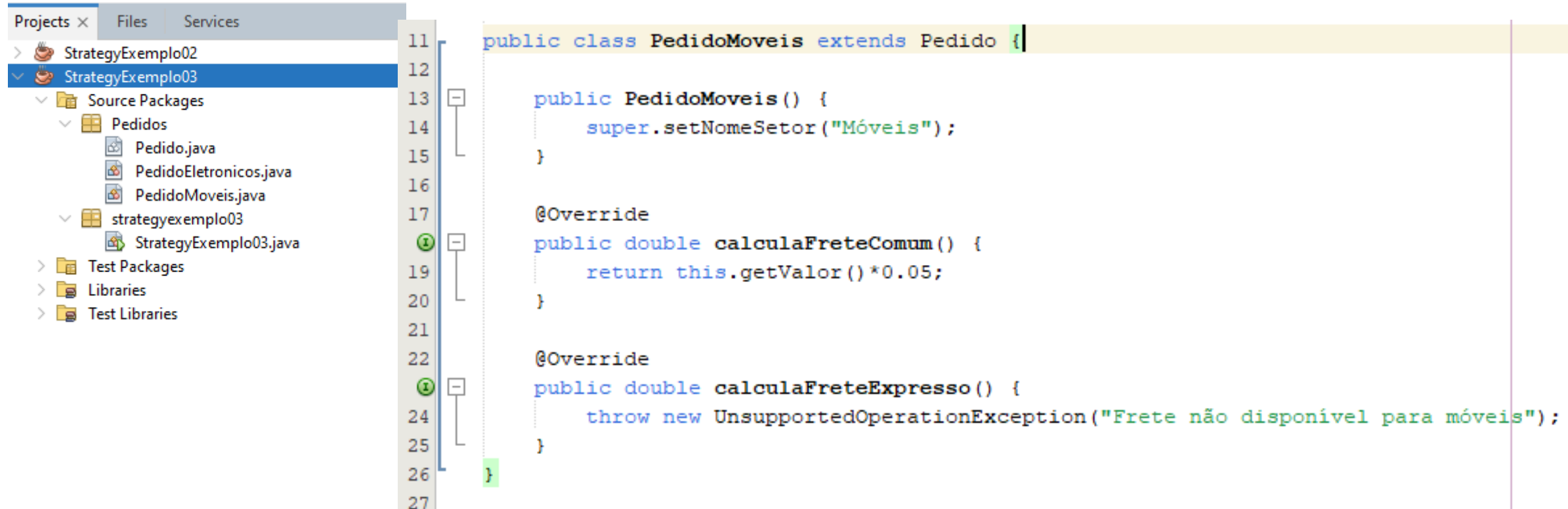
```
15 public abstract class Pedido {
16
17     private double valor;
18     private String nomeSetor;
19
20     public double getValor() {
21         return valor;
22     }
23
24     public void setValor(double valor) {
25         this.valor = valor;
26     }
27
28     public String getNomeSetor() {
29         return nomeSetor;
30     }
31
32     public void setNomeSetor(String nomeSetor) {
33         this.nomeSetor = nomeSetor;
34     }
35
36     public abstract double calculaFreteComum();
37
38     public abstract double calculaFreteExpresso();
39 }
```


• Java - Classe PedidoEletronicos



```
10  */
11  public class PedidoEletronicos extends Pedido {
12
13      public PedidoEletronicos() {
14          super.setNomeSetor("Eletronicos");
15      }
16
17      @Override
18      public double calculaFreteComum() {
19          return this.getValor()*0.05;
20      }
21
22      @Override
23      public double calculaFreteExpresso() {
24          return this.getValor()*0.1;
25      }
26  }
```

• Java - Classe PedidoMoveis

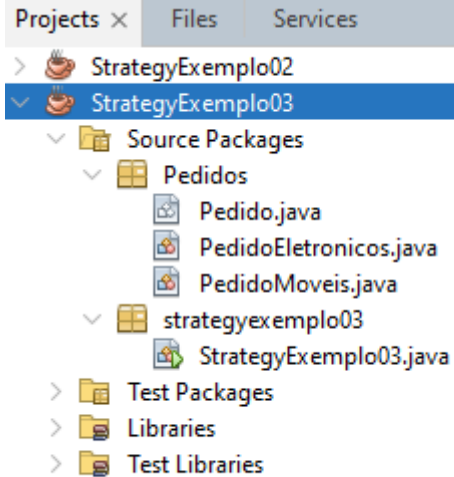


The screenshot displays an IDE with the following components:

- Project Explorer (Left):** Shows a project named 'StrategyExemplo03' under 'Source Packages'. It contains a package 'Pedidos' with files 'Pedido.java', 'PedidoEletronicos.java', and 'PedidoMoveis.java'. There is also a package 'strategyexemplo03' with 'StrategyExemplo03.java'.
- Code Editor (Right):** Shows the implementation of the 'PedidoMoveis' class, which extends 'Pedido'. The code is as follows:

```
11 public class PedidoMoveis extends Pedido {  
12  
13     public PedidoMoveis() {  
14         super.setNomeSetor("Móveis");  
15     }  
16  
17     @Override  
18     public double calculaFreteComum() {  
19         return this.getValor()*0.05;  
20     }  
21  
22     @Override  
23     public double calculaFreteExpresso() {  
24         throw new UnsupportedOperationException("Frete não disponível para móveis");  
25     }  
26 }  
27
```

• Java - Main

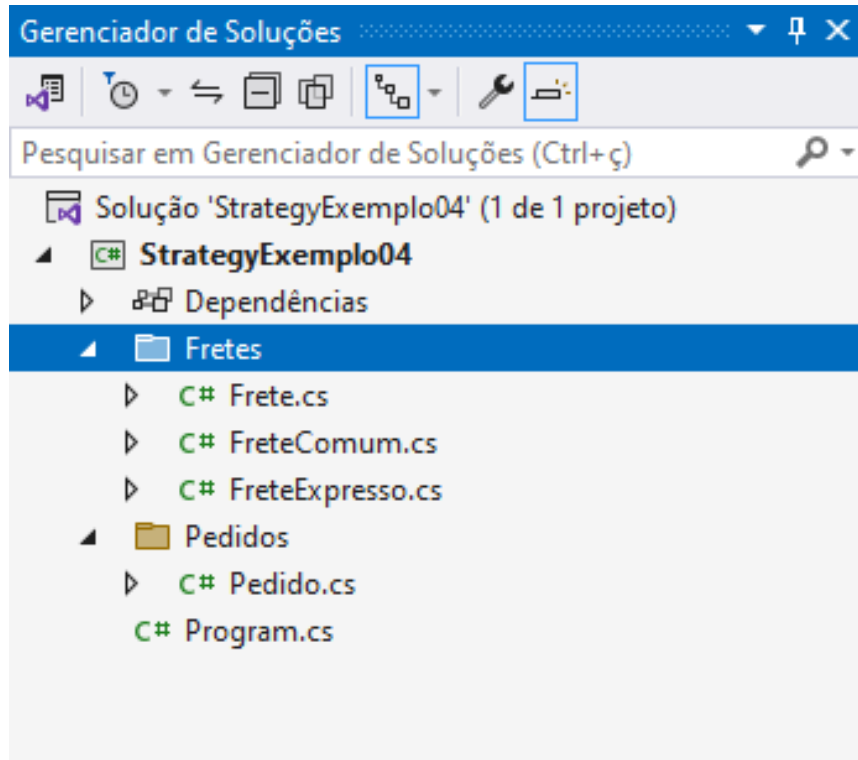


```
15 public class StrategyExemplo03 {
16     /**...3 lines */
19     public static void main(String[] args) {
20         try{
21             //cria pedido
22             Pedido pedidoEletronicos = new PedidoEletronicos();
23             //define valor do pedido
24             pedidoEletronicos.setValor(100);
25             //calcula frete comum
26             System.out.println("Frete Comum - "+pedidoEletronicos.getNomeSetor()+
27                 " R$"+pedidoEletronicos.calculaFreteComum());
28             //calcula frete expresso
29             System.out.println("Frete Comum - "+pedidoEletronicos.getNomeSetor()+
30                 " R$"+pedidoEletronicos.calculaFreteExpresso());
31
32             //cria pedido
33             Pedido pedidoMoveis = new PedidoMoveis();
34             //define valor do pedido
35             pedidoMoveis.setValor(100);
36             //calcula frete comum
37             System.out.println("Frete Comum - "+pedidoMoveis.getNomeSetor()+
38                 " R$"+pedidoMoveis.calculaFreteComum());
39             //calcula frete expresso
40             System.out.println("Frete Comum - "+pedidoMoveis.getNomeSetor()+
41                 " R$"+pedidoMoveis.calculaFreteExpresso());
42
43         } catch (Exception e) {
44             System.out.println(e.getMessage());
45         }
46     }
47
48 }
```

- **Cenário 3**
- Neste cenário cada sub-pedido controla seus fretes;
- As desvantagens dessa abordagem é que não existe reaproveitamento de código;
- Repare que o método `calcularFreteComum()` é exatamente igual nas duas subclasses (se fossem mais tipos o problema aumentaria);
- Além de obrigar lançar uma `exception()` no `calcularFreteExpresso()` para a subclasse `PedidoMoveis`.

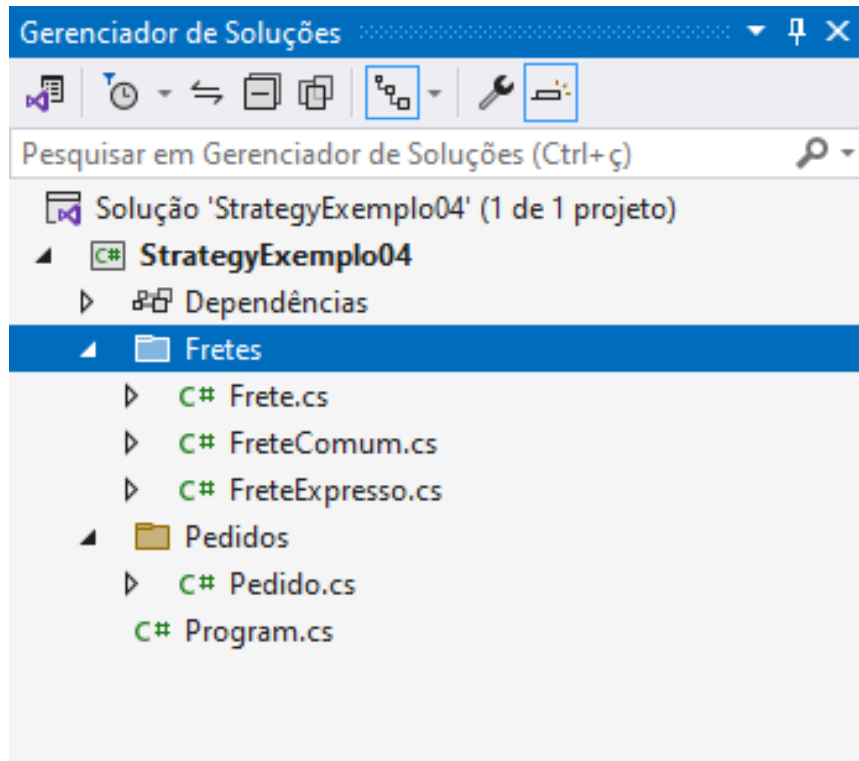
- **Cenário 4**
- Para resolvermos os problemas previamente apresentados nos outros exemplos, vamos aplicar o padrão Strategy.
- O padrão Strategy encapsula algoritmos que representam um comportamento similar, ou seja, isola o código que toma a decisão de modo que ele possa ser editado ou incrementado de forma totalmente independente.
- Vamos começar com uma interface chamada Frete que possui o método calcula().
- Tal interface define uma família de algoritmos, onde cada membro dessa família é capaz de calcular um determinado tipo de frete.

• C# - Interface Frete



```
7 namespace StrategyExemplo04.Fretes
8 {
9     3 referências
10     public interface Frete
11     {
12         2 referências
13         public double calcula(double valorPedido);
14     }
15 }
```

• C# - Classes FreteComum e FreteExpresso

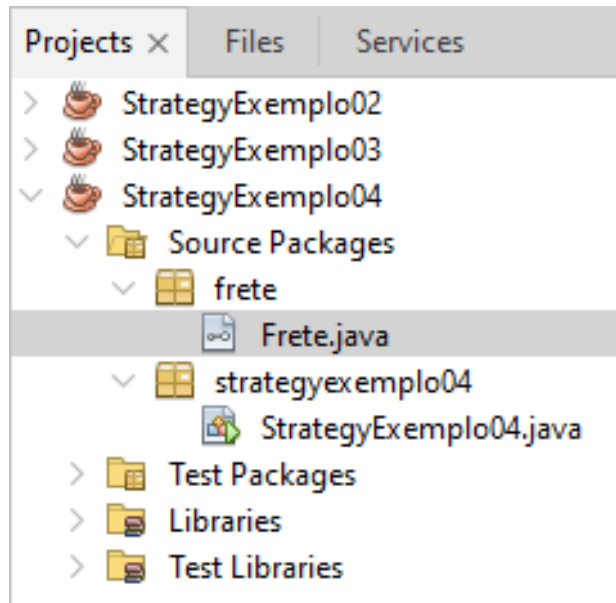


+ Crie as classes FreteComum e FreteExpresso que devem implementar a interface Frete.

```
7 namespace StrategyExemplo04.Fretes
8 {
9     0 referências
10    public class FreteComum : Frete
11    {
12        1 referência
13        public double calcula(double valorPedido)
14        {
15            return valorPedido * 0.05;
16        }
17    }
18 }
```

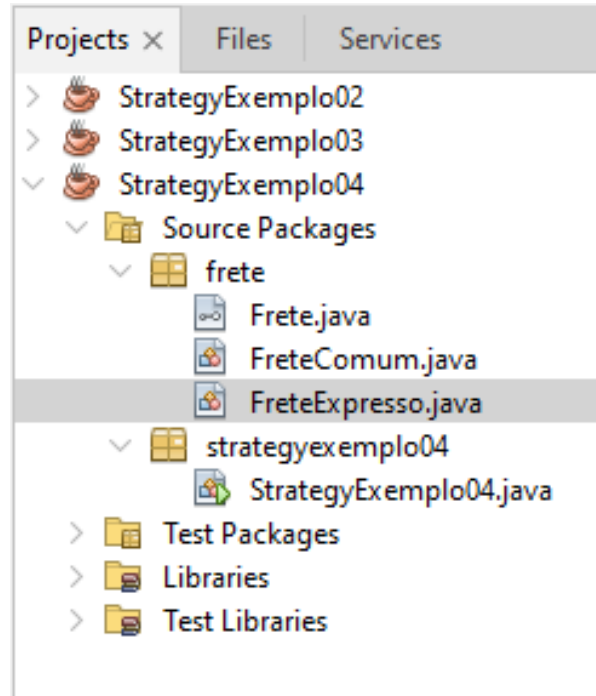
```
7 namespace StrategyExemplo04.Fretes
8 {
9     0 referências
10    public class FreteExpresso : Frete
11    {
12        1 referência
13        public double calcula(double valorPedido)
14        {
15            return valorPedido * 0.1;
16        }
17    }
18 }
```

• Java - Interface Frete



```
5 package frete;
6
7 /**
8  *
9  * @author jeffe
10 */
11 public interface Frete {
12
13     public double calcula(double valorPedido);
14
15 }
16
```


• Java - Classes FreteComum e FreteExpresso



Crie as classes FreteComum e FreteExpresso que devem implementar a interface Frete.

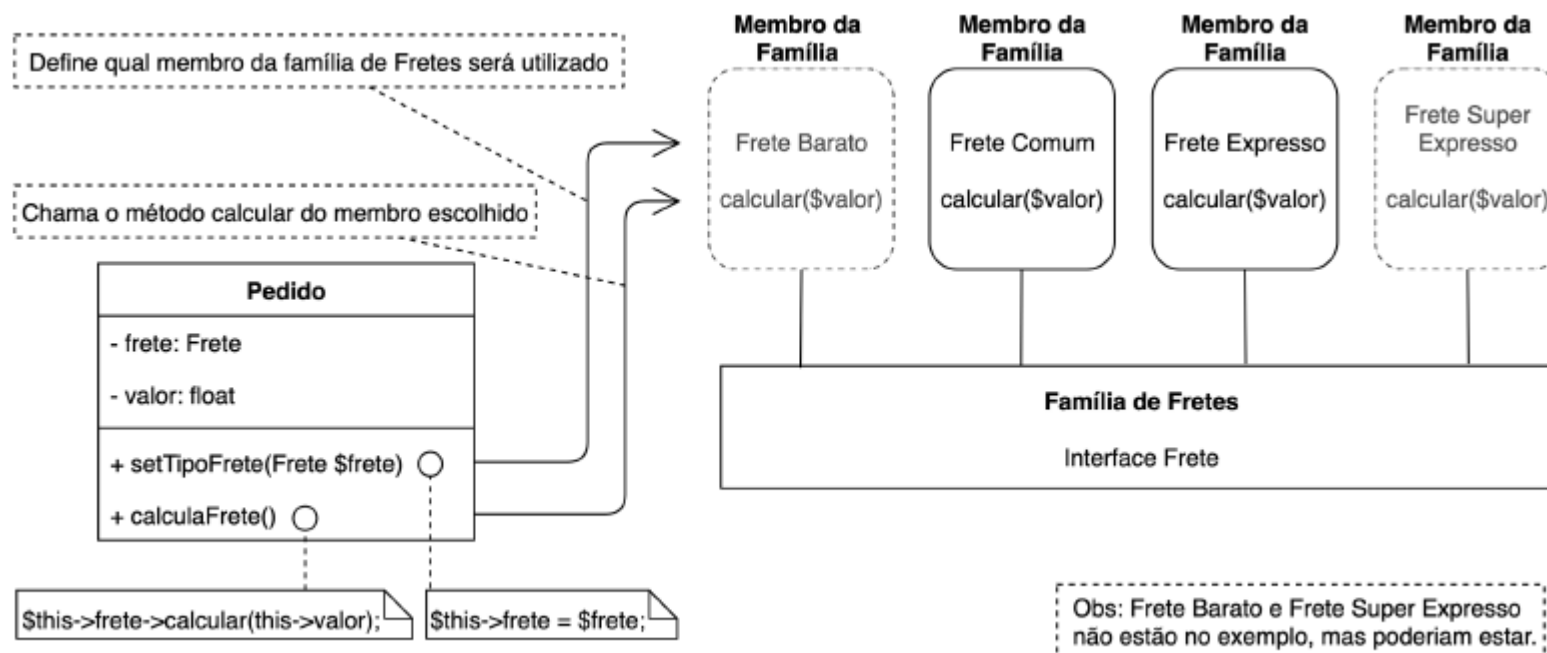
```
5 package frete;
6
7 /**
8  *
9  * @author jeffe
10  */
11 public class FreteComum implements Frete{
12
13     @Override
14     public double calcula(double valorPedido) {
15         return valorPedido*0.05;
16     }
17 }
18
```

```
5 package frete;
6
7 /**
8  *
9  * @author jeffe
10  */
11 public class FreteExpresso implements Frete{
12
13     @Override
14     public double calcula(double valorPedido) {
15         return valorPedido*0.1;
16     }
17 }
18
```

- **Cenário 4**
- Cada uma das classes implementam a interface Frete, portanto todas elas possuem o método calcula() que faz o cálculo conforme sua fórmula interna;
- Agora o cálculo do frete não fica mais na classe Pedido e nem em suas subclasses, este comportamento foi encapsulado em classes separadas.

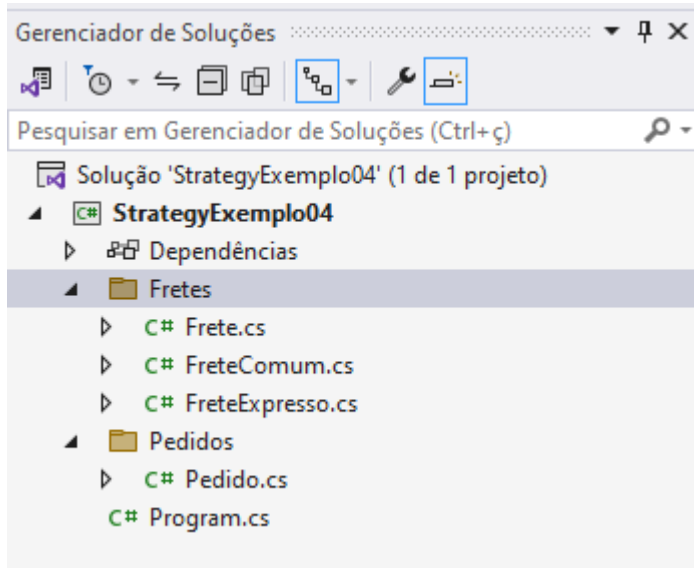
• Cenário 4

- Com a criação das classes Frete, basta adaptar que a classe abstrata Pedido para ela tenha os seguinte métodos:
 - setTipoFrete(Frete tipoFrete) → define o tipo de frete no pedido
 - calculaFrete() → responsável por invocar o método calcula() do objeto recebido em setTipoFrete().



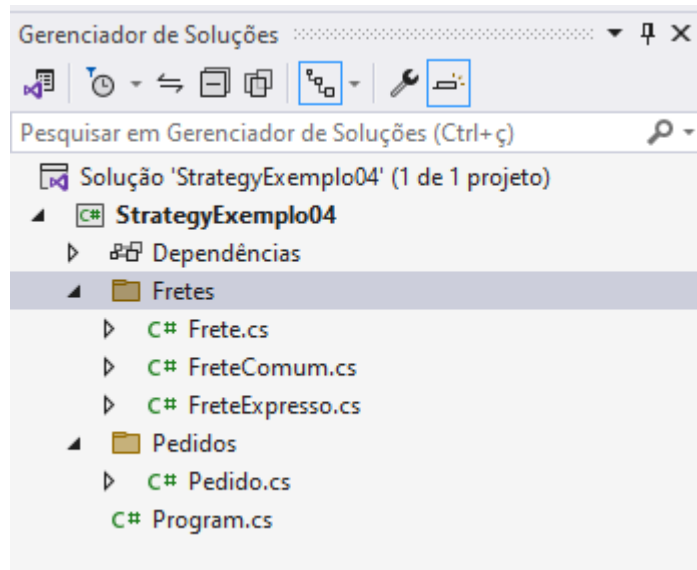
Esquema da utilização de famílias de algoritmos

• C# - Classe Pedido



```
1  using StrategyExemplo04.Fretes;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7
8
9  namespace StrategyExemplo04.Pedidos
10 {
11     1 referência
12     public abstract class Pedido
13     {
14         1 referência
15         public double valor { get; set; }
16         0 referências
17         public string? nomeSetor { get; set; }
18         1 referência
19         public Frete tipoFrete { get; set; }
20
21         0 referências
22         public double calculaFrete()
23         {
24             return tipoFrete.calcula(this.valor);
25         }
26     }
27 }
```

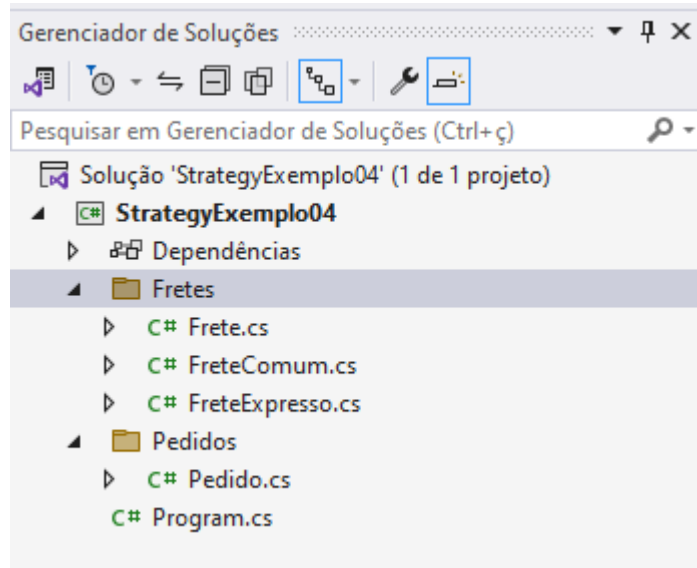
- C# - Classe PedidoEletronicos e PedidoMoveis



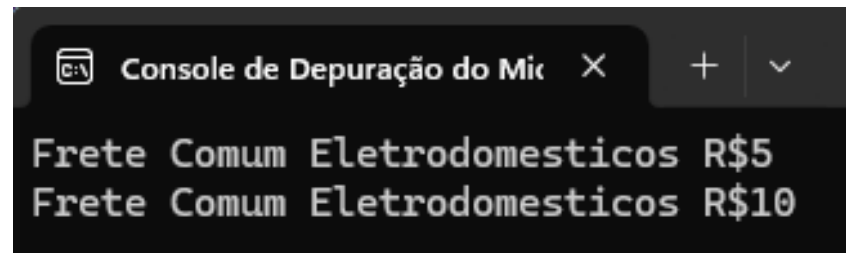
```
7 namespace StrategyExemplo04.Pedidos
8 {
9     1 referência
10     public class PedidoEletronicos : Pedido
11     {
12         0 referências
13         public PedidoEletronicos()
14         {
15             this.nomeSetor = "Eletrodomesticos";
16         }
17     }
```

```
7 namespace StrategyExemplo04.Pedidos
8 {
9     1 referência
10     public class PedidoMoveis : Pedido
11     {
12         0 referências
13         public PedidoMoveis()
14         {
15             this.nomeSetor = "Moveis";
16         }
17     }
```

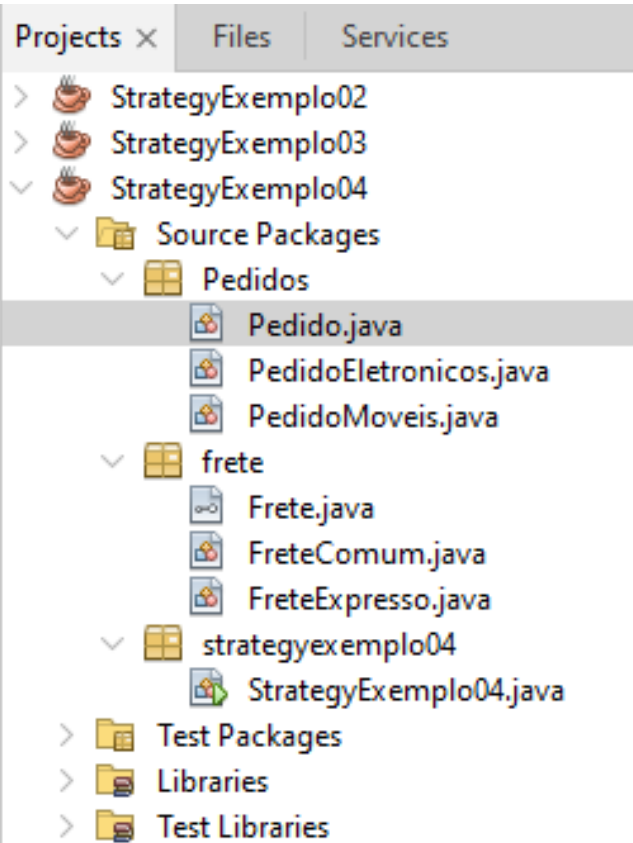
. C# - Main



```
1
2 using StrategyExemplo04.Fretes;
3 using StrategyExemplo04.Pedidos;
4
5 try
6 {
7     //criação dos tipos de frete
8     Frete freteComum = new FreteComum();
9     Frete freteExpresso = new FreteExpresso();
10
11
12     //Cria pedido de eletronicos
13     Pedido pedidoEletronicos = new PedidoEletronicos();
14     //define valor do pedido
15     pedidoEletronicos.valor = 100;
16     //define o tipo de frete
17     pedidoEletronicos.tipoFrete = freteComum;
18     //Calcula frete comum
19     Console.WriteLine("Frete Comum " + pedidoEletronicos.nomeSetor + " R$" + pedidoEletronicos.calculaFrete());
20
21     //altera o tipo de frete
22     pedidoEletronicos.tipoFrete = freteExpresso;
23     //calcula frete expresso
24     Console.WriteLine("Frete Comum " + pedidoEletronicos.nomeSetor + " R$" + pedidoEletronicos.calculaFrete());
25
26 }
27 catch (Exception e)
28 {
29     Console.WriteLine(e.ToString());
30 }
```



• Java - Classe Pedido



```
7 import frete.Frete;
```

```
8  
9 /**  
10 *  
11 * @author jeffe  
12 */
```

```
13 public abstract class Pedido {
```

```
14  
15     private double valor;
```

```
16     private String nomeSetor;
```

```
17     private Frete tipoFrete;
```

```
18  
19     public double getValor() {  
20         return valor;  
21     }
```

```
22  
23     public void setValor(double valor) {  
24         this.valor = valor;  
25     }
```

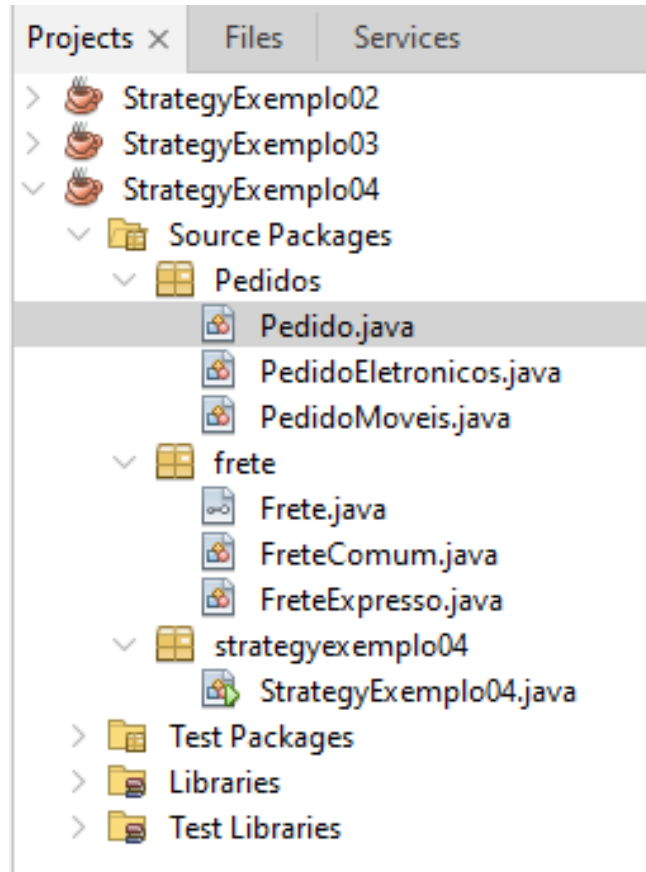
```
26  
27     public String getNomeSetor() {  
28         return nomeSetor;  
29     }
```

```
30  
31     public void setNomeSetor(String nomeSetor) {  
32         this.nomeSetor = nomeSetor;  
33     }
```

```
34  
35     public Frete getTipoFrete() {  
36         return tipoFrete;  
37     }
```

```
38  
39     public void setTipoFrete(Frete tipoFrete) {  
40         this.tipoFrete = tipoFrete;  
41     }  
42  
43     public double calculaFrete() {  
44         return tipoFrete.calcula(valor);  
45     }  
46  
47 }
```

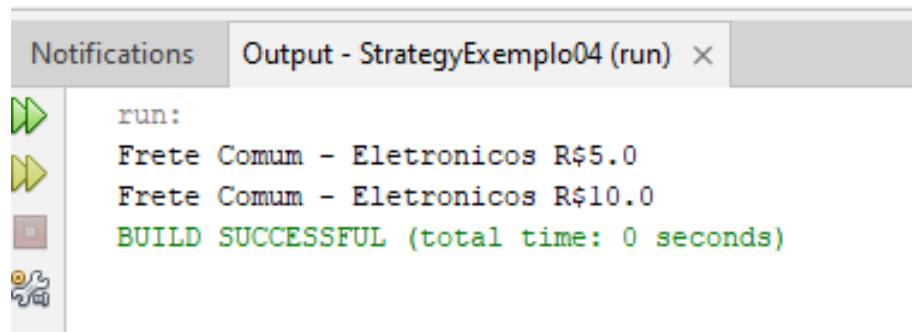
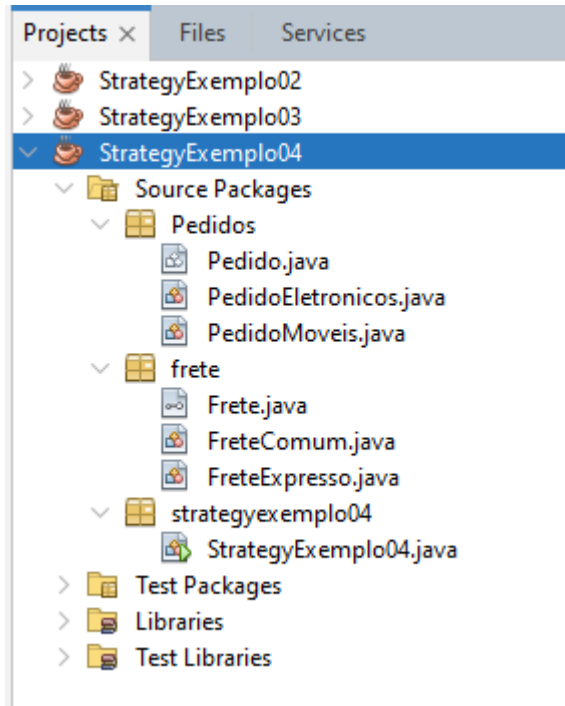
• Java - Classe PedidoEletronicos e PedidoMoveis



```
5 package Pedidos;
6
7 /**
8  *
9  * @author jeffe
10 */
11 public class PedidoMoveis extends Pedido {
12
13     public PedidoMoveis() {
14         this.setNomeSetor("Moveis");
15     }
16
17 }
```

```
5 package Pedidos;
6
7 /**
8  *
9  * @author jeffe
10 */
11 public class PedidoEletronicos extends Pedido {
12
13     public PedidoEletronicos()
14     {
15         this.setNomeSetor("Eletronicos");
16     }
17
18 }
```


• Java - Main



```
18 public class StrategyExemplo04 {
19
20     /**
21      * @param args the command line arguments
22      */
23     public static void main(String[] args) {
24         try{
25             //cria os tipos de frete
26             Frete freteComum = new FreteComum();
27             Frete freteExpresso = new FreteExpresso();
28
29             //cria pedido
30             Pedido pedidoEleetro = new PedidoEletronicos();
31             //define valor do pedido
32             pedidoEleetro.setValor(100);
33
34             //define o tipo de frete para o pedido
35             pedidoEleetro.setTipoFrete(freteComum);
36             //calcula frete comum
37             System.out.println("Frete Comum - "+pedidoEleetro.getNomeSetor()+
38                 " R$"+pedidoEleetro.calculaFrete());
39
40             //altera tipo de frete no pedido
41             pedidoEleetro.setTipoFrete(freteExpresso);
42             //calcula frete expresso
43             System.out.println("Frete Comum - "+pedidoEleetro.getNomeSetor()+
44                 " R$"+pedidoEleetro.calculaFrete());
45
46         } catch (Exception e) {
47             System.out.println(e.getMessage());
48         }
49     }
50 }
```

• Cenário 4

- Com isso o algoritmo fica mais flexível, o tipo de frete passa a ser definido dinamicamente em tempo de execução. Além disso passa a obedecer alguns princípios básicos de OO:
 - Programe para abstrações
 - Open-closed
 - Prioridade a composição em relação a herança

