

# Decorator

Padrões de Projeto Estrutural I

Prof. Me Jefferson Passerini



O padrão **Decorator** anexa responsabilidades adicionais a um objeto dinamicamente. Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.

### **Por que utilizar?**

Às vezes, queremos adicionar responsabilidade a objetos individuais, não a uma classe inteira.

Uma maneira de adicionar responsabilidades é com herança.

Herdar uma característica de outra classe faz com que a subclasse também a tenha.

Isso é inflexível, pois, a definição de tal característica é feita estaticamente, de modo que Cliente não pode controlar como e quando decorar o objeto com ela.

## **Por que utilizar? - Exemplo**

Para ilustrar tal situação, tomemos como exemplo o sistema de uma pizzeria, onde o cliente pode acrescentar características adicionais a sua pizza.

É importante dizer que os donos da pizzeria em breve criarão novos sabores de pizza e novos acréscimos.

## **Por que utilizar? - Exemplo**

**A pizzeria possui 3 sabores de pizza no cardápio:**

Pizza de Frango – R\$19,00

Pizza de Calabresa – R\$25,00

Pizza de Queijo – R\$22,00

**E possui acréscimos:**

Borda Recheada com requeijão – R\$8,50

Massa Integral – R\$5,00

### Por que utilizar? - Exemplo

Cada pizza tem um preço, ao adicionar um acréscimo seu valor deve ser somado ao valor da pizza.

Por exemplo, uma pizza de frango com borda recheada de requeijão custaria **R\$27,50**.

**R\$19,00 pela pizza + 8,50 pela borda de requeijão**

## Por que utilizar? - Exemplo

Todas as pizzas devem possuir um preço e uma descrição.  
Seria inviável criar uma classe para cada combinação possível,  
como *PizzaCalabresaBordaRequeijão* ou  
*PizzaQueijoMassaIntegral*.

A cada novo tipo de pizza ou novo acréscimo o número de classes  
cresceria **exponencialmente**.

Por que utilizar? -  
Exemplo

É possível utilizar  
herança.

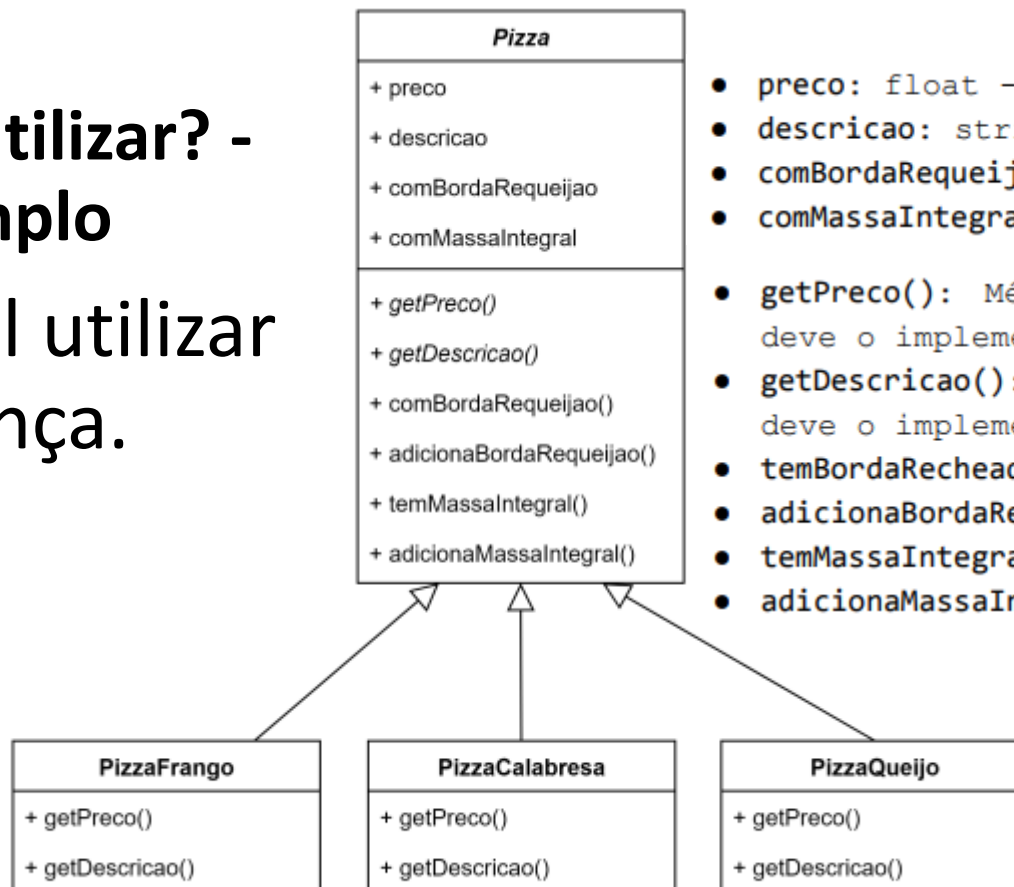


Diagrama de classes do exemplo (cenário 1)

- `preco: float` - Preço da Pizza
- `descricao: string` - Descrição da pizza
- `comBordaRequeijao: bool` - Indica se a pizza tem borda recheada.
- `comMassaIntegral: bool` - Indica se a pizza tem massa integral.
- `getPreco()`: Método abstrato - todas as classes que herdam de `Pizza` deve o implementar - define o preço da pizza.
- `getDescricao()`: Método abstrato - todas as classes que herdam de `Pizza` deve o implementar - define a descrição da pizza.
- `temBordaRecheada()`: retorna o atributo `comBordaRequeijao` [Sem título]
- `adicionaBordaRequeijao()`: define `comBordaRequeijao` como `true`.
- `temMassaIntegral()`: retorna o atributo `comMassaIntegral`.
- `adicionaMassaIntegral()`: define `comMassaIntegral` como `true`.

Deste modo *PizzaFrango*, *PizzaCalabresa* e *PizzaQueijo* teriam que implementar seu próprio método `getPreço()` e herdariam todos os demais métodos e atributos da classe abstrata *Pizza*



O problema do uso de herança no caso das pizzas é a inflexibilidade que ela acarreta.

*Todos os acréscimos são atributos às subclasses em tempo de compilação.*

Imagine que a pizzaria começasse a servir pizzas doces, todas elas teriam o atributo comBordaRequeijao, o que não faz sentido.

Neste caso seria ideal se fosse possível **expandir as pizzas em tempo de execução**, onde os acréscimos pudessem ser adicionados a elas conforme a necessidade surgisse. ***É aí que o padrão decorator pode entrar***

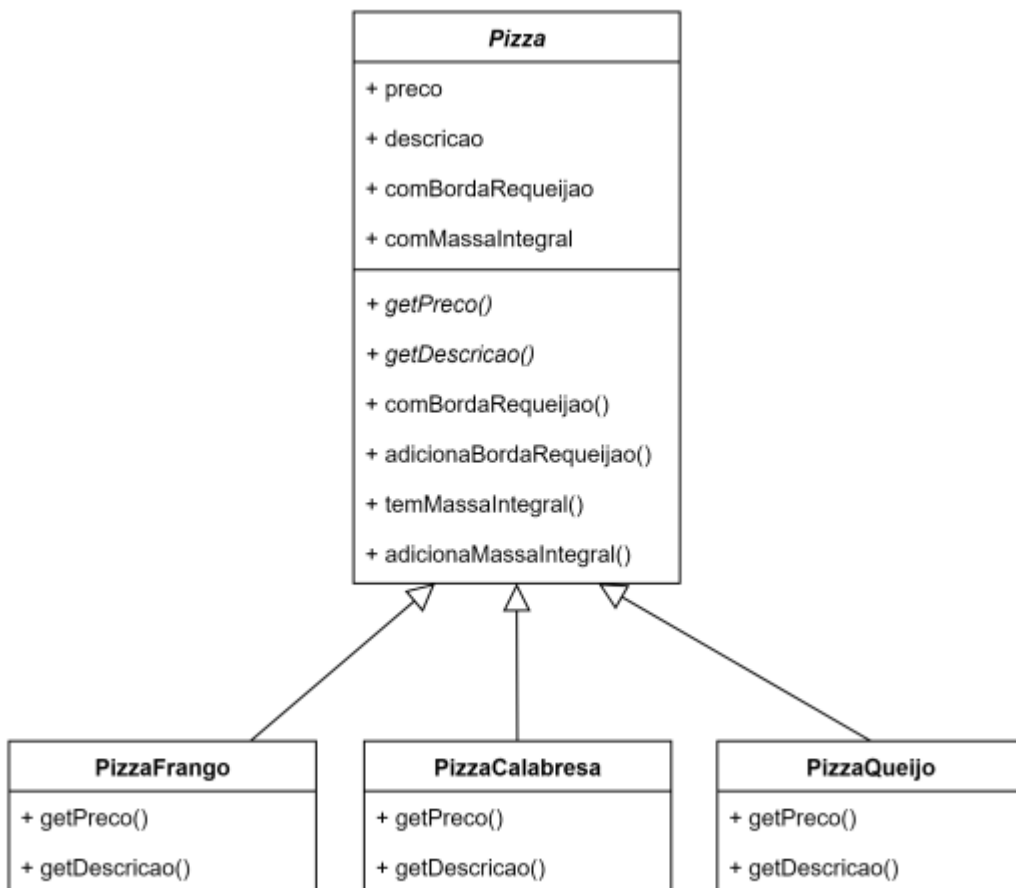


Diagrama de classes do exemplo (cenário 1)

## Padrão Decorator - Componentes

- **Component** → é o supertipo comum entre **componenteConcreto** e **Decorator**. Pode ser uma classe abstrata ou interface. Cada **Component** pode ser usado sozinho ou englobado por um decorator.
- **ComponenteConcreto** → é o objeto ao qual novos comportamentos serão adicionados dinamicamente por meio dos Decorators. Eles estende **Component**.

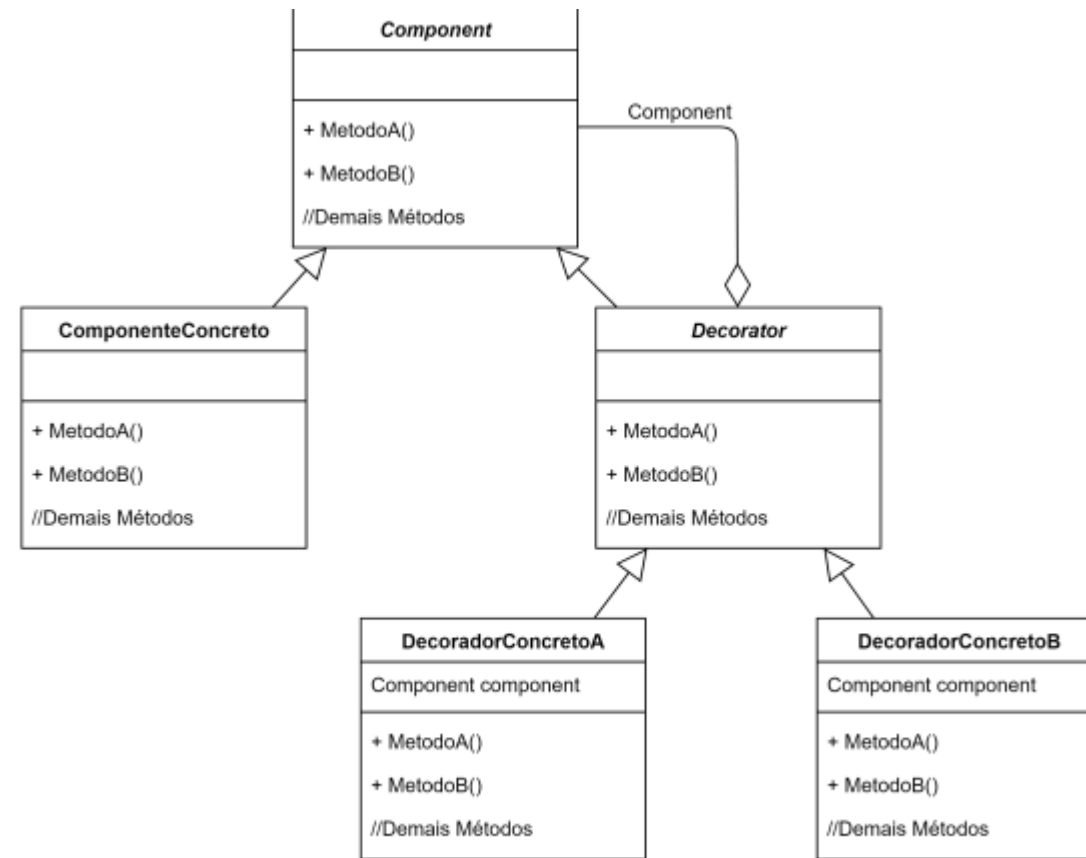


Diagrama de Classes

## Padrão Decorator - Componentes

- **Decorator** → cada decorator TEM-UM (engloba um) Component.
  - Isso significa que todo Decorator deve manter uma referência a um Component.
  - Os Decorators implementam a mesma interface ou classe abstrata que o componente que irão decorar.

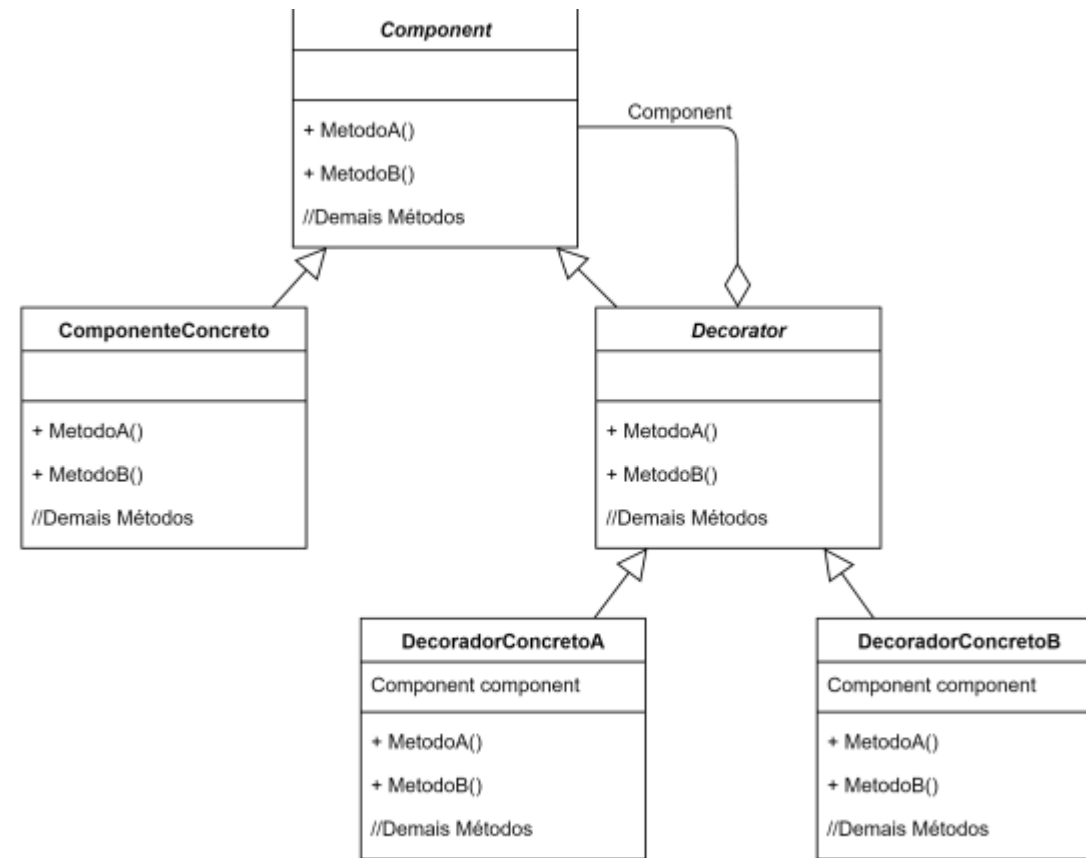


Diagrama de Classes

## Padrão Decorator - Componentes

- **decoratorConcreto** → implementam a classe abstrata ou interface Decorator, graças ao polimorfismo também são do supertipo Component.
- Podem adicionar novos métodos ao componente que decoram, no entanto, novo comportamento geralmente é adicionado fazendo cálculos antes e/ou depois de um método existente no componente.

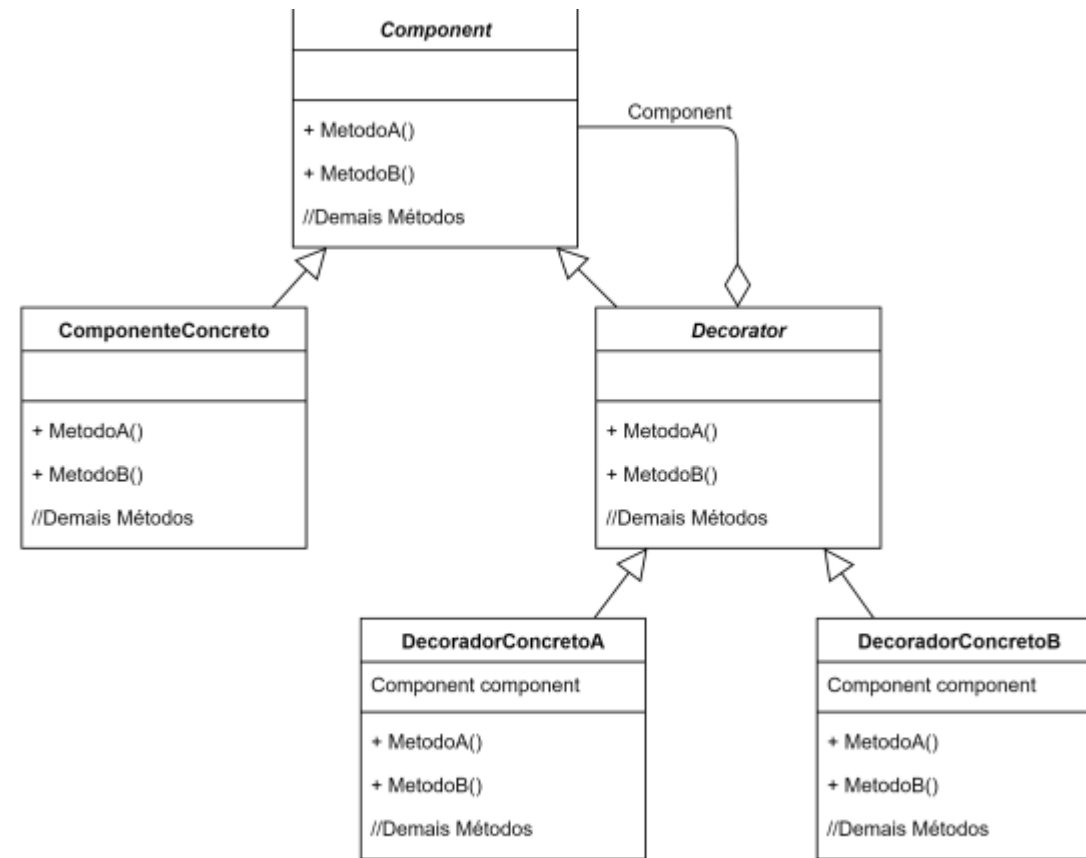


Diagrama de Classes

## **Padrão Decorator - Aplicabilidade**

- Quando for necessário adicionar comportamentos a objetos individuais de forma dinâmica e transparente, sem afetar outros objetos.
- Ao implementar comportamentos que podem ser fundamentais para determinados objetos e ao mesmo tempo desnecessários ou inapropriados a outros.
- Quando um grande número de extensões produziria uma grande quantidade de subclasses para suportar todas as combinações de comportamentos possíveis. Ou quando uma definição de classe estiver oculta ou indisponível para subclassificação.

## Padrão Decorator - Consequências

- Traz mais flexibilidade que herança estática. O padrão Decorator fornece uma maneira mais flexível de adicionar comportamentos aos componentes do que as herdando estaticamente.
- Com os decorators, os comportamentos podem ser adicionados e removidos aos componentes em tempo de execução simplesmente anexando e os desanexando.
- Pode gerar muitas classes e aumentar a complexidade do sistema.
- Fornecer classes de decorators diferentes para uma classe de componente específica permite misturar e combinar comportamentos.

## Padrão Decorator - Consequências

- Decorators facilitam a adição de um comportamento repetidas vezes a um componente.
- Os decorators fornecem comportamentos a um componente conforme a necessidade.
  - Ao invés de tentar prever todos os comportamentos possíveis em uma classe complexa e personalizável, pode-se definir uma classe simples e adicionar comportamentos incrementalmente por meio dos objetos decorators.
  - Isso evita o carregamento de comportamentos desnecessários ou inapropriados a uma classe.

## Padrão Decorator - Consequências



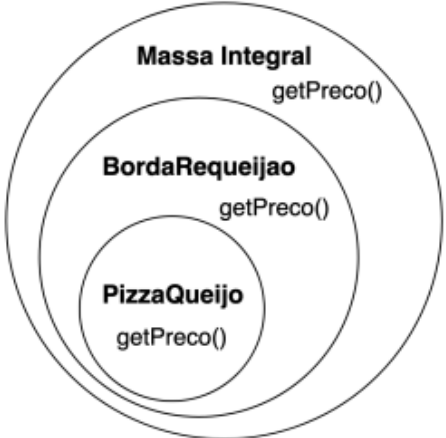
- Um decorator e seu componente não são idênticos.
  - Um decorator atua como um contêiner transparente.
  - Mas, do ponto de vista da identidade do objeto (objetos concretos), um componente decorado não é idêntico ao próprio componente.
  - Portanto, não se deve confiar na identidade do objeto ao usar decorators.
- A utilização do padrão decorator pode resultar em sistemas compostos por muitos objetos pequenos, todos parecidos.
  - Eles diferem apenas na maneira como estão interconectados.
  - Embora esses sistemas sejam fáceis de personalizar por quem os entende, eles podem ser difíceis de aprender e depurar.



# Problema da Pizzaria – Solução (Decorator)

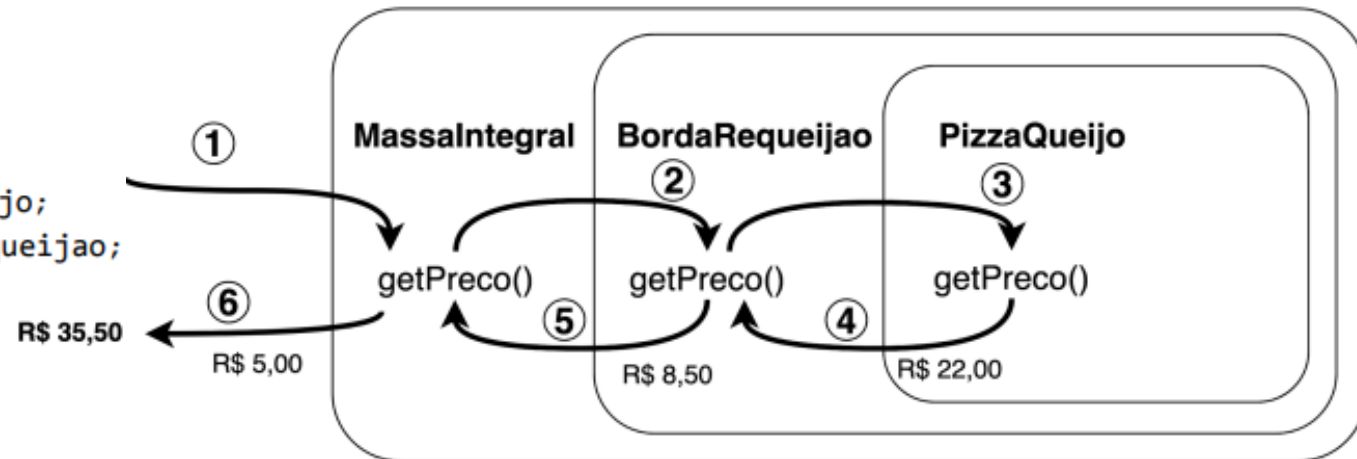
## Padrões de Projetos Estrutural – Decorator

- Suponha que queremos uma pizza de queijo com borda recheada de requeijão e massa integral.
- Vamos iniciar com uma pizza de queijo e decorá-la com os acréscimos.

<p>1 - Começamos com o objeto <code>PizzaQueijo</code>.</p>	<p>2 - Adicionamos borda recheada de requeijão.</p>	<p>3 - Adicionamos também a massa integral.</p>
		
<p><code>PizzaQueijo</code> herda os métodos de <code>Pizza</code> e implementa seu método <code>getPreco()</code>.</p>	<p>O objeto <code>BordaRequeijao</code> é um <i>decorator</i>, seu tipo deve ser igual ao do objeto que está decorando.</p> <p>Então, <code>BordaRequeijao</code> e <code>PizzaQueijo</code> devem ter o mesmo supertipo <code>Pizza</code>.</p> <p>Portanto <code>BordaRequeijao</code> também implementa um método <code>getPreco()</code>, e por meio do polimorfismo qualquer pizza englobada por <code>BordaRequeijao</code> pode ser tratada como um objeto do tipo <code>Pizza</code>.</p>	<p><code>MassaIntegral</code> também é um <i>decorator</i> então também tem o mesmo tipo que <code>PizzaQueijo</code> e implementa seu método <code>getPreco()</code>.</p> <p>Deste modo, <code>PizzaQueijo</code> englobada por <code>BordaRequeijao</code> e por <code>MassaIntegral</code> continua sendo uma <code>Pizza</code>, é possível fazer com ela tudo o que se pode fazer com <code>PizzaQueijo</code>, inclusive chamar seu método <code>getPreco</code>.</p>

- Observe como aconteceria o cálculo do valor total de uma pizza de queijo com borda recheada de requeijão e com massa integral.

1. O método `getPreco()` de `MassaIntegral` é chamado;
2. `MassaIntegral` chama o método `getPreco()` de `BordaRequeijao`.
3. `BordaRequeijao` chama `getPreco()` de `PizzaQueijo`;
4. `PizzaQueijo` retorna seu preço R\$ 22,00;
5. `BordaRequeijao` acrescenta seu custo (R\$ 8,50) a `PizzaQueijo`;
6. `MassaIntegral` acrescenta seu custo ao retorno de `BordaRequeijao`;



Fluxo da recursão das chamadas do método `getPreco()`

- Os passos de 1 a 6 realizam a soma  $R\$22,00 + R\$8,50 + R\$5,00$  que é valor da pizza somado aos valores dos acréscimos, totalizando R\$35,50.
- Pode ser difícil visualizar a implementação do fluxo descrito

# Decorator Implementação C#

Padrões de Projeto Estrutural I

Prof. Me Jefferson Passerini

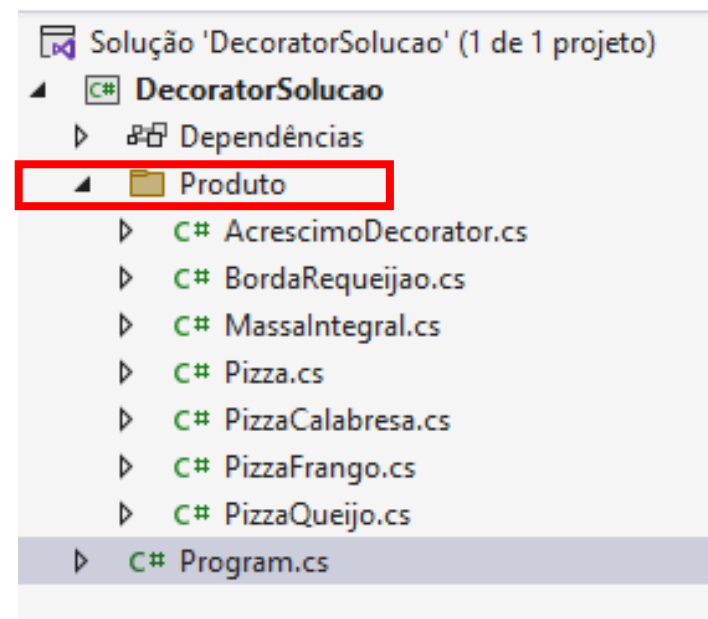






Diagrama de classes do exemplo implementando o padrão decorator

- Crie um projeto em C# denominado **DecoratorSolucao**.
- Crie uma pasta **Produto** que irá armazenar nossas classes
- Onde ao final teremos a seguinte estrutura.



- Inicialmente vamos criar a classe abstrata **Pizza**.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace DecoratorSolucao.Produto
8  {
9      10 referências
10     public abstract class Pizza
11     {
12         6 referências
13         protected string descricao { get; set; }
14         6 referências
15         protected double preco { get; set; }
16
17         11 referências
18         public abstract string getDescricao();
19         11 referências
20         public abstract double getPreco();
21     }
22 }
```

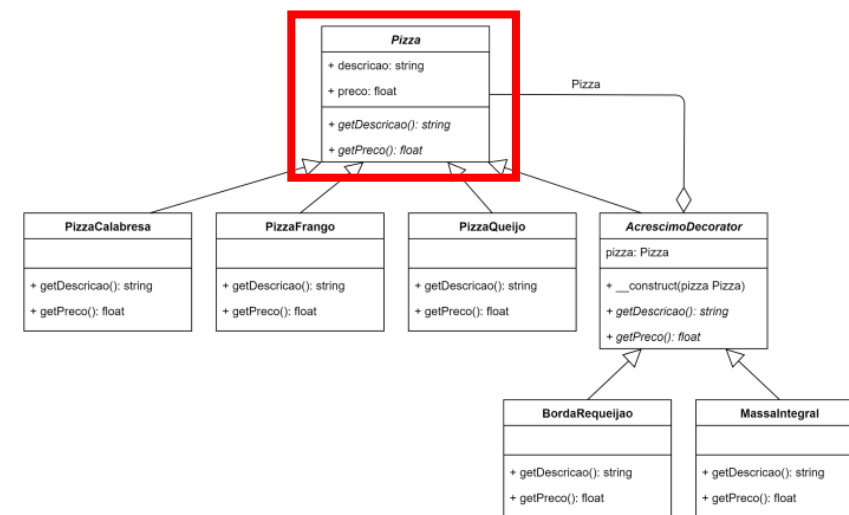
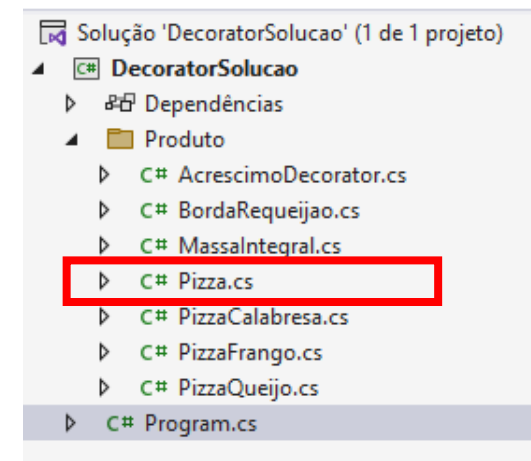


Diagrama de classes do exemplo implementando o padrão decorator



# Problema da Pizzaria – Solução

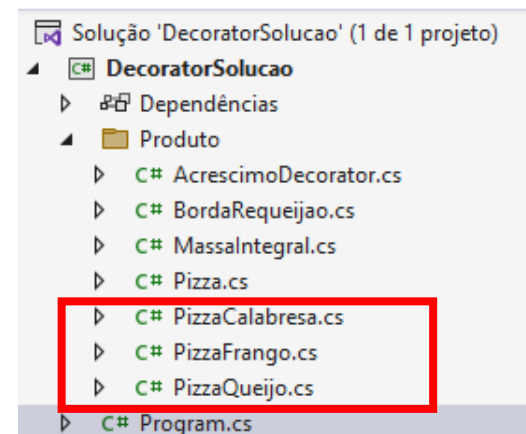
## Padrões de Projetos Estrutural – Decorator

- Crie as classes Concretas **PizzaCalabresa**, **PizzaFrango** e **PizzaQueijo**.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace DecoratorSolucao.Produto
8  {
9      1 referência
10     public class PizzaCalabresa : Pizza
11     {
12         0 referências
13         public PizzaCalabresa() {
14             this.descricao = "Deliciosa pizza de Calabresa";
15             this.preco = 25;
16         }
17
18         6 referências
19         public override string getDescricao()
20         {
21             return this.descricao;
22         }
23
24         6 referências
25         public override double getPreco()
26         {
27             return this.preco;
28         }
29     }
30 }
```



Diagrama de classes do exemplo implementando o padrão decorator



# Problema da Pizzaria – Solução

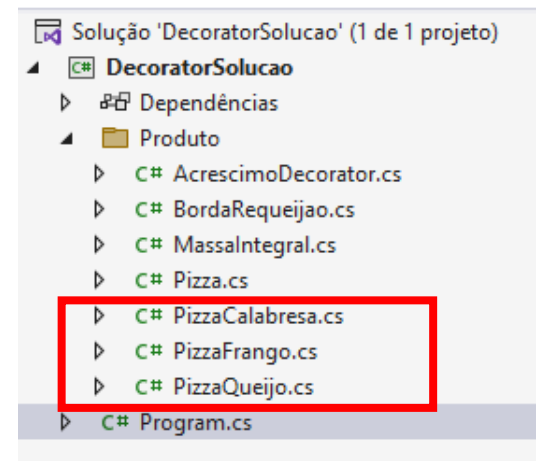
## Padrões de Projetos Estrutural – Decorator

- Crie as classes Concretas **PizzaCalabresa**, **PizzaFrango** e **PizzaQueijo**.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace DecoratorSolucao.Produto
8  {
9      1 referência
10     public class PizzaFrango: Pizza
11     {
12         0 referências
13         public PizzaFrango() {
14             this.descricao = "Deliciosa pizza de frango";
15             this.preco = 19;
16         }
17
18         6 referências
19         public override string getDescricao()
20         {
21             return this.descricao;
22         }
23
24         6 referências
25         public override double getPreco()
26         {
27             return this.preco;
28         }
29     }
30 }
```



Diagrama de classes do exemplo implementando o padrão decorator



# Problema da Pizzaria – Solução

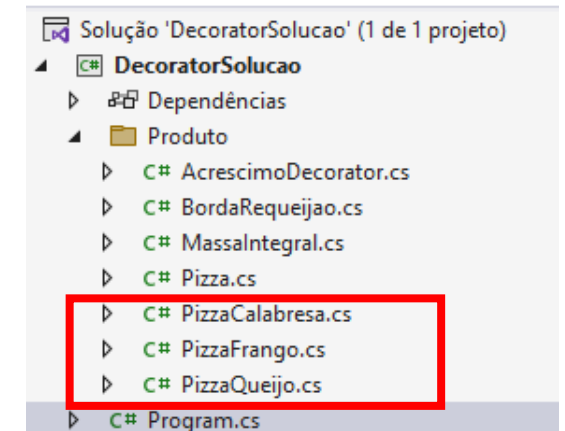
## Padrões de Projetos Estrutural – Decorator

- Crie as classes Concretas **PizzaCalabresa**, **PizzaFrango** e **PizzaQueijo**.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace DecoratorSolucao.Produto
8  {
9      2 referências
10     public class PizzaQueijo : Pizza
11     {
12         1 referência
13         public PizzaQueijo() {
14             this.descricao = "Deliciosa pizza de Queijo";
15             this.preco = 22;
16         }
17         6 referências
18         public override string getDescricao()
19         {
20             return this.descricao;
21         }
22         6 referências
23         public override double getPreco()
24         {
25             return this.preco;
26         }
27     }
```



Diagrama de classes do exemplo implementando o padrão decorator



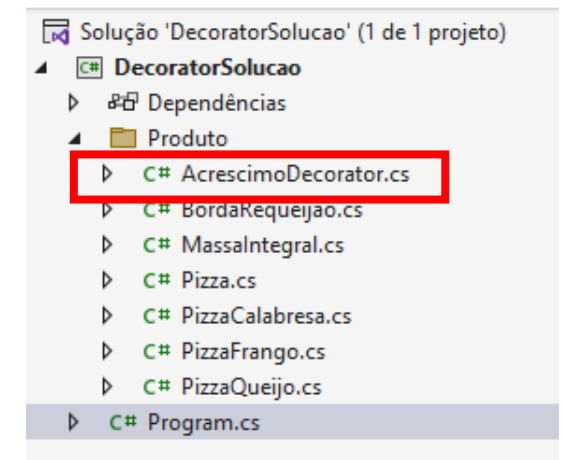


- Crie o Decorator (classe abstrata) – Supertipo (que estende Pizza e agregada um objeto do tipo Pizza).

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace DecoratorSolucao.Produto
8  {
9      2 referências
10     public abstract class AcrescimoDecorator: Pizza
11     {
12         protected Pizza pizza;
13
14         8 referências
15         public override abstract string getDescricao();
16
17         8 referências
18         public override abstract double getPreco();
19     }
20 }
```



Diagrama de classes do exemplo implementando o padrão decorator



- Crie o Decorators Concretos BordaRequeijao e Massa Integral.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace DecoratorSolucao.Produto
8  {
9      2 referências
10     public class BordaRequeijao : AcrescimoDecorator
11     {
12         private double valorBorda = 8.5;
13
14         1 referência
15         public BordaRequeijao(Pizza pizza) {
16             this.pizza = pizza;
17         }
18
19         7 referências
20         public override string getDescricao()
21         {
22             return this.pizza.getDescricao() + " Borda recheada de requeijão";
23         }
24
25         7 referências
26         public override double getPreco()
27         {
28             return this.pizza.getPreco()+valorBorda;
29         }
30     }
31 }
```

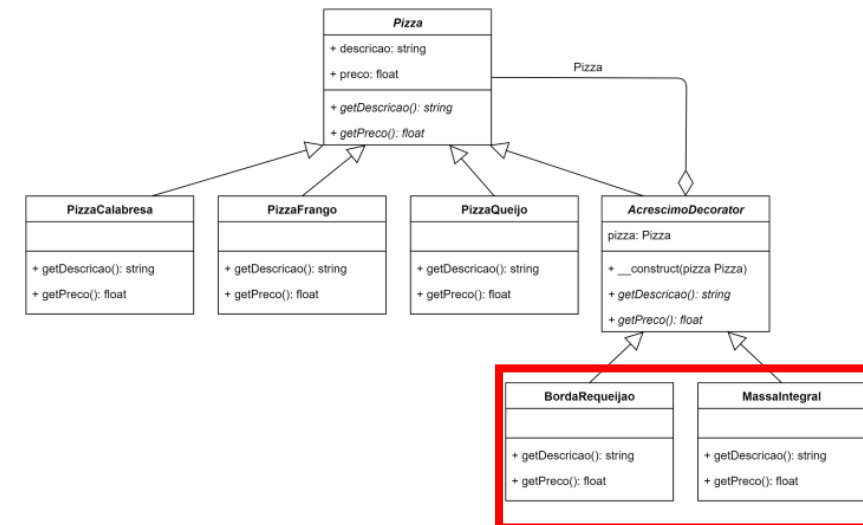
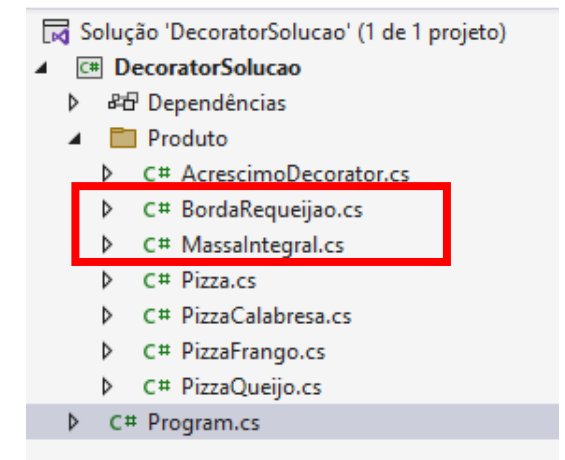


Diagrama de classes do exemplo implementando o padrão decorator



- Crie o Decorators Concretos BordaRequeijao e Massa Integral.

```
1  using System;
2  using System.Collections.Generic;
3  using System.IO.Pipes;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7
8  namespace DecoratorSolucao.Produto
9  {
10     2 referências
11     internal class MassaIntegral : AcrescimoDecorator
12     {
13         private double valorMassaIntegral=5;
14
15         1 referência
16         public MassaIntegral(Pizza pizza) {
17             this.pizza = pizza;
18         }
19
20         7 referências
21         public override string getDescricao()
22         {
23             return this.pizza.getDescricao() + " Massa Integral";
24         }
25
26         7 referências
27         public override double getPreco()
28         {
29             return this.pizza.getPreco() + valorMassaIntegral;
30         }
31     }
32 }
```

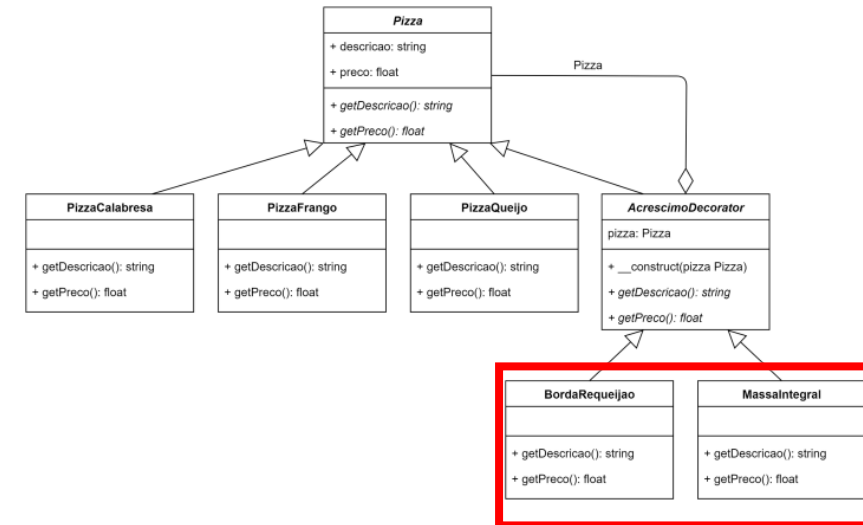
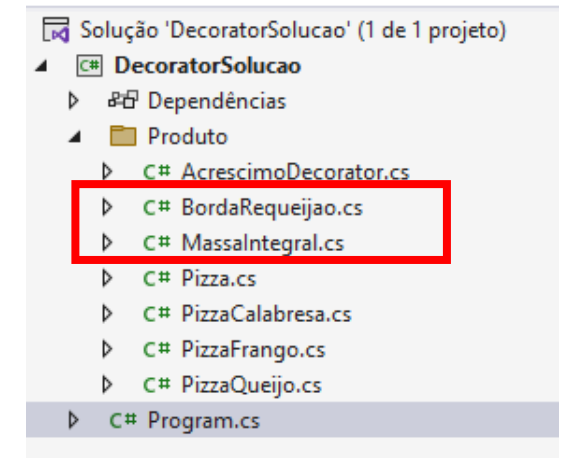


Diagrama de classes do exemplo implementando o padrão decorator



- Implemente os testes

```
1 using DecoratorSolucao.Produto;
2
3 //Criação de uma pizza
4 Console.WriteLine("Cria uma pizza de queijo");
5 Pizza pizzaQueijo = new PizzaQueijo();
6
7 //impressão de sua descrição
8 Console.WriteLine("Produto--> "+pizzaQueijo.getDescricao()+
9     " Valor R$"+pizzaQueijo.getPreco());
10
11
12 Console.WriteLine(" ");
13 //adiciona borda de requeijão
14 Console.WriteLine("Adiciona borda de requeijão");
15 //um decorator é criado para englobar a pizza
16 Pizza pizzaQueijoBorda = new BordaRequeijao(pizzaQueijo);
17
18 Console.WriteLine("Produto--> " + pizzaQueijoBorda.getDescricao()
19     + " Valor R$" + pizzaQueijoBorda.getPreco());
20
21 Console.WriteLine(" ");
22 //adiciona borda de requeijão
23 Console.WriteLine("Adiciona massa integral");
24 //um decorator é criado para englobar a pizza
25 Pizza pizzaQueijoBordaMassaIntegral = new MassaIntegral(pizzaQueijoBorda);
26
27 Console.WriteLine("Produto--> " + pizzaQueijoBordaMassaIntegral.getDescricao()
28     + " Valor R$" + pizzaQueijoBordaMassaIntegral.getPreco());
29
30 --
```

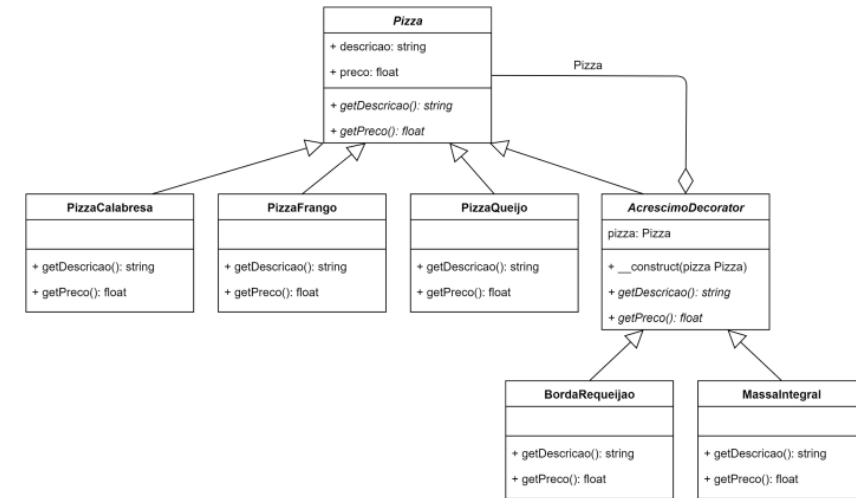
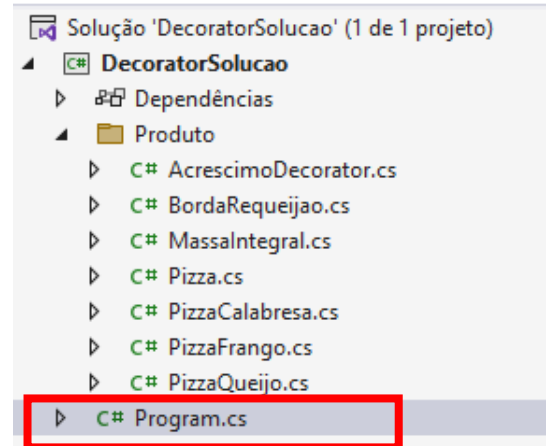
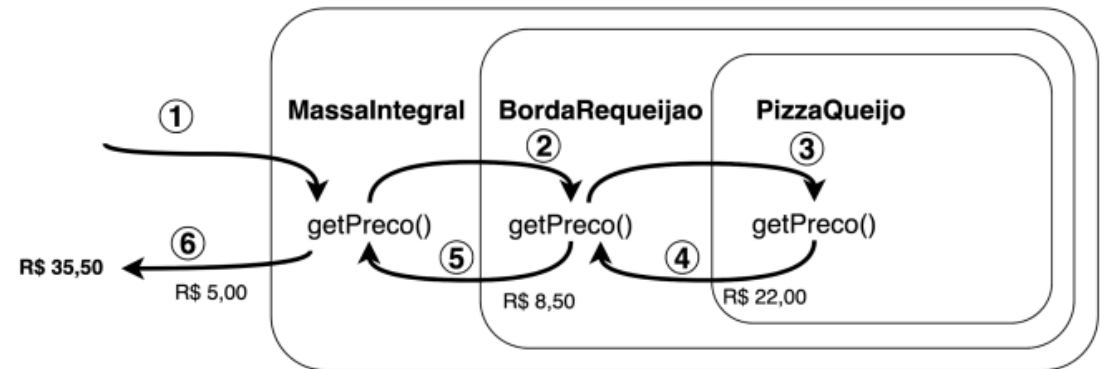


Diagrama de classes do exemplo implementando o padrão decorator



### • Resultado

1. O método `getPreco()` de `MassaIntegral` é chamado;
2. `MassaIntegral` chama o método `getPreco()` de `BordaRequeijao`.
3. `BordaRequeijao` chama `getPreco()` de `PizzaQueijo`;
4. `PizzaQueijo` retorna seu preço R\$ 22,00;
5. `BordaRequeijao` acrescenta seu custo (R\$ 8,50) a `PizzaQueijo`;
6. `MassaIntegral` acrescenta seu custo ao retorno de `BordaRequeijao`;



Fluxo da recursão das chamadas do método `getPreco()`

```
Cria uma pizza de queijo  
Produto--> Deliciosa pizza de Queijo Valor R$22
```

```
Adiciona borda de requeijão  
Produto--> Deliciosa pizza de Queijo Borda recheada de requeijão Valor R$30,5
```

```
Adiciona massa integral  
Produto--> Deliciosa pizza de Queijo Borda recheada de requeijão Massa Integral Valor R$35,5
```



# Decorator Implementação Java

Padrões de Projeto Estrutural I

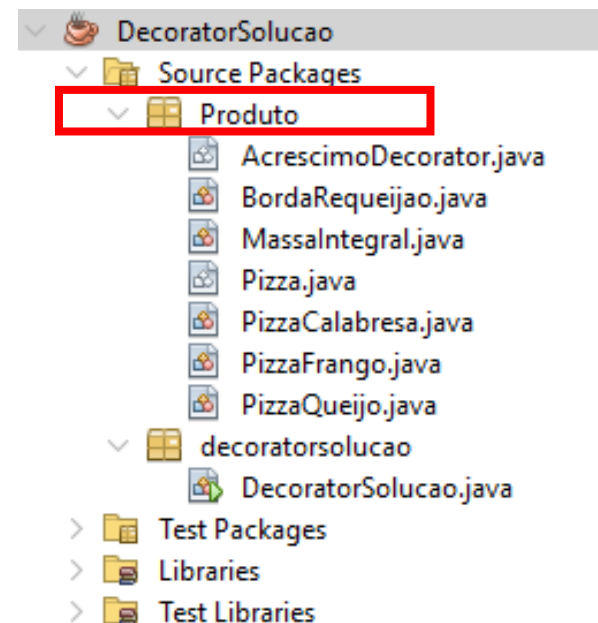
Prof. Me Jefferson Passerini





Diagrama de classes do exemplo implementando o padrão decorator

- Crie um projeto em C# denominado **DecoratorSolucao**.
- Crie uma pacote **Produto** que irá armazenar nossas classes
- Onde ao final teremos a seguinte estrutura.



- Inicialmente vamos criar a classe abstrata **Pizza**.

```
1  [+ ...4 lines
5  package Produto;
6
7  [+ /**...4 lines */
11 public abstract class Pizza {
12
13     protected String descricao;
14     protected double preco;
15
16     public abstract String getDescricao();
17     public abstract double getPreco();
18
19 }
20
```

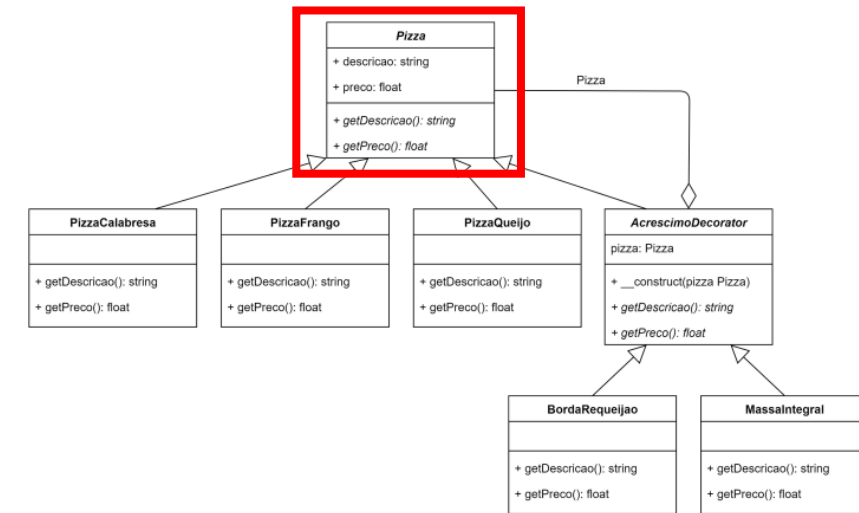
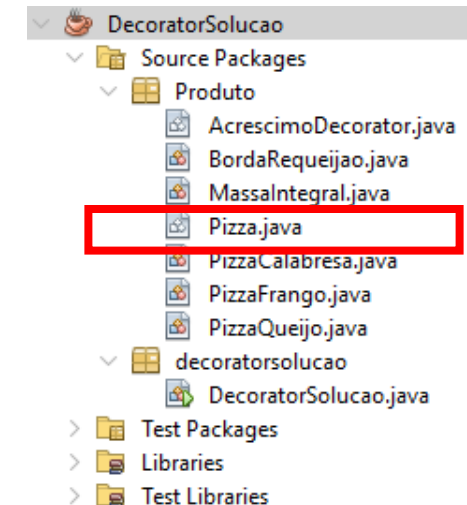


Diagrama de classes do exemplo implementando o padrão decorator



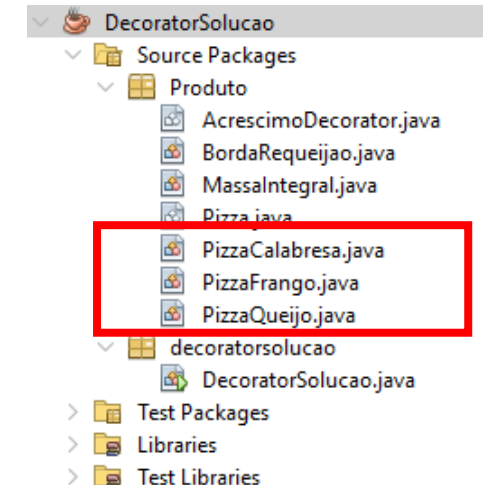


- Crie as classes Concretas **PizzaCalabresa**, **PizzaFrango** e **PizzaQueijo**.

```
1  ...4 lines
5  package Produto;
6
7  /**...4 lines */
11 public class PizzaCalabresa extends Pizza {
12
13     public PizzaCalabresa() {
14         this.descricao = "Deliciosa pizza de calabresa";
15         this.preco = 25;
16     }
17     @Override
18     public String getDescricao() {
19         return this.descricao;
20     }
21
22     @Override
23     public double getPreco() {
24         return this.preco;
25     }
26
27 }
```



Diagrama de classes do exemplo implementando o padrão decorator



# Problema da Pizzaria – Solução

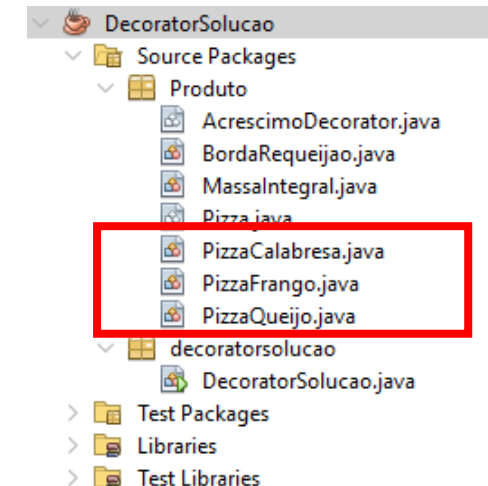
## Padrões de Projetos Estrutural – Decorator

- Crie as classes Concretas **PizzaCalabresa**, **PizzaFrango** e **PizzaQueijo**.

```
1  ...4 lines
5  package Produto;
6
7  /**...4 lines */
11 public class PizzaFrango extends Pizza {
12
13     public PizzaFrango() {
14         this.descricao="Deliciosa pizza de frango";
15         this.preco=19;
16     }
17     @Override
18     public String getDescricao() {
19         return this.descricao;
20     }
21
22     @Override
23     public double getPreco() {
24         return this.preco;
25     }
26
27 }
```



Diagrama de classes do exemplo implementando o padrão decorator

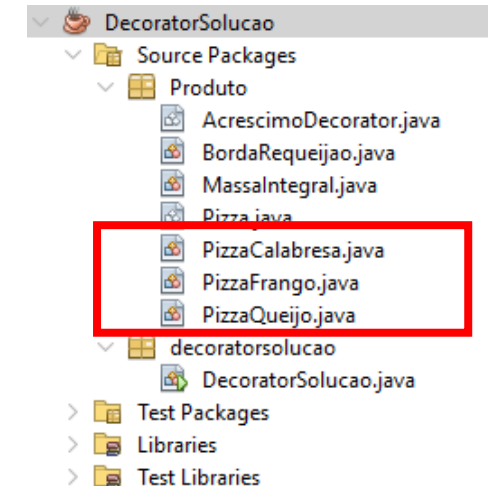


- Crie as classes Concretas **PizzaCalabresa**, **PizzaFrango** e **PizzaQueijo**.

```
1  ...4 lines
5  package Produto;
6
7  /**...4 lines */
11 public class PizzaQueijo extends Pizza {
12
13     public PizzaQueijo() {
14         this.descricao="Deliciosa pizza de queijo";
15         this.preco=22;
16     }
17     @Override
18     public String getDescricao() {
19         return this.descricao;
20     }
21
22     @Override
23     public double getPreco() {
24         return this.preco;
25     }
26
27
28 }
```



Diagrama de classes do exemplo implementando o padrão decorator

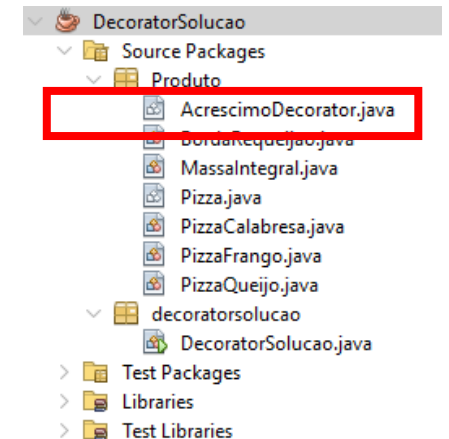


- Crie o Decorator (classe abstrata) – Supertipo (que estende Pizza e agregada um objeto do tipo Pizza).

```
1  ...4 lines
5  package Produto;
6
7  /**...4 lines */
11 public abstract class AcrescimoDecorator extends Pizza {
12
13     protected Pizza pizza;
14
15     @Override
16     public abstract String getDescricao();
17
18     @Override
19     public abstract double getPreco();
20
21 }
```



Diagrama de classes do exemplo implementando o padrão decorator



- Crie o Decorators Concretos BordaRequeijao e Massa Integral.

```
1  ...4 lines
5  package Produto;
6
7  /**...4 lines */
11 public class BordaRequeijao extends AcrescimoDecorator {
12
13     private double valorBorda = 8.5;
14
15     public BordaRequeijao(Pizza pizza) {
16         this.pizza = pizza;
17     }
18     @Override
19     public String getDescricao() {
20         return this.pizza.getDescricao()+" Borda recheada de requeijao";
21     }
22
23     @Override
24     public double getPreco() {
25         return this.pizza.getPreco()+valorBorda;
26     }
27
28
29 }
```

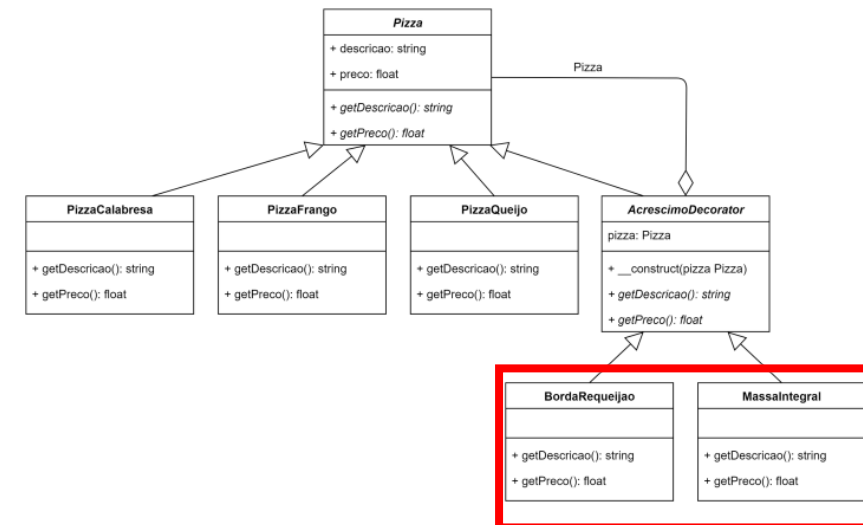
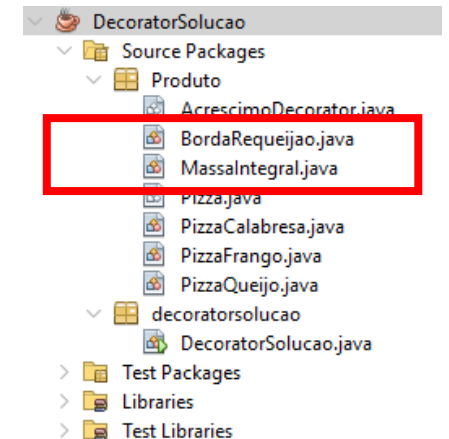


Diagrama de classes do exemplo implementando o padrão decorator



- Crie o Decorators Concretos BordaRequeijao e Massa Integral.

```
1  ...4 lines
5  package Produto;
6
7  /**...4 lines */
11 public class MassaIntegral extends acrescimoDecorator{
12
13     private double valorMassaIntegral = 5;
14
15     public MassaIntegral(Pizza pizza){
16         this.pizza = pizza;
17     }
18
19     @Override
20     public String getDescricao() {
21         return this.pizza.getDescricao()+" Massa Integral";
22     }
23
24     @Override
25     public double getPreco() {
26         return this.pizza.getPreco()+valorMassaIntegral;
27     }
28
29 }
```

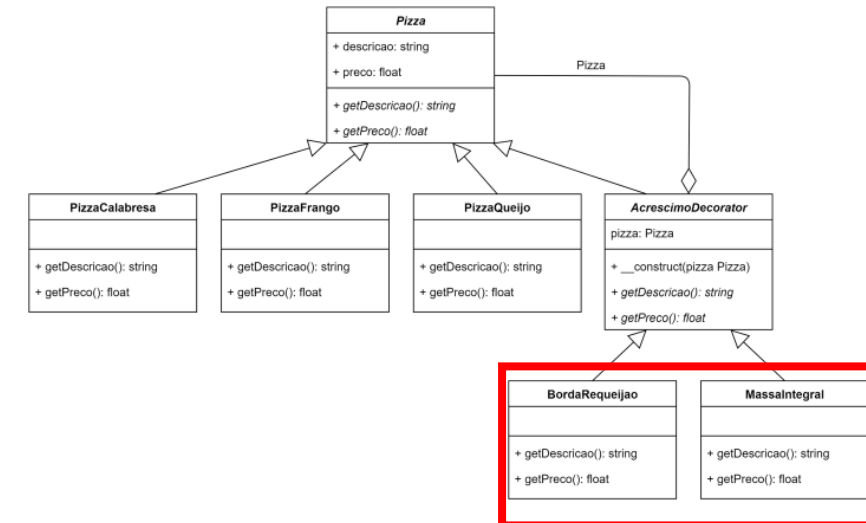
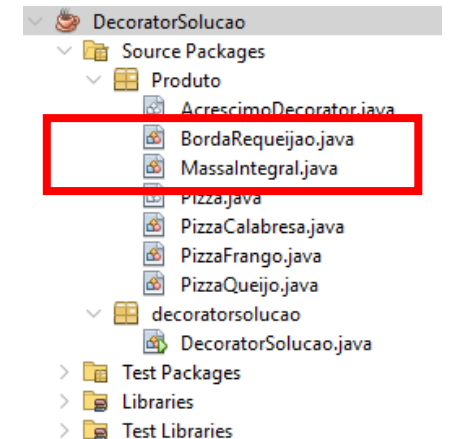


Diagrama de classes do exemplo implementando o padrão decorator



- Implemente os testes

```
1  ...4 lines
5  package decoratorsolucao;
6
7  import Produto.BordaRequeijao;
8  import Produto.MassaIntegral;
9  import Produto.Pizza;
10 import Produto.PizzaQueijo;
11
12 /**...4 lines */
16 public class DecoratorSolucao {
17
18     /**...3 lines */
21     public static void main(String[] args) {
22         System.out.println("Cria uma pizza de queijo");
23         Pizza pizzaQueijo = new PizzaQueijo();
24         System.out.println("Produto --> "+pizzaQueijo.getDescricao()+
25             " Valor R$"+pizzaQueijo.getPreco());
26
27         System.out.println(" ");
28         System.out.println("Adiciona borda de requeijao");
29         Pizza pizzaQueijoBorda = new BordaRequeijao(pizzaQueijo);
30         System.out.println("Produto --> "+pizzaQueijoBorda.getDescricao()+
31             " Valor R$"+pizzaQueijoBorda.getPreco());
32
33         System.out.println(" ");
34         System.out.println("Adiciona Massa Integral");
35         Pizza pizzaQueijoBordaMassaIntegral = new MassaIntegral(pizzaQueijoBorda);
36         System.out.println("Produto --> "+pizzaQueijoBordaMassaIntegral.getDescricao()+
37             " Valor R$"+pizzaQueijoBordaMassaIntegral.getPreco());
38     }
39 }
```

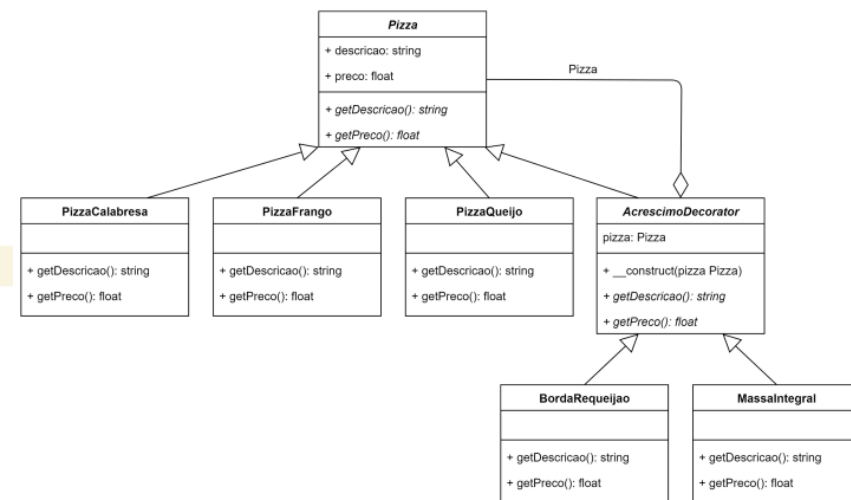
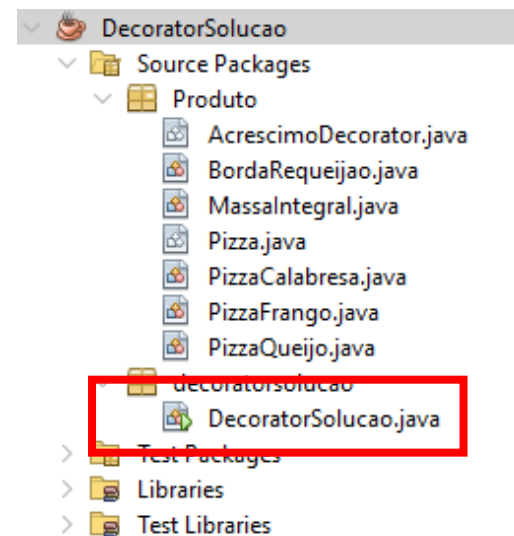


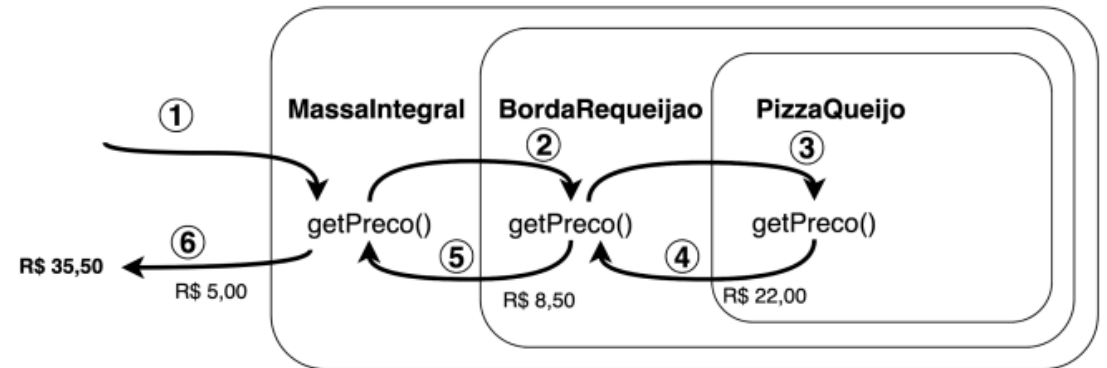
Diagrama de classes do exemplo implementando o padrão decorator





### • Resultado

1. O método `getPreco()` de `MassaIntegral` é chamado;
2. `MassaIntegral` chama o método `getPreco()` de `BordaRequeijao`.
3. `BordaRequeijao` chama `getPreco()` de `PizzaQueijo`;
4. `PizzaQueijo` retorna seu preço R\$ 22,00;
5. `BordaRequeijao` acrescenta seu custo (R\$ 8,50) a `PizzaQueijo`;
6. `MassaIntegral` acrescenta seu custo ao retorno de `BordaRequeijao`;



Fluxo da recursão das chamadas do método `getPreco()`

run:

Cria uma pizza de queijo

Produto --> Deliciosa pizza de queijo Valor R\$22.0

Adiciona borda de requeijao

Produto --> Deliciosa pizza de queijo Borda recheada de requeijao Valor R\$30.5

Adiciona Massa Integral

Produto --> Deliciosa pizza de queijo Borda recheada de requeijao Massa Integral Valor R\$35.5

BUILD SUCCESSFUL (total time: 0 seconds)