

## ***Decorator***

### Padrões Estruturais

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

#### **Motivação (Por que utilizar?)**

Às vezes, queremos adicionar responsabilidades 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.

Para ilustrar tal situação, tomemos como exemplo o sistema de uma Pizzaria, onde o cliente pode acrescentar características adicionais a sua pizza. É importante dizer que os donos da pizzaria em breve criarão novos sabores de pizza e novos acréscimos.

A pizzaria possui 3 sabores de pizza em seu cardápio:

- Pizza de Frango - R\$19,00;
- Pizza de Calabresa - R\$25,00;
- Pizza de Queijo - R\$22,00;

Os acréscimos são:

- Borda recheada com requeijão - R\$8,50;
- Massa Integral - R\$5,00;

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 + R\$8,50 pela borda de requeijão = R\$27,50**

Todas as pizzas devem possuir um preço e uma descrição. Seria inviável criar uma classe para cada combinação possível, como *PizzaCalabresaBordaRequeijao* ou *PizzaQueijoMassaIntragral*. A cada novo tipo de pizza ou novo acréscimo o número de classes cresceria exponencialmente.

É possível utilizar herança.

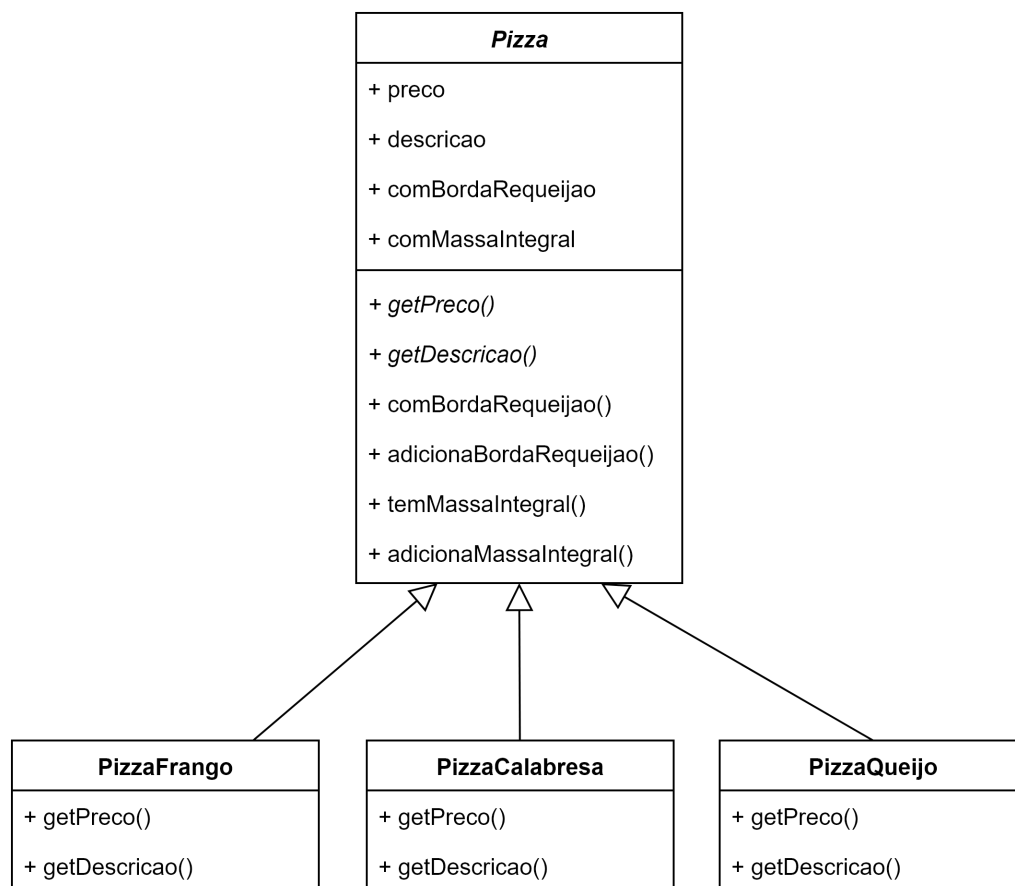


Diagrama de classes do exemplo (cenário 1)

A classe `Pizza` é uma classe abstrata que possui os atributos:

- **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.

E os seguintes métodos:


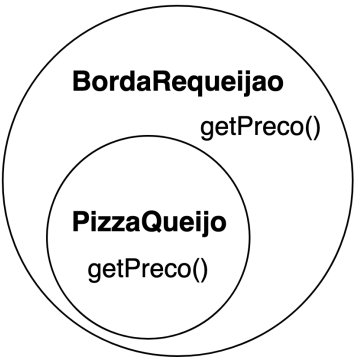
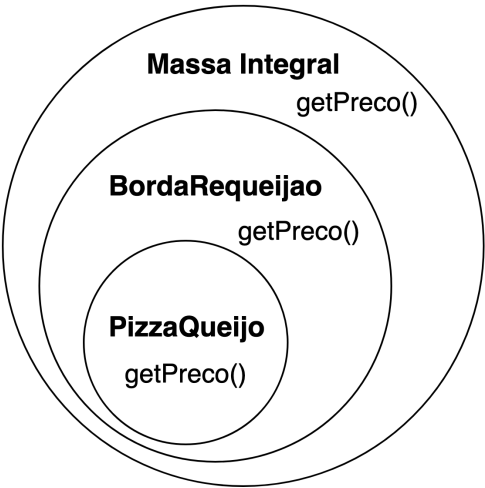
- **getPreco():** Método abstrato - todas as classes que herdam de `Pizza` deve o implementar - define o preço da pizza.
- **getDescricao():** Método abstrato - todas as classes que herdam de `Pizza` deve o implementar - define a descrição da pizza.
- **temBordaRecheada():** retorna o atributo `comBordaRequeijao`.
- **adicionaBordaRequeijao():** define `comBordaRequeijao` como true.
- **temMassaIntegral():** retorna o atributo `comMassaIntegral`.
- **adicionaMassaIntegral():** define `comMassaIntegral` como true.

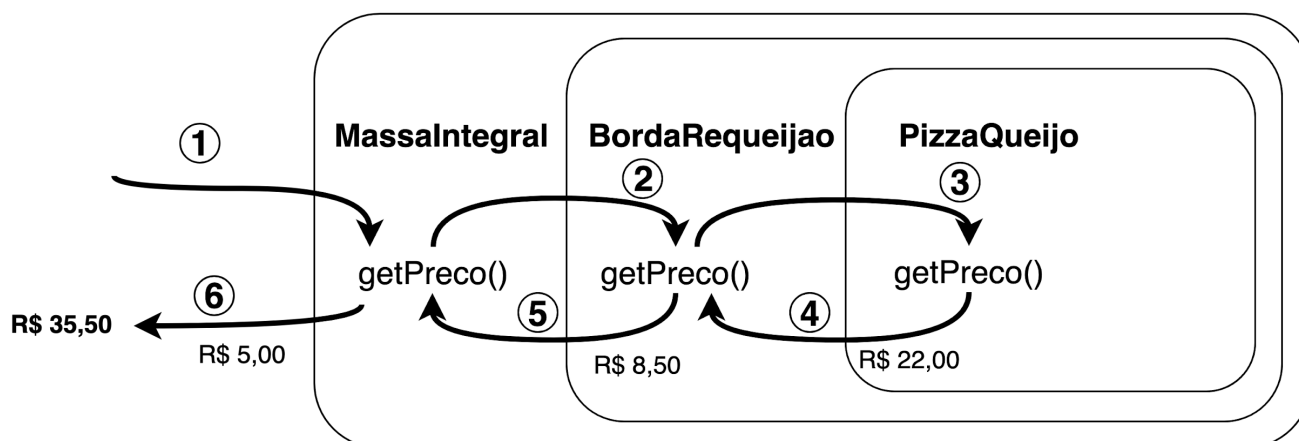
Deste modo **PizzaFrango**, **PizzaCalabresa** e **PizzaQueijo** teriam que implementar seu próprio método `getPreco()` e herdariam todos os demais métodos e atributos da classe abstrata `Pizza`.

O problema do uso de herança no caso das pizzas é a inflexibilidade que ela trás. Todos os acréscimos são atribuídos às subclasses em tempo de compilação. Imagine que a pizzaria comesse 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.

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



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`;

Os passos de 1 a 6 realizam a soma  $R\$22,00 + R\$8,50 + R\$5,00$  que é valor da pizza somado ao valores dos acréscimos, totalizando  $R\$35,50$ .

Por enquanto pode ser difícil imaginar como seria a implementação do fluxo de processamento acima, vamos ver o que sabemos até aqui sobre o padrão *Decorator*.

1 - Os *decorators* e objetos os quais decoram devem ter o mesmo supertipo.

Para que a afirmação acima seja possível precisamos criar um supertipo, então, vamos começar pela classe abstrata **Pizza**.

```
abstract class Pizza
{
    protected string $descricao = 'Descrição não definida';
    protected float $preco;

    //Todas as classes que herdam de pizza precisam implementar este método.
    abstract public function getDescricao(): string;

    //Todas as classes que herdam de pizza precisam implementar este método.
    abstract public function getPreco(): float ;
}
```

Agora que já temos a classe **pizza** podemos criar as subclasses **PizzaFrango**, **PizzaCalabresa** e **PizzaQueijo**.

```
class PizzaFrango extends Pizza
{
    public function __construct()
    {
        //Adicionamos uma descrição para pizza de frango;
        $this->descricao = 'Deliciosa pizza de frango';
    }

    //Implementação de getDescricao feita por PizzaFrango.
    public function getDescricao(): string
    {
        return $this->descricao;
    }

    //Implementação de getPreco feita por PizzaFrango.
    public function getPreco(): float
    {
        return 19;
    }
}
```

```
class PizzaCalabresa extends Pizza
{
    public function __construct()
    {
        //Adicionamos uma descrição para pizza de calabresa;
        $this->descricao = 'Deliciosa pizza de calabresa';
    }

    //Implementação de getDescricao feita por PizzaCalabresa.
    public function getDescricao(): string
    {
        return $this->descricao;
    }

    //Implementação de getPreco feita por PizzaCalabresa.
    public function getPreco(): float
    {
        return 25;
    }
}
```

```

class PizzaQueijo extends Pizza
{
    public function __construct()
    {
        //Adicionamos uma descrição para pizza de queijo;
        $this->descricao = 'Deliciosa pizza de queijo';
    }

    //Implementação de getDescricao feita por PizzaQueijo.
    public function getDescricao(): string
    {
        return $this->descricao;
    }

    //Implementação de getPreco feita por PizzaQueijo.
    public function getPreco(): float
    {
        return 22;
    }
}

```

## 2 - Pode-se utilizar um ou mais decorators para englobar um objeto.

Sabendo que podem existir vários tipos de *decorators* vamos criar uma classe abstrata para eles também. Devemos lembrar que os *decorators* e objetos decorados por eles devem ter o mesmo supertipo então a classe abstrata **AcrescimoDecorator** também deve ser do tipo **Pizza**.

```

abstract class AcrescimoDecorator extends Pizza
{
    protected Pizza $pizza;

    //O decorator precisa manter uma referência ao objeto decorado.
    public function __construct(Pizza $pizza)
    {
        $this->pizza = $pizza;
    }

    //Vamos forçar que cada decorador implemente sua própria descrição.
    //Para concatenar a descrição do acréscimo a descrição da pizza.
    abstract public function getDescricao(): string ;

    //Vamos forçar que cada decorador implemente seu preço.
    //Para somar o preço do acréscimo ao preço da pizza.
    abstract public function getPreco(): float ;
}

```

```
class BordaRequeijao extends AcrescimoDecorator
{
    public function getDescricao(): string
    {
        //Retorna a descrição de Pizza concatenada a descrição de BordaRequeijao.
        return $this->pizza->getDescricao() . ' + Borda recheada de requeijão';
    }

    public function getPreco(): float
    {
        //Retorna o preço de Pizza somado ao preço de BordaRequeijao.
        return $this->pizza->getPreco() + 8.50;
    }
}
```

```
class MassaIntegral extends AcrescimoDecorator
{
    public function getDescricao(): string
    {
        //Retorna a descrição de Pizza concatenada a descrição de MassaIntegral.
        return $this->pizza->getDescricao() . ' + Massa integral';
    }

    public function getPreco(): float
    {
        //Retorna o preço de Pizza somado ao preço de MassaIntegral.
        return $this->pizza->getPreco() + 5;
    }
}
```

3 - O *decorator* adiciona seu comportamento ao objeto decorado antes ou depois de delegar o resto do trabalho para ele.

4 - Objetos podem ser decorados a qualquer momento, ou seja, em tempo de execução.

Vamos fazer um teste para fixar a idéia que o decorator propõe:

```
//==== Criação de uma Pizza ====
$pizzaQueijo = new PizzaQueijo();

//Impressão de sua descrição
echo 'Descrição: ' . $pizzaQueijo->getDescricao() . '<br>';

//Impressão de seu preço
echo 'Preço: R$' . $pizzaQueijo->getPreco() . '<br>';

//==== Adição borda de requeijão a pizza ====
echo '<br> Adição de borda de requeijão<br>';
//Um decorator é criado e passa a englobar a pizza
$pizzaQueijoBorda = new BordaRequeijao($pizzaQueijo);

//Impressão da descrição do decorator + pizza
echo 'Descrição: ' . $pizzaQueijoBorda->getDescricao() . '<br>';

//Impressão do preço do decorator + pizza
echo 'Preço: R$' . $pizzaQueijoBorda->getPreco() . '<br>';

//==== Adição massa integral a pizza ====
echo '<br> Adição de massa integral<br>';
//Mais um decorator é criado e passa a englobar o primeiro decorator e a pizza
$pizzaQueijoBordaMassaIntegral = new MassaIntegral($pizzaQueijoBorda);

//Impressão da descrição do primeiro decorator + segundo decorator + pizza
echo 'Descrição: ' . $pizzaQueijoBordaMassaIntegral->getDescricao() . '<br>';

//Impressão do preço do primeiro decorator + segundo decorator + pizza
echo 'Preço: R$' . $pizzaQueijoBordaMassaIntegral->getPreco() . '<br>';
```

Saída:

```
Descrição: Deliciosa pizza de queijo
Preço: R$22

Adição de borda de requeijão
Descrição: Deliciosa pizza de queijo + Borda recheada de requeijão
Preço: R$30.5

Adição de massa integral
Descrição: Deliciosa pizza de queijo + Borda recheada de requeijão + Massa integral
Preço: R$35.5
```

No teste acima nós adicionamos dois acréscimos a pizza de queijo, e eles foram adicionados em tempo de execução. Isso torna nosso código muito flexível, pois caso um novo acréscimo ou pizza fosse criado pelos donos da pizzaria, bastaria criar uma nova classe sem a necessidade de modificar as classes existentes.



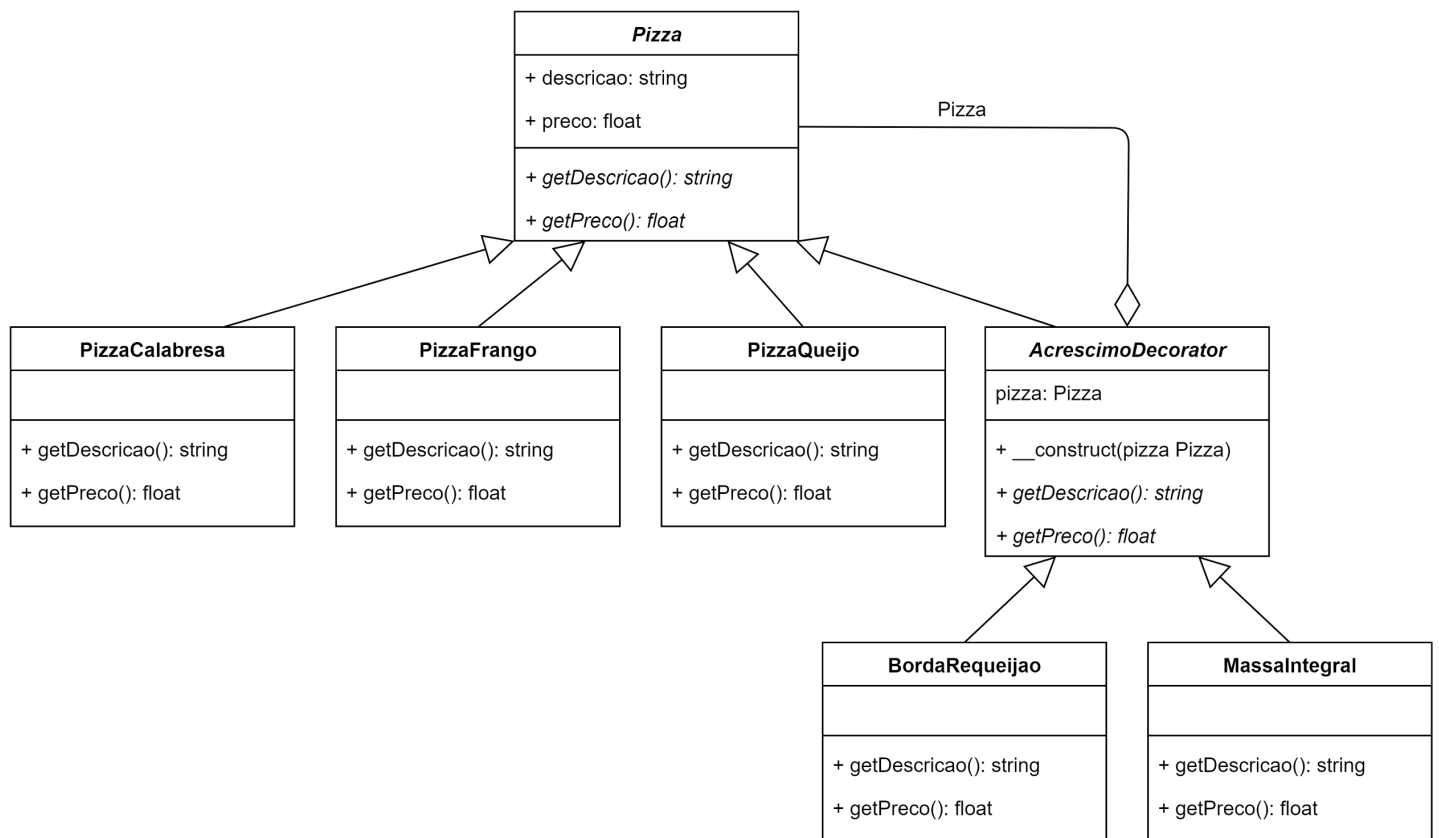


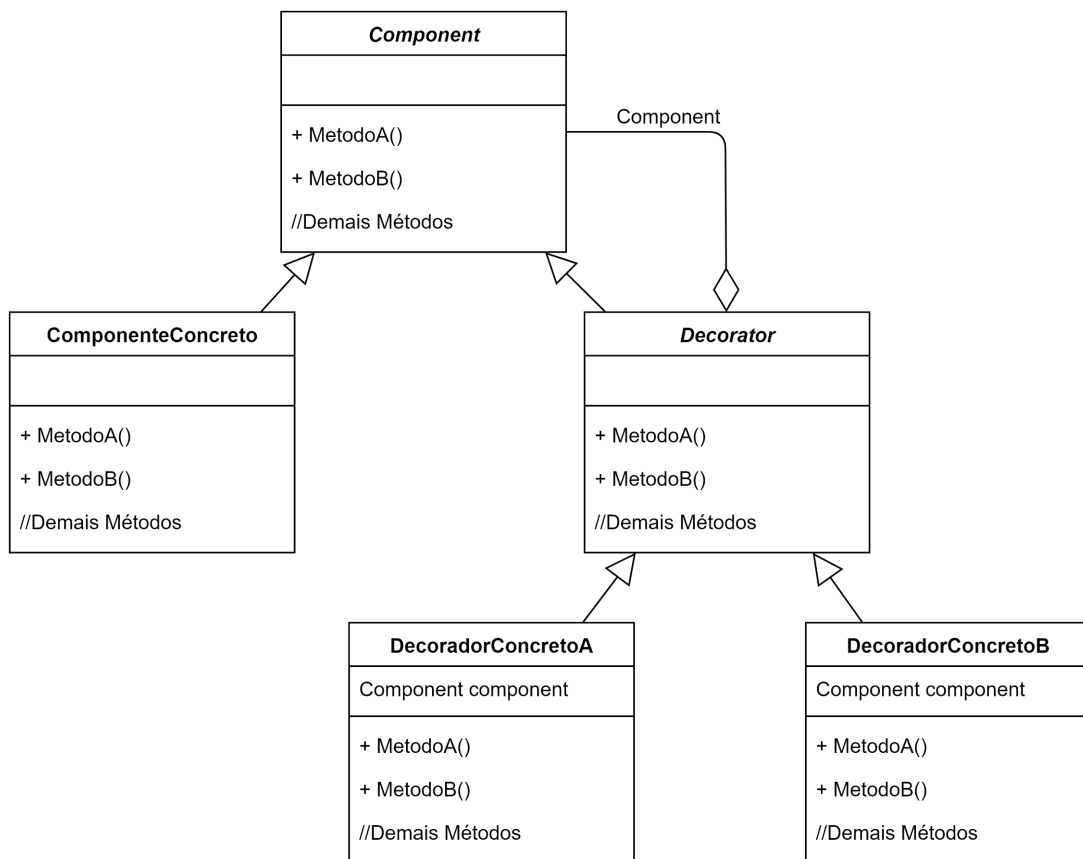
Diagrama de classes do exemplo implementando o padrão *decorator*

### Aplicabilidade (Quando utilizar?)

- 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.

## 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. Ele estende Component.
- **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.
- **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**

## 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.
- *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.
- 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.