Padrão Decorator:

Padrão Decorator: Como Decorar Objetos em C# de Forma Flexível

Rayssa Melissa Alves Ribeiro - RA: 2304618

Guilherme Pazitte - RA: 2307317

1. O que é o Padrão Decorator?

O **Decorator** é um padrão estrutural que permite adicionar funcionalidades a um **objeto** de forma **dinâmica** e **flexível**, sem precisar alterar o código da classe original.

Exemplo Simples:

Imagine que você tem um **carro básico** e quer adicionar algumas funcionalidades extras, como **ar-condicionado** e **rodas especiais**. Ao invés de modificar a classe do carro, você **decoraria** o carro com essas funcionalidades, criando novos objetos que envolvem o carro original.

2. Quando Usar o Padrão Decorator?

Use o padrão Decorator quando:

- Você não quer ou não pode **modificar** o código da classe original.
- Deseja **adicionar funcionalidades de forma modular** e flexível.
- Quer evitar criar **múltiplas subclasses** para cada combinação de funcionalidades (o que ocorre em sistemas com muitas variações).

3. Como Funciona o Padrão Decorator em C#?

O Decorator usa uma **abstração comum** para garantir que tanto o objeto original quanto o decorador tenham a mesma interface. Assim, podemos envolver um objeto com várias camadas, onde cada camada adiciona uma funcionalidade específica.

Estrutura Básica em C#:

1. **Componente Base (Classe Base)**: O objeto que será decorado.
2. **Decorator (Decorador)**: Cada decorador envolve um objeto e adiciona funcionalidades extras.

4. Exemplo Visual: Carro e Acessórios

Passo 1: Carro Base

Primeiro, criamos o nosso **carro básico** (com custo fixo), que vai ser a classe base.

```
public interface ICarro
{
   double Custo();
}
```

```
public class Carro : ICarro
{
    public double Custo()
    {
        return 20000;  // Custo do carro básico
    }
}
```

Passo 2: Criando o Decorator

Agora, criamos uma classe DecoratorCarro, que implementa a interface ICarro e recebe um objeto ICarro como parâmetro para decorar esse objeto.

```
public abstract class DecoradorCarro : ICarro
{
    protected ICarro _carro;

    public DecoradorCarro(ICarro carro)
    {
        _carro = carro;
    }

    public virtual double Custo()
    {
        return _carro.Custo();
    }
}
```

Passo 3: Decorando o Carro com Funcionalidades

Agora, vamos criar os decoradores específicos, como **ar-condicionado** e **rodas especiais**.

```
public class ArCondicionado : DecoradorCarro
{
    public ArCondicionado(ICarro carro) : base(carro) { }

    public override double Custo()
```

```
    {
        return _carro.Custo() + 1500;  // Custo do ar-condicionado
    }
}

public class RodasEspeciais : DecoradorCarro
{
    public RodasEspeciais(ICarro carro) : base(carro) { }

    public override double Custo()
    {
        return _carro.Custo() + 2000;  // Custo das rodas especiais
    }
}
```

Passo 4: Combinando as Funcionalidades

Agora, podemos criar um carro que tenha **ar-condicionado** e **rodas especiais**, combinando os decoradores.

```
class Program
{
    static void Main(string[] args)
    {
        // Carro básico
        ICarro carroSimples = new Carro();

        // Decorando com Ar-condicionado
        ICarro carroComAr = new ArCondicionado(carroSimples);

        // Decorando com Rodas Especiais
        ICarro carroComArERodas = new RodasEspeciais(carroComAr);

        // Custo final
        Console.WriteLine($"Custo do carro: {carroComArERodas.Custo()}");
    }
}
```

Resultado do Código:

A saída do código será:

Custo do carro: 23500

O custo final é 20.000 (carro básico) + 1.500 (ar-condicionado) + 2.000 (rodas especiais) = 23.500.

5. Vantagens do Decorator em C#

O padrão **Decorator** é uma solução muito útil quando:

- Você **precisa adicionar funcionalidades a objetos** de forma modular e reutilizável.
- Quer evitar o uso excessivo de **herança**, o que pode resultar em um sistema difícil de manter.
- Permite **combinar** diferentes comportamentos em **tempo de execução**, sem a necessidade de criar várias classes para cada combinação.

6. Comparando com Herança

Problema com Herança:

Quando você tenta adicionar novas funcionalidades à classe base, criando várias subclasses para cada combinação, o sistema fica difícil de gerenciar e expandir. Vamos ver um exemplo simples.

```
public class CarroComArERodas : Carro
{
   public CarroComArERodas() : base()
   {
      // Adiciona funcionalidades específicas de ar-condicionado e rodas
   }
}
```

À medida que novas funcionalidades são adicionadas (exemplo: **som melhorado**, **teto solar**), você teria que criar uma nova classe para cada combinação, o que resulta em **muitas classes** e torna o sistema difícil de manter.

Com o Decorator:

Usando o Decorator, podemos **compor** as funcionalidades de maneira simples e clara, como mostramos no exemplo do carro, sem precisar criar uma nova classe para cada combinação.

7. Exemplo Real: Sistema de Notificação

Imaginemos um sistema de **notificação** onde podemos ter diferentes formas de envio, como **e-mail**, **SMS**, **push** e **WhatsApp**. Usaremos o padrão Decorator para adicionar essas funcionalidades de forma modular.

Passo 1: Interface de Notificação

Primeiro, criamos a interface INotificacao.

```
public interface INotificacao
{
   void Enviar();
}
```

Passo 2: Notificação Básica (E-mail)

Aqui, criamos a notificação básica que envia um **e-mail**.

```
public class NotificacaoEmail : INotificacao
{
   public void Enviar()
   {
      Console.WriteLine("Enviando e-mail");
   }
}
```

Passo 3: Decorando com SMS

Agora, criamos o decorador **SMS**.

```
public class NotificacaoSms : INotificacao
{
   private readonly INotificacao _notificacao;

   public NotificacaoSms(INotificacao notificacao)
   {
      _notificacao = notificacao;
   }
```

```
    public void Enviar()
    {
        _notificacao.Enviar();
        Console.WriteLine(" + Enviando SMS");
    }
}
```

Passo 4: Decorando com Push

Agora, decoramos com o **Push**.

```
public class NotificacaoPush : INotificacao
{
    private readonly INotificacao _notificacao;

    public NotificacaoPush(INotificacao notificacao)
    {
        _notificacao = notificacao;
    }

    public void Enviar()
    {
        _notificacao.Enviar();
        Console.WriteLine(" + Enviando Push");
    }
}
```

Passo 5: Combinando as Funcionalidades

Agora, podemos criar uma **notificação completa** que envie **e-mail**, **SMS** e **Push**.

csharp

Copiar código

```
class Program
{
    static void Main(string[] args)
    {
        // Criando a notificação básica (E-mail)
        INotificacao notificacao = new NotificacaoEmail();

        // Decorando com SMS
        notificacao = new NotificacaoSms(notificacao);
```

```
        // Decorando com Push
        notificacao = new NotificacaoPush(notificacao);

        // Enviando a notificação
        notificacao.Enviar();
    }
}
```

Resultado do Código:

A saída será:

```
Enviando e-mail
 + Enviando SMS
 + Enviando Push
```

8. Conclusão

O **padrão Decorator** é uma excelente maneira de adicionar funcionalidades a objetos de forma **flexível** e **modular**, sem alterar a classe original. Em C#, ele permite que você componha funcionalidades de maneira simples, usando **abstração**, **composição** e **decoradores**. Isso traz benefícios como:

- **Flexibilidade** para adicionar ou remover comportamentos.
- **Evita heranças complexas**, permitindo um sistema mais simples e manutenível.
- **Facilidade para adicionar novas funcionalidades** sem afetar o código existente

9. Link GITHUB

GuilhermePazitte/
Decorator-Pattern

0 Contributors

0 Issues

0 Stars

0 Forks