Programação Orientada a Objetos
Aula 9

- Revisão
- SOL - ID

Programação Orientada a Objetos

- SOLID
  - Boas práticas de programação
  - Código limpo
  - Manutenção
  - Fácil de testar
  - Escalabilidade

Programação Orientada a Objetos

- “S” OLID “SRP”
- Princípio da responsabilidade única
- Uma classe deve ter somente uma razão para existir e alterar, com isso o que quero dizer é que ele vai ter somente uma responsabilidade
    - Exemplo que não segue o solid
        - class Empregado{
          public void SalvarNoBanco(){};
          public void GerarRelatorio(){}/
          }
    - Exemplo que segue o solid

```
class Empregado{
        public void SalvarNoBanco(){};
}
class GeradorDeRelatorio{
        public void GerarRelatorio(){};
}
```

Programação Orientada a Objetos

- “S” OLID “SRP”

- Manutenibilidade Aprimorada:
  - Ao separar responsabilidades em classes ou módulos distintos, torna-se mais fácil compreender, modificar e corrigir o código. Cada componente lida com uma única responsabilidade, reduzindo a complexidade geral do sistema.
- Facilita a Reutilização de Código:
  - Classes ou módulos com responsabilidades únicas são mais propensos a serem reutilizados em diferentes partes do sistema. Isso promove a modularidade e a reutilização do código em diversos contextos.
- Testabilidade Aprimorada:
  - Componentes com responsabilidades únicas são mais fáceis de testar, pois é mais simples isolar e verificar o comportamento de uma única responsabilidade. Isso contribui para a criação de testes mais específicos e eficazes.
- Aumenta a Coesão:
  - O SRP promove a coesão, que é a medida da relação entre os membros de uma classe. Uma classe coesa tem métodos e atributos que estão logicamente relacionados. Isso leva a um código mais organizado e compreensível.
- Redução do Acoplamento:
  - Componentes que seguem o SRP tendem a ter baixo acoplamento, pois cada um lida com uma única responsabilidade e não depende excessivamente de outros componentes. Isso torna o sistema mais flexível e menos propenso a efeitos colaterais não desejados.

Programação Orientada a Objetos

- “S” OLID “SRP”

- Facilita a Evolução do Sistema:
  - Quando uma responsabilidade específica precisa ser modificada, adicionada ou removida, é mais fácil fazer essas alterações sem afetar outras partes do sistema. O SRP facilita a evolução do sistema ao longo do tempo.
- Promove Boas Práticas de Design:
  - Ao seguir o SRP, os desenvolvedores são incentivados a adotar boas práticas de design, como a criação de classes e métodos concisos, coesos e focados. Isso resulta em um código mais limpo e sustentável.
- Aprimora a Legibilidade do Código:
  - Classes que aderem ao SRP são mais legíveis, pois têm uma única razão para mudar. Isso facilita a compreensão do propósito de cada componente no código.
- Simplifica o Entendimento do Negócio:
  - Ao dividir o sistema em componentes que refletem responsabilidades de negócios específicas, o SRP contribui para um entendimento mais claro e alinhado com os requisitos do negócio.
- Melhora a Escalabilidade:
  - Sistemas com componentes independentes e responsabilidades bem definidas são mais escaláveis. Novas funcionalidades podem ser adicionadas de maneira mais eficiente, e a manutenção do código é facilitada.

Em resumo, o SRP promove uma arquitetura de software mais sustentável, modular e fácil de entender, tornando o código mais resiliente às mudanças e facilitando o desenvolvimento e a manutenção de sistemas de software ao longo do tempo.

Programação Orientada a Objetos

- SOLID “SRP” Exercício do Princípio da Responsabilidade Única:
- Temos um programa que é responsável para armazenar informações de um produto e calcular o preço do produto baseado em diferentes descontos. Quero que seja modificado o código abaixo para aplicar o SRP

```
public class Produto
{
    public string Nome{ get; set; }
    public decimal Preco{ get; set; }
    public decimal Desconto{ get; set; }
public void SalvarNoBanco()
    {
        // Lógica para salvar o produto no banco de dados
    }
public decimal CalcularPrecoFinal()
    {
        // Lógica para calcular o preço final com desconto
        decimal precoComDesconto= Preco - (Preco  * Desconto/ 100);
        return precoComDesconto;
    }
}
```

Programação Orientada a Objetos

- S ”O” LID “OPEN CLOSED”
- Este princípio declara que uma classe deve ser aberta para extensão, mas fechada para modificação. Ou seja, você deve poder adicionar novas funcionalidades a um sistema sem alterar o código existente.
- Extensibilidade sem Modificação:
    - A principal vantagem do OCP é permitir a extensão do sistema sem a necessidade de modificar o código existente. Novas funcionalidades podem ser adicionadas por meio da introdução de novas classes ou módulos, sem alterar o código já existente.
- Menor Risco de Introdução de Bugs:
    - Como o código existente não é alterado para adicionar novas funcionalidades, há menos risco de introduzir erros ou bugs. Isso facilita a manutenção e evolução do sistema ao longo do tempo.
- Facilita a Reutilização de Código:
    - Classes e módulos que seguem o OCP são mais propensos a serem reutilizáveis em diferentes contextos. A capacidade de estender um sistema sem modificar o código existente favorece a reutilização de código em novos cenários.
- Promove a Hierarquia de Abstrações:
    - O OCP muitas vezes leva à criação de hierarquias de abstrações, onde classes base definem comportamentos comuns e classes derivadas estendem ou especializam esses comportamentos. Isso melhora a organização e a compreensão do código.

Programação Orientada a Objetos

- S ”O” LID “OPEN CLOSED”

- Encoraja o Design por Contrato:
  - O OCP está relacionado ao conceito de Design por Contrato, onde as classes base e derivadas estabelecem contratos claros sobre seu comportamento. Isso leva a uma melhor compreensão das responsabilidades de cada classe na hierarquia.
- Adaptação a Mudanças de Requisitos:
  - A conformidade com o OCP torna o sistema mais adaptável a mudanças nos requisitos. Novas funcionalidades podem ser incorporadas de maneira mais eficiente, e a introdução de alterações é menos propensa a causar efeitos colaterais.
- Facilita a Manutenção e Evolução:
  - O OCP torna o código mais fácil de manter e evoluir ao longo do tempo. As extensões podem ser feitas de maneira isolada, sem a necessidade de revisitar e modificar partes já existentes do sistema.
- Aumentar a Flexibilidade do Sistema:
  - Sistemas que seguem o OCP tendem a ser mais flexíveis, permitindo que novos comportamentos e funcionalidades sejam adicionados de maneira modular. Isso facilita a adaptação do sistema a requisitos em constante mudança.
- Melhora a Escalabilidade do Código:
  - O OCP contribui para a escalabilidade do código, permitindo que o sistema cresça em termos de funcionalidades e complexidade sem aumentar proporcionalmente a complexidade do código existente.

Programação Orientada a Objetos

```
public class Retangulo
{
    public double Altura { get; set; }
    public double Largura { get; set; }
}
```

```
public class CalculadoraArea
{
    public double CalcularArea(Retangulo retangulo)
    {
        return retangulo.Altura * retangulo.Largura;
    }
}
```

```
public class CalculadoraVolume
{
    public double CalcularVolume(Retangulo retangulo, double profundidade)
    {
        return retangulo.Altura * retangulo.Largura * profundidade;
    }
}
```

Programação Orientada a Objetos

```
// Violando o OCP
public class CalculadoraArea
{
    public double CalcularArea(object forma)
    {
        if (forma is Retangulo)
        {
            var retangulo = (Retangulo)forma;
            return retangulo.Altura * retangulo.Largura;
        }
        else if (forma is Circulo)
        {
            var circulo = (Circulo)forma;
            return Math.PI * circulo.Raio * circulo.Raio;
        }
        else
        {
            throw new ArgumentException("Tipo de forma não suportado");
        }
    }
}
```

Problemas:

A classe CalculadoraArea precisa ser modificada sempre que uma nova forma é adicionada.

A cada nova forma, a classe fica mais complexa e difícil de manter.

Violamos o OCP, pois a classe está aberta para modificação.

Programação Orientada a Objetos

```
public abstract class Forma
{
    public abstract double CalcularArea();
}

public class Retangulo : Forma
{
    public double Altura { get; set; }
    public double Largura { get; set; }
    public override double CalcularArea()
    {
        return Altura * Largura;
    }
}
```

```
public class Circulo : Forma
{
    public double Raio { get; set; }
    public override double CalcularArea()
    {
        return Math.PI * Raio * Raio;
    }
}

public class Calculadora
{
    public double CalcularArea(Forma forma)
    {
        return forma.CalcularArea();
    }
}
```

Programação Orientada a Objetos

Vantagens:

- Adicionamos a capacidade de calcular a área de diferentes formas sem modificar a classe Calculadora.
- Introduzimos uma hierarquia de herança que nos permite estender o comportamento sem modificar o código existente.
- A classe Calculadora é fechada para modificação e aberta para extensão.

- O OCP nos incentiva a projetar sistemas de software de forma que possamos adicionar novas funcionalidades sem modificar o código existente. Isso torna o sistema mais flexível, extensível e fácil de manter ao longo do tempo. Em C#, isso é frequentemente alcançado através do uso de herança e polimorfismo.

Programação Orientada a Objetos

- S O ”L” ID “Liskov Substitution Principle (Princípio da Substituição de Liskov - L)”
- Este princípio afirma que os objetos de uma classe derivada devem poder substituir objetos da classe base sem afetar a integridade do programa.
- Flexibilidade na Hierarquia de Classes:
    - Obedecendo ao LSP, classes derivadas podem ser usadas de forma transparente no lugar de suas classes base. Isso proporciona uma maior flexibilidade na construção e extensão de hierarquias de classes.
- Polimorfismo Sem Surpresas:
    - O LSP garante que o polimorfismo (capacidade de um objeto ser referenciado por uma referência de sua classe base) funcione de maneira intuitiva. Isso significa que, ao substituir uma classe base por uma derivada, o comportamento esperado é mantido.
- Manutenção Simplificada:
    - Classes derivadas podem ser adicionadas ao sistema sem a necessidade de modificar o código existente. Isso facilita a manutenção do código, reduzindo o risco de introduzir erros ao estender a funcionalidade do sistema.
- Reusabilidade do Código:
    - O LSP promove a reusabilidade do código ao permitir que classes base e derivadas sejam intercambiáveis. Isso significa que as classes derivadas podem ser utilizadas em diferentes partes do sistema, aumentando a modularidade.

Programação Orientada a Objetos

- S O ”L” ID “Liskov Substitution Principle (Princípio da Substituição de Liskov - L)”

- Testabilidade Melhorada:
    - Classes que seguem o LSP são mais fáceis de testar. Se uma classe derivada pode ser substituída pela classe base sem impactar os resultados dos testes, é mais provável que a hierarquia de classes seja robusta e livre de erros.
- Encoraja o Uso Adequado da Herança:
    - Ao seguir o LSP, os desenvolvedores são incentivados a usar a herança de maneira apropriada, criando relações significativas entre classes base e derivadas. Isso evita hierarquias de herança confusas ou mal concebidas.
- Facilita Design por Contrato:
    - O LSP está relacionado ao conceito de Design por Contrato, onde as classes base e derivadas estabelecem contratos claros sobre seu comportamento. Isso leva a uma melhor compreensão das responsabilidades de cada classe na hierarquia.
- Adaptação a Mudanças:
    - A conformidade com o LSP torna o sistema mais adaptável a mudanças futuras. Adicionar novas classes derivadas ou modificar as existentes não impactará o código que já depende das classes base.

Programação Orientada a Objetos

- S O ”L” ID “Liskov Substitution Principle (Princípio da Substituição de Liskov - L)”

- Facilita a Compreensão do Código:
    - O LSP promove uma estrutura de código mais intuitiva. Ao substituir uma classe por outra sem surpresas, a compreensão do comportamento do código torna-se mais direta.

- o Liskov Substitution Principle contribui para um design orientado a objetos mais sólido e facilita a criação de sistemas que são mais flexíveis, extensíveis e fáceis de manter.

Programação Orientada a Objetos

- Exercicios

Programação Orientada a Objetos

- Exercicios
- Vamos criar um software em que inicialmente não seguimos o princípio Open/Closed (OCP). Temos alguns carros e devemos calcular o preço de aluguel de acordo com o tipo de carro.

Programação Orientada a Objetos

```
public class Carro
{
    public string Modelo { get; set; }
    public int Ano { get; set; }
}
public class CalculadoraAluguel
{
    public double CalcularPrecoAluguel(Carro carro)
    {
        double preco = 0;
        if (carro.Ano >= 2020)
        {
            preco = 100;
        }
        else
        {
            preco = 50;
        }
        return preco;
    }
}
```

```
public class CarroElétrico : Carro{    public int AutonomiaBateria { get; set; }}
public class CalculadoraAluguel{
    public double CalcularPrecoAluguel(Carro carro)    {
        double preco = 0;
        if (carro.Ano >= 2020)
        {           preco = 100;       }
        else if (carro is CarroElétrico)       {
            // Violando o OCP - Adicionando lógica específica para CarroElétrico
            var carroEletrico = (CarroElétrico)carro;
            if (carroEletrico.AutonomiaBateria > 300)           {
                preco += 50;           }       }
        else
        {
            preco = 50;
        }

        return preco;
    }
}
```

Programação Orientada a Objetos

- Exercicios
- Vamos criar um software em que inicialmente não seguimos o princípio LISKOV (LSP). Temos um guarda chuva comum e outro automático, vamos implementar como eles devem abrir.

Programação Orientada a Objetos

```
public class GuardaChuva
{
    public virtual void Abrir()
    {
        Console.WriteLine("Guarda-chuva aberto.");
    }

    public virtual void Fechar()
    {
        Console.WriteLine("Guarda-chuva fechado.");
    }
}
```

```
public class GuardaChuvaAutomatico : GuardaChuva{
    private bool _estaAberto;
    public override void Abrir()    {
        Console.WriteLine("Guarda-chuva automático abrindo.");
        _estaAberto = true;
    }
    public override void Fechar()    {
        Console.WriteLine("Guarda-chuva automático fechando.");
        _estaAberto = false;
    }
    public void Agitar()    {
        if (_estaAberto)
        {
            Console.WriteLine("Guarda-chuva automático agitado.");
        }
        else       {
            Console.WriteLine("Não é possível agitar um guarda-chuva fechado.");
        }
    }
}
```

Programação Orientada a Objetos

- SOLID “SRP” Exercício do Princípio da Responsabilidade Única:
- Agora que aplicou o SRP no nosso problema, temos uma nova solicitação da loja que usa o sistema, ela quer aplicar uma promoção de natal para os produtos que contém no seu nome a palavra “natal”, para estes produtos será aplicado um desconto de 20% os outros produtos terão um desconto de 15%
- A loja só tinha opção de pagamento em dinheiro, mas agora ela implementou o meio de pagamento por PIX e Cartão de crédito, com isso foi solicitado para que quando for pago com PIX tenha um desconto a mais de 10% e quando o pagamento for em cartão será aumentado o valor do produto em 5%, com isso altere o software para calcular o preço final com estas novas regras

Programação Orientada a Objetos

- SOLID “SRP” Exercício do Princípio da Responsabilidade Única:
- Agora a loja quer também salvar os preços e produtos em um banco de dados, para isso vamos usar o banco SQLITE, vai ajudar a lembrar os conceitos de banco de dados vistos. Para poder fazer isso segue um exemplo de código que salva em um banco de dados SQLite.
    - Abra o Visual Studio e selecione o seu projeto.
    - Vá para Tools > NuGet Package Manager > Manage NuGet Packages for Solution.
    - Na guia Browse, procure por System.Data.SQLite.
    - Selecione o pacote System.Data.SQLite e clique em Install para adicionar o pacote ao seu projeto.

Programação Orientada a Objetos

```
    private const string ConnectionString = "Data Source=Products.db;Version=3;";
    public void SaveToDatabase(Product product)
    {
        using (var connection = new SQLiteConnection(ConnectionString))
        {
            connection.Open();
            using (var command = connection.CreateCommand())
            {
                command.CommandText = "INSERT INTO Products (Name, Price, DiscountPercentage) VALUES (@Name, @Price, @DiscountPercentage)";
                command.Parameters.AddWithValue("@Name", product.Name);
                command.Parameters.AddWithValue("@Price", product.Price);
                command.Parameters.AddWithValue("@DiscountPercentage", product.DiscountPercentage);
                command.ExecuteNonQuery();
            }
        }
    }
}
```

Programação Orientada a Objetos

- Agora queremos mostrar os produtos que tem na loja, para isso implemente

```
public void MostrarProdutos()
 {
    using (var connection = new SQLiteConnection(ConnectionString))
    {
       connection.Open();
       using (var command = connection.CreateCommand())
       {
          command.CommandText = "SELECT * FROM Products";
          using (var reader = command.ExecuteReader())
          {
             while (reader.Read())
             {
                Console.WriteLine($"Product: {reader["Name"]}, Price: {reader["Price"]}, Discount: {reader["DiscountPercentage"]}");
             }
          }
       }
    }
 }
```

Programação Orientada a Objetos

- Baseado no código anterior, altere crie um novo metodo, para Trazer todos os Produtos
- public Produtos[] TragaTodosOsProdutos(), e use está implementação no MostrarProdutos()

SENAI

Serviço Nacional de Aprendizagem Industrial

PELO FUTURO DO TRABALHO

0800 048 1212 **sc.senai.br**

Rodovia Admar Gonzaga, 2765 - Itacorubi - 88034-001 - Florianópolis, SC