



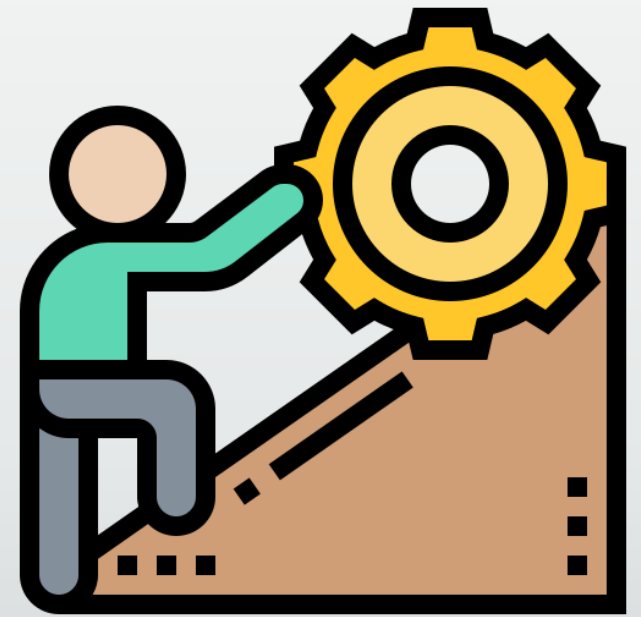
# Princípios de Projeto de Software

Programação III

Prof. Edson Mota, PhD, MSc, PMP

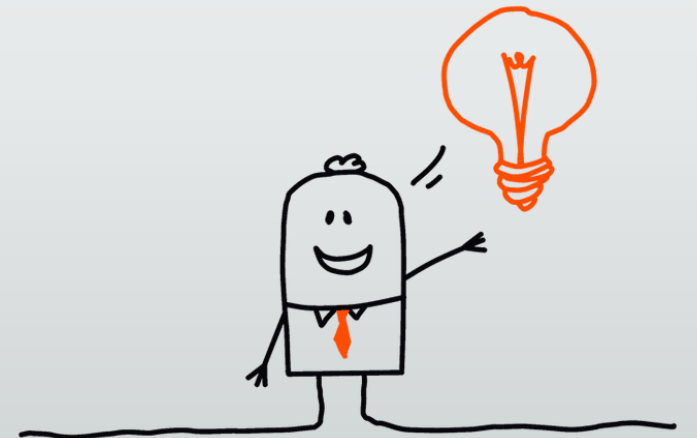
# Projetos de Software

- Projetos para sistemas confiáveis e extensíveis exigem atenção para diversos aspectos relacionados à sua construção.
- Desenvolver software não é uma tarefa trivial.
- Desenvolver softwares bem projetados, capazes de se manterem atualizados por grandes períodos de tempo é ainda mais desafiador.



# Projetos de Software

- Dois aspectos **essenciais** podem aumentar as chances de se produzir projetos de qualidade:
  1. Projetos criados sobre os fundamentos da **orientação a objetos**;
  2. **Não reinventar a roda**, tomar como base **casos de sucesso** em problemas similares (**Design Partner**).



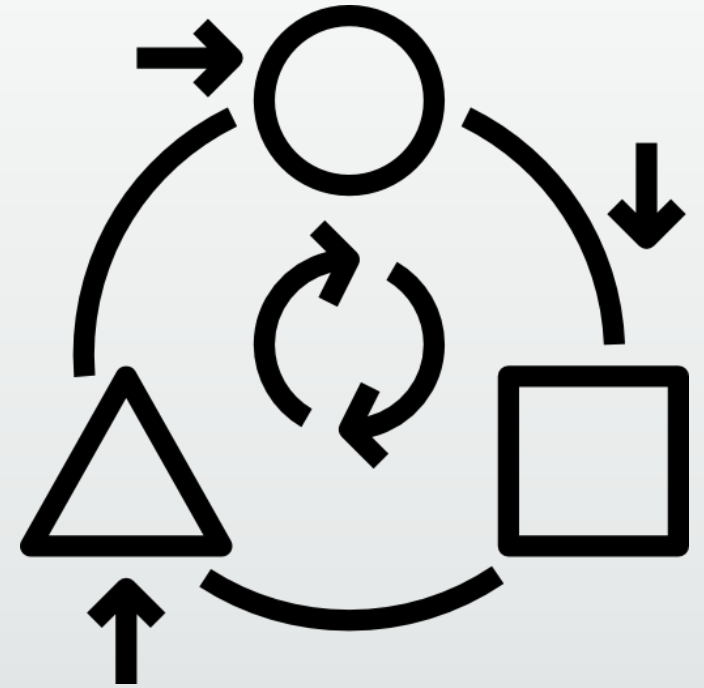
# Sintomas de Problemas

- Algumas características podem indicar a propensão de um software a apresentar problemas: (Martin, 2000)
  1. Rigidez / Adaptabilidade
  2. Fragilidade
  3. Imobilidade
  4. Viscosidade



# Rigidez / Adaptabilidade

- **Dificuldade** de se **realizar modificações** em um software;
- A dificuldade decorre do **efeito cascata** das mudanças em módulos relacionados;
- Isso pode tornar uma simples mudança em um tipo de variável em algo muito mais **complexo** e **caro**.



(Martin, 2000)

# Fragilidade

- Forte relação com o sintoma anterior;
- Tendência ao surgimento de erros em diferentes partes de um sistema em decorrência de alguma alteração feita;
- Essas falhas, muitas vezes, não são localizadas no ponto de modificação, mas são efeitos colaterais da mudança.



(Martin, 2000)

# Imobilidade

- **Incapacidade de reutilização** dos recursos do software (Martin, 2000)



“**Para** que uma **técnica** de **reuso** seja **efetiva**, tem que ser mais **fácil reusar** os artefatos do que desenvolvê-los da estaca zero.” (Krueger, 1992)



# Viscosidade

- **Dificuldade em se manter a concepção original do projeto, ou requisitos do ambiente de desenvolvimento, quando se executam modificações no sistema. (Martin, 2000)**

Trata-se de **seguir padrões arquiteturais** e de infraestrutura **pré-definidos**. A fuga destes **padrões** pode **gerar muita dor de cabeça**





# Qualidade x Desenvolvimento de Software

- Essa realidade motivou a criação de diferentes abordagens envolvendo **boas práticas** e **princípios** de projetos de software.
- Essas abordagens **têm norteado** os processos de **construção** e **manutenção** dos sistemas computadorizados.
  - Algumas mais conhecidas:

## **S.O.L.I.D**

D.R.Y      *Don't repeat yourself*

K.I.S.S      *Keep It Simple, Stupid*

Y.A.G.N.I      *You Aren't Gonna Need It*



Single  
responsibility



Open-Closed  
Principle



Liskov  
substitution



Interface  
segregation



Dependency  
inversion



# Mas então, o que é S.O.L.I.D

- Uma abordagem proposta **por Robert C. Martin** (também conhecido como "Uncle Bob").
- Início dos anos 2000
- Preconiza um conjunto de boas práticas no desenvolvimento de software.



Uncle Bob

# Princípio da Responsabilidade Única!



“Uma **classe** deve ter uma, e **apenas**  
**uma razão para mudar**” (Martin, 2000)



# Mudanças Vão Acontecer!

- Cada classe e cada método devem ter uma e **apenas uma** função.
- Redução da complexidade
- Redução do acoplamento
- Facilidade na leitura e entendimento



- Esse trecho de código atende ao princípio da **responsabilidade única**?

```
0 referências
public class Relatorio {

    0 referências
    public void Calcular()
    {
        // Geração do relatório
    }

    0 referências
    public void Formatar_e_Imprimir()
    {
        // Formata e imprime o relatório
    }

    0 referências
    public void Salvar()
    {
        // Salva o relatório
    }
}
```



# ▪ E agora?



```
0 referências
public class Relatorio {

    0 referências
    public void Calcular()
    {
        // Geração do relatório
    }

    0 referências
    public void Formatar()
    {
        // Formata o relatório
    }

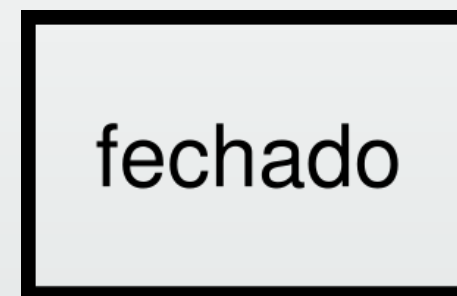
    0 referências
    public void Imprimir()
    {
        // Imprime o relatório
    }

    0 referências
    public void Salvar()
    {
        // Salva o relatório
    }
}
```

# Princípio do Aberto / Fechado



“Entidades de software (classes, módulos, funções) devem ser **abertas** para extensões e **fechadas** para modificações.”





# O Impacto das Mudanças

- Impacto direto na **evolução consistente** de um **software**.



Em última análise, trata-se de **perder o medo** de criar **novos bugs**, a partir de uma **mudança realizada**.

# Vamos analisar esse trecho de código?

```
public abstract class Compra
{
    0 referências
    public void AdicionarAoCarrinho()
    {
        // Código para adicionar um item ao carrinho
    }

    // A implementação desse método será feita
    // pelo usuário que quiser consumir essa classe!
    2 referências
    public abstract void CalculaDesconto();
}
```

[...]abertas para extensões e  
fechadas para modificações[...]

```
0 referências
public class CompraAVista : Compra
{
    1 referência
    public override void CalculaDesconto()
    {
        // Código de desconto a ser aplicado
        // para compras a vista
    }
}

0 referências
public class CompraAPrazo : Compra
{
    1 referência
    public override void CalculaDesconto()
    {
        // Código de desconto a ser aplicado
        // para compras a prazo
    }
}
```

# Princípio da Substituição de Liskov



“Deve ser possível substituir a sua classe base pela sua classe derivada.”

# Princípio da Substituição de Liskov

- Proposto por Barbara Liskov, esse princípio está associado ao conceito de Tipo Abstrato de Dados – Abstract Data Type (ADT).
- Baseia-se na noção de subtipo (ou subclasse):
  - Definição: *Dado um programa  $P$  que faz uso de um objeto  $O1$ ;  $O2$  será subtipo de  $O1$  se for possível substituir  $O1$  por  $O2$  no programa  $P$ , sem que  $P$  altere seu comportamento.* (Liskov, 1987).

**Vamos ver um exemplo prático**

Vamos supor que precisamos desenvolver um programa para calcular a área de um retângulo com a formula básica:

$$(\text{altura} * \text{largura})$$

3 referências

```
internal class Retangulo
```

```
{
```

6 referências

```
public virtual int Largura { get; set; }
```

6 referências

```
public virtual int Altura { get; set; }
```

---

1 referência

```
public virtual int Area()
```

```
{
```

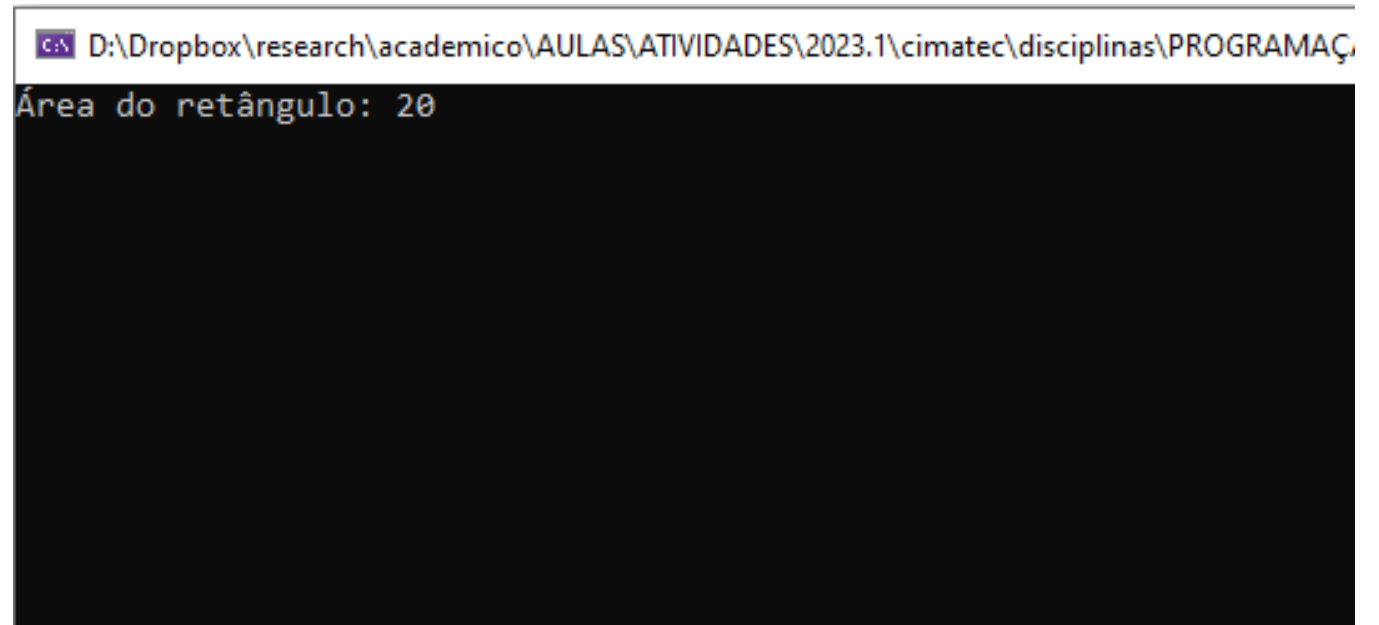
```
    return Largura * Altura;
```

```
}
```

```
}
```

```
Retangulo retangulo = new Retangulo();  
retangulo.Largura = 5;  
retangulo.Altura = 4;  
Console.WriteLine("Área do retângulo: " + retangulo.Area());
```

Agora, precisamos incluir  
uma nova forma, um  
quadrado



The screenshot shows a Windows command prompt window with a title bar indicating the file path: D:\Dropbox\research\academico\AULAS\ATIVIDADES\2023.1\cimatec\disciplinas\PROGRAMAÇÃO. The command prompt has a black background and white text. The first line of output is "Área do retângulo: 20".

```
D:\Dropbox\research\academico\AULAS\ATIVIDADES\2023.1\cimatec\disciplinas\PROGRAMAÇÃO  
Área do retângulo: 20
```



0 referências

```
internal class Quadrado : Retangulo
```

```
{
```

```
    // Assumindo que um quadrado tem lados iguais, poderíamos
```

```
    // apenas igualar a largura e a altura para calcular a área (será?)
```

6 referências

```
    public override int Largura
```

```
    {
```

```
        get { return base.Largura; }
```

```
        set { base.Largura = value; base.Altura = value; }
```

```
    }
```

6 referências

```
    public override int Altura
```

```
    {
```

```
        get { return base.Altura; }
```

```
        set { base.Altura = value; base.Largura = value; }
```

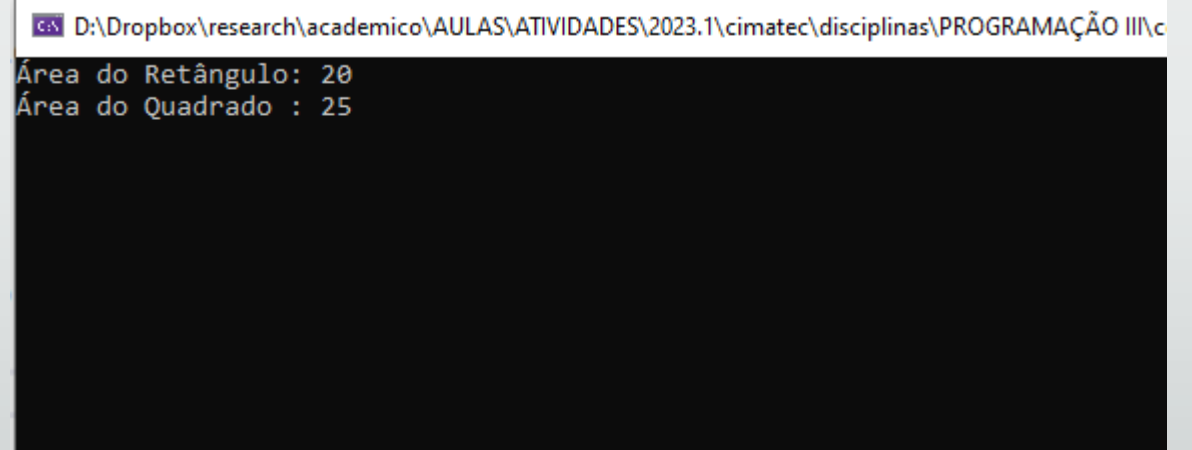
```
    }
```

```
}
```

```
Retangulo retangulo = new Retangulo();  
retangulo.Largura = 4;  
retangulo.Altura = 5;  
Console.WriteLine("Área do Retângulo: " + retangulo.Area());
```

```
Retangulo quadrado = new Quadrado();  
quadrado.Largura = 4;  
quadrado.Altura = 5;  
Console.WriteLine("Área do Quadrado : " + quadrado.Area());
```

**Apesar** de ter uma saída "**correta**", o resultado **não condiz** com os **parâmetros** apresentados!  
Largura=4, Altura=5



```
D:\Dropbox\research\academico\AULAS\ATIVIDADES\2023.1\cimaterc\disciplinas\PROGRAMAÇÃO III\c  
Área do Retângulo: 20  
Área do Quadrado : 25
```

O que precisamos fazer para **atender** ao  
**Princípio da Substituição de Liskov?**



4 referências

```
public interface IForma
```

```
{
```

```
  |
```

4 referências

```
  public int Largura { get; set; }
```

4 referências

```
  public int Altura { get; set; }
```

4 referências

```
  int Area();
```

```
}
```

1 referência

```
internal class Retangulo : IForma
```

```
{
```

3 referências

```
public int Largura { get; set; }
```

3 referências

```
public int Altura { get; set; }
```

3 referências

```
public int Area()
```

```
{
```

```
    return Largura * Altura;
```

```
}
```

```
}
```

2 referências

```
internal class Quadrado : IForma
```

```
{
```

3 referências

```
public int Lado { get; set; }
```

2 referências

```
public int Largura { get; set; }
```

2 referências

```
public int Altura { get; set; }
```

3 referências

```
public int Area()
```

```
{
```

```
    return Lado * Lado;
```

```
}
```

```
}
```

```
// Cria um objeto do tipo Retangulo
IForma retangulo = new Retangulo();
retangulo.Largura = 5;
retangulo.Altura = 4;
```

```
// Calcula a área do retângulo
Console.WriteLine("Área do retângulo: " + retangulo.Area());
```

```
// Cria um objeto do tipo Quadrado
IForma quadrado = new Quadrado();
((Quadrado)quadrado).Lado = 5;
```

```
// Calcula a área do quadrado
Console.WriteLine("Área do quadrado: " + quadrado.Area());
```

C:\D:\Dropbox\research\academico\AULAS\ATIVIDADES\2023.1\cimatic\disciplinas\PROGRAMAÇÃO III\c

```
Área do retângulo: 20
Área do quadrado: 25
```

# Princípio da Segregação de Interfaces



“Muitas **interfaces específicas** do cliente são **melhores** de que uma **interface de uso geral**”

(Martin, 2000)



# Princípio da Segregação de Interfaces

- Clientes não devem ser forçados a depender de interfaces que eles não vão utilizar.
- Este princípio facilita a implementação de interfaces.
- Contribuir para a construção de interfaces mais específicas.

**Vamos ver um exemplo prático**

```
namespace segregacao_interface_smell.interfaces
{
    2 referências
    internal interface IAnimal
    {
        2 referências
        void Voa();
        2 referências
        void Corre();
        2 referências
        void Late();
    }
}
```

```
namespace segregacao_interface_smell
{
    0 referências
    internal class Cachorro : IAnimal
    {
        1 referência
        public void Corre()
        {
            throw new NotImplementedException();
        }

        1 referência
        public void Late()
        {
            throw new NotImplementedException();
        }

        1 referência
        public void Voa()
        {
            throw new NotImplementedException();
        }
    }
}
```

```
namespace segregacao_interface_smell
{
    0 referências
    internal class Passaro : IAnimal
    {
        1 referência
        public void Corre()
        {
            throw new NotImplementedException();
        }

        1 referência
        public void Late()
        {
            throw new NotImplementedException();
        }

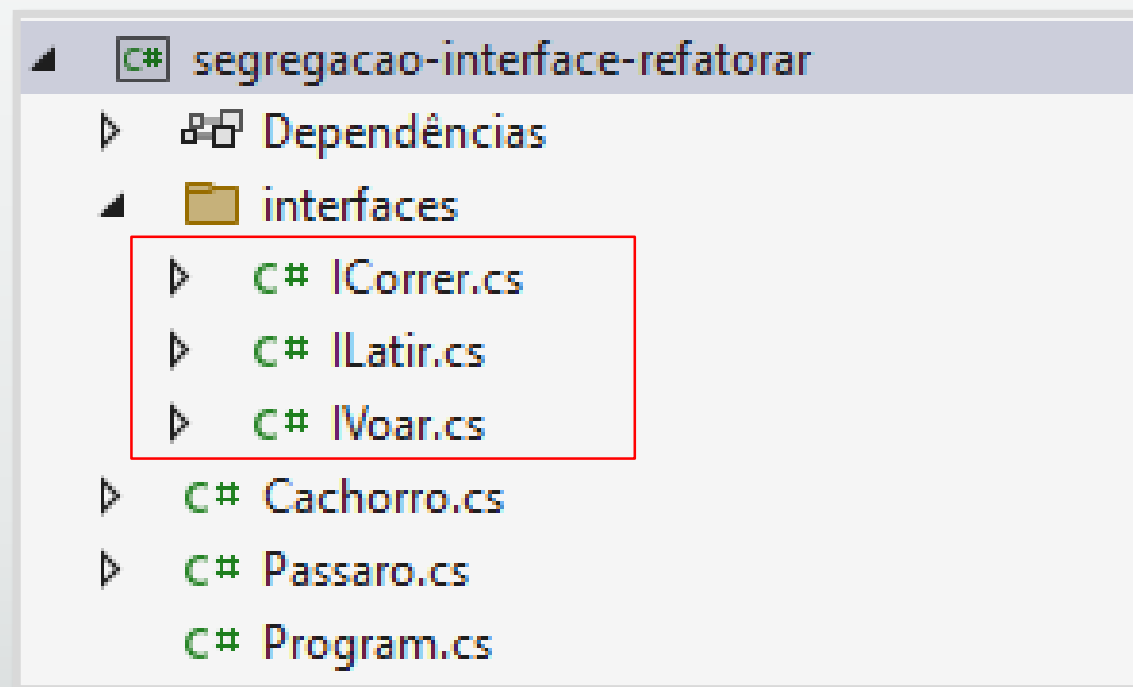
        1 referência
        public void Voa()
        {
            throw new NotImplementedException();
        }
    }
}
```

O que precisamos fazer para **atender** ao  
**Princípio da Segregação de Interfaces?**



# Princípio da Segregação de Interfaces

- Construindo interfaces mais específicas



```
namespace segregacao_interface_refatorar.interfaces
{
    1 referência
    internal interface ICorrer
    {
        1 referência
        void Corre();
    }
}
```

```
namespace segregacao_interface_refatorar.interfaces
{
    1 referência
    internal interface ILatir
    {
        1 referência
        void Late();
    }
}
```

```
namespace segregacao_interface_refatorar.interfaces
{
    1 referência
    internal interface IVoar
    {
        1 referência
        void Voa();
    }
}
```



```
namespace segregacao_interface_refatorar
{
    0 referências
    internal class Cachorro : ICorrer, ILatir
    {
        1 referência
        public void Corre()
        {
            throw new NotImplementedException();
        }

        1 referência
        public void Late()
        {
            throw new NotImplementedException();
        }
    }
}
```

```

namespace segregacao_interface_refatorar
{
    0 referências
    internal class Passaro : IVoar
    {
        1 referência
        public void Voa()
        {
            throw new NotImplementedException();
        }
    }
}

```

# Princípio da Inversão de Dependência



“Depender das **Abstrações**. Não depender das **Concretizações**.”

(Martin, 2000)

# Princípio da Inversão de Dependência

- **Duas regras fundamentais**

1. Módulos de alto nível **não devem depender** de módulos de baixo nível, ambos devem depender de abstrações.
2. Abstrações **não devem depender** de detalhes, detalhes devem depender de abstrações.

- Ou seja,

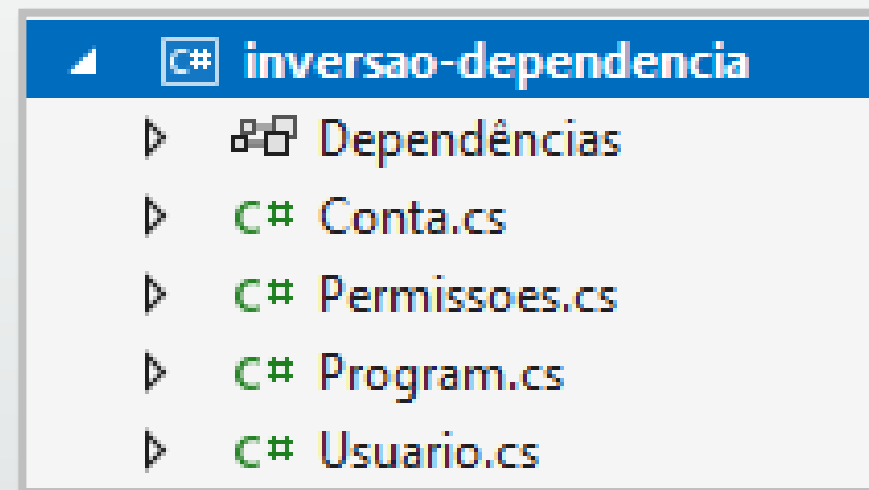
- Em vez de se criar **dependências diretas** entre os diferentes componentes de um **sistema**,
  - **Deve-se depender** de **abstrações** e **interfaces**, deixando a implementação específica (**detalhes**) para as **classes concretas** que a implementam.



**Vamos ver um exemplo prático**

# Vamos analisar um código?

- Nosso projeto atual é composto de 4 classes:
  - **Usuário:** Armazena a estrutura de dados do usuário
  - **Conta:** Permite a ativação de uma conta
  - **Permissões:** Permite a atribuição de permissões ao usuário
  - **Program:** Responsável por inicializar a aplicação



```
namespace inversao_dependencia
```

```
{
```

```
    4 referências
```

```
    public class Usuario
```

```
    {
```

```
        private string? nome;
```

```
        private string? login;
```

```
        private string? senha;
```

```
        3 referências
```

```
        public string? Nome { get => nome; set => nome = value; }
```

```
        2 referências
```

```
        public string? Login { get => login; set => login = value; }
```

```
        2 referências
```

```
        public string? Senha { get => senha; set => senha = value; }
```

```
    }
```

```
}
```

- Usuário é um módulo de baixo nível (não tem dependências)
- Mantém apenas as propriedades
- Encapsulamentos

- Conta é um módulo de Alto Nível (Possui dependência)
- Necessita da classe usuário para funcionar
- Acessa a classe usuário diretamente
- Viola o princípio da inversão de dependência

```
namespace inversao_dependencia
{
    2 referências
    public class Conta
    {
        1 referência
        public void AtivarConta(Usuario usuario, string login, string senha)
        {
            Console.WriteLine($"\\nUma nova conta foi criada para o usuário: {usuario.Nome}\\n");
        }
    }
}
```



- Permissões é um módulo de Alto Nível (Possui dependência)
- Necessita da classe usuário para funcionar
- Acessa a classe usuário diretamente
- Viola o princípio da inversão de dependência

```
namespace inversao_dependencia
{
    2 referências
    public class Permissoes
    {
        1 referência
        public void CriaPermissoes(Usuario usuario, string permissao)
        {
            Console.WriteLine($"Foi atribuída a permissão de {permissao} ao usuário {usuario.Nome}");
        }
    }
}
```

```
using inversao_dependencia;
```

0 referências

```
public class Program
```

```
{
```

0 referências

```
public static void Main(string[] args)
```

```
{
```

```
    Usuario usuario = new Usuario { Nome = "Jose", Login = "jse", Senha = "1234"};
```

```
    Conta conta = new Conta();
```

```
    conta.AtivarConta(usuario, usuario.Login, usuario.Senha);
```

```
    Permissoes permissoes = new Permissoes();
```

```
    permissoes.CriaPermissoes(usuario, "Administrador");
```

```
    Console.ReadLine();
```

```
}
```

```
}
```

- Acessa todas as classes diretamente
- Alto acoplamento
- Os problemas relacionados ao uso indiscriminado do 'new'

D:\Dropbox\research\academico\AULAS\ATIVIDADES\2023.1\cimatec\disciplinas\PROGRAMAÇÃO III\codigos\CodeSOLID\inversao-dependencia4\b

Uma nova conta foi criada para o usuário: Jose

Foi atribuida a permissão de Administrador ao usuário Jose

=> nome

=> log

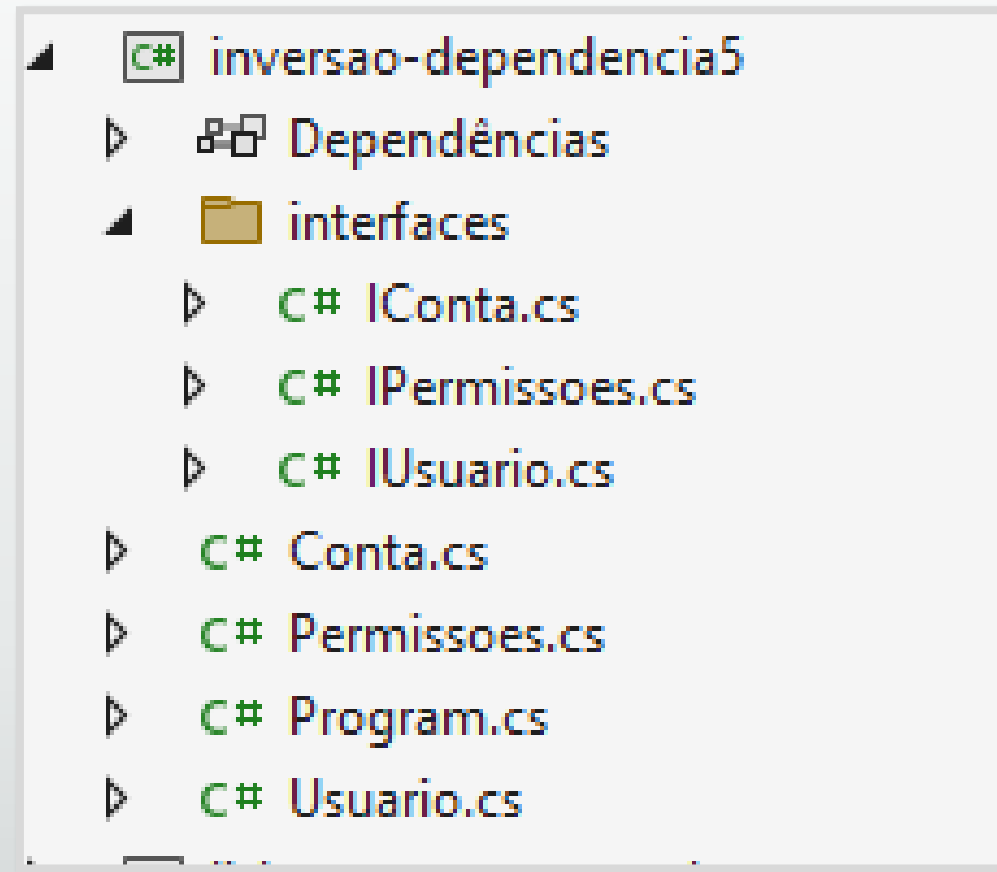
=> sen

O que precisamos fazer para **atender** ao  
**Princípio da Inversão de dependência?**



# Refatoração

- Para atender ao princípio da inversão de dependência precisamos fazer com que todos os módulos dependam de abstrações,
- Evitando acessos diretos às classes concretas.
- Para isso, podemos criar interfaces que vão intermediar a relação entre essas classes



```

namespace inversao_dependencia5.interfaces
{
    6 referências
    public interface IUsuario
    {
        2 referências
        string? Login { get; set; }
        4 referências
        string? Nome { get; set; }
        2 referências
        string? Senha { get; set; }
    }
}

```

- Interface para a classe usuário
- Evitar que qualquer outra classe acesse diretamente a classe concreta usuário

- Implementando a interface na classe concreta usuário

```
namespace inversao_dependencia5
```

```
{
```

```
    1 referência
```

```
    public class Usuario : IUsuario
```

```
    {
```

```
        private string? nome;
```

```
        private string? login;
```

```
        private string? senha;
```

```
        4 referências
```

```
        public string? Nome { get => nome; set => nome = value; }
```

```
        2 referências
```

```
        public string? Login { get => login; set => login = value; }
```

```
        2 referências
```

```
        public string? Senha { get => senha; set => senha = value; }
```

```
    }
```

```
}
```

- Criando uma interface para a classe conta
- A assinatura do método usando a interface usuário

```
namespace inversao_dependencia5.interfaces
{
    2 referências
    public interface IConta
    {
        2 referências
        void AtivarConta(IUsuario usuario, string login, string senha);
    }
}
```



- Implementação da classe conta
- Observe que ela não tem mais acesso a classe usuário diretamente
- Não sabe o que há na implementação da classe usuário

```
namespace inversao_dependencia5
{
    1 referência
    public class Conta : IConta
    {
        2 referências
        public void AtivarConta(IUsuario usuario, string login, string senha)
        {
            Console.WriteLine($"Uma nova conta foi criada para o usuário: {usuario.Nome}\n");
        }
    }
}
```

- Criando uma interface para a classe Permissões

```
namespace inversao_dependencia5.interfaces
{
    2 referências
    public interface IPermissoes
    {
        2 referências
        void CriaPermissoes(IUsuario usuario, string permissao);
    }
}
```

- Implementação da classe Permissões
- A Classe permissões não tem mais acesso a classe usuário diretamente

```
namespace inversao_dependencia5
{
    1 referência
    public class Permissoes : IPermissoes
    {
        2 referências
        public void CriaPermissoes(IUsuario usuario, string permissao)
        {
            Console.WriteLine($"Foi atribuida a permissão de {permissao} ao usuário {usuario.Nome}");
        }
    }
}
```

```
public class Program
```

```
{
```

```
    0 referências
```

```
    public static void Main(string[] args)
```

```
    {
```

```
        IUserario usuario = new Usuario { Nome = "Jose", Login = "jse", Senha = "1234" };
```

```
        IConta conta = new Conta();
```

```
        conta.AtivarConta(usuario, "jse", "1234");
```

```
        IPermissoes permissoes = new Permissoes();
```

```
        permissoes.CriaPermissoes(usuario, "Administrador");
```

```
        Console.ReadLine();
```

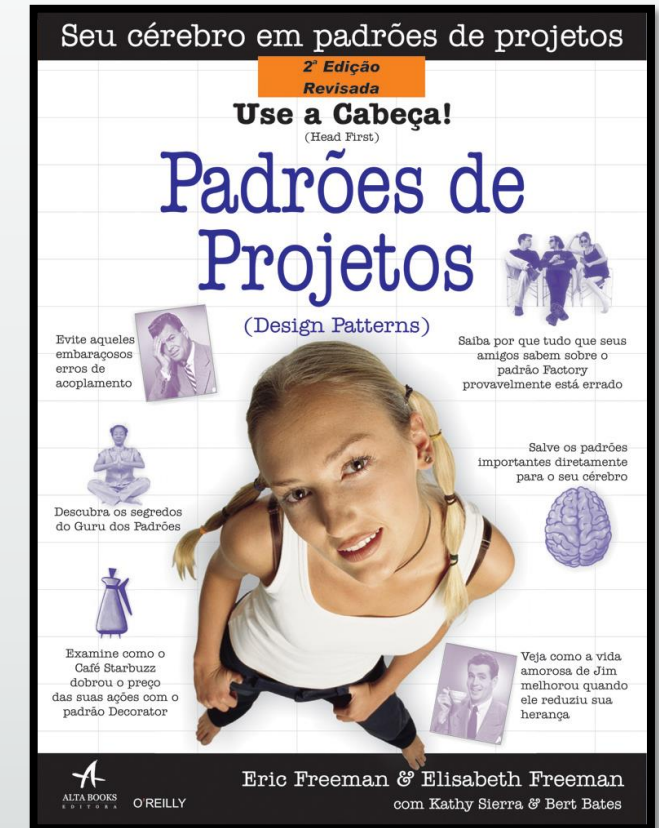
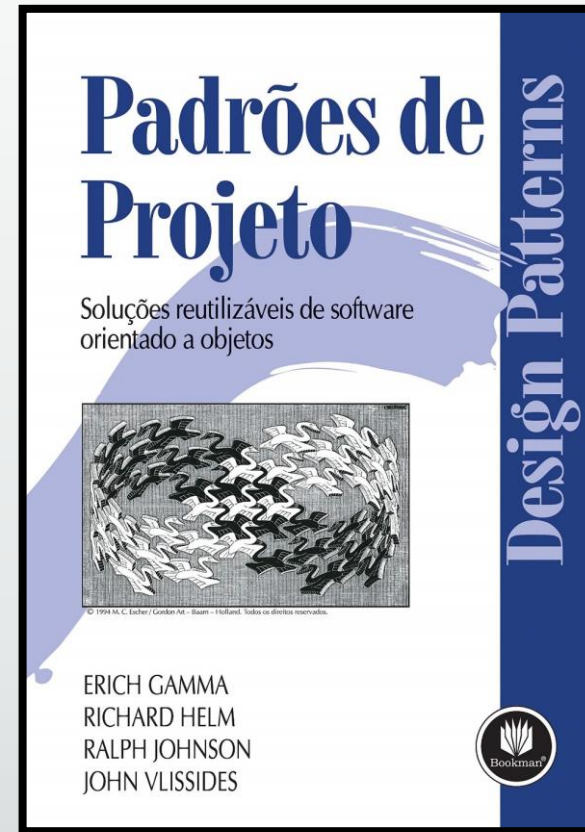
```
    }
```

```
}
```

- Na execução, a instancia é atribuída a uma interface
- Nessa classe, apesar do acoplamento ter sido reduzido, ainda existem problemas com muitas instâncias
- Alto acoplamento

# Outras Fontes...

- Utilização de padrões de projeto
- Soluções para problemas comuns estruturados em um conjunto de boas práticas de programação.
- Alguns padrões são:
  - ✓ Decorator
  - ✓ Factory
  - ✓ Singleton
  - ✓ Repository
  - ✓ Observer
  - ✓ Entre outros...



**Bons estudos!**