Disciplina: POO - Programação Orientada a Objetos

SOLID – Princípios

Profª. Drª. Giovana Angélica Ros Miola
giovana.miola@fatec.sp.gov.br

Acoplamento e Coesão entre Classes

- Coesão e acoplamento são conceitos interligados
- Classes coesas tendem a gerar baixo acoplamento
- No desenvolvimento de sistemas preze por alta coesão e baixo acoplamento
- Uma das maneiras de obter baixo acoplamento é utilizando interfaces

Acoplamento entre classes

- Baixo acoplamento: é o grau em que uma classe conhece a outra.

- Se o conhecimento da classe A sobre a classe B for através de sua interface, temos um baixo acoplamento. BOM

- Se a classe A depende de membros da classe B que não fazem parte da interface de B, então temos um alto acoplamento. RUIM.

Coesão entre classes

- Coesão é o grau em que uma classe tem um único e bem focado propósito. Quanto maior a coesão, melhor é o projeto do sistema

  - Quando temos uma classe elaborada com um único e bem focado propósito, dizemos que ela tem alta coesão. BOM.

  - Quando temos uma classe com propósitos que não pertencem apenas a ela, temos uma baixa coesão. RUIM.

SOLID - Histórico

- SOLID é um acrônimo dos cinco primeiros princípios da programação orientada a objetos e design de código

- Os princípios foram identificados por Robert C. Martin (chamado de tio Bob/uncle Bob) em 1990.

- The Clean Code Blog - by Robert C. Martin (Uncle Bob), https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

- Mais tarde o acrônimo SOLID (modo de programação consistente) foi proposto por Michael Feathers, após observar que os cinco princípios poderiam se encaixar nesta palavra.

Problemas comuns que o SOLID pode evitar

- Dificuldade na testabilidade / criação de testes de unidade;
- Código macarrônico, sem estrutura ou padrão;
- Dificuldades de isolar funcionalidades;
- Duplicação de código, uma alteração precisa ser feita em N pontos;
- Fragilidade, o código quebra facilmente em vários pontos após alguma mudança.

SOLID – Aplicação para benefícios da OO

- Os princípios SOLID devem ser aplicados para se obter os benefícios da Orientação a Objetos, tais como:

  - Códigos mais fáceis de se manter, de serem adaptados e de se ajustarem as mudanças
  - Códigos testáveis e de fácil entendimento
  - Códigos que fornecem o máximo de reaproveitamento
  - Aplicações com ciclo de vida maiores

SOLID

- Para garantir os benefícios da OO e obter uma boa conduta no desenvolvimento de sistemas, utiliza-se os princípios SOLID :

  - SRP – **S**ingle Responsability Principle (Princípio da Responsabilidades Única)
  - OCP – **O**pen/Closed Principle (Princípio Aberto/Fechado)
  - LSP – **L**iskov Substitution Principle (Princípio de Substituição de Liskov)
  - ISP – **I**nterface Segregation Principle (Princípio da Segregação de Interface)
  - DIP – **D**ependency Inversion Principle (Princípio da Inversão de Dependência)

Princípios

Princípio da responsabilidade única - uma classe deve ter apenas uma única responsabilidade (mudanças em apenas uma parte da especificação do software, devem ser capazes de afetar a especificação da classe).
Princípio de aberto/fechado - entidades de software/classes devem ser abertas para extensão, mas fechadas para modificação.
Princípio da substituição de Liskov - objetos em um programa devem ser substituíveis por instâncias de seus subtipos, sem alterar a funcionalidade do programa. Deve ser capaz de afetar apenas a especificação da classe.
Princípio da segregação de Interface - muitas interfaces de clientes específicas, são melhores do que uma para todos propósitos.
Princípio da inversão de dependência - deve-se depender de abstrações, não de objetos concretos.

Identificação de desenvolvimento

- Identifique seus códigos por meio destas perguntas:
  - Você possui teste na sua aplicação? Eles são essenciais no desenvolvimento
  - Suas classes estão muito grandes, de difícil leitura?
  - A sua implementação está muito complexa, com muitos níveis de decisão (ifs)?
  - Você consegue reutilizar o seu código de modo prático, dentro do próprio ou em novos projetos?
  - A sua implementação é frágil a mudanças?

Exemplo:
Carrinho de compras

- Serão desenvolvidas algumas características (atributos) e ações (métodos).
- Abstraindo do mundo real o carrinho de compras

- Testando a classe:

```
class Program{
    static void Main(string[] args){
       CarrinhoCompras c1 = new CarrinhoCompras();
       Console.WriteLine("Itens: " + c1.ExibirItens());
    }
}
```

```
class CarrinhoCompras{
    private string itensDescricao;
    private double itensValor;
    private string status;
    private double valorTotal;

    public CarrinhoCompras(){
        this.itensDescricao = "";
        this.itensValor = 0.0;
        this.status = "aberto";
        this.valorTotal = 0;
    }
    public string ExibirItens(){
        return this.itensDescricao;
    }
}
```

Substituir por:
private List<Itens>
vetItens;

- Os atributos itensDescrição e itensValor, não são atributos que deveriam pertencer a esta classe e sim a outra como, por exemplo, Itens

Inserção de novas funcionalidades Código sem SRP

```
class Program{
    static void Main(string[] args){
        CarrinhoCompras c1 = new CarrinhoCompras();
        c1.ExibirItens();
        Console.WriteLine("\nValor total: " +
                          c1.ExibirValotTotal());

        Itens it1 = new Itens("Impressora",1000);
        Itens it2 = new Itens("Monitor",2000);
        c1.AdicionarItem(it1);
        c1.AdicionarItem(it2);
        c1.ExibirItens();
        Console.WriteLine("\nValor total: " +
                          c1.ExibirValotTotal());

        if (c1.ConfirmarPedido())
           Console.WriteLine("\nPedido realizado
                              com sucesso.");
        else
           Console.WriteLine("\nErro, carrinho não
                              possui itens.");
        c1.ExibirStatus();
    }
}
```

```
class CarrinhoCompras{
    private List<Itens> vetItens = new List<Itens>();
    private string status;
    private double valorTotal;

    public CarrinhoCompras(){
        this.status = "aberto";
        this.valorTotal = 0;
    }
    public void ExibirItens(){
        foreach (Itens it in vetItens)
            it.MostraItens();
    }
    public bool AdicionarItem(Itens it){
        vetItens.Add(it);
        valorTotal += it.Valor;
        return true;
    }
    public double ExibirValotTotal(){
        return valorTotal;
    }
    public string ExibirStatus(){
        return status;
    }
    public bool ConfirmarPedido(){
    //apenas confirma quando tiver
    //pedidos cadastrados
        if (ValidarCarrinho()){
            status = "confirmado";
            EnviarEmailConfirmacao();
            return true;
        }
        return false;
    }
    public void EnviarEmailConfirmacao{
        Console.WriteLine("Envia e-mail de
                           confirmação");
    }
    public bool ValidarCarrinho()
    {//qtde de itens > 0 retorna true ou false
        return vetItens.Count > 0;
    }
}
```

```
class Itens{
    private string descricao;
    private double valor;

    public Itens(string descricao,
                 double valor){
        Descricao = descricao;
        Valor = valor;
    }
    public string Descricao{
        get { return descricao; }
        set { descricao = value; }
    }
    public double Valor{
        get { return valor; }
        set { valor = value; }
    }
    public void MostraItens(){
        Console.WriteLine("Descrição: “
        + Descricao + "\tValor: " +
        Valor );
    }
}
```

Carrinho de compras, está correto?

- A implementação deste projeto atendeu aos princípios básicos de Orientação a Objetos?
- Foram abstraídos atributos e métodos para a classe?
- Mas será que este projeto atende aos princípios SOLID?

- Será que existem questões que podem ser melhoradas, para garantir os benefícios da OO?
- Sim e para isso serão abordados os princípios SOLID.

Abstrair, observar (um ou mais elementos de um todo), avaliando características e propriedades em separado

1º Princípio SOLID - SRP – Single Responsability Principle - Princípio da Responsabilidades Única

- Uma classe deve ter apenas um motivo para mudar, ou seja, uma classe deve ter uma e apenas uma responsabilidade
- Analisem os métodos implementados pelo carrinho de compras:
  1. ExibirItens()
  2. AdicionarItem()
  3. ExibirValotTotal()
  4. ExibirStatus()
  5. ConfirmarPedido()
  6. EnviarEmailConfirmacao()
  7. ValidarCarrinho()

1º Princípio - Princípio da Responsabilidades Única

- O projeto passará por mudanças para readequar os papéis de cada classe, ou seja, o código será refatorado.
- As quatro responsabilidades encontradas, darão origem a quatro classes:

CarrinhoCompras

Itens

Pedido

EmailService

Código com SRP

```
class Pedido{
    private string status;
    private double valorPedido;
    private CarrinhoCompra
                        _carrinhoCompra;

    public Pedido() {
        Status = "aberto";
        _carrinhoCompra = new
                        CarrinhoCompra();
    }
    public bool Confirmar() {
        if(CarrinhoCompra.ValidarCarrinho())
        {
            Status = "Confirmado";
            return true;
        }
        return false;
    }
    public string Status {
        get { return status; }
        set { status = value; }
    }
    public _CarrinhoCompra {
        get { return _carrinhoCompra; }
        set { _carrinhoCompra= value; }
    }
    public double ValorPedido {
        get { return valorPedido; }
        set { valorPedido = value; }
    }
}
```

```
class EmailService {
    private string de =
"contato@site.com.br";
    private string para;
    private string assunto;
    private string conteudo;
    public EmailService(string para,
    string assunto,  string conteudo) {
        Para = para;
        Assunto = assunto;
        Conteudo = conteudo;
    }
    public static void DispararEmail(){
        // static para não ter que
        // instanciar objeto
        Console.WriteLine("Envia
                            e-mail.");
    }
    public string De {
        get { return de; }
        set { de = value; }
    }
    public string Para {
        get { return para; }
        set { para = value; }
    }
    public string Assunto {
        get { return assunto; }
        set { assunto = value; }
    }
    public string Conteudo {
        get { return conteudo; }
        set { conteudo = value; }
    }
}
```

```
class CarrinhoCompras{
        private List<Itens> vetItens =
                        new List<Itens>();
    public CarrinhoCompras(){
        this.vetItens = new List<Itens>();
    }
    public List<Itens> VetItens{
        get { return vetItens; }
    }
    public bool AdicionarItem(Itens it){
        vetItens.Add(it);
        return true;
    }
    public bool ValidarCarrinho()
    {//qtde de itens > 0 retorna
     //true ou false
        return vetItens.Count > 0;
    }
}
```

Código com SRP

```
class Program{
    static void Main(string[] args) {
        Pedido p = new Pedido();

        Itens it1 = new Itens("Impressora", 1000);
        Itens it2 = new Itens("Monitor", 2000);

        //p obtem a instância de carrinho
        //para depois adicionar
        p.CarrinhoCompra.AdicionarItem(it1);
        p.CarrinhoCompra.AdicionarItem(it2);

        Console.WriteLine("\nItens do carrinho");
        var pp = p.CarrinhoCompra.VetItens;
        double total = 0;
        foreach (Itens it in pp){
            it.MostrarItens();
            total += it.Valor;
        }
        Console.WriteLine("\nValor total: " + total);
        Console.WriteLine("\nCarrinho está válido? "+
                    p.CarrinhoCompra.ValidarCarrinho());
        Console.WriteLine("\nStatus do pedido: " + p.Status);
        Console.WriteLine("\nConfirmar pedido: " +
                                p.Confirmar());
        Console.WriteLine("\nE-mail");
        if (p.Status == "confirmado")
            EmailService.DispararEmail();
        Console.ReadKey();
    }
}
```

```
class Itens{
    private string descricao;
    private double valor;

    public Itens(string descricao,
                 double valor){
        Descricao = descricao;
        Valor = valor;
    }
    public string Descricao{
        get { return descricao; }
        set { descricao = value; }
    }
    public double Valor{
        get { return valor; }
        set { valor = value; }
    }
    public void MostrarItens(){
        Console.WriteLine("Descrição: “
        + Descricao + "\tValor: " +
        Valor );
    }
   public bool ItemValido() {
       if (Descricao == "")
           return false;
       if (Valor == 0)
           return false;
       return true;
   }
}
```

Avaliando as vantagens do SRP

- Novas validações são necessárias:
  - Não permitir itens com valores zerados ou negativos. Se o valor do item for zerado ou negativo o sistema deve retornar false, caso contrário true.
  - Não permitir itens com descrição vazias. Se a descrição do item estiver vazia o sistema deve retornar false, caso contrário true.

Projeto SEM Solid – class CarrinhoCompras

```
public bool ItemValido(Itens it){
    if (it.Descricao == "")
        return false;
    if (it.Valor == 0)
        return false;
    return true;
}
public bool AdicionarItem(Itens it) {
    if (ItemValido(it)) {
        vetItens.Add(it);
        valorTotal += it.Valor;
        return true;
    }
    return false;
}
```

Projeto COM Solid – class Itens

```
public bool ItemValido(){
    if (Descricao == "")
        return false;
    if (Valor == 0)
        return false;
    return true;
}
```

Avaliando as vantagens do SRP

- Veja que a complexidade vai aumentando e se não definir as responsabilidades corretas de cada classe, ela possuirá muitas linhas de implementação, as quais não são dela e sim de outra classe, como já apresentado.
- Nota-se que, se não aplicados os princípios, como neste caso o SRP, as classes tornam-se confusas.
- O SRP está ligado a um dos pilares da Orientação a Objetos a abstração, pois deve-se abstrair o máximo possível, os objetos, com suas respectivas responsabilidades e para fazer isso é necessário verificar as ações desses objetos, para identificar de qual classe é a responsabilidade.
- Se for notado que algo, de um objeto já instanciado, possa ser abstraído para outro, ainda mais especializado, **refatore** (readeque) sua implementação e deixe seu código, ainda mais organizado, reutilizável, claro, ou seja, mais fácil de dar manutenção, caso no futuro, seja necessário, sofrer algumas alterações, modificações ou melhorias.

2º Princípio SOLID - OCP – Open Close Principle – Princípio Aberto e Fechado

- Entidades de software, tais como classes, módulos, funções, entre outros, devem sempre estar abertas para extensões, mas fechadas para modificações

- Quando novos comportamentos e recursos precisam ser adicionados no software, deve-se estender e não alterar o código fonte original.

2º Princípio – Princípio Aberto e Fechado

- O que é uma alteração? (Fechado)
  - Precisamos acessar uma classe já existente para inserir ou modificar comportamentos (métodos)
- O que é uma expansão? (Aberto)
  - Requer uma abstração de código mais sofisticada no momento em que esta implementando as classes, ou seja, é necessário pensar em como as classes serão capazes de serem estendidas, de modo que quando um novo comportamento for necessário, que a classe já existente, seja estendida, ao invés de ser modificada
- Veja que no princípio aberto fechado, está sugerindo que você deve pensar em extensibilidade, antes de implementar os códigos
- O OCP, é considerado o princípio mais polêmico e mais complexo e menos adotado, justamente por ser complexo de ser entendido

```
class Funcionario{
    private double salario;
    public double Salario{
        get { return salario; }
        set { salario = value; }
    }
}
class Estagiario{
    private double bolsaEstagio;
    public double BolsaEstagio{
        get { return bolsaEstagio; }
        set {bolsaEstagio = value; }
    }
}
class Program{ //Main()
    Funcionario f = new Funcionario();
    Estagiario e = new Estagiario();
    FolhaPagamento fPag = new FolhaPagamento();
    fPag.CalcularFolha(f, e);
```

```
class FolhaPagamento {
    private double saldoFolha;
    public double SaldoFolha() {
        get {return saldoFolha; }
        set { saldoFolha = value; }
    } // possibilidade de solução
    public double CalcularSalarioFuncionario(
                                            Funcionario f){
        return f.Salario * 7 / 100;
    }
    public double CalcularSalarioEstagiario (Estagiario e){
        return e.Salario + 50;
    } // 50, exemplo de valor de seguro
    public double CalcularFolha(Funcionario f,
                                Estagiario e) {
        saldoFolha += CalcularSalarioFuncionario
                                            (Funcionario f)
        saldoFolha += CalcularSalarioFuncionario
                                            (Estagiario e)
        return saldoFolha;
    }
}
```

Tem problema nesta implementação, não é funcional e a classe Folha de Pagamento não é coesa.

```
class Funcionario{
    private double salario;
    public double Salario{
        get { return salario; }
        set { salario = value; }
    }
    public virtual double CalcularSalario() {
        return Salario * 7 / 100;
    }
}
class Estagiario: Funcionario{

    public override double CalcularSalario()
    {
        return Salario + 50;
    }
}
```

```
class FolhaPagamento {
    private double saldoFolha;
    public double SaldoFolha {
        get { return saldoFolha; }
        set { saldoFolha = value; }
    }
    public double CalcularSalarios(Funcionario f)
    {
       if //verifica se é Funcionário
            SaldoFolha = f.CalcularSalario();
        if (f.GetType().IsInstanceOfType(Estagio))
            SaldoFolha = f.CalcularSalario();
        return SaldoFolha;
    }
}
```

Não executa, pois teriam mais tratamentos, a serem feitos como regra de negócio

2º Princípio - Princípio Aberto e Fechado

- A classe “FolhaDePagamento” precisa verificar o funcionário para aplicar a regra de negócio correta na hora do pagamento.
- No futuro a empresa resolveu trabalhar com funcionários PJ, neste caso, seria necessário modificar essa classe e consequentemente o princípio Open-Closed do SOLID seria quebrado.
- A modificação mais comum, seria adicionar um **IF** e verificar o novo tipo de funcionário PJ, aplicando as regras para essa nova funcionalidade, mas é exatamente este o problema, ao alterar uma classe já existente para adicionar um novo comportamento, corre-se um sério risco de introduzir problemas, em algo que já estava funcionando.

2º Princípio – Princípio Aberto e Fechado

- Possível solução para este problema:
  - A classe “FolhaDePagamento” não precisa mais saber quais métodos chamar para calcular.
  - Ela será capaz de calcular o pagamento corretamente de qualquer novo tipo de funcionário que seja criado no futuro, desde que ele implemente a interface “IRemuneravel”, sem qualquer necessidade de alteração do seu código fonte.
  - Este princípio é base para um dos **padrão de projeto/design patterns** mais conhecidos, o **Strategy**.

```
interface IRemuneravel {
    double CalcularSalario();
}
class Funcionario {
    //atributos e métodos necessários
}
class ContratoClt : Funcionario,
                    IRemuneravel {
    public double CalcularSalario() {
        return Salario * 7 / 100;
    }
}
class Estagiario : Funcionario,
                   IRemuneravel {
    public double CalcularSalario() {
        return 800;
    }
}
```

```
class FolhaPagamento {
    private double saldoFolha;
    public double CalcularFolha(IRemuneravel f){
        SaldoFolha +=  f. CalcularSalario();
        return SaldoFolha;
    }
    public double SaldoFolha {
        get { return saldoFolha; }
        set { saldoFolha = value; }
    }
}
class Program {
    static void Main(string[] args) {
        Estagiario e = new Estagiario();
        ContratoClt c = new ContratoClt();
        FolhaPagamento f = new FolhaPagamento();
        Console.WriteLine("Saldo da folha R$ " +
                                  f.CalcularFolha(e));
        Console.WriteLine("Saldo da folha R$ " +
                                  f.CalcularFolha(c));
    }
}
```

```
Saldo da folha R$ 800
Saldo da folha R$ 2000
```

3º Princípio SOLID - LSP – Liskov Substitution Principle – Princípio da Substituição de Liskov

- Barbara Liskov apresentou em 1987, um artigo científico, sobre este princípio que ficou mais conhecido em 1994, com uma publicação junto com Jannette Wing.

- Este princípio informa que “Uma classe derivada deve ser substituível por sua classe base”, ou de maneira mais simples, se um objeto B é um subtipo de um outro objeto A, este objeto A pode substituir B em qualquer lugar no código, sem que este código pare de funcionar

3º Princípio – Princípio da Substituição de Liskov

```
class ClasseA {
    public string BuscarNome() {
        return "Nome A";
    }
}
class ClasseB : ClasseA {
    public string BuscarNome() {
        return "Nome B";
    }
}
```

```
class Program {
    static void Main(string[] args) {
        ClasseA a = new ClasseA();
        ClasseB b = new ClasseB();
        ImprimirNome(a);
        ImprimirNome(b);
    }
    public static void ImprimirNome (ClasseA obj) {
        Console.WriteLine (obj.BuscarNome());
    }
}
```

```
Resultado:    Nome A
              Nome A
```

3º Princípio – Princípio da Substituição de Liskov

- Como demonstrado, é passado como parâmetro tanto a classe base como a classe derivada e (entretanto) o código continua apresentando a frase da classe base.
- Alguns exemplos de violação do princípio de Liskov:
  - Sobrescrever, implementar um método que não faz nada
  - lançar uma exceção inesperada ou
  - retornar valores de tipos diferentes da classe base.
- Para não violar o LSP, além de estruturar muito bem as suas abstrações, em alguns casos, você precisará usar a injeção de dependência e também usar outros princípios do SOLID, como por exemplo, o Princípio Aberto Fechado e o Princípio da Segregação da Interface, que será abordado mais à frente.

3º Princípio – Princípio da Substituição de Liskov

- Injeção de dependência - *Dependency Injection* é um padrão de desenvolvimento de programas de computadores utilizado quando é necessário manter baixo o nível de acoplamento entre diferentes módulos de um sistema. A Injeção de dependência se relaciona com o padrão Inversão de controle, mas não pode ser considerada um sinônimo deste.
- Seguir o LSP permite usar o polimorfismo com mais confiança.
- Pode-se chamar as classes derivadas referindo-se à sua classe base sem preocupações com resultados inesperados.

3º Princípio – Princípio da Substituição de Liskov

- A utilização do recurso de herança não deve ser usado somente para reaproveitar código, e sim quando realmente faz sentido herdar as características e comportamentos da classe base.

- Se implementada as classes utilizando o OCP-Princípio Aberto-Fechado, se torna mais fácil aplicar o LSP.

- Uma classe passa a herdar o comportamento de outra geralmente quando a resposta de perguntas como “objeto A é um objeto B?” ou “Conta Poupança é uma Conta

3º Princípio – Princípio da Substituição de Liskov

- É necessária uma atenção maior com a sentença **“É um”**. Ainda que um “objeto A” tenha as mesmas características de “objeto B”, nem sempre uma herança entre esses objetos estará coerente.

- Por exemplo, Conta Poupança é uma Conta?

- No primeiro momento sim, então pode-se herdar o comportamento de Conta em Conta Poupança que será atendido o princípio LSP?

3º Princípio – Princípio da Substituição de Liskov

```
class Conta {
    private double saldo;
    public double Saldo {
        get { return saldo; }
        set { saldo = value; }
    }
    public virtual void Sacar(double valor) {
        Saldo -= valor;
    }
}
class ContaPoupanca : Conta {
    public override void Sacar(double valor) {
        if (Saldo >= valor)
            Saldo -= valor;
    }
}
```

Usando o LSP essa herança está correta?

```
class Program {
    static void Main(string[] args) {
        Conta c = new Conta();
        c.Saldo = 100;
        c.Sacar(250);
        Console.WriteLine("A conta tem o saldo de R$ " + c.Saldo);
        //A conta tem o saldo de R$ -150
        ContaPoupanca p = new ContaPoupanca();
        p.Saldo = 100;
        p.Sacar(250);
        Console.WriteLine("A conta tem o saldo de R$ " + p.Saldo);
        //A conta tem o saldo de R& 100
        Conta a = new ContaPoupanca();
        a.Saldo = 100;
        a.Sacar(250);
        Console.WriteLine("A conta tem o saldo de R$ " + a.Saldo);
        //A conta tem o saldo de R& 100
        Console.ReadKey();
    }
}
```

O LSP diz que se trocar a classe pai pela filha, nada deverá ser modificado será?

3º Princípio – Princípio da Substituição de Liskov

- Algo está errado, o resultado alterou com a classe derivada, apresentando A conta tem o saldo de R$ 100 reais
- Apesar das classes possuírem características (métodos e propriedades) idênticas, o **comportamento externo** de cada uma é diferente, pois na conta corrente, por exemplo, pode ter valores negativos, já na conta poupança, só é possível sacar valores menor ou igual ao saldo, logo fazendo o código quebrar
- Aplicar o LSP é uma questão de análise do que realmente codificar, para este princípio ser atendido, em alguns casos aplicar princípio do ISP, resolveria o problema, mas na maioria deles, a melhor forma é não usar heranças, onde o LSP é violado, pois certamente o código estará sujeito a futuros problemas

4º Princípio SOLID - ISP – Interface Segregation Principle – Princípio da Segregação da Interface

- Este princípio informa que “uma classe não deve ser forçada a implementar interfaces e métodos que não irá utilizar”.
- Indica que é melhor criar interfaces mais específicas ao invés de ter uma única interface genérica.
- O uso das interfaces torna o código **menos acoplado**, extensível e assim gerando menos problemas em manutenções ou adições de novas funcionalidades futuras.
- Isto trata da coesão em interfaces, da implementação de módulos mais enxutos, com poucos comportamentos

4º Princípio – Princípio da Segregação da Interface

- Frase usadas no ISP:
  - Muitas interfaces específicas são melhores do que uma interface única
  - Clientes/classes não devem ser forçados a depender de métodos que não usam

- O ISP ajuda quando uma determinada interface tem muitos comportamentos ou, genérica demais com muitas responsabilidades e aplicando esse padrão, pode-se subdividir essa interface em partes menores com responsabilidades mais específicas

4º Princípio – Princípio da Segregação da Interface

```
// SEM SOLID
interface ICadastro {
        void ValidarDados();
        void SalvarBanco();
        void EnviarEmail();
}
class CadastroProduto : ICadastro {
    public void ValidarDados() {
        // Validar dados
    }
    public void SalvarBanco() {
        // Inserir na tabela Produto
    }
    public void EnviarEmail() {
        // Produto não tem e-mail
        // Mas porque ser obrigado a
        // usar este método?
    }
}
```

```
class CadastroCliente : ICadastro {
    public void ValidarDados() {
        // Validar CPF, Email
    }
    public void SalvarBanco() {
        // Inserir na tabela Cliente
    }
    public void EnviarEmail() {
        // Enviar e-mail para o cliente
    }
}
class Program {
    static void Main(string[] args) {
        //realizar instâncias e testar
    }
}
```

4º Princípio – Princípio da Segregação da Interface

```
// COM SOLID
interface ICadastro {
    void SalvarBanco();
}

interface ICadastroCliente : ICadastro {
    void ValidarDados();
    void EnviarEmail();
}

interface ICadastroProduto : ICadastro {
    void ValidarDados();
}

class CadastroProduto : ICadastroProduto {
    public void ValidarDados() {
        // Validar dados
    }
    public void SalvarBanco() {
        // Inserir na tabela Produto
    }
}
```

```
class CadastroCliente : ICadastroCliente {
    public void SalvarBanco() {
        // Inserir na tabela Cliente
    }
    public void ValidarDados() {
        // Validar CPF, Email
    }
    public void EnviarEmail() {
        // Enviar e-mail para o cliente
    }
}

class Program {
    static void Main(string[] args) {
        //realizar instâncias e testar
    }
}
```

5º Princípio SOLID - DIP – Dependency Inversion Principle – Princípio da Inversão da Dependência

- Será apresentado algo comum que demonstra a dependência de uma classe para outra, por exemplo, de **Classe1** para a **Classe2**:

```
class Classe1 {
    private Classe2 obj = new Classe2();
}
```

- A Classe1 necessita usar a **Classe2**, mostrando que ocorre uma **dependência** de **implementação**
- Por exemplo, utilizando o padrão de projeto DAO (*Data Access Object*/Objeto de acesso a dados), abstrai e encapsula os mecanismos de acesso a dados, escondendo os detalhes da execução da origem dos dados

5º Princípio – Princípio da Inversão da Dependência

- A classe ClienteDAO encapsula os métodos de acesso a base de dados e realiza operações de CRUD (*Create, Read, Update* e *Delete*)

```
public class ClienteDAO {
    public void Cadastrar (Cliente cliente) {
       // acessa a base e cadastra
       // um cliente
    }
}
```

- A classe ClienteBO representa uma classe de regra de negócio, que manipula o objeto e ao término de sua ação, chama o método de gravação no banco de dados

```
public class ClienteBO {
    private ClienteDAO obj;

    public void CadastrarCliente (Cliente cliente){
        obj = new ClienteDAO();
        obj.Cadastrar(cliente);
    }
}
```

5º Princípio – Princípio da Inversão da Dependência

- Encapsular o acesso a dados é o correto a se fazer, no exemplo anterior, tem a vantagem do reuso, assim, evitando duplicação de código, mas tem o problema da dependência da implementação da classe ClienteDAO
- Será que não seria interessante a classe ClienteBO (BO - *Business Objects* – Objetos de Negócio) saber que necessita do comportamento de ClienteDAO, mas sem precisar saber quem é ClienteDAO?
- O conceito de interface/abstração, novamente será utilizado para extrair o comportamento de ClienteDAO

5º Princípio – Princípio da Inversão da Dependência

```
interface IClienteDAO {
    void Cadastrar(Cliente cliente);
}

public class Cliente {
    //atributos e métodos necessários
}

public class ClienteDAO : IClienteDAO {
    public void Cadastrar(Cliente cliente) {
        //acessa a base e
        //cadastra um cliente
    }
}
```

```
public class ClienteBO {
     private IClienteDAO clienteDAO;

     public ClienteBO(IClienteDAO clienteDAO)
     {
          this.clienteDAO = clienteDAO;
     }

     public void CadastrarCliente(Cliente cliente)
     {

         clienteDAO.Cadastrar(cliente);
     }
}
```