Princípios SOLID

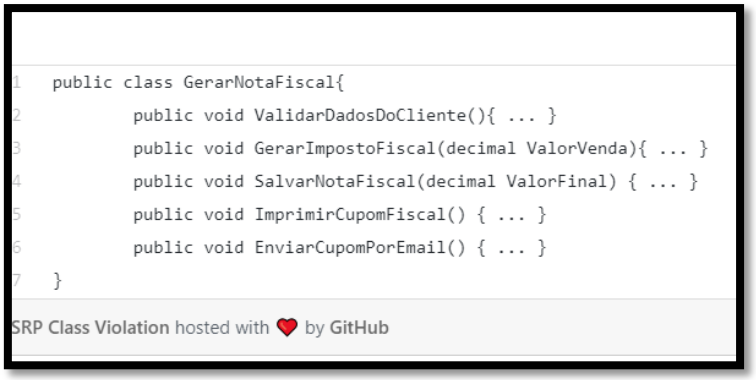
O que é SOLID?

SOLID é um conjunto de princípios e boas práticas de programação orientada a objetos.

Lista de princípios:

1- Princípio da Responsabilidade única

Uma classe deve ter apenas um motivo para mudar. Cada classe deve realizar uma responsabilidade.



A classe GerarNotaFiscal tem uma série de responsabilidades estas que não são inerentes a ela:

- Validar dados do Cliente;
- Gerar Imposto;
- Salvar Cupom Fiscal;
- Imprimir Cupom Fiscal;
- Enviar Cupom por e-mail;

A classe apresentada possui 5 responsabilidades, portanto ela precisa ser modificada.

Suponhamos que o método ***ValidarDadosDoCliente()*** necessitasse ser alterado e após a alteração, um bug tenha passado despercebido. Toda a estrutura da classe foi comprometida, ou seja, o sistema deixaria de emitir notas fiscais por conta de um bug em um método. Esse tipo de acoplamento faz com que os testes sejam mais complexos de serem realizados e menos eficazes.

Quando se trabalha com o tipo de arquitetura acima, tem-se a falsa impressão de que o sistema está sendo construído de forma prática e simples, pois se tem poucas classes para se dar manutenção, mas, quando elas começam a se expandir, é que vemos como o sistema ficou muito mais complexo e de difícil manutenção.

Quando trabalhamos com esse princípio tem-se o sentimento de estarmos criando muitas classes com pouco código e meio inúteis, mas quando as funcionalidades do sistema crescem, podemos perceber que a expansão nada mais é do que criar novas classes nos lugares corretos e conecta-las de forma coesa.

Para fazer essa classe ser coesa e seguir a premissa do SRP é simples, separar as responsabilidades da classe tornando-as únicas. Desse modo, teremos várias classes com apenas uma responsabilidade cada.

```
1  public class DadosDoCliente{
2          Validar();
3  }
4
5  public class ImpostosCupomFiscal{
6          GerarImpostos(decimal ValorVenda);
7  }
8
9  public class PersistenciaCupomFiscal{
10         Salvar (decimal ValorFinal);
11 }
12
13 public class EmissaoCupomFiscal{
14         Imprimir();
15 }
16
17 public class ComunicacaoCupomFiscal{
18         EnviarPorEmail();
19 }
```

SRP Class Solution hosted with ♥ by GitHub view raw

Vantagens da aplicação do SRP na classe:

- Facilidade de manutenção e evolução do código;
- Código limpo e fácil de entender;
- Facilidade de testar;
- Redução do acoplamento;
- Complexidade reduzida e coesão das classes

2- Princípio Aberto/fechado

Você deve ser capaz de estender um comportamento de uma classe sem a necessidade de modificá-lo.

O principio tem como objetivo trabalhar diretamente com a manutenção das classes. Ou seja, entidades como classes, módulos e funções devem estar abertas para extensão, porém fechadas para modificação.

Em outras palavras significa que esta classe pode ter seu comportamento alterado com facilidade quando necessário, sem a alteração do seu código fonte. Essa extensão pode ser feita através de herança, interface e composição.

```
public enum TipoEmail {
        Texto,
        Html,
        Criptografado
}

public void class EnviarEmail(string mensagem, string assunto, TipoEmail tipo){
        if(tipo == TipoEmail.Texto)
        {
                mensagem = this.RemoverFormatacao(mensagem);
        }
        else if(tipo == TipoEmail.Html)
        {
                mensagem = this.InserirHtml(mensagem);
        }
        else if(tipo == TipoEmail.Criptografado)
        {
                mensagem = this.CriptografarMensagem(mensagem);
        }

        this.EnviarMensagem();
}
```

OCP Class Violation hosted with ♥ by GitHub view raw

Acima temos o exemplo de uma classe que valida o tipo de e-mail para tratar a mensagem de acordo com o seu tipo, mas, quando um novo tipo de mensagem for criado, a classe deverá ser editada e um novo if deverá ser acrescentado.

Mas aí você, estudante espertinho deve estar pensando: “so coloca mais uns if q ta suave hehe so bom demais seloko”. Mas, desse modo, violaríamos o princípio do OCP.

O conceito de aberto/fechado (OCP) tem a premissa de criarmos novas classes para funcionalidades de tipos semelhantes, e caso tenhamos novas funcionalidades, novas classes sejam criadas.

Vantagens do OCP:

Extensibilidade: Ao termos uma nova funcionalidade ou comportamento, não precisaremos alterar a classe já existente, e sim estendê-la. Com isso mantemos o código original confiável e intacto, e criamos um código com design duradouro, de qualidade e manutenibilidade altas.

Abstração: Se as abstrações forem bem feitas, conseguimos de forma fácil estender os métodos da nossa aplicação (se não forem, aí o azar é seu).
O conceito OCP indica principalmente o uso da herança para praticarmos o extensão dos métodos.

Para solucionar o impasse e colocar o trem em OCP, criamos várias classes, cada uma com uma responsabilidade definida, suas próprias regras de negócios e sem a necessidade de alterarmos a funcionalidade padrão devido à criação de uma nova regra.

```
1  public abstract class Email
2  {
3          public abstract void Enviar(string assunto, string mensagem);
4  }
5
6  public class TextoEmail : Email
7  {
8          public override void Enviar(string assunto, string mensagem)
9          {
10                 Util.RemoverFormatacao(mensagem);
11         }
12 }
13
14 public class HtmlEmail : Email
15 {
16         public override void Enviar(string assunto, string mensagem)
17         {
18                 Util.InserirHtml(mensagem);
19         }
20 }
21
22 public class CriptografadoEmail : Email
23 {
24         public override void Enviar(string assunto, string mensagem)
25         {
26                 Util.CriptografarMensagem(mensagem);;
27         }
28 }
```

OCP Class Solution hosted with ❤ by GitHub

3- Princípio da substituição de Liskov

As classes derivadas devem ser substituíveis para as classes base.

O Liskov Substitution Principle (LSP) ou Princípio de Substituição de Liskov está diretamente ligado ao OCP (Open Closed Principle).

O LSP tem como objetivo nos alertar quanto a utilização da herança, que é um poderoso mecanismo e deve ser utilizado com extrema parcimônia.

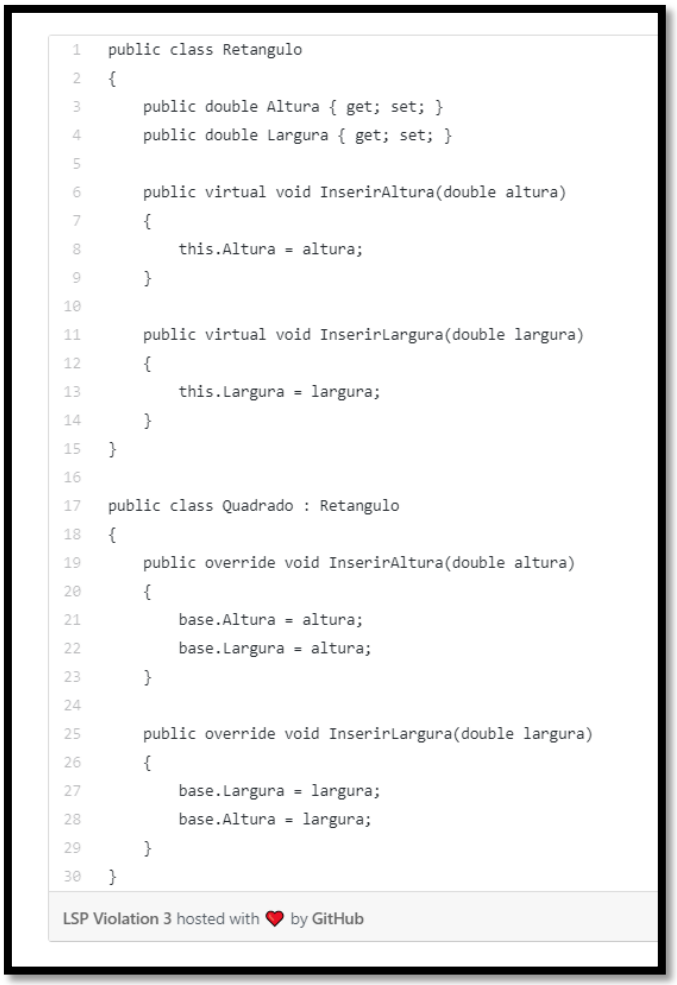
> *"Se para cada objeto x1 do tipo S há um objeto x2 do tipo T de tal forma que, para todos os programas P definidos em termos de T, o comportamento de P não muda quando x1 é substituído por x2 então S é um subtipo de T."*
>
> *- Clarice Lispector*

A afirmação acima será expressa em código abaixo:

```
1  public class T { //... }
2
3  public class S : T { //... }
4
5  public static class ProgramP
6  {
7      public static string AcceptObject(T obj)
8      {
9          return "ok !";
10     }
11 }
12
13 class Program
14 {
15     static void Main(string[] args)
16     {
17         var x1 = new T();
18         var x2 = new S();
19
20         //Aceita o objeto do tipo T
21         Console.WriteLine(ProgramP.AcceptObject(x1));
22
23         //Aceita objeto do tipo S que é um subtipo de T
24         Console.WriteLine(ProgramP.AcceptObject(x2));
25     }
26 }
```

LSP Premise hosted with ❤ by GitHub

Vou mostrar um exemplo de um cara que violou o princípio LSP



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

    public virtual void InserirAltura(double altura)
    {
        this.Altura = altura;
    }

    public virtual void InserirLargura(double largura)
    {
        this.Largura = largura;
    }
}

public class Quadrado : Retangulo
{
    public override void InserirAltura(double altura)
    {
        base.Altura = altura;
        base.Largura = altura;
    }

    public override void InserirLargura(double largura)
    {
        base.Largura = largura;
        base.Altura = largura;
    }
}
```

LSP Violation 3 hosted with ❤ by GitHub

A geometria nos afirma que todo quadrado é um retângulo, mas, quando estamos desenvolvendo não podemos levar essa afirmativa ao pé da letra. Perceba que no exemplo acima definimos o valor da altura igual ao da sua largura no quadrado, dessa forma deixamos explicito que nem sempre um quadrado é um retângulo e vemos um tipo clássico de violação do LSP.

Outra maneira de violar o LSP é lançar uma exceção inesperada ou sobrescrever um método e deixa-lo vazio.

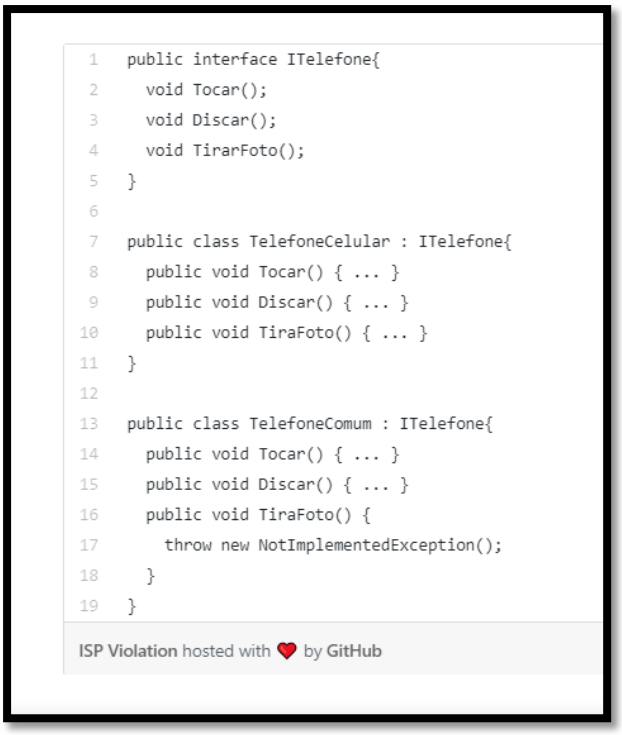
Usar o LSP é essencial para que o código fique coerente e faça sentido, além de facilitar a manutenibilidade e a realização de testes.

4- Princípio da segregação de interfaces

Muitas interfaces específicas são melhores do que uma interface única geral. Em suma, esse princípio é gordofóbico: ele joga na cara as desvantagens de interfaces “gordas”. Uma interface não pode forçar uma classe a implementar coisas que ela não utilizará.

Interfaces que tem muitos comportamentos (gordas) geralmente se espalham pelo sistema trazendo complexidade e dificuldade de manutenção ao código.

Dê uma boa olhada no exemplo de violação desse princípio abaixo:



```
1  public interface ITelefone{
2    void Tocar();
3    void Discar();
4    void TirarFoto();
5  }
6
7  public class TelefoneCelular : ITelefone{
8    public void Tocar() { ... }
9    public void Discar() { ... }
10   public void TiraFoto() { ... }
11 }
12
13 public class TelefoneComum : ITelefone{
14   public void Tocar() { ... }
15   public void Discar() { ... }
16   public void TiraFoto() {
17     throw new NotImplementedException();
18   }
19 }
```

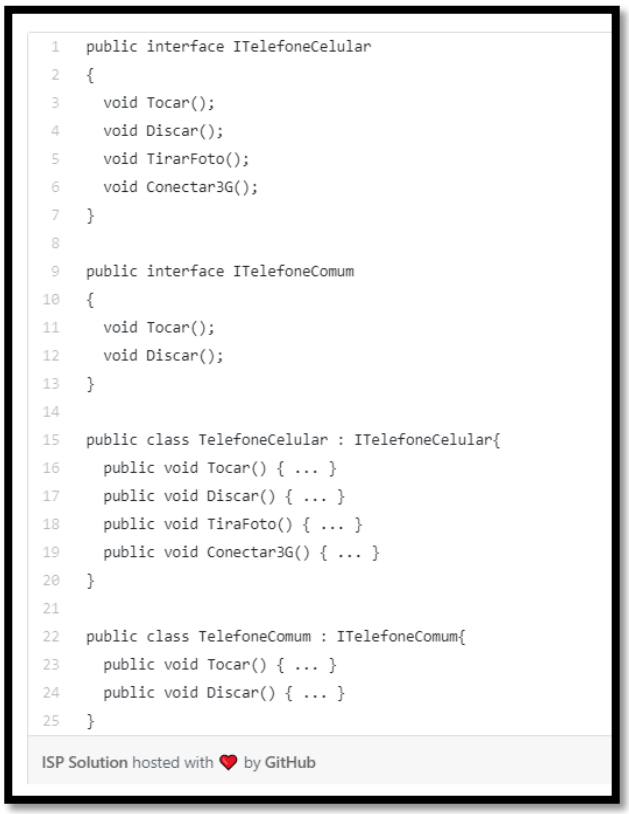
ISP Violation hosted with ❤ by GitHub

Veja que a classe TelefoneCelular implementou a interface corretamente e todos os métodos eram usuais a classe.

Já para a classe TelefoneComum tivemos um método que lançou uma Exception, pois aquele metódo não tinha utilidade para a classe. (um telefone comum não tira foto).

Percebemos que, ao, criarmos uma Interface genérica e nada específica às nossas classes, pode ficar chatinho e difícil de fazer manutenção posterior ao código.

Pois bem, como a gente faz de tudo aqui, bora corrigir isso:



```
1  public interface ITelefoneCelular
2  {
3    void Tocar();
4    void Discar();
5    void TirarFoto();
6    void Conectar3G();
7  }
8
9  public interface ITelefoneComum
10 {
11   void Tocar();
12   void Discar();
13 }
14
15 public class TelefoneCelular : ITelefoneCelular{
16   public void Tocar() { ... }
17   public void Discar() { ... }
18   public void TiraFoto() { ... }
19   public void Conectar3G() { ... }
20 }
21
22 public class TelefoneComum : ITelefoneComum{
23   public void Tocar() { ... }
24   public void Discar() { ... }
25 }
```

ISP Solution hosted with ❤ by GitHub

Tá vendo que é suave? É só criar interfaces específicas para as classes

Respeitando a premissa do ISP geramos facilidade de manutenção, pois temos especificidade nas classes clientes, quebramos o acoplamento entre as classes que a implementação de interfaces “gordas” traz e ainda ganhamos coesão e eficiência no código. Usar isso aí é bom demais, só alegria.

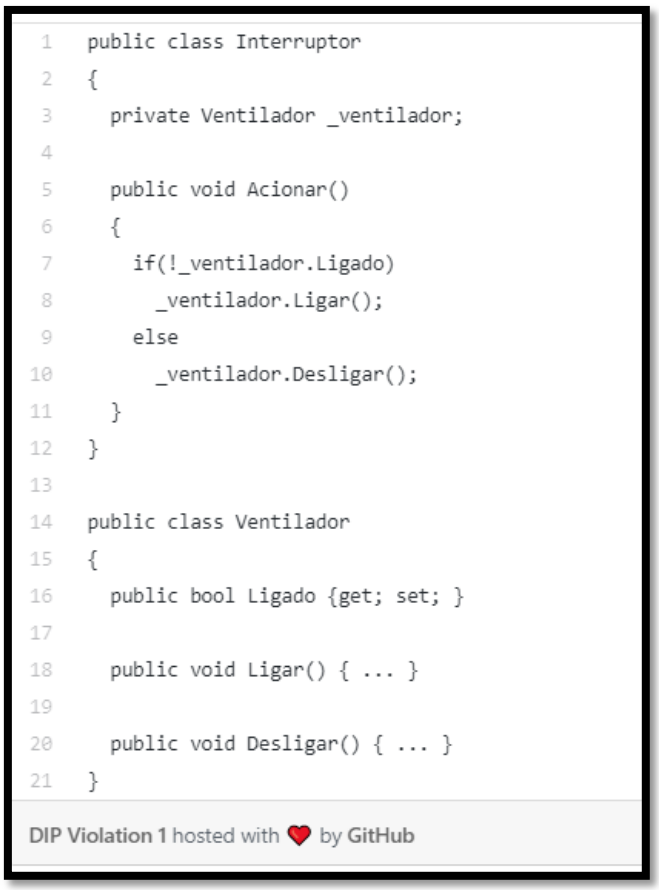
5- Princípio da inversão de dependência

Dependa de abstrações e não de implementações.

O DIP ou Princípio da Inversão de Dependência é a base para termos um projeto com um excelente design orientado a objetos, focado no domínio e com uma arquitetura flexível.

De uma forma objetiva o princípio nos faz entender que sempre devemos depender de abstrações e não das implementações, afinal de contas, as abstrações mudam menos e facilitam a mudança de comportamento e as evoluções do código.

Vamos observar um exemplo de violação desse princípio?



```
1  public class Interruptor
2  {
3    private Ventilador _ventilador;
4
5    public void Acionar()
6    {
7      if(!_ventilador.Ligado)
8        _ventilador.Ligar();
9      else
10       _ventilador.Desligar();
11   }
12 }
13
14 public class Ventilador
15 {
16   public bool Ligado {get; set; }
17
18   public void Ligar() { ... }
19
20   public void Desligar() { ... }
21 }
```

DIP Violation 1 hosted with ❤ by GitHub

No exemplo, a classe concreta Interruptor depende de uma outra classe concreta (Ventilador).

O Interruptor deveria ser capaz de acionar qualquer dispositivo independente de ser um ventilador uma lâmpada ou até mesmo um carro.

Vão corrigir então:

```
interface IDispositivo
{
  bool Ligado { get; set; }
  void Acionar();
  void Ligar();
  void Desligar();
}

public class Ventilador : IDispositivo
{
  public bool Ligado { get; set; }

  public void Acionar ()
  {
    if (!this.Ligado)
      this.Ligar();
    else
      this.Desligar();
  }

  public void Ligar() { ... }

  public void Desligar() { ... }
}

public class Lampada : IDispositivo
{
  public bool Ligado { get; set; }

  public void Acionar ()
  {
    if (!this.Ligado)
      this.Ligar();
    else
      this.Desligar();
  }

  public void Ligar() { ... }

  public void Desligar() { ... }
}

public class Interruptor
{
  private readonly IDispositivo _dispositivo;

  public void AcionarDispositivo()
  {
    _dispositivo.Acionar();
  }
}
```

DIP Solution hosted with ❤ by GitHub

Agora a classe concreta Interruptor depende da abstração de um IDispositivo e não mais de uma classe concreta.

Dica essencial:

Identificar as abstrações é importante para que mantenhamos o projeto flexível, robusto e preparado para que as futuras implementações não sejam difíceis e complexas.

O DIP trás uma série de benefícios, principalmente em relação a arquitetura de software. O principio torna o aplicação focada na resolução dos problemas, fazendo da implementação um mero detalhe.

Tendo como base a afirmativa acima, podemos perceber que a abstração IDispositivo está diretamente vinculado com o cliente (Interruptor), tornando sua implementação (Ventilador, Lampada) um detalhe.

Feito por: João Francisco Braga Cardoso