THE DEVELOPER'S CONFERENCE

Trilha Design de Código

# SOLID numa abordagem real

# Lucas Souto Maior


Projeto CIn/Samsung
Engenheiro de Software

lucassoutomaior@outlook.com

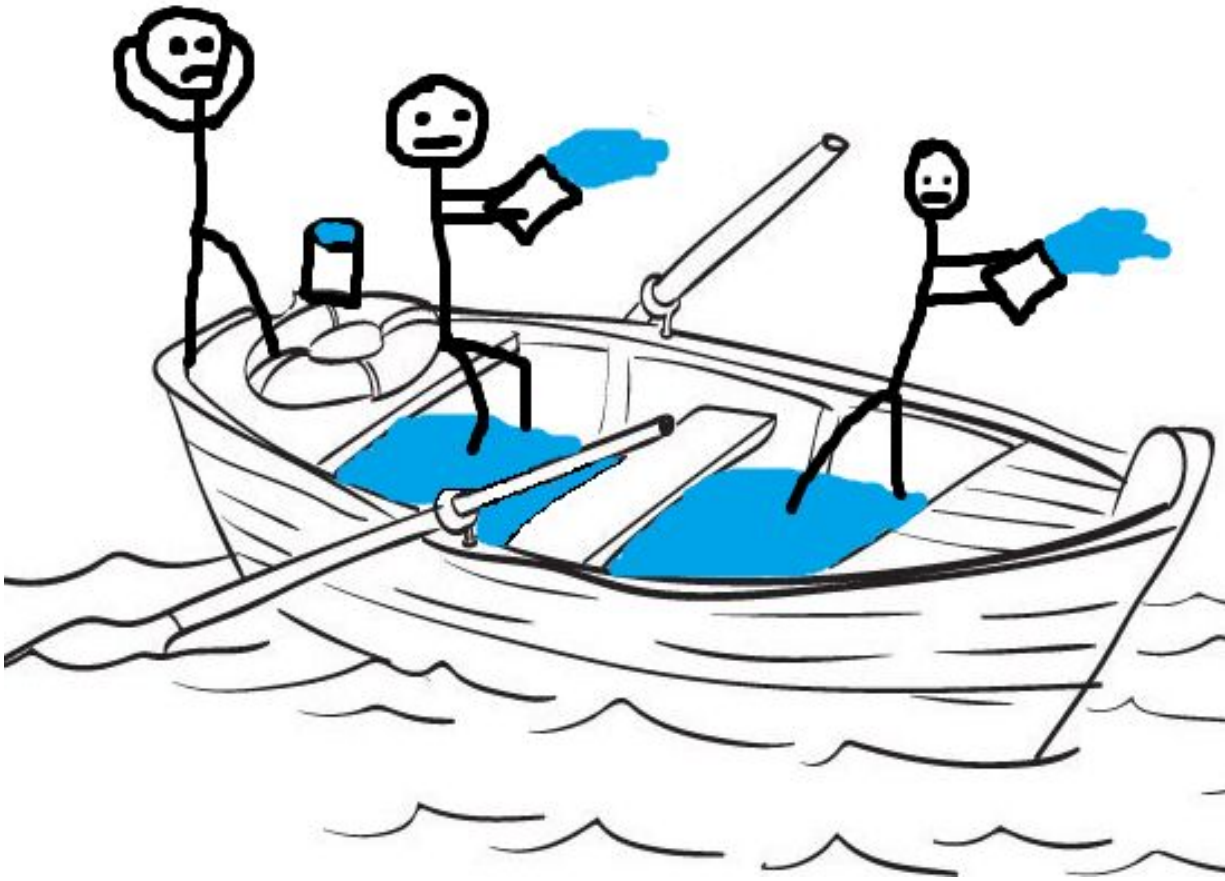
linkedin.com/in/lucas-souto-maior-b69b2083

github.com/soutolucas

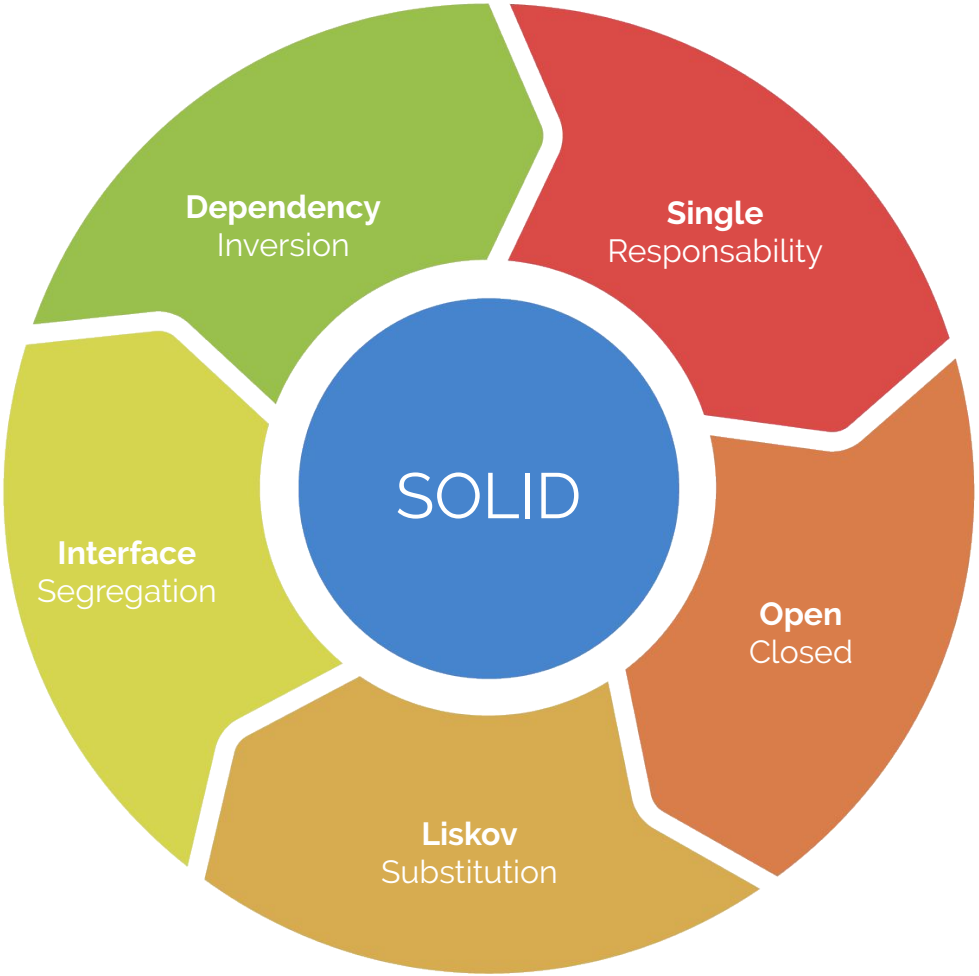
# Proposta da apresentação

- Falar das minhas experiências ao fazer uso do SOLID no meu dia-a-dia de desenvolvimento de software
- Exemplificar através de cenários mais reais, não apenas exemplos de código didáticos
- Os códigos exibidos foram adaptados devido a confidencialidade dos projetos

# Alguém já se sentiu assim?!



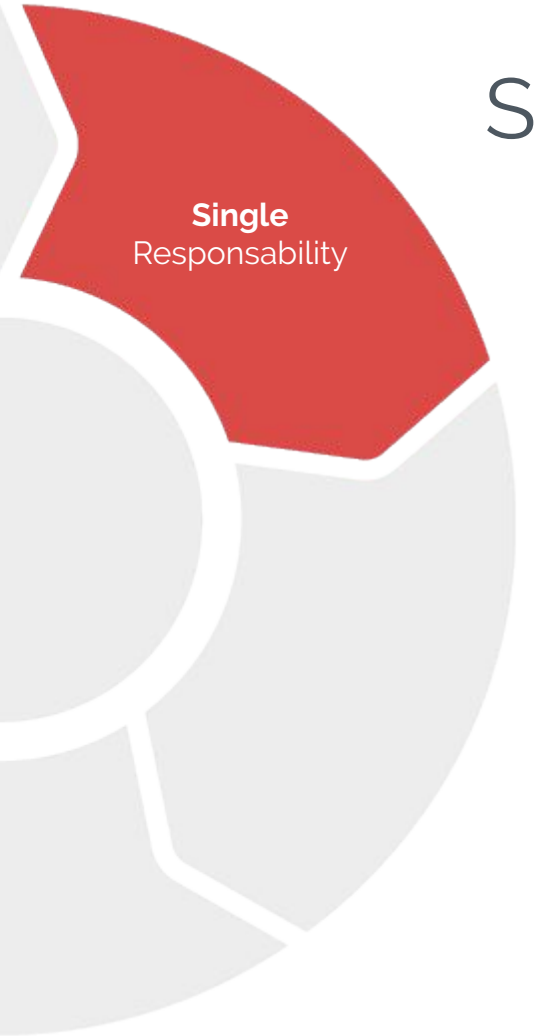

SOLID é um acrônimo para 5 princípios da programação orientada a objetos e design de código identificados por **Robert C. Martin** (Uncle Bob)


Dependency
Inversion
Single
Responsability
SOLID
Interface
Segregation
Open
Closed
Liskov
Substitution

# Benefícios

Facilidade em manter, estender, ajustar, testar

(Usar todo o potencial da Programação Orientada a Objetos)




# S - Single responsibility principle

**“Uma classe deve ter somente uma razão para mudar”**

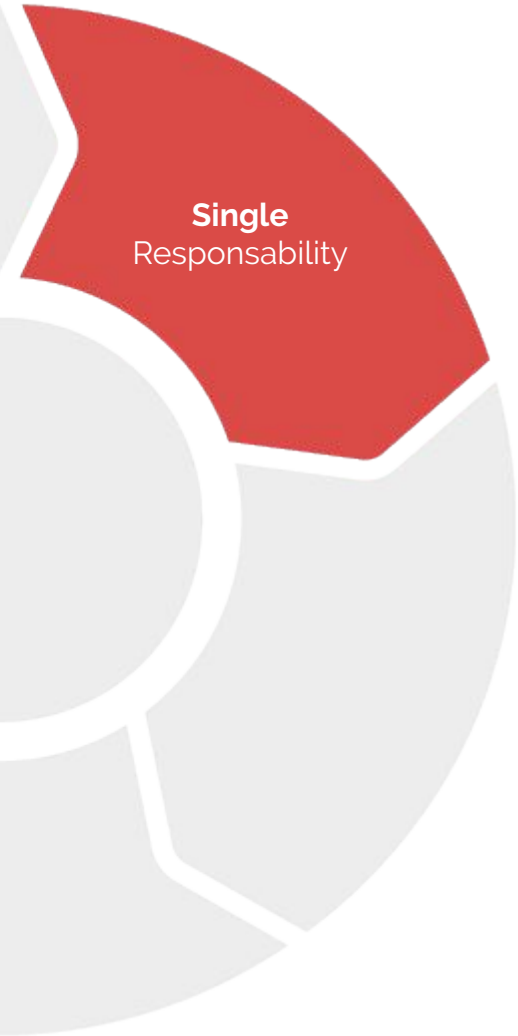
Isso se aplica não apenas a classes, mas também a funções, arquivos, etc

# S - Single responsibility principle

**Single**
Responsability

```
public double CalculateAndSaveArea(double a, double b)
{
    //Code to calculate Area
    //...

    //Code to save Area
    //...
}
```

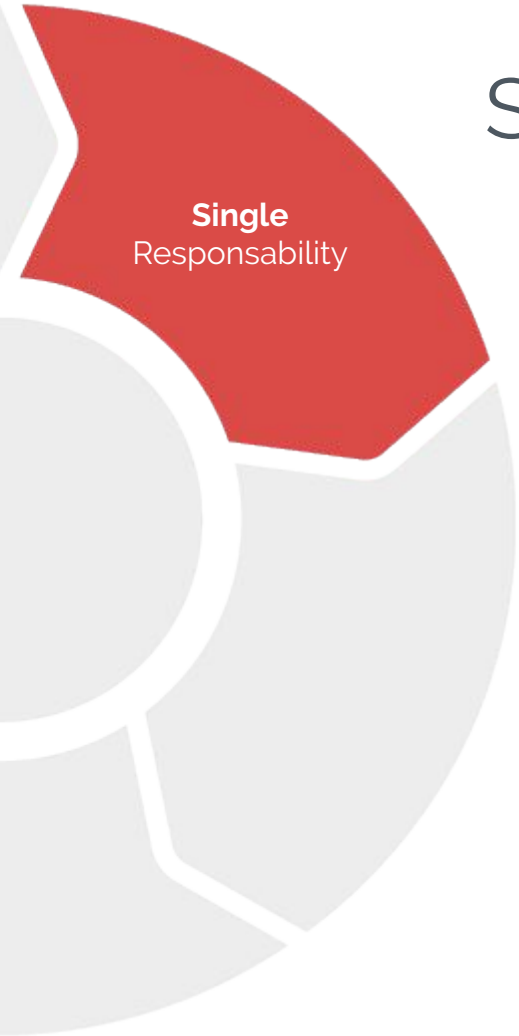
```
public class Aircraft
{
    public void SpeedUp() { /* Code to speed up */ }
    public void Brake() { /* Code to brake */ }
    public void FuelUp() { /* Code to fuel up */ }
}
```




# Problemas encontrados por falta de coesão:

- Dificuldade de compreensão e reuso
- Muitas responsabilidades podem tornar difícil alterar uma parte sem comprometer outra
- A classe tem um número excessivo de dependências (alto acoplamento), ficando mais sujeita a mudanças

# S - Single responsibility principle




```
public double CalculateArea(double a, double b)
{
    //Code to calculate Area
    //...
}

public double SaveArea(double area)
{
    //Code to save Area
    //...
}
```

```
public class Aircraft
{
    public void SpeedUp() { /* Code to speed up */ }
    public void Brake() { /* Code to brake */ }
}

public class FuelService
{
    public void FuelUp() { /* Code to fuel up */ }
}
```

# S - Single responsibility principle

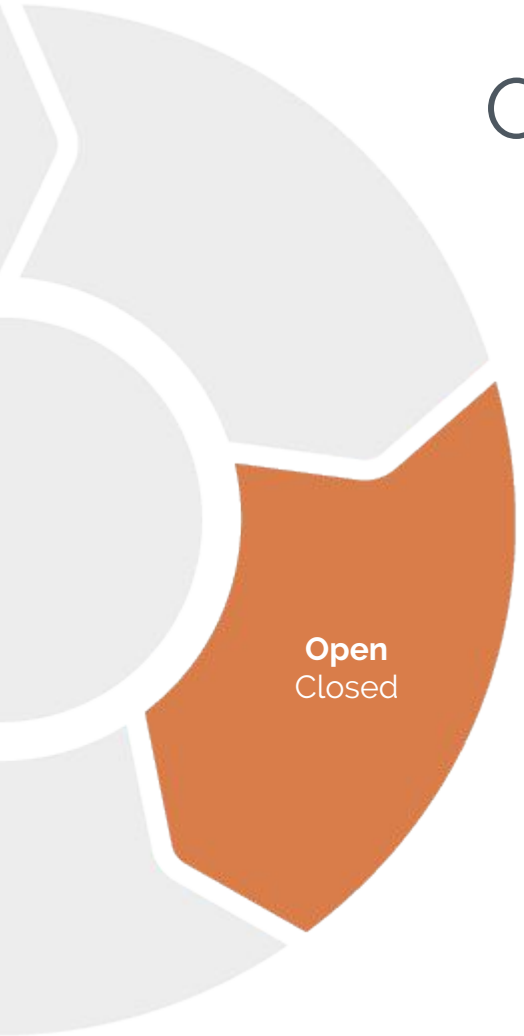
**Single**
Responsability

```
public void AddItemsLastPosition(IEnumerable<IItem> items)
{
    try
    {
        semaphore.Wait();

        foreach (var item in items)
        {
            list.AddLast(item);
            TotalValue += item.Value;
        }
    }
    finally
    {
        semaphore.Release();
    }
}
```

TotalValue?!

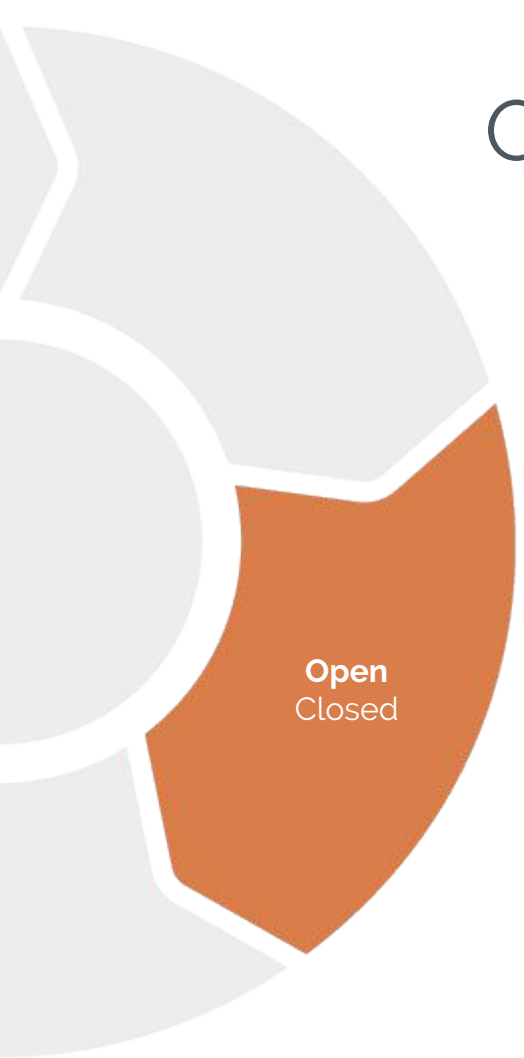
**Nem sempre é fácil...**

# O - Open/closed principle




**“Entidades de software (classes, módulos, funções etc) devem ser abertas para extensão mas fechadas para modificação”**

# O - Open/closed principle

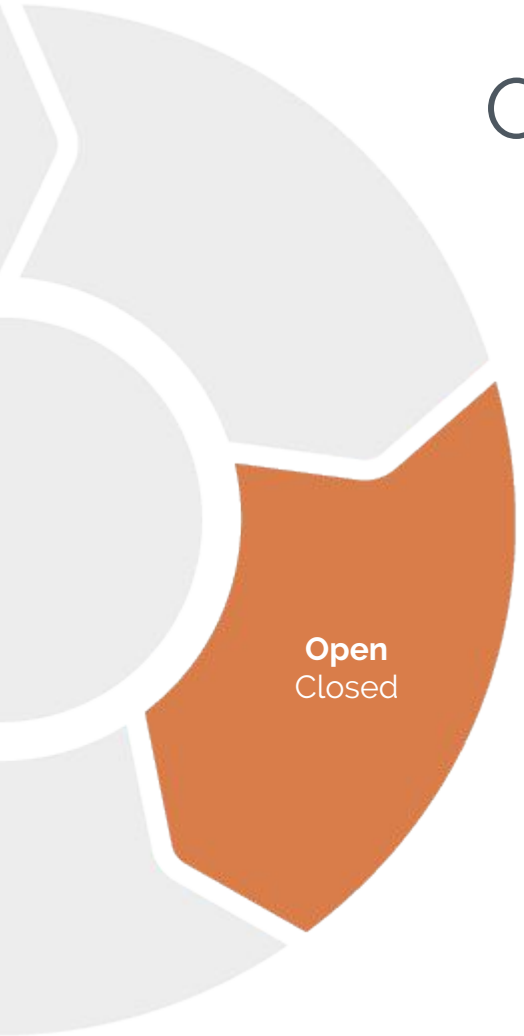



**Problema:**

Gostaria de uma classe que efetue pagamentos

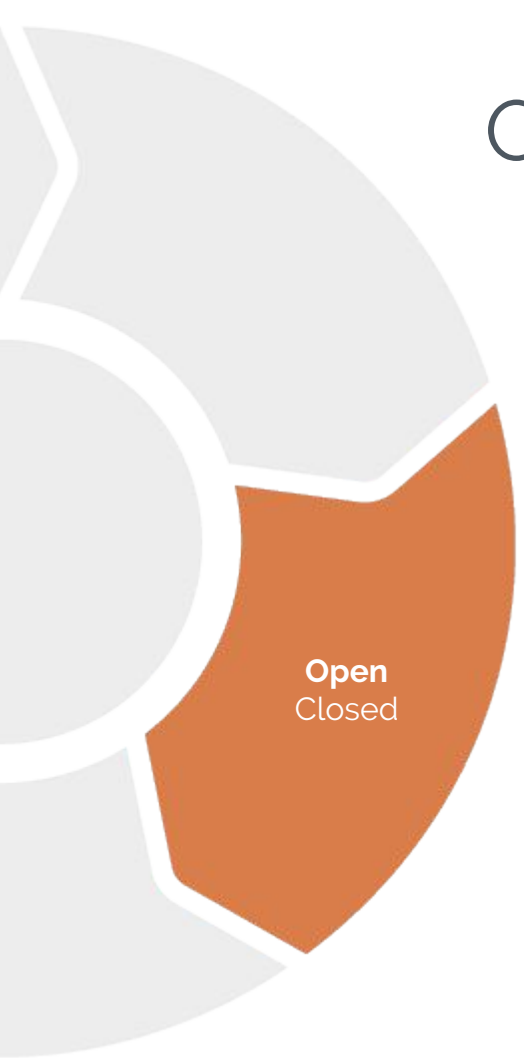
- Preciso checar se há saldo antes de executar o pagamento;
- O pagamento pode ser realizado em dinheiro, cartão ou boleto bancário

# O - Open/closed principle




```
public class Payment
{
    private readonly PaymentType paymentType;

    public Payment(PaymentType paymentType)
    {
        this.paymentType = paymentType;
    }

    public void Perform()
    {
        if (HasBalance())
        {
            if (paymentType == PaymentType.Money) { /*Code for money payment*/ }
            else if (paymentType == PaymentType.Card) { /*Code for card payment*/ }
            else if (paymentType == PaymentType.BankSlip) { /*Code for bank slip payment*/ }
        }
    }

    private bool HasBalance() { /*Code to check balance*/ }
}
```

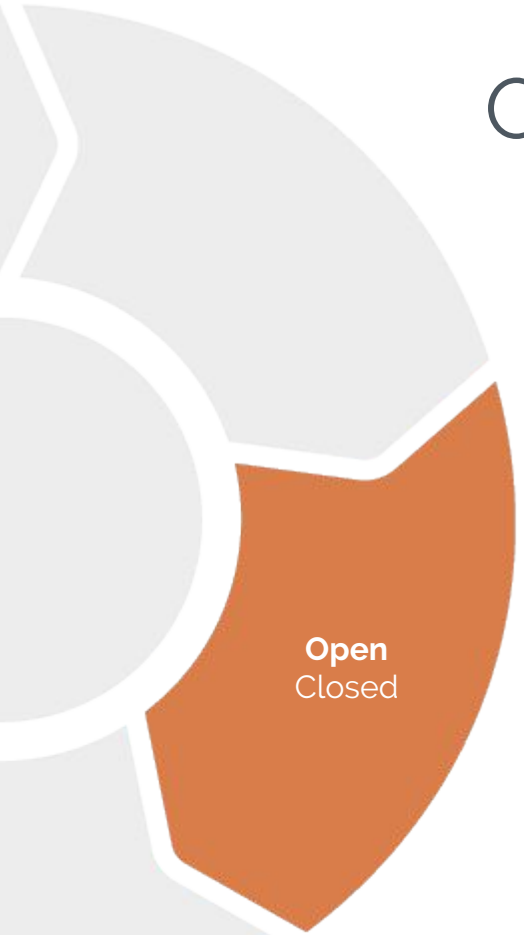
# O - Open/closed principle

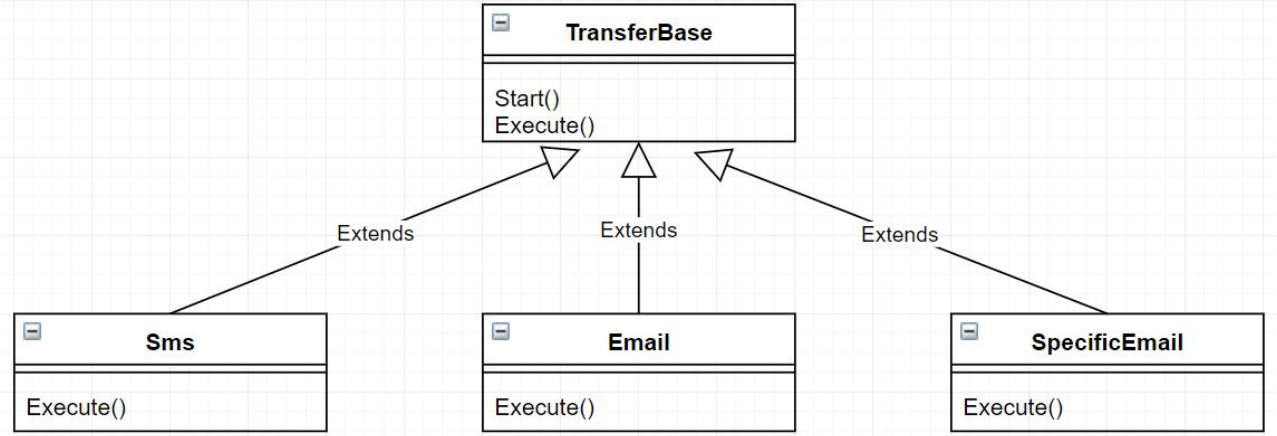



Apenas para encapsular a chamada para um serviço de Balance.

```
public abstract class Payment
{
    protected abstract void Execute();

    public void Perform()
    {
        if (HasBalance())
            Execute();
    }

    private bool HasBalance() { /*Code to check balance*/ }
}

public class MoneyPayment : Payment
{
    protected override void Execute()
    {
        //Code for money payment
    }
}

public class CardPayment : Payment { /* Code... */}
public class BankSlipPayment : Payment { /* Code... */}
```

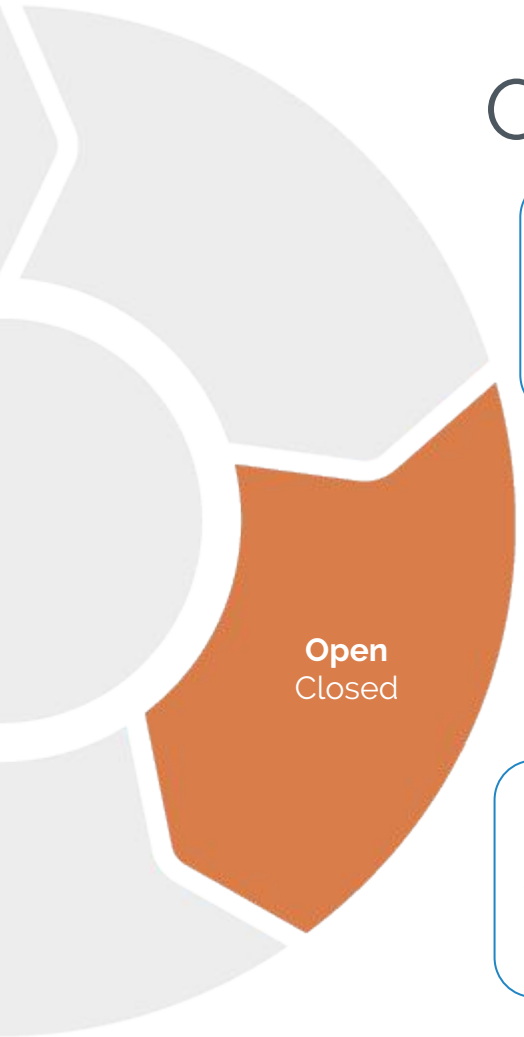
# O - Open/closed principle



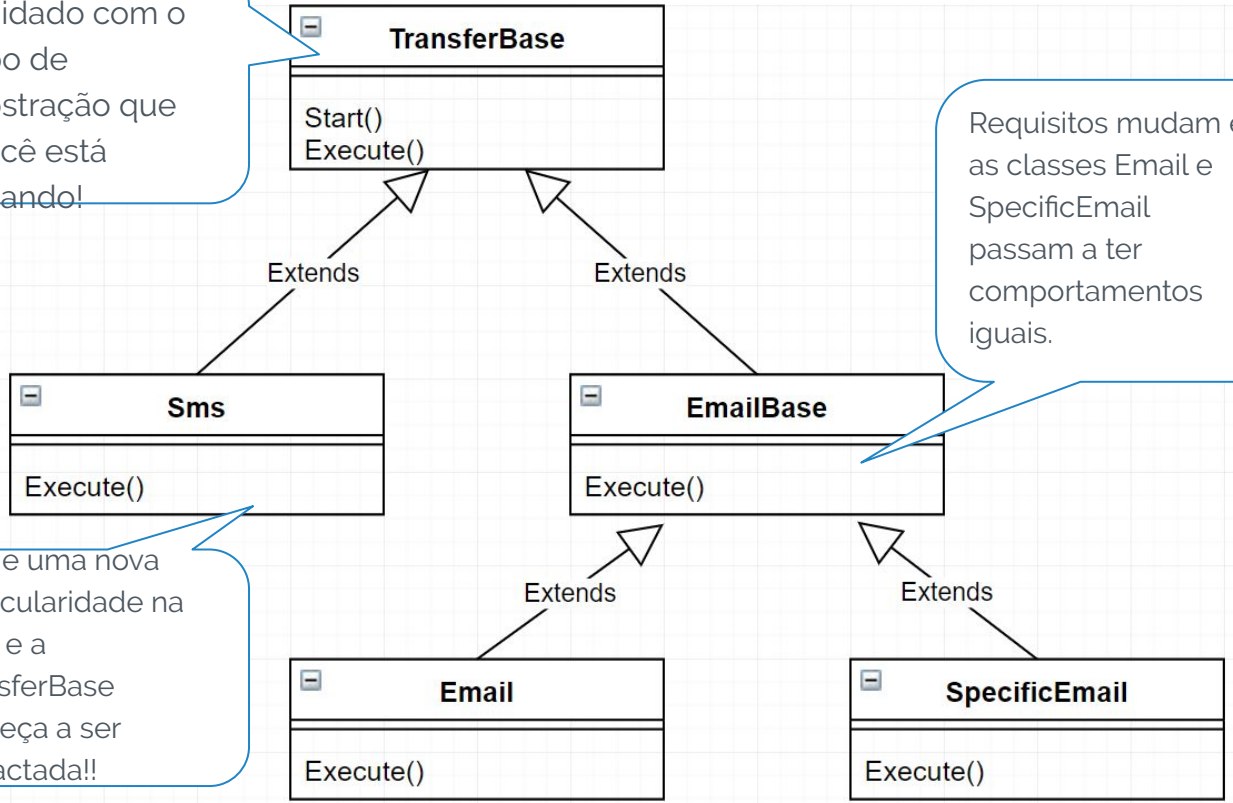




**Uma experiência real frustrada...**

# O - Open/closed principle




Tenha muito cuidado com o tipo de abstração que você está criando!



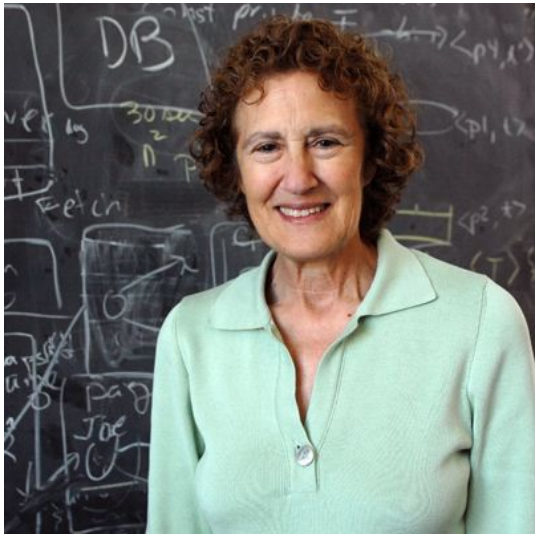

Requisitos mudam e as classes Email e SpecificEmail passam a ter comportamentos iguais.

Surge uma nova particularidade na Sms e a TransferBase começa a ser impactada!!

# L - Liskov substitution principle

O Princípio de Substituição de Liskov leva esse nome por ter sido criado por **Barbara Liskov**, em 1988.

**“As classes base devem ser substituíveis por suas classes derivadas.”**



Barbara Liskov

# L - Liskov substitution principle

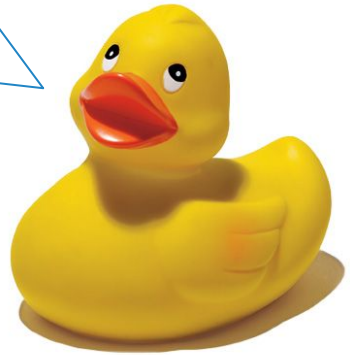
**O problema do pato...**

# L - Liskov substitution principle

```
public class Duck
{
    public void Quack() { }

    public void Fly() { }
}

public class RubberDuck : Duck
{
    //Rubber duck doesn't fly
}
```

# L - Liskov substitution principle

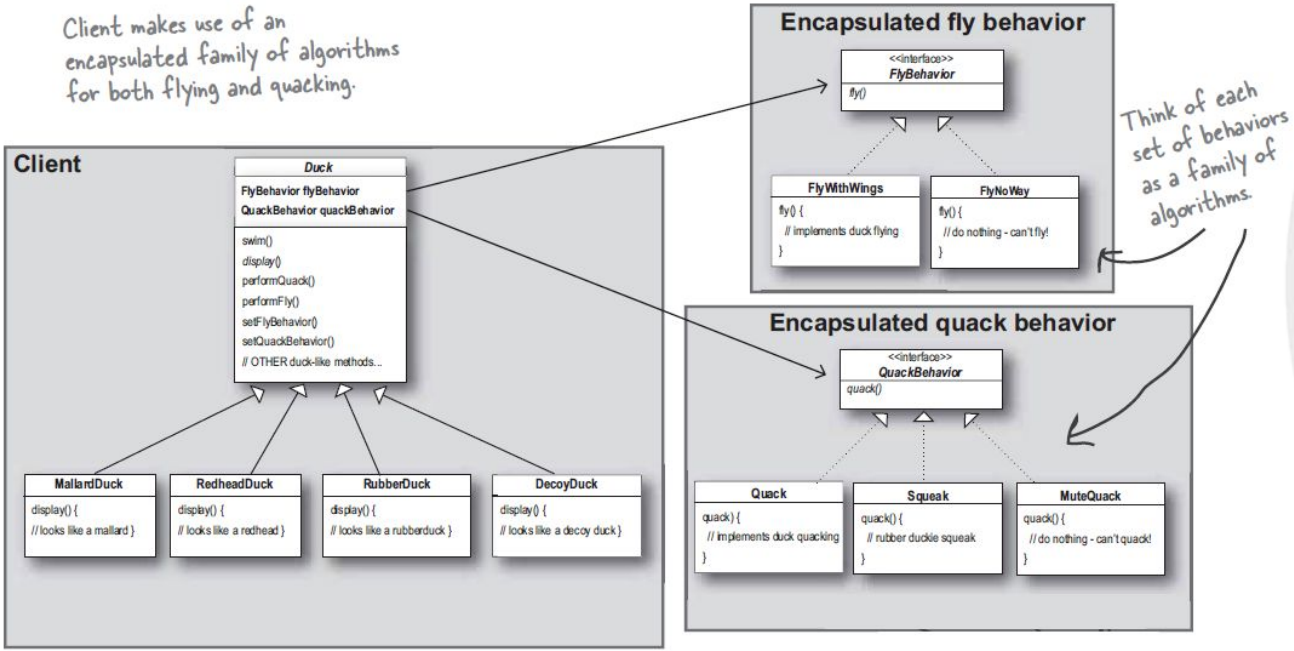
Poderíamos criar uma classe para patos que voam e outra para patos que não voam, ambas herdando de Pato. A minha classe herdaria da classe “PatosQueNaoVoam” e as demais aves herdariam da classe “PatosQueVoam”. O que vocês acham?



# L - Liskov substitution principle


**...surge um pato de madeira.**

# L - Liskov substitution principle




**Liskov**
Substitution

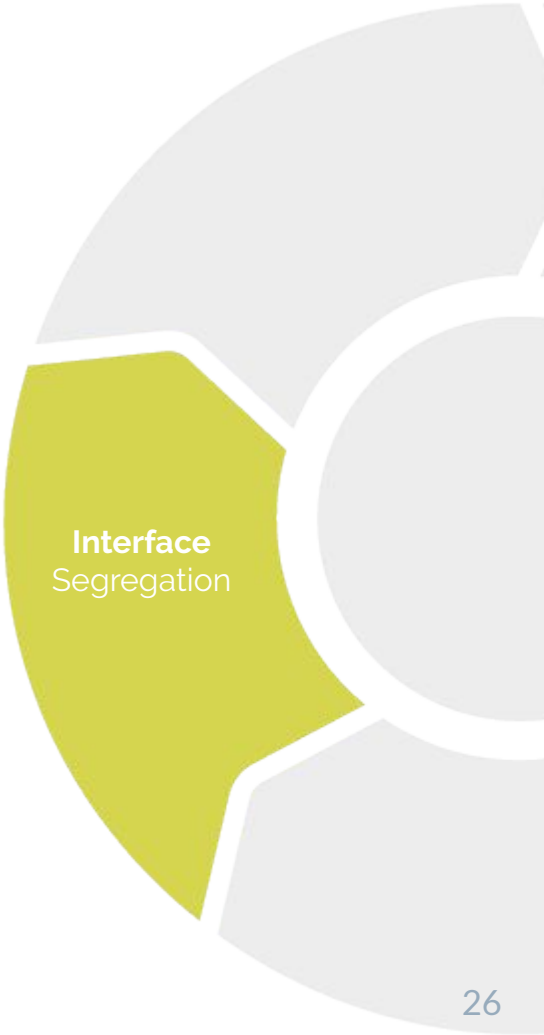
# L - Liskov substitution principle

**“Crie suas classes pensando em herança, ou então proíba-a”**
**Joshua Bloch**

- No começo das linguagens orientadas a objeto, a herança era usada para vender a ideia
- Mas, utilizar herança pode não ser tão simples. É fácil cair em armadilhas criadas por hierarquias de classes longas ou confusas

# I - Interface segregation principle

**“Clientes não devem ser forçados a depender de métodos que não usam”**



# I - Interface segregation principle

```
public interface IDevice
{
    void TurnOn();
    void TurnOff();
    void ToCall();
}

public class Smartphone : IDevice
{
    public void TurnOn() { /* Code to turn on */ }
    public void TurnOff() { /* Code to turn off */ }
    public void ToCall() { /* Code to call */ }
}

public class Laptop : IDevice
{
    public void TurnOn() { /* Code to turn on */ }
    public void TurnOff() { /* Code to turn off */ }
    public void ToCall()
    {
        throw new NotImplementedException("Laptop doesn't perform call");
    }
}
```

# I - Interface segregation principle

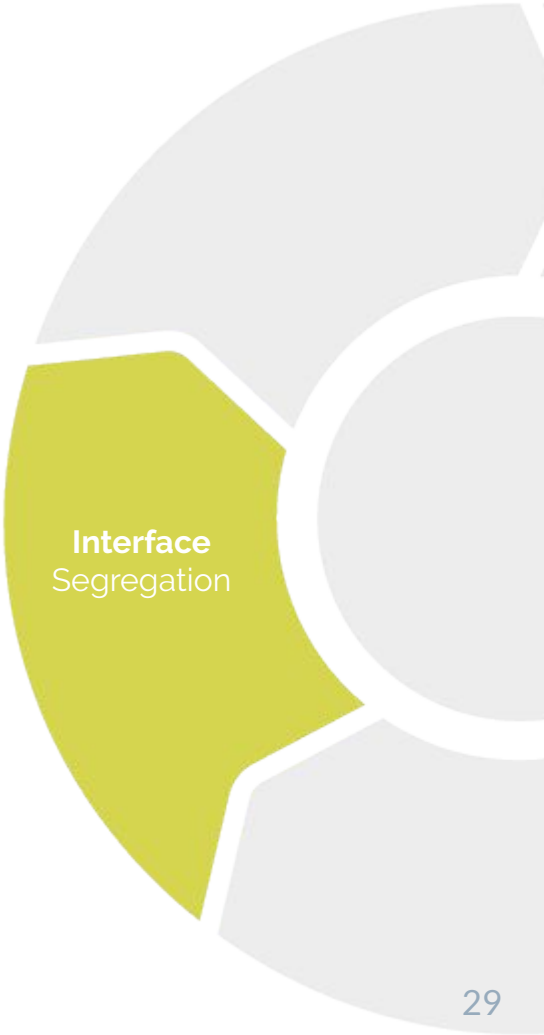
```
public interface ISmartphone
{
    void TurnOn();
    void TurnOff();
    void ToCall();
}

public interface ILaptop
{
    void TurnOn();
    void TurnOff();
}

public class Smartphone : ISmartphone
{
    public void TurnOn() { /* Code to turn on */ }
    public void TurnOff() { /* Code to turn off */ }
    public void ToCall() { /* Code to call */ }
}

public class Laptop : ILaptop
{
    public void TurnOn() { /* Code to turn on */ }
    public void TurnOff() { /* Code to turn off */ }
}
```

# I - Interface segregation principle

**Já tentei de outras formas. Deu certo?**

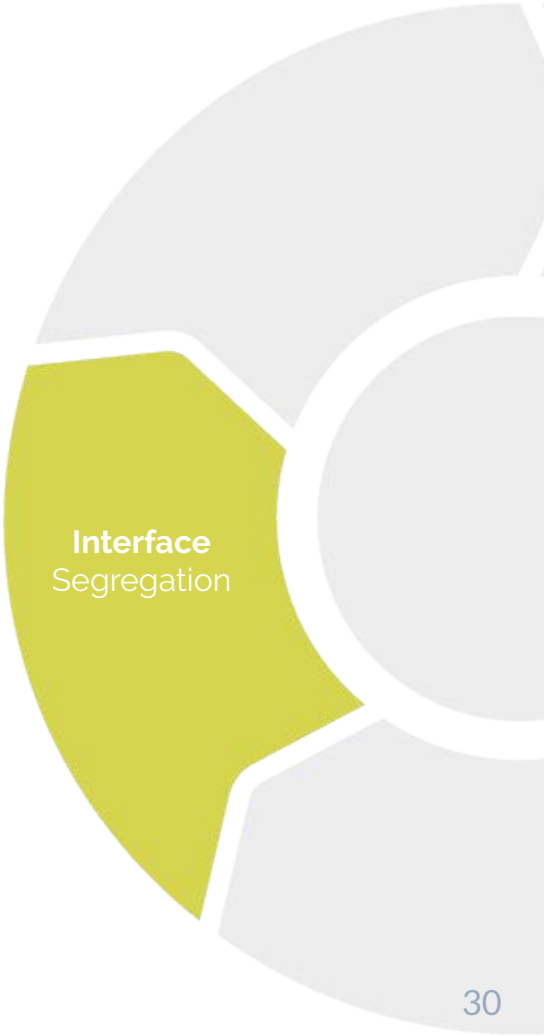
```
public interface IDevice
{
    void TurnOn();
    void TurnOff();
}

public interface ISmartphone : IDevice
{
    void ToCall();
}
```

```
public interface ITurnOnOff
{
    void TurnOn();
    void TurnOff();
}

public interface ICaller
{
    void ToCall();
}

public class Smartphone : ITurnOnOff, ICaller
{
    //Code...
}
```



# I - Interface segregation principle

**E a quantidade de métodos numa interface?**



# D - Dependency inversion principle


**“Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.**

**Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.”**

# D - Dependency inversion principle

```
public class TaxCalculator
{
    public void Calculate(bool someCondition)
    {
        //Code to Calculate Tax...

        if (someCondition)
            new SimpleRule().Calculate();
        else
            new CompoundRule().Calculate();
    }
}

public class SimpleRule
{
    public double Calculate() { /* Code to calculate simple rule */}
}

public class CompoundRule
{
    public double Calculate() { /* Code to calculate compound rule */}
}
```

# D - Dependency inversion principle

```
public class TaxCalculator
{
    private readonly ICalculateRule calculateRule;
    public TaxCalculator(ICalculateRule calculateRule)
    {
        this.calculateRule = calculateRule;
    }

    public void Calculate()
    {
        //Code to Calculate Tax...

        this.calculateRule.Calculate();
    }
}
```

```
public interface ICalculateRule
{
    double Calculate();
}

public class SimpleRule : ICalculateRule
{
    public double Calculate()
    {
        //Code to calculate simple rule
    }
}

public class CompoundRule : ICalculateRule
{
    public double Calculate()
    {
        //Code to calculate compound rule
    }
}
```

# D - Dependency inversion principle

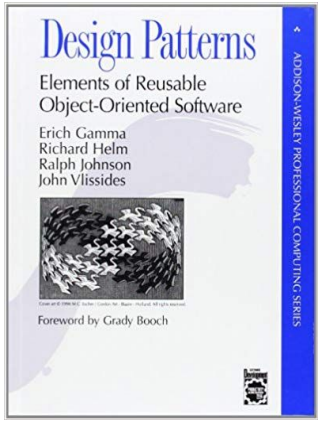
**Mais um ponto importante**
**sobre depender de interfaces...**

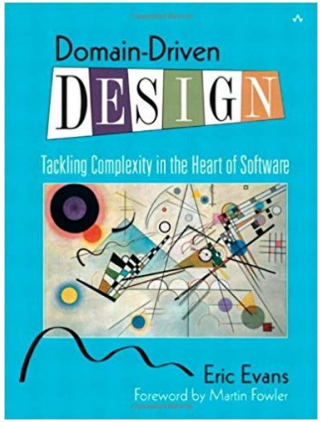
# Considerações finais

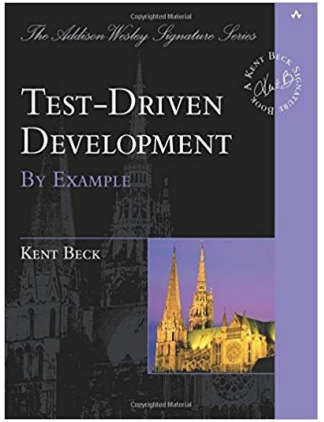
- Use os princípios SOLID como guia. Não leve tudo ao "pé da letra"
- Após conhecê-los, cuidado com a vontade de sair refatorando tudo
- Não tenha muito apego pelo design de código que você criou, mude caso seja necessário
- Ficou ruim pra testar o código? Reavalie
- Compartilhe conhecimento com sua equipe

# Indicações de leitura





# Obrigado!

**Dúvidas?**