

# S.O.L.I.D.

PRINCÍPIOS PARA UM BOM CÓDIGO OO

SOFTPLAN, 2019



1

VOCÊ PRODUZ CÓDIGO DE QUALIDADE ?



SOFTPLAN, 2019

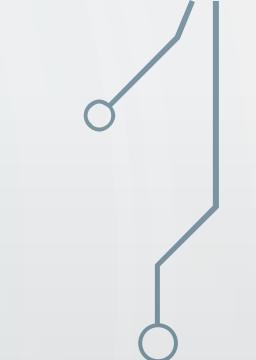
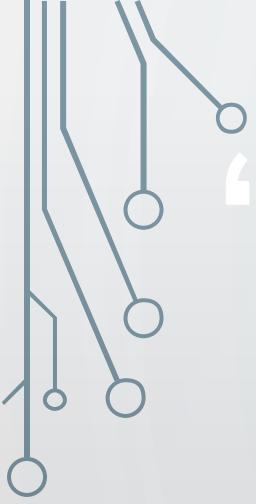
# CLARO QUE SIM, MAS....

- Mas seu projeto possui algum desses sintomas ?
  - Repetição de código
  - Falta de coesão
  - Falta de padronização
  - Rigidez e fragilidade dificultando as alterações
  - Falta de testabilidade
  - Falta de modularidade



# OS 5 PRINCÍPIOS S.O.L.I.D.

- Por volta de 2000
- Robert C. Martin (*Uncle Bob*)
- Paper: Design Principles and Design Patterns
- Caminhos para evitar alguns problemas da OO
- São Princípios, não regras, que nos ajudam a criar Softwares melhores



## SRP - SINGLE RESPONSIBILITY PRINCIPLE

”

*A class should have one, and only one, reason to change*

# OCP - OPEN/CLOSED PRINCIPLE

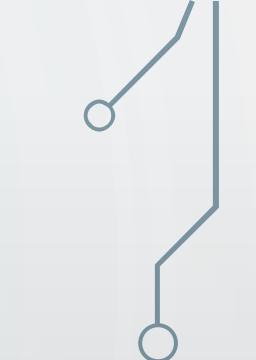
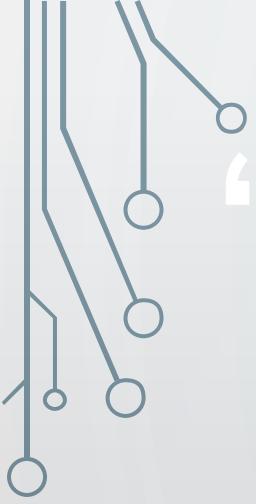
You should be able to extend a classes behavior, without modifying it.

# LSP - LISKOV SUBSTITUTION PRINCIPLE

*Derived classes must be substitutable for their base classes.*

# ISP - INTERFACE SEGREGATION PRINCIPLE

*Make fine grained interfaces that are client specific.*



# DIP - DEPENDENCY INVERSION PRINCIPLE

”

*Depend on abstractions, not on concretions.*

# RESULTADOS DOS S.O.L.I.D.

- Melhor design de código, com melhor Organização e Padronização
- Manutenções e Evoluções mais fáceis e seguras
- Melhor Testabilidade
- Melhor Modularização
- Mais clareza

SOFTPLAN, 2019

# SRP

# SINGLE RESPONSIBILITY PRINCIPLE

SOFTPLAN, 2019



# COESÃO

- “*União harmônica entre uma coisa e outra; harmonia: a coesão das partes de um Estado.*” (dicio.com)
- Uma das primeiras lições das aulas de programação
- Cada classe ou método precisa ter apenas uma única responsabilidade



OK.  
MAS COMO ESCREVER CLASSES COESAS ?

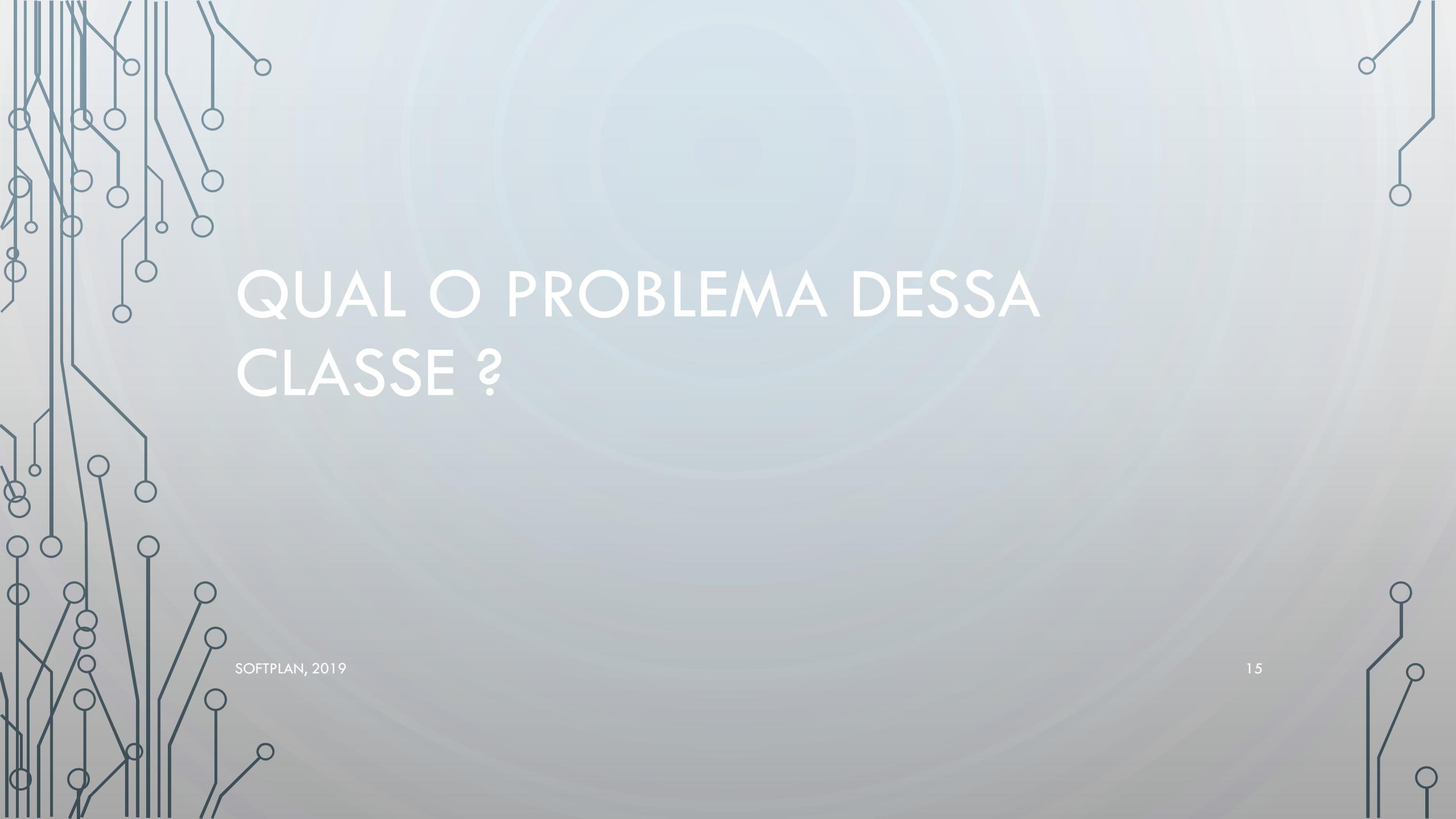
VAMOS AO NOSSO PRIMEIRO EXEMPLO.

SOFTPLAN, 2019

13

# PROBLEMAS DE COESÃO

```
public class CalculadoraDeBonus {  
  
    public double calculaBonus(Funcionario funcionario) {  
        if(DESENVOLVEDOR.equals(funcionario.getCargo())) {  
            return dezOuVintePorcento(funcionario);  
        }  
        if(DBA.equals(funcionario.getCargo()) ||  
            TESTER.equals(funcionario.getCargo())) {  
            return quinzeOuVinteCincoPorcento(funcionario);  
        }  
        throw new RuntimeException("Cargo invalido");  
    }  
  
    private double dezOuVintePorcento(Funcionario funcionario) {  
        if(funcionario.getSalarioBase() > 3000.0) {  
            return funcionario.getSalarioBase() * 0.9;  
        } else {  
            return funcionario.getSalarioBase() * 0.8;  
        }  
    }  
  
    private double quinzeOuVinteCincoPorcento(Funcionario funcionario) {  
        if(funcionario.getSalarioBase() > 3000.0) {  
            return funcionario.getSalarioBase() * 0.85;  
        } else {  
            return funcionario.getSalarioBase() * 0.75;  
        }  
    }  
}
```



# QUAL O PROBLEMA DESSA CLASSE ?

SOFTPLAN, 2019

15

# QUAL O PROBLEMA DESSA CLASSE ?

- Quantos cargos existem na Softplan ?
- Quantos IFs seriam necessários para implementar essa classe ?
- Classes como essas tendem a crescer e mudar muito
- Imagine precisar alterar esta classe após 5 anos de criada !

# O PROBLEMA DAS CLASSES NÃO COESAS

- Alta complexidade
- Baixo reutilização
- Excesso de dependências
- Fragilidade
- Não param de crescer..... e mudar .... e crescer mais

# ENTÃO VAMOS EM BUSCA DA COESÃO

- Vamos entender o problema: Por que nossa classe não é coesa ?
  - Implementar um regra e associar a regra ao cargo podem ser responsabilidades distintas ?
  - Quando um novo **Cargo** surge, a classe **CalculadoraDeBonus** precisará **crescer**
  - Quando uma nova **Regra de Cálculo** surge, a classe **CalculadoraDeBonus** precisará **mudar**
- Precisamos generalizar este problema e organizar um pouco

# RESOLVENDO O PROBLEMA DE COESÃO COM AS REGRAS

SOFTPLAN, 2019

19

# VAMOS GENERALIZAR O PROBLEMA E FIXAR UM CONTRATO

```
public interface RegraDeCalculo {  
    double calcula(Funcionario f);  
}
```

# VAMOS IMPLEMENTAR CADA REGRA EM SUA PRÓPRIA CLASSE

```
public interface RegraDeCalculo {  
    double calcula(Funcionario f);  
}
```

```
public class DezOuVintePorCento implements RegraDeCalculo {  
    public double calcula(Funcionario funcionario) {  
        if(funcionario.getSalarioBase() > 3000.0) {  
            return funcionario.getSalarioBase() * 0.9;  
        } else {  
            return funcionario.getSalarioBase() * 0.8;  
        }  
    }  
}
```

```
public class QuinzeOuVinteCincoPorCento implements RegraDeCalculo {  
    public double calcula(Funcionario funcionario) {  
        if(funcionario.getSalarioBase() > 3000.0) {  
            return funcionario.getSalarioBase() * 0.85;  
        } else {  
            return funcionario.getSalarioBase() * 0.75;  
        }  
    }  
}
```

# RESULTADO DA REFATORAÇÃO

- Design de classes mais claro
  - Fica evidente onde as regras devem ser implementadas
  - Fica claro o formato de uma regra: parâmetros, retorno, assinatura
- Redução da complexidade da `CalculadoraDeBonus`
- Isolamos a complexidade das regras de cálculo
- Testabilidade
- Manutenibilidade

# MAS, NOSSA CALCULADOR CONTINUA COMPLEXA

```
public class CalculadoraDeBonus {  
  
    public double calcula(Funcionario funcionario) {  
        if(DESENVOLVEDOR.equals(funcionario.getCargo())) {  
            return new DezOuVintePorCento().calcula(funcionario);  
        }  
        if(DBA.equals(funcionario.getCargo()) ||  
            TESTER.equals(funcionario.getCargo())) {  
            return new QuinzeOuVinteCincoPorCento().calcula(funcionario);  
        }  
        throw new RuntimeException("Cargo invalido");  
    }  
}
```

# MELHORANDO NOSSA CALCULADORA

- Design e Interações claras
- A mágica do Encapsulamento bem definido
- Conectando os Cargos e as Regras no local certo
- Nenhum cargo ficará sem regra

```
public enum Cargo {  
    DBA(new QuinzeOuVinteCincoPorCento()),  
    TESTER(new QuinzeOuVinteCincoPorCento()),  
    DESENVOLVEDOR(new DezOuVintePorCento());  
  
    private RegraDeCalculo regra;  
}
```

# MELHORANDO O ENCAPSULAMENTO DA NOSSA CALCULADORA

```
public class CalculadoraDeBonus {  
    public double calcula(Funcionario funcionario) {  
        return funcionario.getCargo()  
            .getRegra().calcula(funcionario);  
    }  
}
```

# SRP

# SINGLE RESPONSIBILITY PRINCIPLE

UMA CLASSE DEVE TER UM, E SOMENTE UM,  
MOTIVO PARA MUDAR

SOFTPLAN, 2019

26

# SRP - SINGLE RESPONSIBILITY PRINCIPLE

- Uma classe ter uma, e somente uma, RESPONSABILIDADE
- Classes coesas, ou que obedecem o SRP, tendem:
  - A ser menores e mais simples
  - A ser menos suscetíveis a problemas
  - A ser mais reutilizáveis
  - A ter uma a ter uma única responsabilidade

# SRP - SINGLE RESPONSIBILITY PRINCIPLE

TKS!

SOFTPLAN, 2019





# DIP DEPENDENCY INVERSION PRINCIPLE

SOFTPLAN, 2019



## ANTERIORMENTE EM...

- Discutimos a Coesão e o SRP
- Resolvemos o SRP e criamos um outro problema: o **Acoplamento**

```
public class CalculadoraDeBonus {  
  
    public double calcula(Funcionario funcionario) {  
        if(DESENVOLVEDOR.equals(funcionario.getCargo())) {  
            return new DezOuVintePorCento().calcula(funcionario);  
        }  
        if(DBA.equals(funcionario.getCargo()) ||  
            TESTER.equals(funcionario.getCargo())) {  
            return new QuinzeOuVinteCincoPorCento().calcula(funcionario);  
        }  
        throw new RuntimeException("Cargo invalido");  
    }  
}
```

# ACOPLAMENTO

- Quem nunca ouviu esta frase nas aulas de programação ?

“As classes precisam ser muito Coesas e pouco Acopladas!”

- Mas por que o acoplamento é algo ruim ?
  - Ou será que nem todo acoplamento é ruim ?

# NOSSO EXEMPLO DE ACOPLAMENTO

```
public class GeradorDeNotaFiscal {  
  
    private final EnviadorDeEmail email;  
    private final NotaFiscalDao dao;  
  
    public GeradorDeNotaFiscal(EnviadorDeEmail email, NotaFiscalDao dao) {  
        this.email = email;  
        this.dao = dao;  
    }  
  
    public NotaFiscal gerarNota(Fatura fatura) {  
        double valor = fatura.getValorMensal();  
  
        NotaFiscal nf = new NotaFiscal(valor, calculaImpostoSimples(valor));  
  
        email.enviaEmail(nf);  
        dao.persiste(nf);  
  
        return nf;  
    }  
  
    private double calculaImpostoSimples(double valor) {  
        return valor * 0.06;  
    }  
}
```

# QUAL O PROBLEMA DO NOSSO GeradorDeNotaFiscal ?

- Está acoplada ao EnviadoDeEmail e NotaFiscalDao
- E se amanhã, além de email passarmos a enviar SMS, WhatsApp, etc
- E se passarmos a gravar na nuvem, além do nosso banco local ?
- E se precisarmos consumir/enviar dados para outros sistemas ?
- E se ...

# MUDANÇAS E MAIS MUDANÇAS!!!!

- O problema do acoplamento está no impacto das mudanças.
- Nossos `GeradorDeNotaFiscal` parará de funcionar se:
  - O `EnviadorDeEmail` parar de funcionar
  - O `NotaFiscalDao` parar de funcionar
  - O `EnviadorDeEmail` evoluir e mudar os métodos
  - O `NotaFiscalDao` evoluir e mudar os métodos
- E quanto mais dependências, maior as chances de mudanças indesejadas serem propagadas.

OK.  
ENTÃO ADEUS DEPENDÊNCIAS!!

MAS ISSO É POSSÍVEL ?



SOFTPLAN, 2019

35

# SE TODO ACOPLAMENTO É RUIM, COMO FICA O REUSO ?

- Nem todo acoplamento é ruim. Eles são necessários
- O segredo é escolher bem com quem se acoplar
- Que tal List, String ou Boolean ? É confiável ?
- E aquela lib que seu amigo fez ontem ?

QUAL O SEGREDO DOS BONS ACOPLAMENTOS ?



# ESTABILIDADE É A RESPOSTA!



SOFTPLAN, 2019

38

# PERGUNTO?

- Quantas implementações diferentes existem da interface **List** do java ?
  - `AbstractList`, `AbstractSequentialList`, `ArrayList`, `AttributeList`, `CopyOnWriteArrayList`,  
`LinkedList`, `RoleList`, `RoleUnresolvedList`, `Stack`, `Vector`
- Você teria coragem de mudar a interface **List** na próxima versão do Java ?
- E a classe `EnviadorDeEmail` na próxima versão de sua Lib ?

# ENTÃO COMO TORNAR NOSSO EnviadoDeEmail ESTÁVEL ?

- Interfaces!
- Interfaces são como contratos. Ele tendem a ser estáveis!
- Interfaces são GENERALIZAÇÕES de problemas e permitem que várias soluções sejam construídas ao mesmo tempo que fornecem um meio único de trata-las

# REFATORANDO NOSSO EXEMPLO

- Estratégia: identificar o comportamento comum entre as ações que o `GeradorDeNotaFiscal` precisa executar
- Criar um contrato para que todas as ações sigam
- Criar ações que implementem o contrato
- Fazer com que o `GeradorDeNotaFiscal` conheça apenas nosso Contrato

# REFATORANDO NOSSO EXEMPLO

- Nosso contrato

```
public interface AcaoAposGerarNota {  
    void executa(NotaFiscal nf);  
}
```

# REFATORANDO NOSSO EXEMPLO

- Nossas ações

```
public class NotificarFinanceiro implements AcaoAposGerarNota {  
  
    @Override  
    public void executa(NotaFiscal nf) {  
        // Alguma lógica aqui  
    }  
}
```

```
public class NotificarAlmoxarifado implements AcaoAposGerarNota {  
  
    @Override  
    public void executa(NotaFiscal nf) {  
        // Alguma lógica aqui  
    }  
}
```

```
public class NotaFiscalDao implements AcaoAposGerarNota {  
  
    public void persiste(NotaFiscal nf) {  
        // Alguma lógica aqui  
    }  
  
    @Override  
    public void executa(NotaFiscal nf) {  
        persiste(nf);  
    }  
}
```

# REFATORANDO NOSSO EXEMPLO

- Antes

```
public class GeradorDeNotaFiscal {  
  
    private final EnviadorDeEmail email;  
    private final NotaFiscalDao dao;  
  
    public GeradorDeNotaFiscal(EnviadorDeEmail email, NotaFiscalDao dao) {  
        this.email = email;  
        this.dao = dao;  
    }  
  
    public NotaFiscal gerarNota(Fatura fatura) {  
        double valor = fatura.getValorMensal();  
  
        NotaFiscal nf = new NotaFiscal(valor, calculaImpostoSimples(valor));  
        email.enviaEmail(nf);  
        dao.persiste(nf);  
  
        return nf;  
    }  
}
```

# REFATORANDO NOSSO EXEMPLO

- Depois

```
public class GeradorDeNotaFiscal {  
  
    private final List<AcaoAposGerarNota> acoes;  
  
    public GeradorDeNotaFiscal(List<AcaoAposGerarNota> acoes) {  
        this.acoes = acoes;  
    }  
    public NotaFiscal gerarNota(Fatura fatura) {  
        double valor = fatura.getValorMensal();  
        NotaFiscal nf = new NotaFiscal(valor, calculaImpostoSimples(valor));  
  
        for(AcaoAposGerarNota acao : acoes) {  
            acao.executa(nf);  
        }  
        return nf;  
    }  
    private double calculaImpostoSimples(double valor) { return valor * 0.06; }  
}
```

```
public class GeradorDeNotaFiscal {  
  
    private final List<AcaoAposGerarNota> acoes;  
  
    public GeradorDeNotaFiscal(List<AcaoAposGerarNota> acoes) {  
        this.acoes = acoes;  
    }  
    public NotaFiscal gerarNota(Fatura fatura) {  
        double valor = fatura.getValorMensal();  
        NotaFiscal nf = new NotaFiscal(valor, calculaImpostoSimples(valor));  
  
        for(AcaoAposGerarNota acao : acoes) {  
            acao.executa(nf);  
        }  
        return nf;  
    }  
    private double calculaImpostoSimples(double valor) { return valor * 0.06; }  
}
```



# DIP DEPENDENCY INVERSION PRINCIPLE

*DEPEND ON ABSTRACTIONS, NOT ON CONCRETIONS*

SOFTPLAN, 2019

# DIP - DEPENDENCY INVERSION PRINCIPLE

- Sempre que uma classe for depender de outra, ela deve depender sempre de outro classe mais estável do que ela mesma.
  - Interfaces > Implementações
- Este princípio tem tudo a ver com estabilidade. E como vimos: sempre que uma classe for depender de outra, ela deve depender sempre de outro módulo mais estável do que ela mesma

# IMPLEMENTANDO ABSTRAÇÕES

- Módulos de alto nível não podem depender de módulos de baixo nível.  
Ambos devem depender de abstrações.
- Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

# DIP - DEPENDENCY INVERSION PRINCIPLE

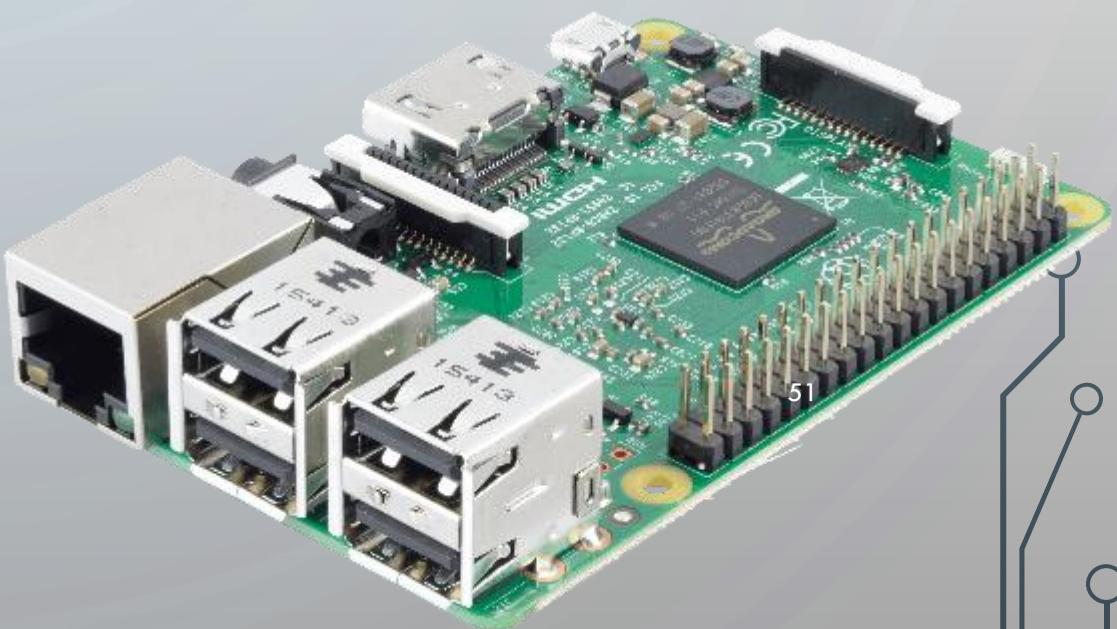
TKS!

SOFTPLAN, 2019



# OCP OPEN/CLOSED PRINCIPLE

SOFTPLAN, 2019





## ANTERIORMENTE EM ...

- Até aqui, discutimos bastante sobre **coesão** e **acoplamento**. Entendemos que classes não coesas devem ter suas responsabilidades divididas em pequenas classes, e que classes devem tentar ao máximo se acoplar com classes que são **estáveis**.
- Agora estamos prontos para fazer com que nossos sistemas evoluam!

# EVOLUIR SEMPRE!

SE HÁ UMA CERTEZA, É QUE OS SISTEMAS SEMPRE MUDAM.

E SÃO ATUALIZADOS

E SÃO EVOLUÍDOS .....

SOFTPLAN, 2019

53



OU SEJA

EVOLUIR PRECISA SER ALGO NATURAL

SOFTPLAN, 2019

54

# PARA EVOLUIR CÓDIGO MAL FEITO

- Precisamos de Ctrl+F/grep
- Precisamos de muito Debug
- Precisamos mudar, testar, mudar mais um pouco, testar
- Perdemos um expediente de trabalho
- Perguntamos para o colega mais experiente o que está faltando
- .....

# NOSSO EXEMPLO DE CÓDIGO

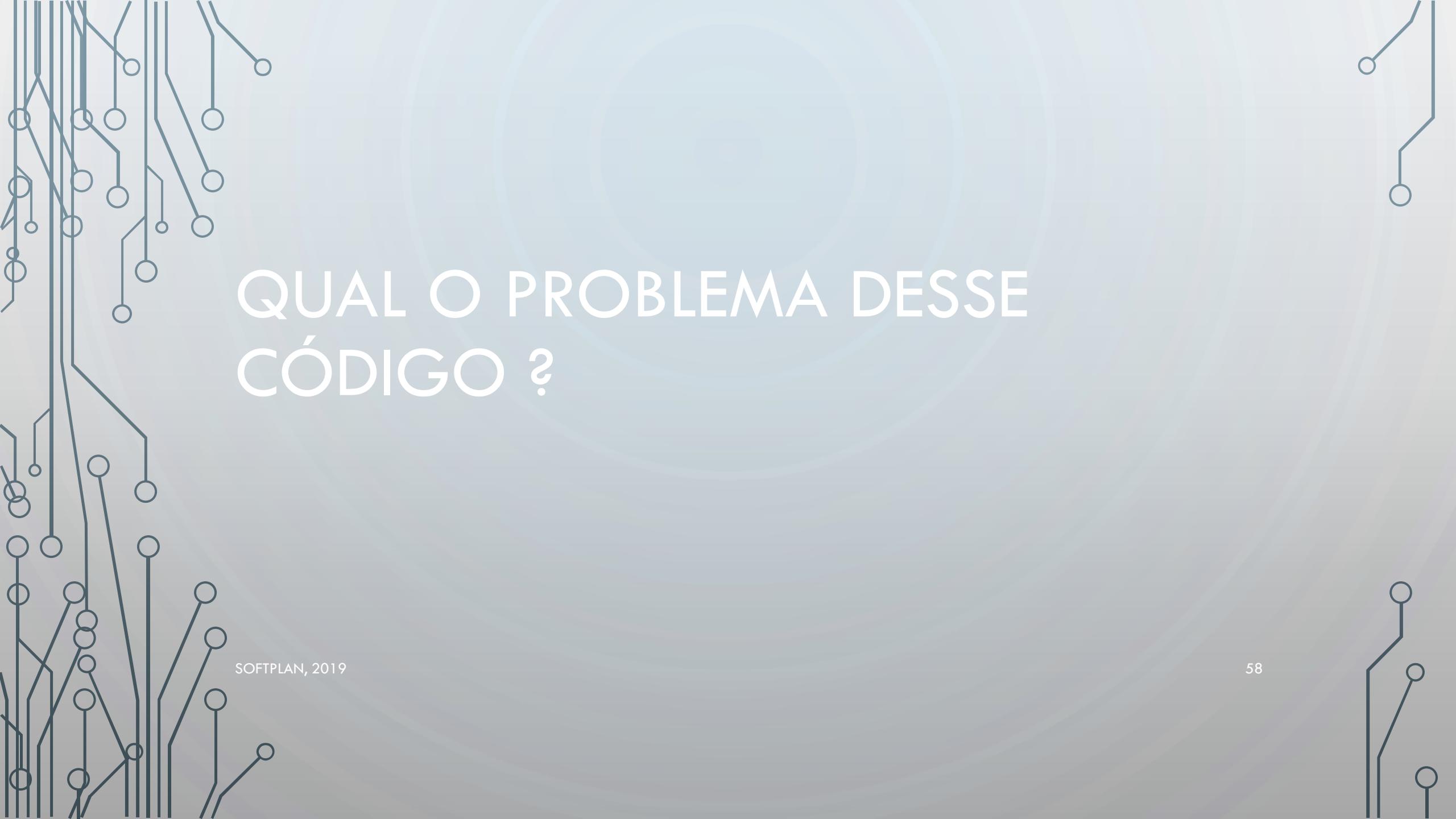
```
public class CalculadoraDePrecos {  
  
    public double calcula(Produto produto) {  
        TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();  
        Frete correios = new Frete();  
  
        double desconto = tabela.descontoPara(produto.getValor());  
  
        double frete = correios.para(produto.getCidade());  
        return produto.getValor() * (1 - desconto) + frete;  
    }  
}
```

```
public class Frete {  
    public double para(Cidade cidade) {  
        if("SP".equals(cidade.getCodigo())) {  
            return 15;  
        }  
        return 30;  
    }  
}
```

```
public class TabelaDePrecoPadrao {  
    public double descontoPara(double valor) {  
        if (valor > 5000) return 0.03;  
        if (valor > 1000) return 0.05;  
        return 0;  
    }  
}
```

# ANÁLISE DO NOSSO EXEMPLO

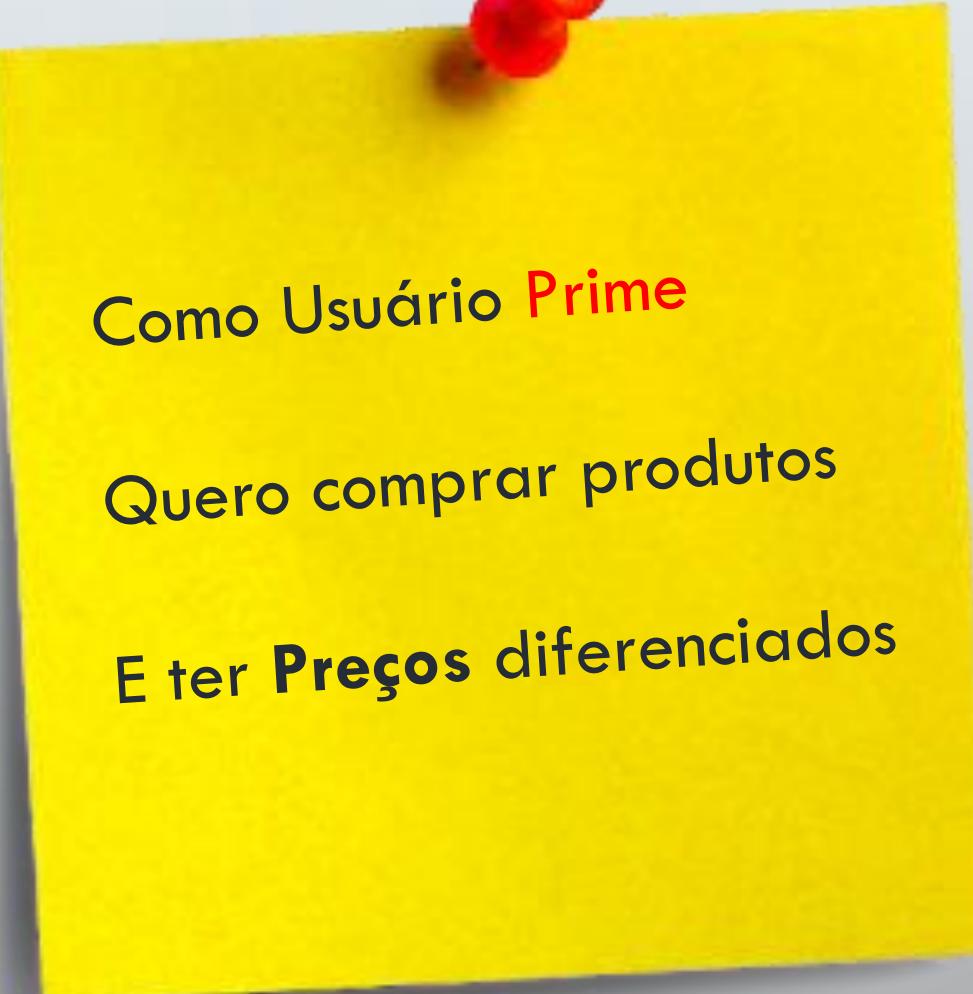
- É simples ?
- É Coeso ?
- O acoplamento está ruim ?
- O código está muito Grep Oriented ?



# QUAL O PROBLEMA DESSE CÓDIGO ?

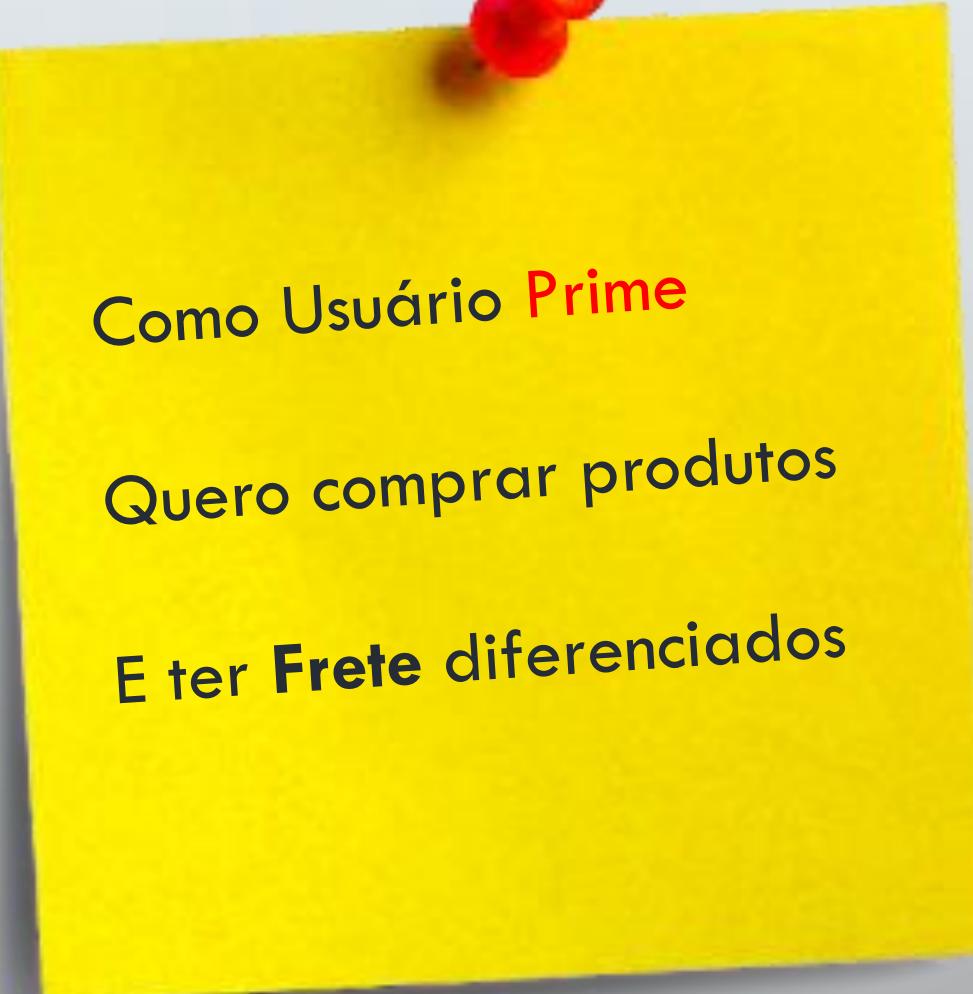
SOFTPLAN, 2019

58



Como Usuário Prime  
Quero comprar produtos  
E ter Preços diferenciados

```
public class CalculadoraDePrecos {  
  
    public double calcula(Produto produto, Regra regra) {  
  
        Frete correios = new Frete();  
  
        double desconto = 0;  
        if (Regra.REGRA1.equals(regra)) {  
            TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();  
            desconto = tabela.descontoPara(produto.getValor());  
        }  
        if (Regra.REGRA2.equals(regra)) {  
            TabelaDePrecoDiferenciada tabela = new TabelaDePrecoDiferenciada();  
            desconto = tabela.descontoPara(produto.getValor());  
        }  
        // .....  
        double frete = correios.para(produto.getCidade());  
        return produto.getValor() * (1 - desconto) + frete;  
    }  
}
```



Como Usuário Prime

Quero comprar produtos

E ter Frete diferenciados

```
public class Frete {  
    public double para(Cidade cidade, Regra regra) {  
  
        if (Regra.REGRA1.equals(regra)) {  
  
            if ("SP".equals(cidade.getCodigo())) {  
                return 15;  
            }  
            return 30;  
  
        } else if (Regra.REGRA2.equals(regra)) {  
            // .....  
        } else if (Regra.REGRA3.equals(regra)) {  
            // .....  
        } else if (Regra.REGRA4.equals(regra)) {  
            // .....  
        }  
        return 50;  
    }  
}
```

# NOSSA SOLUÇÃO JÁ NÃO PARECE TÃO BOA

- Nossas classes ficarão complicadas. Serão muitos Ifs
- Ficará difícil testar
- Perderemos em coesão
- A cada PostIt precisaremos **abrir** as classes para adicionar a REGRA **X**
- A cada nova regra, poderão ser necessários novos parâmetros

# OCP OPEN/CLOSED PRINCIPLE

YOU SHOULD BE ABLE TO EXTEND A CLASSES BEHAVIOR, WITHOUT  
MODIFYING IT

SOFTPLAN, 2019

64

# EVOLUIR PRECISA SER FÁCIL

- Escrever IFs pode ser uma solução ruim
- As mudanças devem ser propagadas automaticamente, sem Ifs
- As classes devem estar abertas para extensão e fechadas para a modificação, ou seja, queremos poder estender o seu **comportamento** sem a necessidade de **modificá-las**

## NO NOSSO EXEMPLO

- Como criar ou modificar as regras de **Frete** sem precisar alterar a classe **Frete sempre?**
- Como criar ou modificar as regras de **Preço** sem precisar alterar a classe **CalculadoraDePrecos sempre?**

# ABSTRAIR

Este é o primeiro passo

# NOSSAS ABSTRAÇÕES

```
public interface ServicoDeEntrega {  
    double para(Cidade cidade);  
}
```

```
public interface TabelaDePreco {  
    double descontoPara(double valor);  
}
```

# DESSA FORMA FICOU FÁCIL CRIAR NOVAS REGRAS

```
public interface TabelaDePreco {  
    double descontoPara(double valor);  
}
```

```
public class TabelaDePreco1 implements TabelaDePreco {  
    @Override  
    public double descontoPara(double valor) {  
        return 0;  
    }  
}  
  
public class TabelaDePreco2 implements TabelaDePreco {  
    @Override  
    public double descontoPara(double valor) {  
        return 0;  
    }  
}  
  
public class TabelaDePreco3 implements TabelaDePreco {  
    @Override  
    public double descontoPara(double valor) {  
        return 0;  
    }  
}
```

```
public interface ServicoDeEntrega {  
    double para(Cidade cidade);  
}
```

```
public class Frete1 implements ServicoDeEntrega {  
    @Override  
    public double para(Cidade cidade) {  
        return 0;  
    }  
}  
  
public class Frete2 implements ServicoDeEntrega {  
    @Override  
    public double para(Cidade cidade) {  
        return 0;  
    }  
}  
  
public class Frete3 implements ServicoDeEntrega {  
    @Override  
    public double para(Cidade cidade) {  
        return 0;  
    }  
}
```

# COMO FAZER NOSSA CALCULADORA UTILIZAR AS ABSTRAÇÕES ?

- A principal mudança será: A CalculadoraDePrecos receberá o ServicoDeEntrega e a TabelaDePreco pelo construtor ao invés de instâncias.
- Dessa forma poderemos **alterar o comportamento** da CalculadorDePrecos **sem a necessidade** de qualquer modificação nela
- Ou seja: A classe CalculadorDePrecos estará aberta para a extensão do seu comportamento e fechada para modificações da sua estrutura.

```
public class CalculadoraDePrecos {  
  
    private TabelaDePreco tabela;  
    private ServicoDeEntrega entrega;  
  
    public CalculadoraDePrecos(TabelaDePreco tabela, ServicoDeEntrega entrega) {  
        this.tabela = tabela;  
        this.entrega = entrega;  
    }  
  
    public double calcula(Produto produto) {  
        //TabelaDePrecoPadrao tabela = new TabelaDePrecoPadrao();  
        //Frete correios = new Frete();  
  
        double desconto = tabela.descontoPara(produto.getValor());  
  
        double frete = entrega.para(produto.getCidade());  
        return produto.getValor() * (1 - desconto) + frete;  
    }  
}
```

# RESULTADO

- ✓ SRP
- ✓ DIP
- ✓ Simples
- ✓ Testável
- ✓ Fácil de evoluir
- ✓ Não Grep Oriented ou Debug Oriented

SOFTPLAN, 2019

# OPC OPEN/CLOSED PRINCIPLE

TKS!

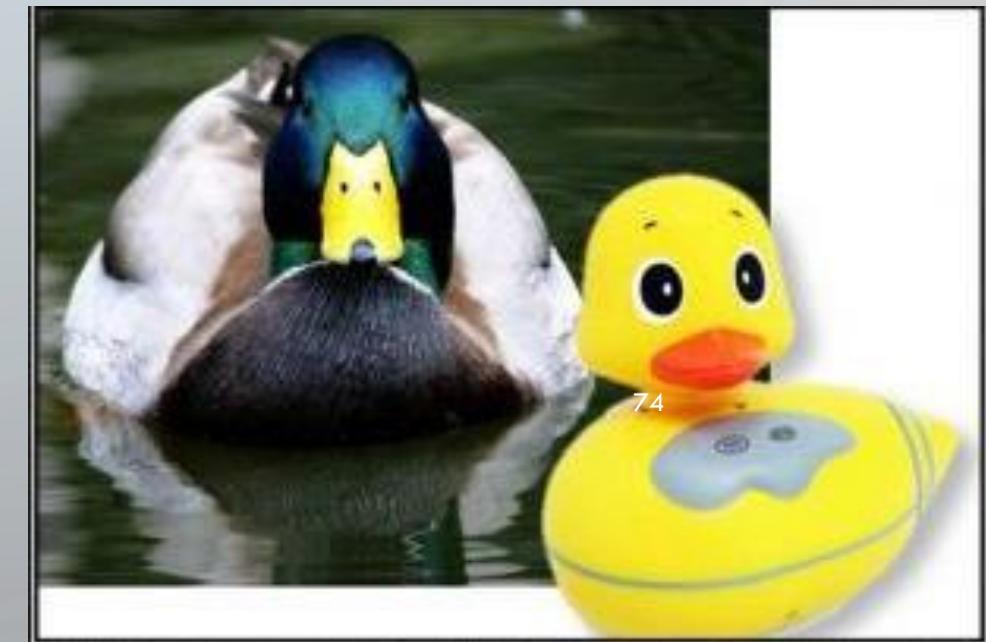
SOFTPLAN, 2019



# LSP

# LISKOV SUBSTITUTION PRINCIPLE

SOFTPLAN, 2019



## ANTERIORMENTE EM ...

- Coesão: criamos uma grande quantidade de classes e cada uma ficou com a sua responsabilidade bem definida
- Encapsulamento: colocamos o código onde ele precisa estar e garantimos que cada classe conheça apenas o necessário da demais, sem intimidades inapropriadas
- Acoplamento: organizamos a relação entre essas classes e agora cada uma se relaciona e sobre influência apenas das classes realmente necessárias
- Abstrações: definimos os contratos e ganhamos segurança e flexibilidade

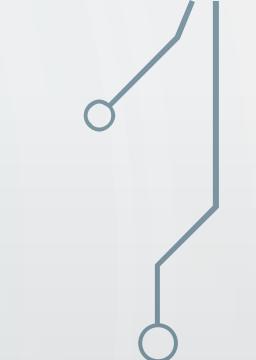
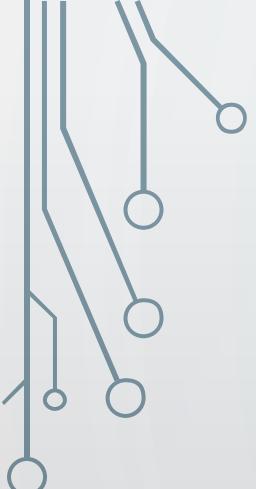
# BASTANTE CÓDIGO CRIADO

- Muitas classes foram criadas
- Muitas interfaces foram criadas
- .....
- Nenhuma herança foi criada ?

# BASTANTE CÓDIGO CRIADO

- Muitas classes foram criadas
- Muitas interfaces foram criadas
- .....
- Nenhuma herança foi criada ?





PQ ?

- Heranças são coisas delicadas
- São fonte as das maiores armadilhas da OO
- **Já se deparou com uma hierarquia de classes grande e confusa ?**
  - Uma maravilha, não ....

# VAMOS AO NOSSO PRIMEIRO EXEMPLO

```
public class ContaComum {  
    protected double saldo;  
  
    public ContaComum() {  
        this.saldo = 0;  
    }  
  
    public void deposita(double valor) throws ValorInvalidoException {  
        if (valor <= 0)  
            throw new ValorInvalidoException();  
        this.saldo += valor;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
  
    public void rende() {  
        this.saldo *= 1.1;  
    }  
}
```



SIMPLES, NÃO ? SEM PROBLEMAS.

SOFTPLAN, 2019

80

AGORA VAMOS IMPLEMENTAR A CONTA DE ESTUDANTE. ELA NÃO RENDE()

```
public class ContaDeEstudante extends ContaComum {  
    public void rende() {  
        this.saldo *= 1.0;  
    }  
}
```

# OU TALVEZ ISSO

```
public class ContaDeEstudante extends ContaComum {  
    public void rende() throws ContaNaoRendeException {  
        throw new ContaNaoRendeException();  
    }  
}
```



# OK. RESOLVE O PROBLEMA

APARENTEMENTE .....

SOFTPLAN, 2019

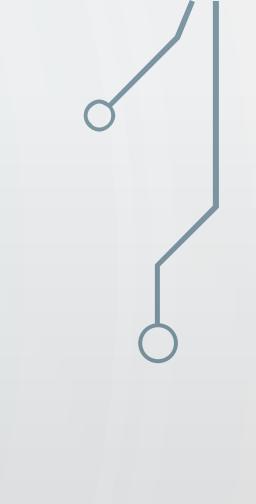
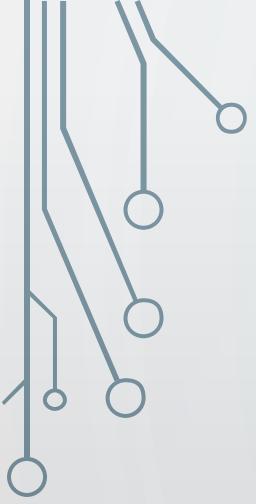
83

MAS ESSA  
**ContaNaoRendeException** ?  
ESTAVA PREVISTA NA CLASSE PAI ?

NÃO ESTAVA, MAS É APENAS UMA EXCEPTION, CERTO ?

# AGORA IMAGINE O SEGUINTE

```
public class ProcessadorDeInvestimentos {  
    Run | Debug  
    public static void main(String[] args) {  
        for (ContaComum conta : contasDoBanco()) {  
            conta.rende();  
            System.out.println("Novo Saldo:");  
            System.out.println(conta.getSaldo());  
        }  
  
        private static List<ContaComum> contasDoBanco() {  
            return Arrays.asList(new ContaComum(), new ContaDeEstudante());  
        }  
    } SOFTPLAN, 2019
```



E ENTÃO

- Compila ?
- Funciona ?
  - Sim
  - Não
  - Depende

# QUAL O PROBLEMA ?

- A classe filha não respeitou **contrato** definido pela classe pai
- A classe ProcessadorDeInvestimentos recebeu um comportamento não esperado
- A execução parou por causa da exceção
  - Try catch ? KKKKK

# LSP

## LISKOV SUBSTITUTION PRINCIPLE

*DERIVED CLASSES MUST BE SUBSTITUTABLE FOR THEIR BASE CLASSES*

OU

*SE S É UM SUBTIPO DE T, ENTÃO OS OBJETOS DO TIPO T, EM UM PROGRAMA, PODEM SER SUBSTITUÍDOS PELOS OBJETOS DE TIPO S SEM QUE SEJA NECESSÁRIO ALTERAR AS PROPRIEDADES DESTE PROGRAMA.—*

SOFTPLAN, 2019

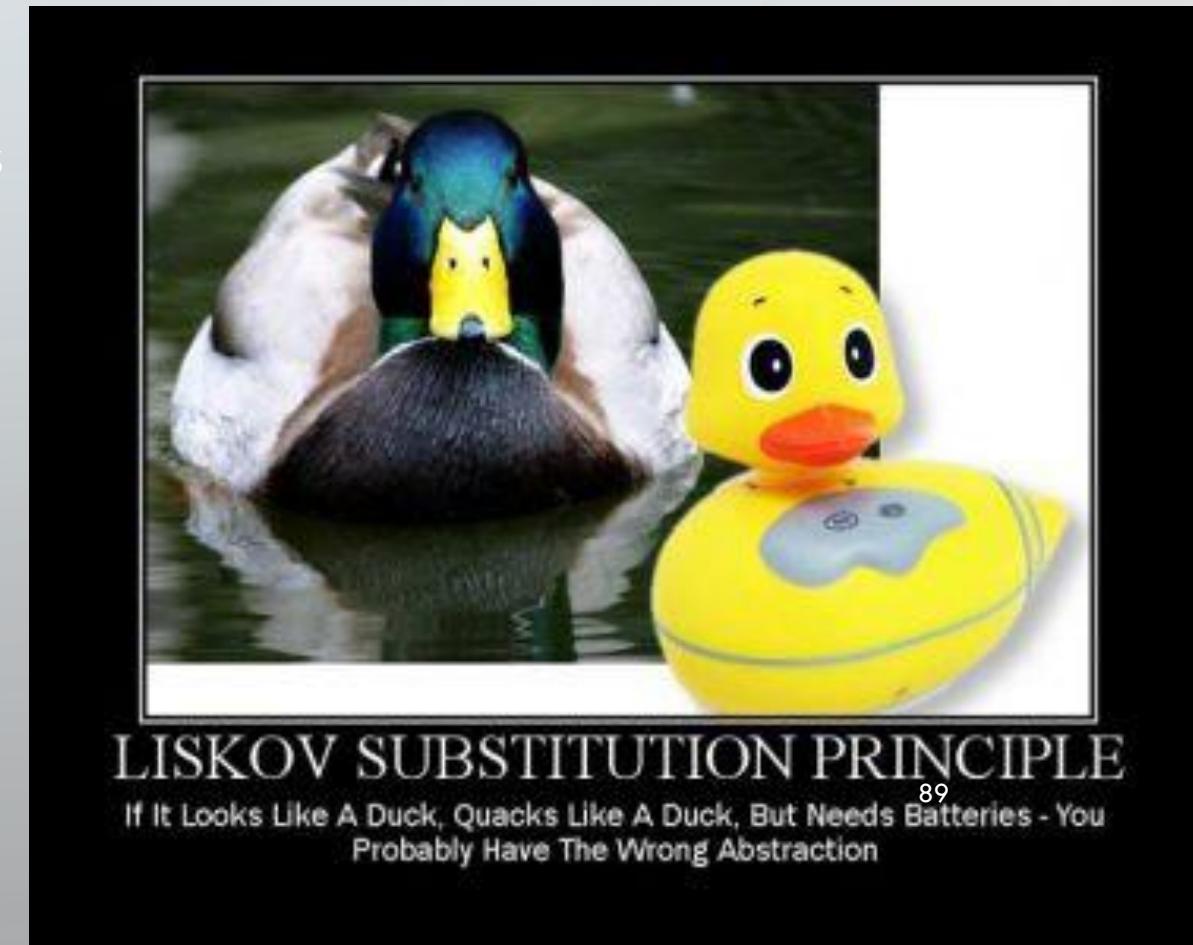
[WIKIPEDIA.](#)

88

# LSP - LISKOV SUBSTITUTION PRINCIPLE

- Criado por Barbara Liskov
- Nos alerta para os perigos das heranças
- Determina que as classes Base devem poder substituir suas classes filhas sem surpresas
- Os contratos entre pais e filhos definem mais que assinaturas de métodos
  - Valores aceitos

SOFTPLAN, 2018  
Comportamentos



## UMA REGRA INTERESSANTE

“A classe filho só pode afrouxar as pré-condições  
e apertar as pós-condições.”

Maurício Aniche

SOFTPLAN, 2019

# PRÉ E PÓS CONDIÇÕES

- Contratos são mais que assinaturas
- Definem também condições de utilização. Comportamentos esperados.
- Imagine o problema se:
  - `deposita()`
    - Na classe Pai, aceita inteiros maiores que 1
    - Imagine se uma classe filha alterar isso para inteiro > 10!
  - `rende()`
    - Na classe Pai, não é prevista uma exceção
    - Imagine uma classe filha lançando exceções !

# PRÉ E PÓS CONDIÇÕES

- Por outro lado
  - Pré condições podem ser afrouxadas
    - Na classe Pai, aceita inteiros maiores que  $1 < 100$
    - Uma classe filha pode alterar isso  $1 < 200$
  - Pós condições poderão ser apertadas
    - Na classe Pai, aceita inteiros maiores que  $1 < 100$
    - Uma classe filha pode alterar isso  $1 < 50$

# UM EXEMPLO MAIS DIDÁTICO

```
public class Retangulo {  
  
    private int x;  
    private int y;  
  
    public Retangulo(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
}  
SOFTPLAN, 2019
```

```
public class Quadrado extends Retangulo {  
  
    private int x;  
    private int y;  
  
    public Quadrado(int x) {  
        super(x, x);  
    }  
}
```

# UM QUADRADO É UM RETÂNGULO DE ALTURA IGUAL À LARGURA ?

- Perceba que a precondição da classe Quadrado é mais forte que a da classe filho. Em um quadrado, ambos os lados precisam ser iguais. Em um retângulo, não.
- Está é uma situação em que a herança não é possível.
- Ao utilizarmos heranças desse tipo, teremos comportamentos indesejados

# QUE TAL COMPOSIÇÃO ?

- Para utilizar o método concat() de da classe String, é necessário herdar de String?
- Então herança é não é a única compartilhar comportamentos, ok ?
- “*Em ciência da computação, composição de objetos (não confundir com composição de funções) é uma maneira de se combinar objetos simples ou tipos de dados em objetos mais complexos.*” Wikipedia
- Se a afirmação: Minha classe A É UM B não faz sentido, então composição provavelmente é uma opção melhor que herança
- Tente: Minha classe A TEM UM B ou minha classe A FAZ USO DE B

# UTILIZANDO COMPOSIÇÃO

```
public class ManipuladorDeSaldo {  
    private double saldo;  
  
    public void adiciona(double valor) {  
    }  
  
    public void retira(double valor) {  
    }  
  
    public void juros(double taxa) {  
    }  
  
    public double getSaldo() {  
        return 0;  
    }  
}
```

# UTILIZANDO COMPOSIÇÃO

```
public class ContaComum {  
  
    private ManipuladorDeSaldo manipulador;  
  
    public ContaComum() {  
        this.manipulador = new ManipuladorDeSaldo();  
    }  
    public void saca(double valor) {  
        manipulador.adiciona(valor);  
    }  
  
    public void rende() {  
        manipulador.juros(0.1);  
    }  
}
```

```
public class ContaDeEstudante {  
  
    private ManipuladorDeSaldo manipulador;  
  
    public ContaDeEstudante() {  
        this.manipulador = new ManipuladorDeSaldo();  
    }  
  
    public void saca(double valor) {  
        manipulador.adiciona(valor);  
    }  
}
```

# CONCLUINDO

- Herança deve ser usada com muito cuidado
- Se a relação é X É UM Y, então ok, pode ser herança
  - Gato é um animal
  - Professor é um Funcionário
- Se a relação é X tem um Y ou X faz uso de Y, melhor usar composição
  - Um Carro tem um Motor
  - Uma Conta tem um saldo

# LSP

# LISKOV SUBSTITUTION PRINCIPLE

TKS!

SOFTPLAN, 2019



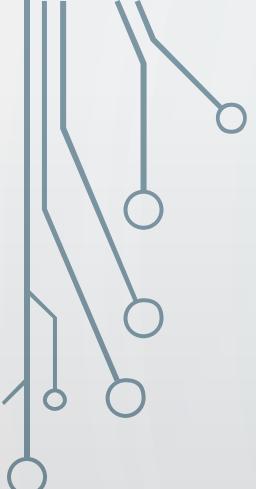


# ISP

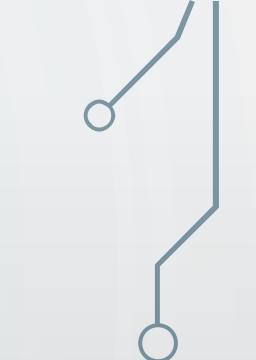
# INTERFACE SEGREGATION PRINCIPLE

SOFTPLAN, 2019

100



## ANTERIORMENTE EM ...

- Discutimos muito sobre coesão de **classes**.
  - Criamos e organizamos nossas **classes**.
  - E quanto as **interfaces** ?
    - Elas não precisam ser coesas e organizadas ?
- 

# VAMOS AO PRIMEIRO EXEMPLO

```
public interface Imposto {  
    NotaFiscal geraNota();  
    double imposto(double valorCheio);  
}
```

# AGORA VAMOS IMPLEMENTAR O ISS

- Imposto sobre 10% do valor da nota e então gera a NF

```
public class ISS implements Imposto {  
  
    public double imposto(double valorCheio) {  
        return 0.1 * valorCheio;  
    }  
  
    public NotaFiscal geraNota(String client) {  
        return new NotaFiscal(/*magicamente*/);  
    }  
}
```

# AGORA VAMOS IMPLEMENTAR ISCMF

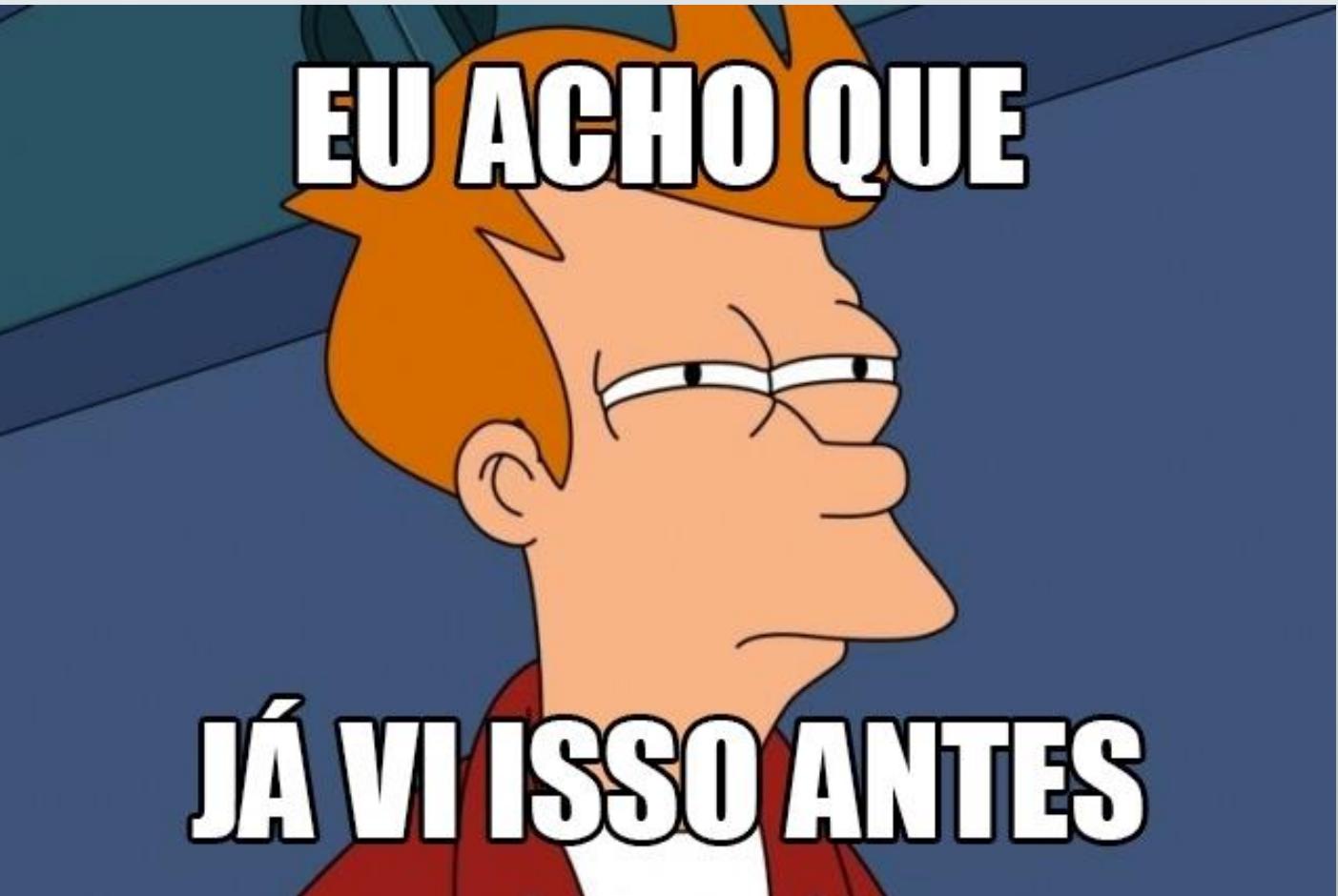
- Imposto sobre 50% do valor da nota e NÃO gera a NF

```
public class ISCMS implements Imposto {  
  
    public double imposto(double valorCheio) {  
        return 0.5 * valorCheio;  
    }  
  
    public NotaFiscal geraNota() {  
        return null;  
    }  
}
```

# AGORA VAMOS IMPLEMENTAR ISCMF

- Ou então....

```
public class ISCMF implements Imposto {  
  
    public double imposto(double valorCheio) {  
        return 0.5 * valorCheio;  
    }  
  
    public NotaFiscal geraNota() {  
        throw new NaoGeraNotaException();  
    }  
}
```





ENTÃO, ESTE CÓDIGO ESTÁ OK ?

SOFTPLAN, 2019

107



# A INTERFACE NÃO ESTÁ COESA!

ENTRE OUTRAS COISAS

SOFTPLAN, 2019

108

# INTERFACES COESAS

- Já discutimos bastante sobre coesão. Então que tal isso ?

```
public interface CalculadorDeImposto {  
    double imposto(double valorCheio);  
}
```

```
public interface GeradorDeNota {  
    NotaFiscal geraNota();  
}
```

# VANTAGEM DAS INTERFACES COESAS

```
public class ISS implements CalculadorDeImposto, GeradorDeNota {  
    public double imposto(double valorCheio) {  
        return 0.1 * valorCheio;  
    }  
  
    public NotaFiscal geraNota() {  
        return new NotaFiscal(/*magicamente*/);  
    }  
}
```

```
public class ISCMSS implements CalculadorDeImposto {  
    public double imposto(double valorCheio) {  
        return 0.5 * valorCheio;  
    }  
}
```

# COESÃO

ORGANIZAÇÃO, REUSO, FLEXIBILIDADE, ESTABILIDADE, .....

SOFTPLAN, 2019

111

# UM OUTRO EXEMPLO PARA FICAR MAIS CLARO

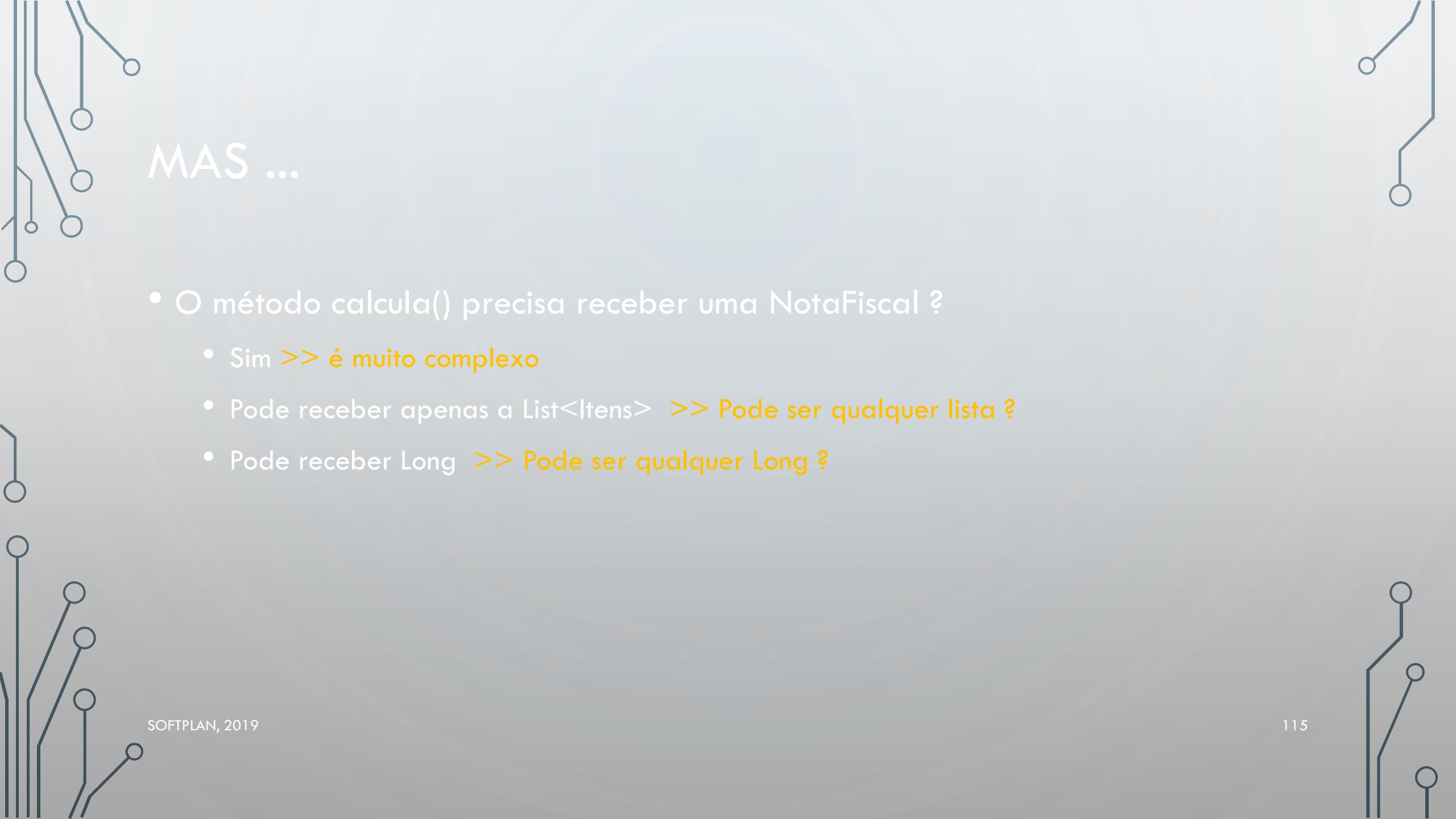
```
public class NotaFiscal {  
  
    public NotaFiscal(  
        Cliente cliente,  
        List<Item> itens,  
        List<Desconto> descontos,  
        Endereco entrega,  
        Endereco cobranca,  
        FormaDePagamento pago,  
        double valorTotal  
    ) { }  
}
```

# AGORA UM CLIENTE QUE CALCULA O VALOR DA NOTA

```
public class CalculadorDeImposto {  
  
    public double calcula(NotaFiscal nf) {  
        double total = 0;  
        for(Item item : nf.getItens()) {  
            if(item.getValor()>1000)  
                total+= item.getValor() * 0.02;  
            else  
                total+= item.getValor() * 0.01;  
        }  
        return total;  
    }  
}
```

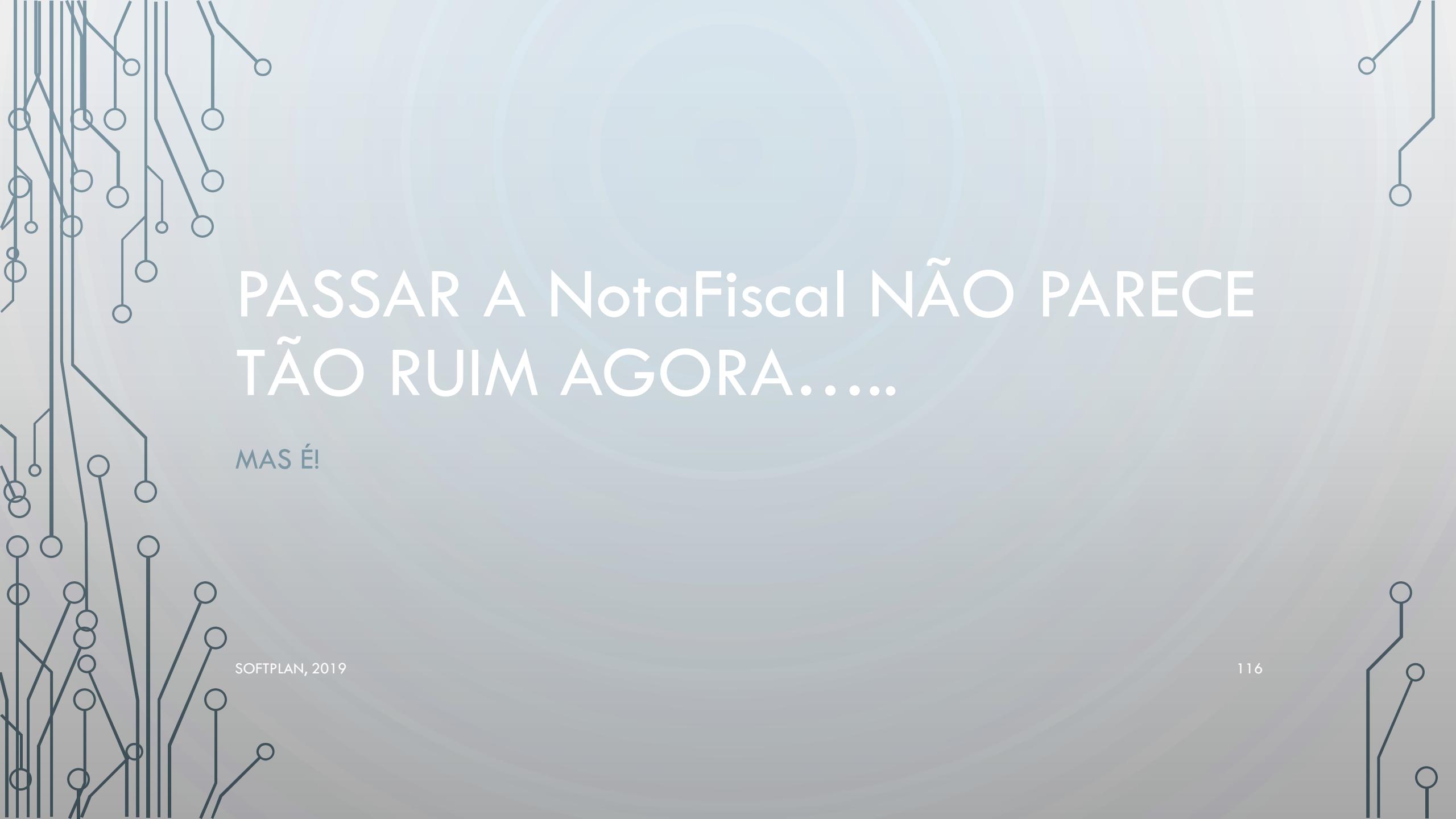
## MAS.....

- O método `calcula()` precisa receber uma `NotaFiscal` ?
  - ✓ Sim
  - ✓ Pode receber apenas a `List<Itens>`
  - ✓ Pode receber `Long`
- Não esqueçamos do acoplamento.....
- Apenas uma lista fica mais simples, e ficamos imunes as alterações da `NotaFiscal`
- Apenas um `Long` fica mais simples ainda, e ficamos imunes as alterações da `NotaFiscal` e não precisamos entender como navegar nos itens



MAS ...

- O método `calcula()` precisa receber uma `NotaFiscal` ?
  - Sim >> é muito complexo
  - Pode receber apenas a `List<Itens>` >> Pode ser qualquer lista ?
  - Pode receber `Long` >> Pode ser qualquer `Long` ?



PASSAR A NotaFiscal NÃO PARECE  
TÃO RUIM AGORA.....

MAS É!

SOFTPLAN, 2019

116

# QUE TAL

```
public interface Tributavel {  
    List<Item> itensASeremTributados();  
}
```

```
public class NotaFiscal implements Tributavel {  
  
    public NotaFiscal(  
        Cliente cliente,  
        List<Item> itens,  
        List<Desconto> descontos,  
        Endereco entrega,  
        Endereco cobranca,  
        FormaDePagamento pagto,  
        double valorTotal  
    ) {  
        // ...  
    }  
    public List<Item> itensASeremTributados() {  
        return Collections.singletonList(new Item());  
    }  
}
```

```
public class CalculadorDeImposto {  
  
    public double calcula(Tributavel nf) {  
        double total = 0;  
        for (Item item : nf.itensASeremTributados()) {  
            if (item.getValor() > 1000)  
                total += item.getValor() * 0.02;  
            else  
                total += item.getValor() * 0.01;  
        }  
        return total;  
    }  
}
```

# QUE TAL AGORA ?

- Temos uma interface (e ela é estável)
- Nossa interface é específica
  - Tem uma finalidade específica e apenas os interessados implementarão
- Diminuímos o acoplamento
  - CalculadorDelImpostos não depende de NotaFiscal, que é instável, mas sim de Tributavel, que é estável
- Fica muito claro o que o método `calcula()` precisa receber
- Podemos delegar para a NotaFiscal a tarefa de reunir os itens da forma correta, e ninguém melhor que ela



# ISP

# INTERFACE SEGREGATION PRINCIPLE

TKS!

SOFTPLAN, 2019



# ISP - INTERFACE SEGREGATION PRINCIPLE

- Muitas interfaces específicas são melhores do que uma interface única geral.
- Clientes não devem ser forçados a depender de interfaces que eles não usam.
- Não devemos obrigar os clientes a implementar comportamentos dos quais eles não necessitam

TKS

THIAGO.SOARES@SOFTPLAN.COM.BR

SOFTPLAN, 2019

121



# CREDITOS

- Fontes:

- <https://www.casadocodigo.com.br/products/livro-oo-solid>
- <https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmltNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/view>
- <https://medium.com/thiago-aragao/solid-princ%C3%ADpios-da-programa%C3%A7%C3%A3o-orientada-a-objetos-ba7e31d8fb25>
- <https://medium.com>equals-lab/princ%C3%ADpios-s-o-l-i-d-o-que-s%C3%A3o-e-porque-projetos-devem-utiliz%C3%A1-los-bf496b82b299>
- <https://www.lambda3.com.br/2019/01/lambda3-podcast-126-principios-solid/>

- Imagens:

- <http://profpv.blogspot.com/2013/04/arquitetura-solid.html>