SOLID e Clean Code

PARTE I

SOLID

# Introdução

* Contexto Teórico:

**Objetivo geral ->** tornar o *design de software* mais:

* + “Entendível”
  + Flexível
  + Fácil manutenção

# Introdução

### Contextualização realista -> Observações da comunidade

*“Os princípios ajudam muito devs Jr. a entender o que dev experiente aplica ao escrever código...”*

*“É extremamente útil em cenários de complexidade e evolução.* *”*

*“Funciona melhor quando adaptado ao contexto da equipe, do projeto e do produto”*

# Introdução

* Contextualização realista -> Observações da comunidade

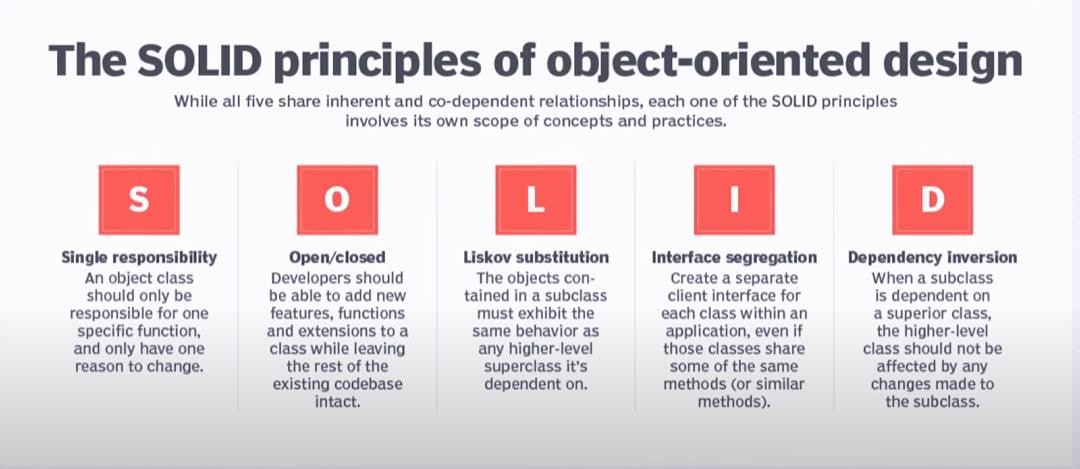
*“Aplicar rigidamente pode gerar estruturas*

*desnecessárias em contextos simples”*

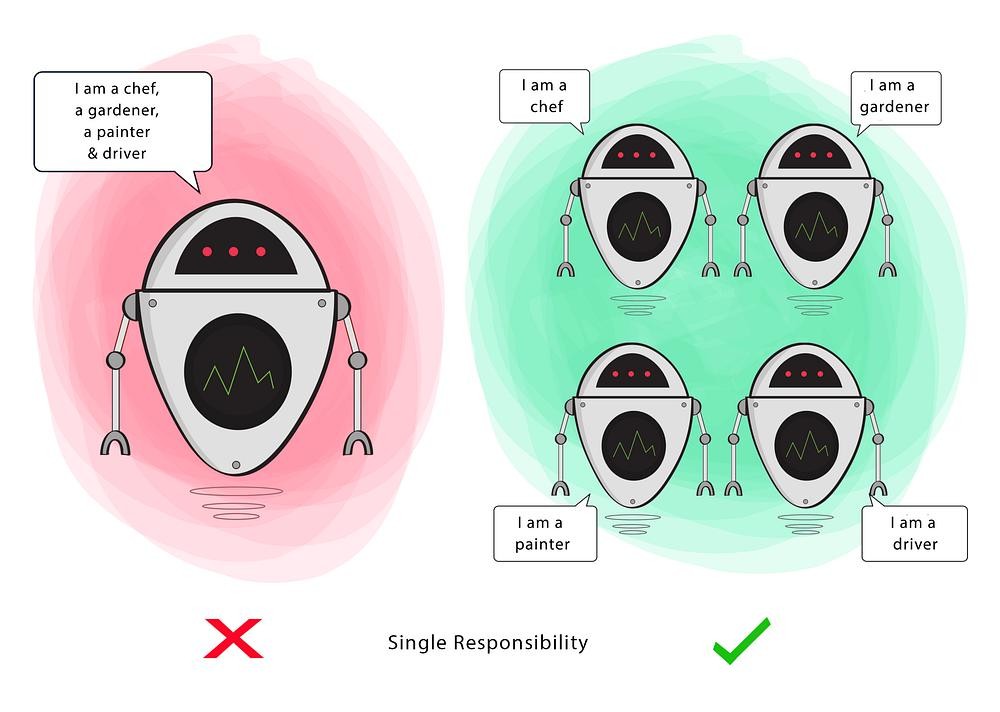
*“Quando o SOLID vira dogma, o time perde capacidade de avaliar trade-offs... o ideal é aplicá-los com bom senso.”*

* Resumo: os princípios são guias ou diretrizes MAS não mandamentos absolutos MAS ampliam a nossa percepção “das coisas”.

# S.O.L.I.D.



## S (Single Responsibility Principle): *SRP*

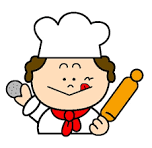


* Cada classe deve ter apenas uma simples responsabilidade

**Qual a ideia?**

Separar comportamentos para que, se surgirem bugs como resultado da sua mudança, eles não afetem outros comportamentos não relacionados.

## S (Single Responsibility Principle):



### Analogia: Um restaurante

* Imagine que há um cozinheiro que prepara pratos, atende pedidos e ainda lava a louça.
* Ele está sobrecarregado e tudo pode dar errado.
* Se cada função for entregue a um especialista — cozinheiro, garçom, faxineiro — tudo flui melhor.

## S (Single Responsibility Principle):

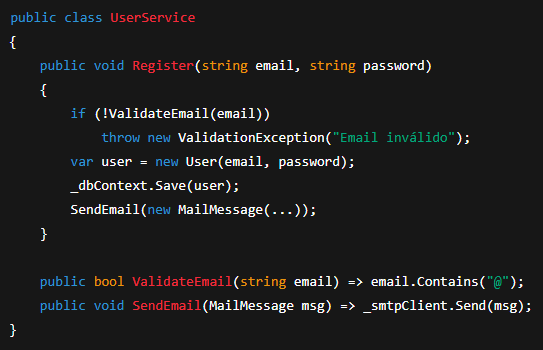
### Exemplos Práticos

* 1) Uma classe que calcula salário e gera relatório — deve ser dividida em duas responsabilidades distintas
* 2) Em vez de uma Classe *EmailManager* que tem tudo: cria o corpo

da mensagem, envia o e-mail, cuida da conexão SMTP.

* *EmailCompositor* (responsável apenas por criar o e-mail)
* *EmailSender* (responsável por enviá-lo)
* *EmailManager* à coordenação dessas partes

## S (Single Responsibility Principle):

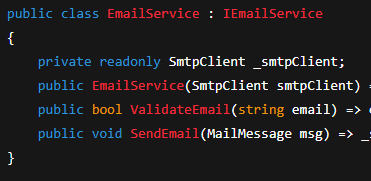
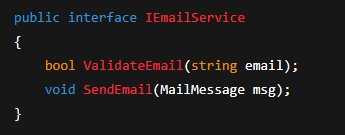
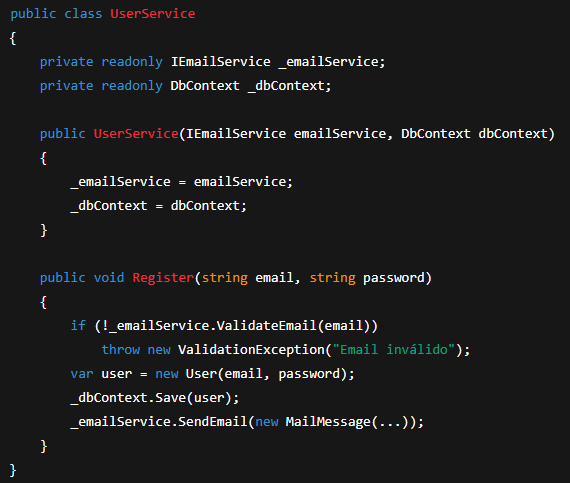


* Código “*Problema*” **O que tá pegando aqui?**

Cada classe como um especialista em uma única tarefa

## S (Single Responsibility Principle):

* Código *Solução* -> aplicando SRP:



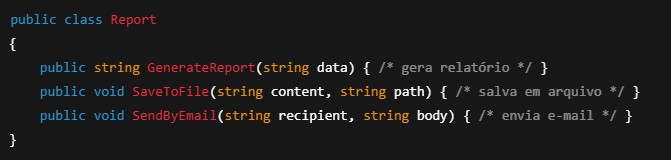
**Funções distintas agora estão**

**bem isoladas**

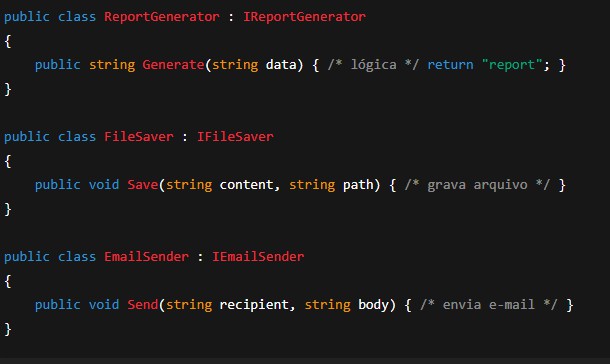
## S (Single Responsibility Principle):

* Código “*Problema*”

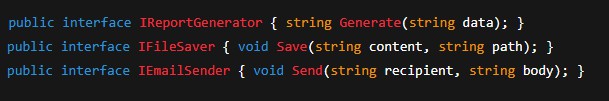
**O que tá pegando aqui?**



## S (Single Responsibility Principle):



* Código *Solução* -> aplicando SRP:

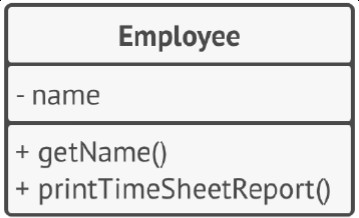


**Agora cada classe tem uma única responsabilidade, facilitando manutenção e testes**

## S (Single Responsibility Principle):

**O que tá pegando aqui?** O formato do relatório pode mudar com o tempo

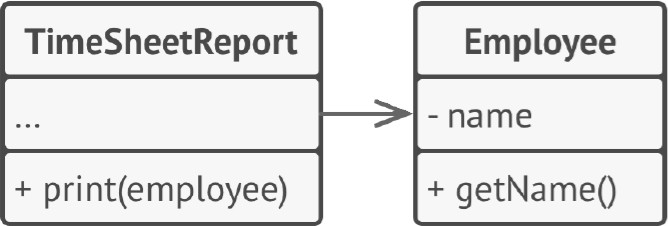
exigindo que se altere o código dentro da classe.



Resolvemos movendo o comportamento relacionado à

impressão de relatórios para uma classe separada.

Esta mudança permite mover outras coisas relacionadas ao relatório para a nova classe



## *https://miro.medium.com/v2/resize:fit:1000/1*0MtFBmm6L2WVM04qCJOZPQ.png*O (Open-Closed Principle): *OCP*

### As classes devem ser abertas para extensão, mas fechadas para modificação

**Qual a ideia?**

Estender o comportamento de uma Classe sem alterar o comportamento existente dela.

Isso evita causar bugs onde quer que a Classe esteja sendo usada

## O (Open-Closed Principle):



### Analogia: Um carro modular

* Imagine que você deseja trocar o sistema de freios de um carro.
* Basta trocar os componentes do freio por melhores sem mexer ou alterar o comportamento básico do freio do carro
  + Ao mexer nisso não vou ter surpresas caso queira freiar e ele acaba acelerando...

## O (Open-Closed Principle):

### Exemplos práticos

* 1) Usar interfaces para permitir novos cálculos de desconto sem modificar o método existente
* 2) Em vez de uma classe que trate tipos fixos de funcionários com

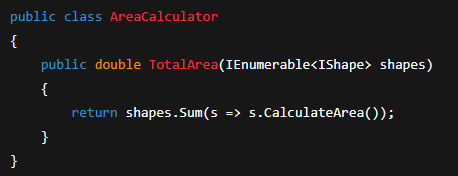
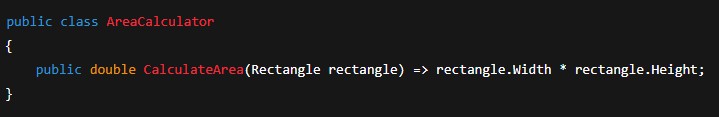
condicionais

* Criar uma interface como *Funcionario* e permita que novas categorias (como *Estagiário*, *Freelancer*) sejam adicionadas com novas classes que a estendem, sem alterar o núcleo existente

## O (Open-Closed Principle):

* Código “*Problema*”

**O que tá pegando aqui?**



**Agora é possível introduzir**

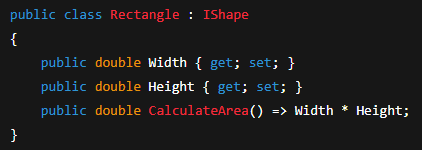
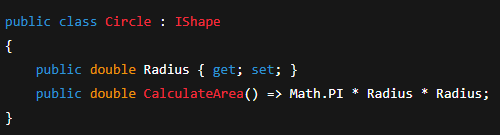
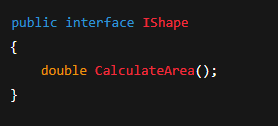
**novas formas sem alterar**

**AreaCalculator**

## O (Open-Closed Principle):

Entidades devem estar abertas para extensão, mas fechadas para modificação.

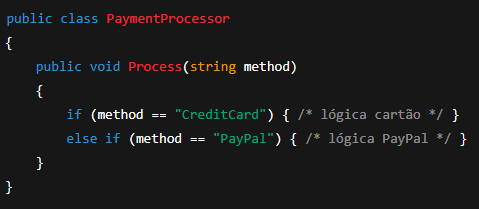
### Código *Solução* -> aplicando OCP com interfaces:



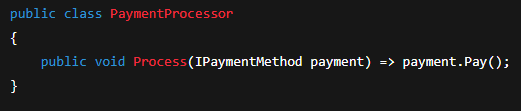
## O (Open-Closed Principle):

* Código “*Problema*”

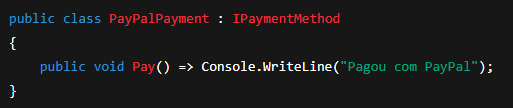
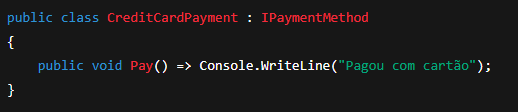
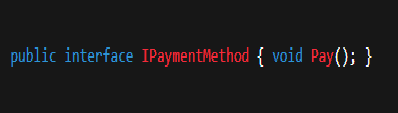
**O que tá pegando aqui?**



## O (Open-Closed Principle):

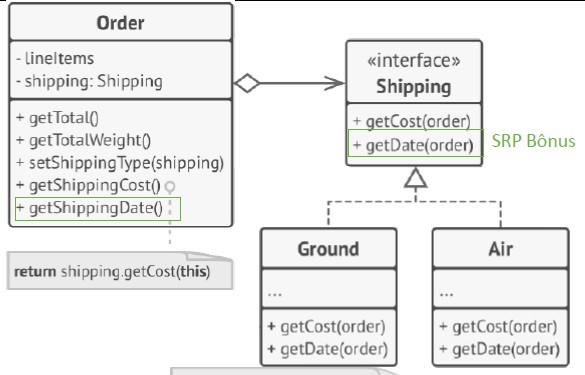


### Código *Solução* -> aplicando OCP com interfaces:



**Agora é possível estender com novos métodos de pagamento sem alterar o *PaymentProcessor*.**

## O (Open-Closed Principle):



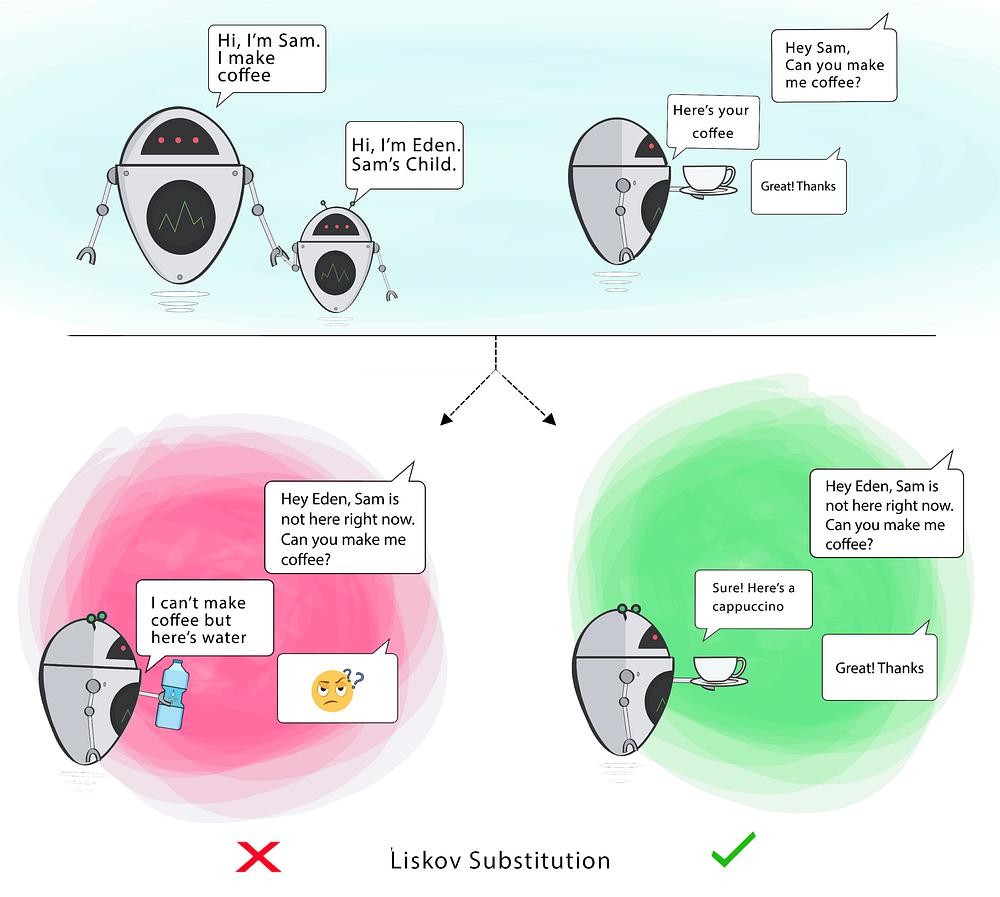
**O que tá pegando aqui?** O que acontece se quisermos adicionar um novo

método de frete (*getShippingCost()*)?

## L (Liskov Substitution Principle): *LSP*

* Se S (*filha*) for um subtipo de T (*pai*)
* Então objetos do tipo T em um programa podem ser substituídos por objetos do tipo S sem alterar nenhuma das propriedades desejáveis desse programa.
* Em outras palavras: Quando uma classe filha não consegue executar as mesmas ações que sua classe pai, isso pode causar bugs.

## L (Liskov Substitution Principle): *LSP*



**Qual a ideia?**

A classe pai entrega café (pode ser qualquer tipo de café).

É aceitável que a classe filha entregue cappuccino, pois é um tipo específico de café

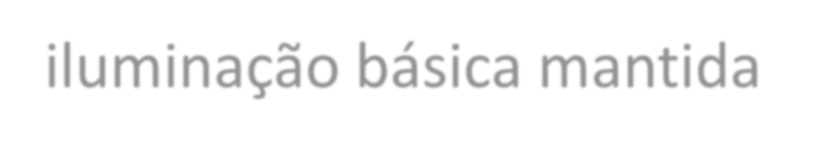
Mas NÃO é aceitável entregar água.

## L (Liskov Substitution Principle):



### Analogia: Lâmpadas em uma luminária

* Se trocar uma lâmpada incandescente por uma fluorescente e ela ainda funcionar normalmente — iluminação básica mantida — o princípio está sendo respeitado.



* Mas se trocar por uma lâmpada decorativa (que pisca, muda cores), a funcionalidade se rompe

## L (Liskov Substitution Principle):

### Exemplos práticos

* 1) *EstudanteDePosGraduacao* não deveria ter que *entregarTCC* caso ele não se aplique — isso quebra o LSP
* 2) A ideia de modelagem na qual um *Quadrado* é um tipo de (herda)

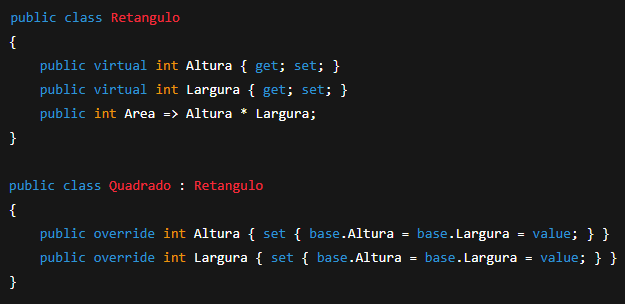
de *Retângulo* viola LSP

* Definir a *largura* e *altura* causa comportamento inesperado.
* No *Retângulo* eles são elementos diferentes mas no *Quadrado* são os

mesmos

## L (Liskov Substitution Principle):

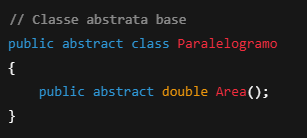
* Código “*Problema*” **O que tá pegando aqui?**



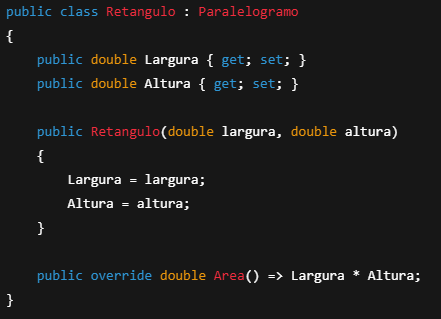
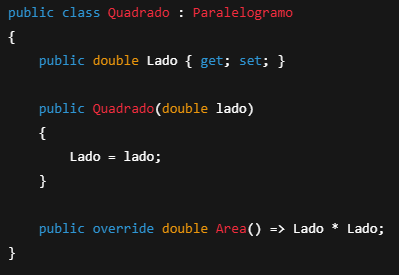
Subclasses devem ser substituíveis por suas superclasses sem alterar o comportamento esperado

## L (Liskov Substitution Principle):

* Código *Solução* -> aplicando LSP:



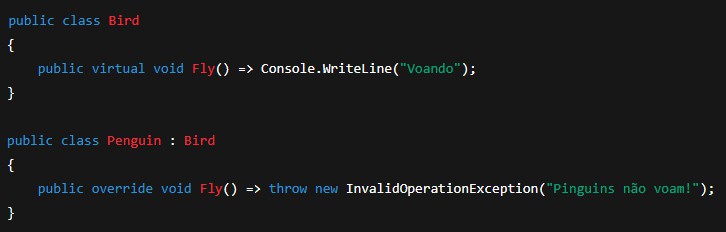
**Polimorfismo puro: ambos herdam de Paralelogramo e implementam Area() sem alterar o significado “das propriedades para o cálculo da Area”, da classe base**



## L (Liskov Substitution Principle):

* Código “*Problema*”

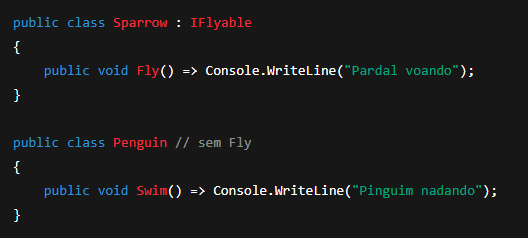
**O que tá pegando aqui?**



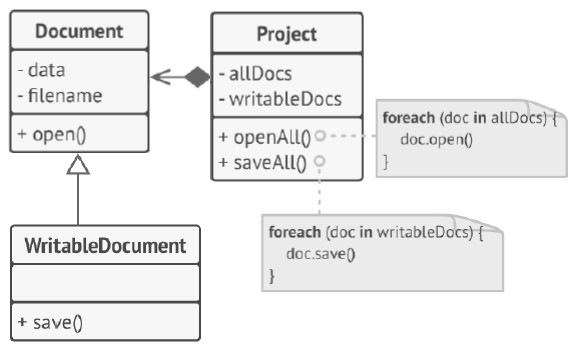
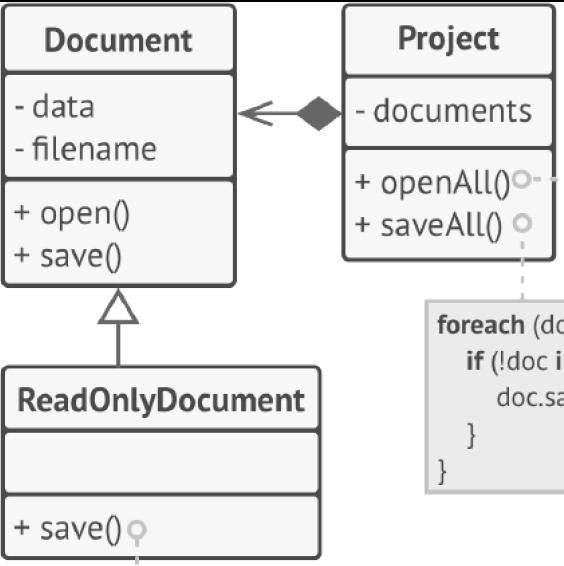
## L (Liskov Substitution Principle):

* Código *Solução* -> aplicando LSP:

**Cada classe implementa apenas comportamentos que fazem sentido, garantindo LSP.**



## L (Liskov Substitution Principle):



**O que tá pegando aqui?** O código será danificado se não verificarmos o tipo

de documento antes de salvá-lo.

*Parece um documento, abre como um documento*

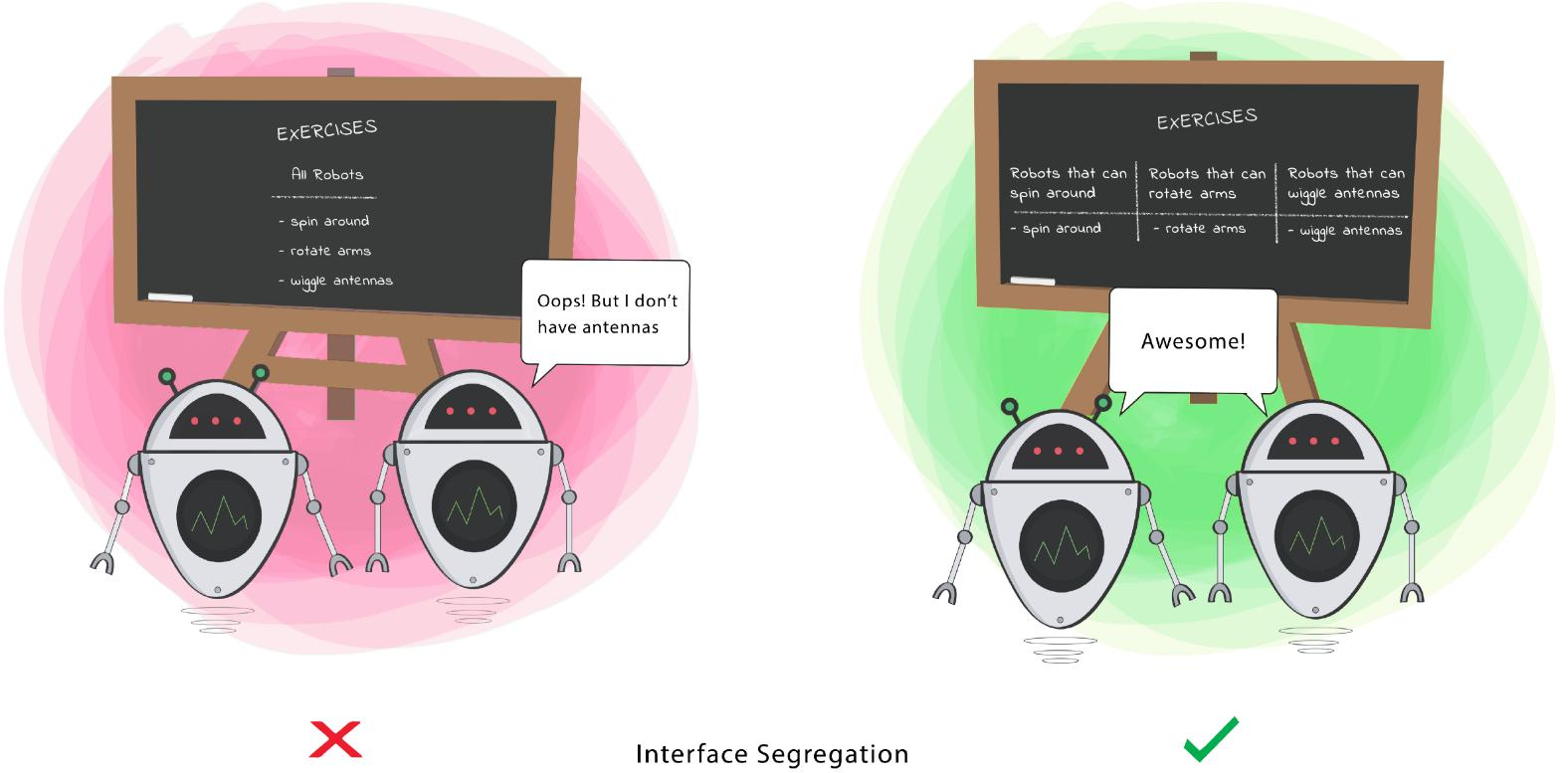
*mas salva lançando exceções*

## I (Interface Segregation Principle): *ISP*

* As classes não devem ser forçados a depender de métodos que não usam.
* Uma classe deve executar apenas as ações necessárias para cumprir sua função.
* Qualquer outra ação deve ser removida completamente ou movida para outro lugar.

## I (Interface Segregation Principle): *ISP*

Várias categorias de robôs



(com e sem antena)

**Qual a ideia?**

Visa dividir um conjunto de ações em conjuntos menores

Para que uma Classe execute SOMENTE o conjunto de ações que ela requer.

Mexer antenas

## I (Interface Segregation Principle):



### Analogia: Ferramentas específicas X genéricas

* Imagine uma caixa de ferramentas cheias de recursos que você nunca usará — você acaba ignorando tudo.
* Melhor ter ferramentas individuais e eficientes que fazem só o que

precisam.

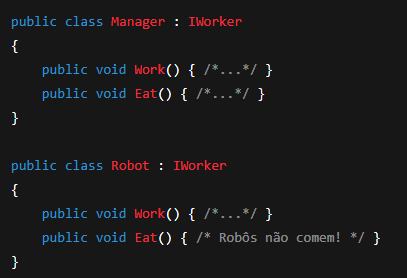
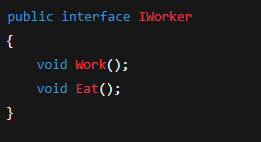
## I (Interface Segregation Principle):

### Exemplos práticos

* 1) Criar uma interface *Aves* genérica *com setAltitude()* obriga pássaros que não voam, como pinguins, a implementar algo que não faz sentido — violação duplamente do ISP e do LSP.
* Melhor dividir a interface:
  + *Aves* (para *localização* e *renderização*)
  + *AvesQueVoam* com *setAltitude*

## I (Interface Segregation Principle):

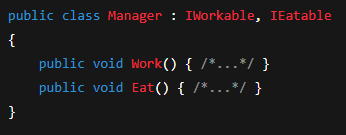
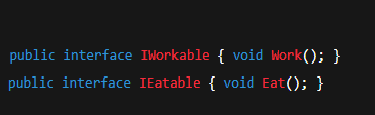
* Código “*Problema*” **O que tá pegando aqui?**



É melhor ter várias interfaces específicas do que uma única genérica

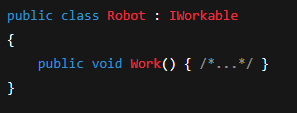
## I (Interface Segregation Principle):

### Código *Solução* -> aplicando ISP com interfaces específicas:



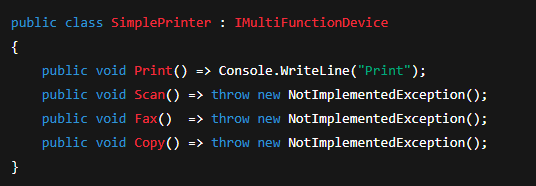
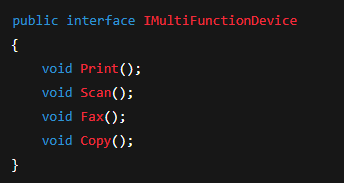
**Cada classe implementa apenas o**

**necessário**



## I (Interface Segregation Principle):

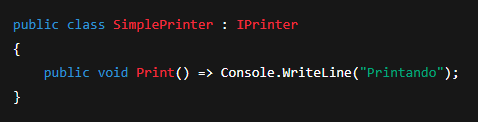
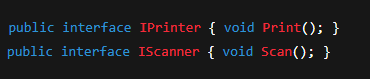
* Código “*Problema*”



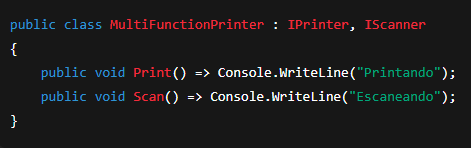
**O que tá pegando aqui?**

## I (Interface Segregation Principle):

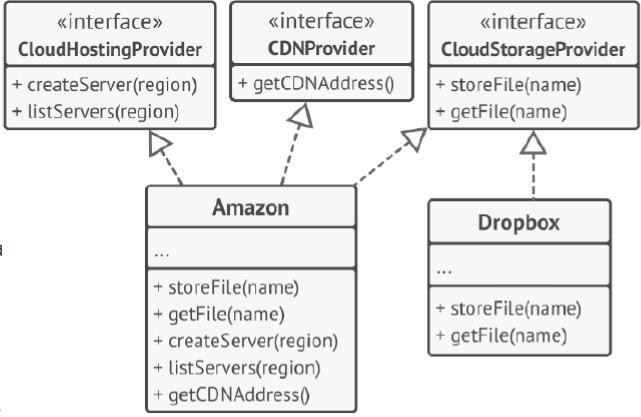
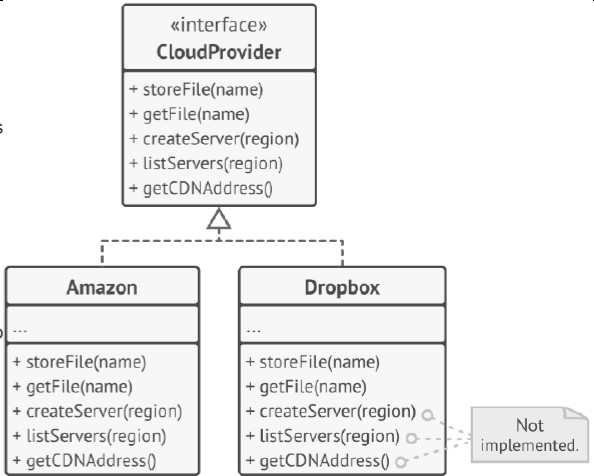
### Código *Solução* -> aplicando ISP com interfaces específicas:



**Cada classe implementa apenas o que realmente usa**



## I (Interface Segregation Principle):



**O que tá pegando aqui?** Na versão inicial oferecesse suporte apenas à Amazon

Cloud.

Ao implementar o suporte para outro provedor, a maioria das interfaces são muito amplas.

## D (Dependency Inversion Principle): *DIP*

### Módulos de alto nível não devem depender de Módulos de baixo nível.

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

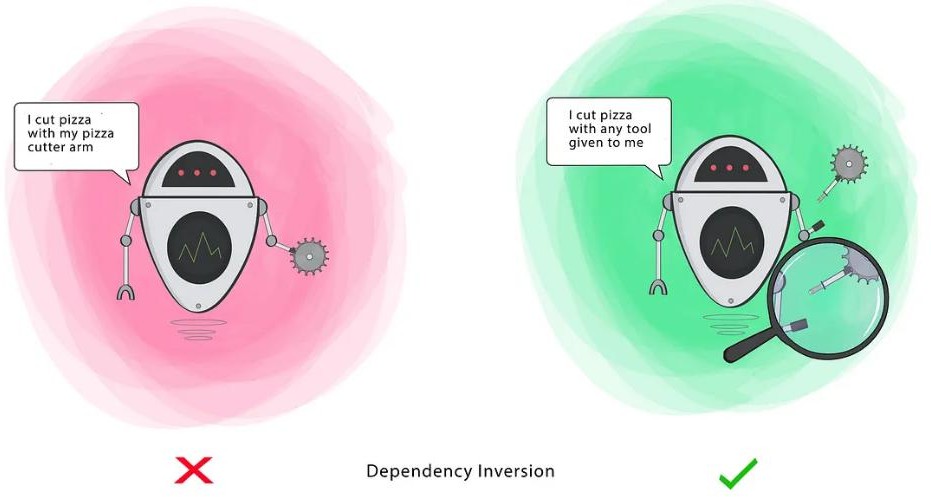
**Módulo (ou Classe) de Alto Nível:** Classe que executa uma ação com uma ferramenta.

**Módulo (ou Classe) de Baixo Nível:** A ferramenta necessária para executar a ação.

**Abstração:** Representa uma interface que conecta as duas Classes.

**Detalhes:** Como funciona a ferramenta

## D (Dependency Inversion Principle): *DIP*

**Qual a ideia?**

Visa reduzir a dependência de uma Classe de alto nível em relação à Classe de baixo nível

Introduzindo uma interface.

## D (Dependency Inversion Principle):

### Analogia: Plug & play de dispositivos eletrônicos.

* Não se liga uma lâmpada diretamente na tomada com fios
* Ela depende de um soquete (abstração).
* Assim, diferentes lâmpadas podem ser usadas pelo adaptador.
* Resumo: Usar "plug & play" de componentes que se conectam via abstrações, como plugues e tomadas

## D (Dependency Inversion Principle):

### Exemplos práticos

* Em vez de uma classe *PasswordReminder* instanciar diretamente um

***Postgre****SQLConnection*,

* Faça com que ela dependa de uma interface abstrata como

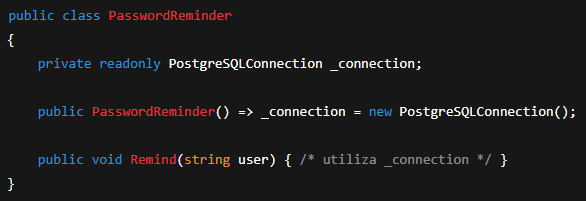
*IConnection*.

* Assim, você pode injetar diferentes tipos de conexão (*MySQL, Oracle,*

*simulação em testes*) sem mudar *PasswordReminder*

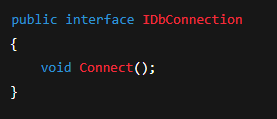
## D (Dependency Inversion Principle):

* Código “*Problema*” **O que tá pegando aqui?**

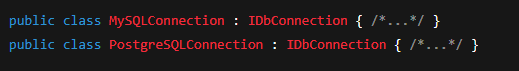


Módulos de alto nível não devem depender de módulos de baixo nível, e sim de abstrações.

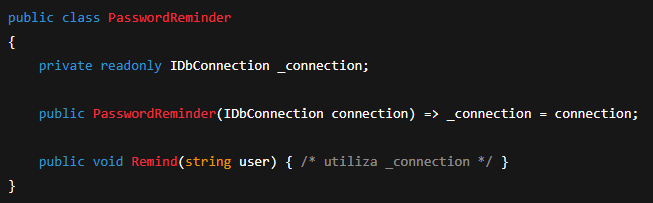
## D (Dependency Inversion Principle):



* Código *Solução* -> aplicando DIP:



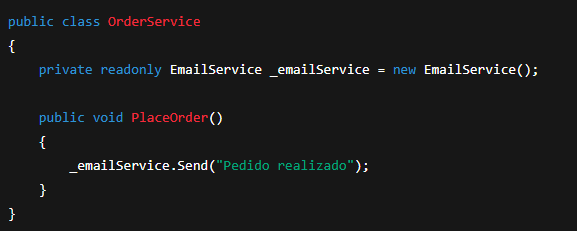
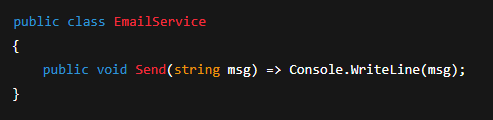
**Agora *PasswordReminder* depende de abstração, não de uma implementação específica.**



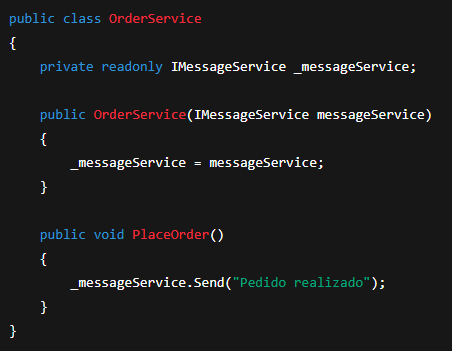
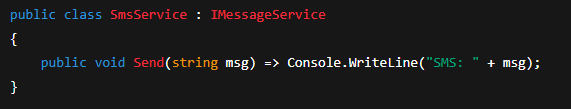
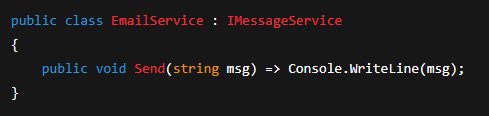
## D (Dependency Inversion Principle):

* Código “*Problema*”

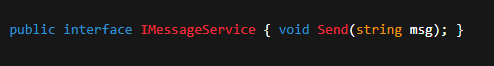
**O que tá pegando aqui?**



## D (Dependency Inversion Principle):



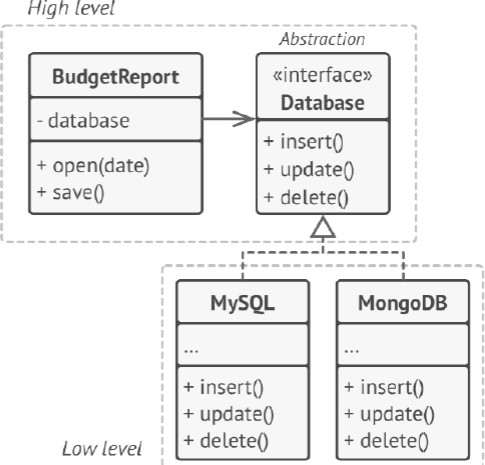
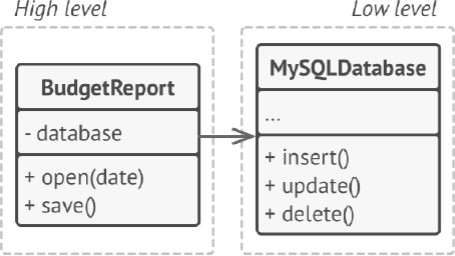
* Código *Solução* -> aplicando DIP:



**Agora *OrderService* depende da abstração, não de uma implementação específica, favorecendo a flexibilidade.**

**O que tá pegando aqui?**

## D (Dependency Inversion Principle):



A classe de relatório (***alto nível***) usa uma classe de banco de dados (***baixo nível***) para ler e manter seus dados.

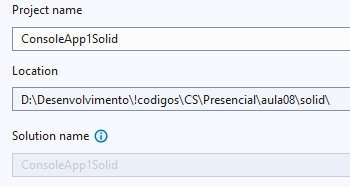
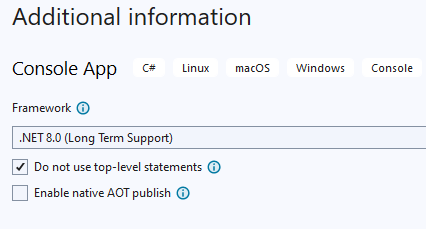
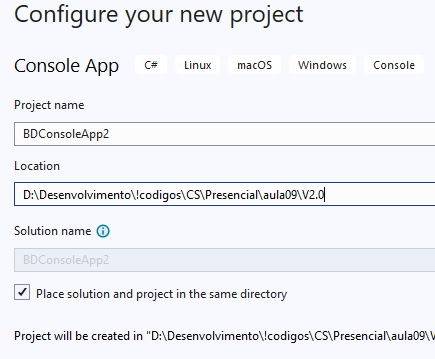
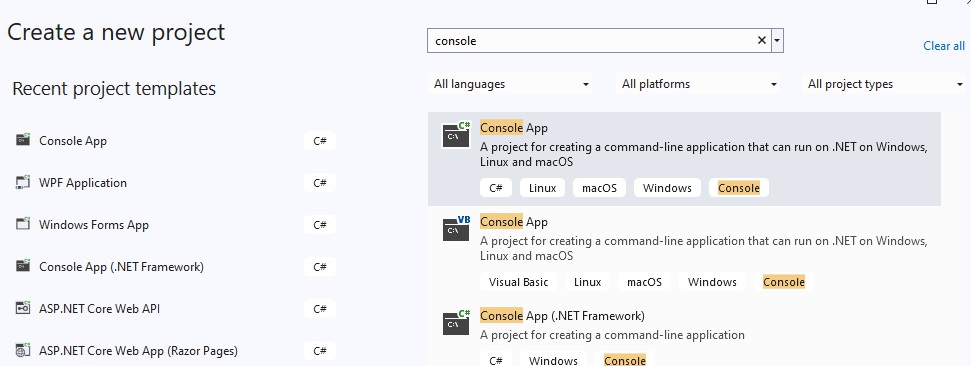
Qualquer mudança na classe de baixo nível (*uma nova versão do banco de dados*), pode afetar a classe de alto nível.

Resumo dos Princípios

|  |  |  |  |
| --- | --- | --- | --- |
| **Acrônimo** | **Princípio** | **Essência** | **Benefício-chave** |
| **S** | Single  Responsibility  (SRP) | “Só uma responsabilidade por classe” | Coesão e clareza estrutural |
| **O** | Open–Closed  (OCP) | "Aberto à extensão, fechado à  modificação" | Estabilidade e  extensibilidade |
| **L** | Liskov Substitution  (LSP) | "Substituição segura entre base e  derivadas” | Polimorfismo seguro |
| **I** | Interface  Segregation  (ISP) | “Interfaces pequenas e diversas,  não uma só inchada” | Modularidade e baixo  acoplamento |
| **D** | Dependency  Inversion  (DIP) | “Dependa de contratos, não de  implementações” | Flexibilidade e testabilidade |

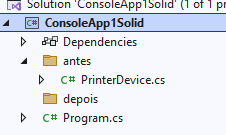
Resumo das Analogias

|  |  |  |
| --- | --- | --- |
| **Princípio** | **Analogia** | **Objetivo** |
| **SRP** | Restaurante com profissionais | Mostrar foco e reduzido acoplamento |
| **OCP** | Carro com peças intercambiáveis | Extender sem modificar o núcleo (a essência) |
| **LSP** | Substituição de lâmpadas | Garantir comportamento consistente |
| **ISP** | Ferramentas específicas | Evitar interfaces inchadas e inúteis |
| **DIP** | Plug & play de tomada | Promover flexibilidade por abstrações |



Atividade Juntos

* Vamos abrir o seguinte projeto e analisar o código
* Vamos rodar o código que está na pasta

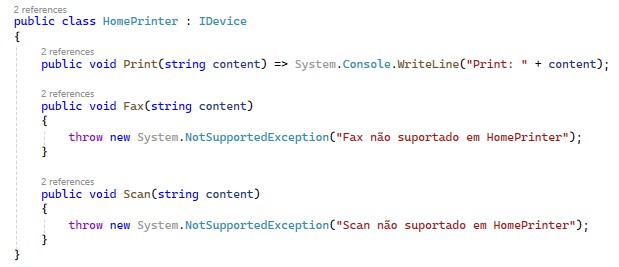


ConsoleApp1Solid - Aluno.zip

“*antes*”

* O que tá pegando?
* Como podemos melhorar? Vamos implementar

na pasta “*depois*”



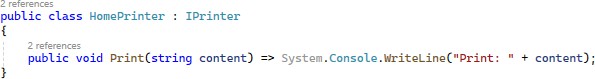
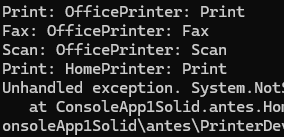
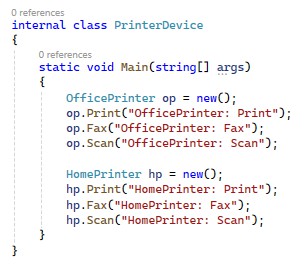
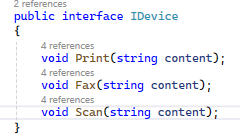
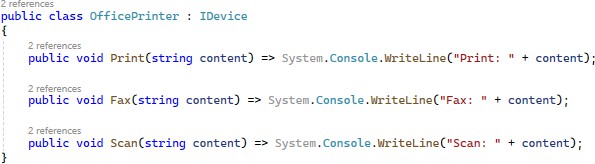
em métodos vazios por

causa que a interface força implementar métodos não são suportados

Não está respeitando:

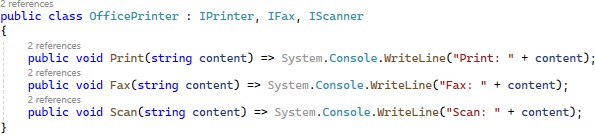
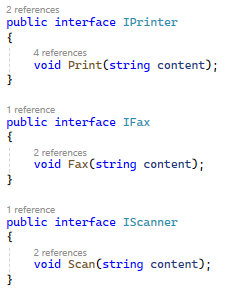
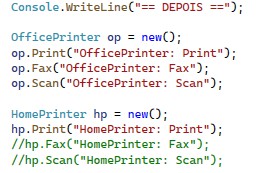
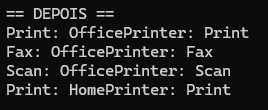
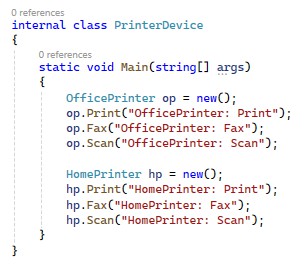
* ISP
* LSP

Está lançando exceções



lança exceção em métodos vazios; não implementa métodos que não suporta.

Violações resolvidas:



* + ISP: cada classe implementa só os métodos que faz sentido.
  + LSP: HomePrinter não

# Atividades - 01

* 1) Escolha um domínio com o qual se sinta confortável — por exemplo: cafeteria, restaurante, aplicativo de mensagens, etc.
* 2) Para cada princípio (S, O, L, I, D), pense num exemplo desse domínio que:
* Explique claramente a intenção do princípio;
* Mostre o que daria errado se o princípio **não** fosse seguido;
* Mostre como seria uma boa aplicação do princípio no contexto da analogia.
* 3) Registre em tabela com colunas
* “Princípio”, “O que seria feito incorretamente (violação)”, “Analogia (domínio)”, “Como seria com o princípio aplicado”.

Exemplo: Domínio “**Cafeteria**”

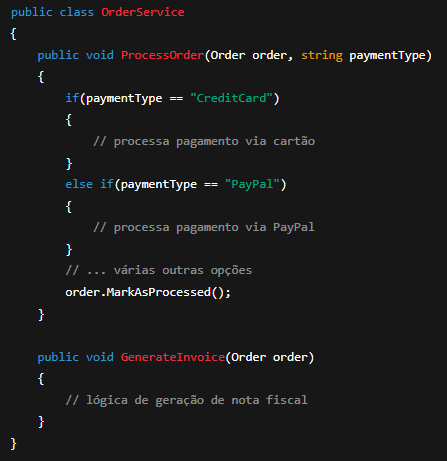
|  |  |  |  |
| --- | --- | --- | --- |
| **Princípio** | **Violação (na analogia)** | **Analogia** | **Com o princípio é aplicado** |
| **SRP**  **Single Responsibility** | O barista faz tudo: prepara o café, limpa a máquina, atende clientes, faz cobranças — tudo por uma pessoa só. Se algo mudar (ex: método de pagamento), o barista sofre impacto em muitas responsabilidades. | Na cafeteria, ter um funcionário que cuida de tudo | Separar funções: um funcionário só prepara cafés; outro cuida da limpeza; outro atende no caixa. Cada um tem uma única responsabilidade. |
| **OCP**  **Open/Closed** | A cafeteria tem um cardápio fixo; quando querem adicionar um novo tipo de café, precisam redesenhar o cardápio físico, mudar decoração, reaprender todo o processo de produção. | Cardápio de café que você tem que reimprimir sempre que inserir um novo tipo (ex: café gelado novo) | Ter um cardápio digital ou quadros modulares: você pode adicionar uma nova bebida sem refazer tudo. A cafeteria está aberta para novos tipos de bebida, mas não precisa reescrever/reestruturar o estabelecimento. |
| **LSP**  **Liskov Substitution** | Suponha que “Café Especial” herde de “Café Base” e redefina certas etapas. Mas se alguém espera que “Café Base” sempre possa ter adição de leite ou açúcar, e “Café Especial” proíbe ou altera essas etapas, isso quebra expectativas. | “Café Base” com opções de personalização; “Café Especial” herda e remove ou altera opções esperadas | Fazer com que “Café Especial” implemente uma interface base comum, respeitando que todos os cafés permitam pelo menos as opções básicas. Ou não usar herança se “Café Especial” tiver comportamentos radicalmente diferentes. |
| **ISP**  **Interface Segregation** | Forçar todos os clientes do café a ter o mesmo menu completo: bebidas quentes, frias, sobremesas, cafés especiais — mesmo que algumas pessoas só queiram café expresso simples. Eles pagam um pacote “menu completo” embora usem só parte. | Menu único enorme que inclui tudo, mesmo para quem quer só um café simples | Ter menus específicos: “Menu Café  Simples”, “Menu Bebidas Geladas”, “Menu Sobremesas”. Clientes escolhem o que usar; os funcionários só precisam conhecer o menu relevante para seu tipo de atendimento. |
| **DIP**  **Dependency Inversion** | A cafeteria compra máquinas específicas de café, tipo uma marca fixa, sem possibilidade futura de trocar ou melhorar; tudo depende de uma máquina concreta; se quebrar, tudo para. | Funcionário (alto nível) depende de  máquina concreta, sem abstração | Usar um padrão em que o barista trabalha com uma interface de máquina de café; diferentes marcas de máquina (concreções) podem ser usadas, desde que implementem essa interface. Assim é possível trocar de máquina sem refatorar toda operação. |

# Atividades - 02

### Leia os seguintes trechos de código (em C#). Para cada trecho:

* 1) Identifique qual(is) princípio(s) SOLID está(ão) sendo violado(s).
* 2) Explique brevemente o que está incorreto no código — qual comportamento ou estrutura causa a violação.
* 3) Sugira uma pequena correção ou refatoração que eliminaria a

violação.



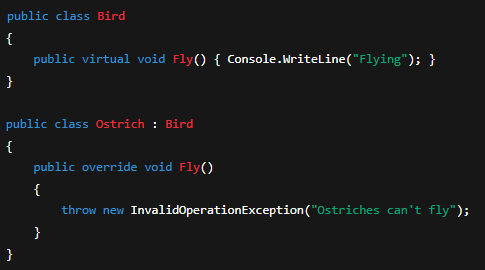
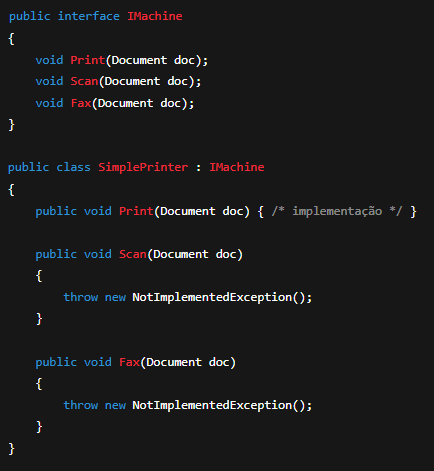
Exemplo: “**OrderService**”

**Código A**

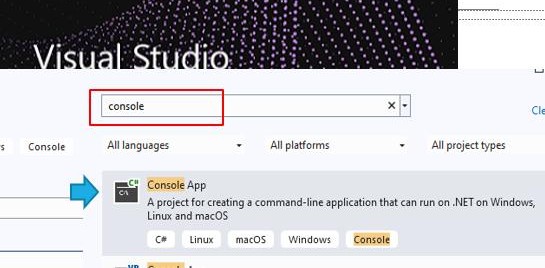
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Princípio(s) violado(s)** | **O que está violado / Por que é violação** | **Sugestão de correção / refatoração** |
|  |
| OCP  (Open/Closed  Principle),  SRP  (Single Responsibility Principle) | O método ProcessOrder mistura lógica de decidir tipo de pagamento (vários if/else) – sempre que se adicionar um novo tipo de pagamento será necessário modificar esse método.  Também a classe tem múltiplas responsabilidades: processar pedidos e gerar faturas (*invoice*). | Introduzir abstração de pagamento: interface *IPaymentMethod* com implementações para cada tipo (Cartão, PayPal, etc.), passar o método desejado.  Separar geração de fatura em classe separada (por exemplo *InvoiceGenerator*). |

**Código B**

**Código C**



|  |  |  |
| --- | --- | --- |
| **Princípio(s) violado(s)** | **O que está violado / Por que é violação** | **Sugestão de correção / refatoração** |

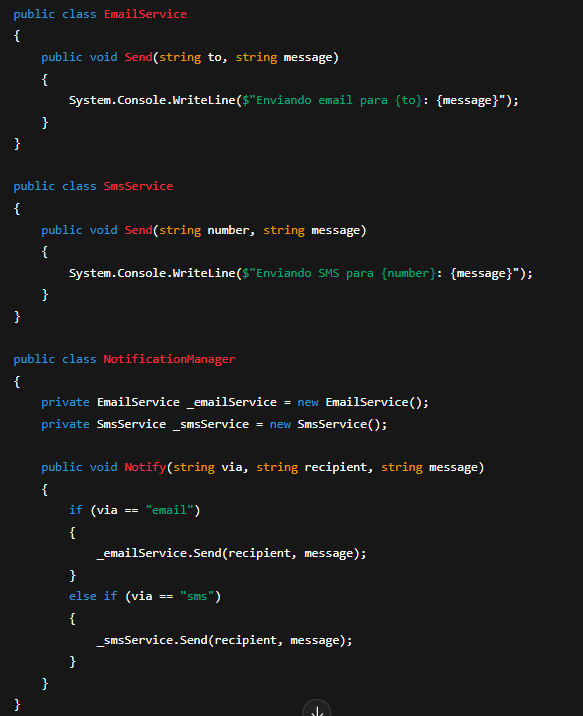


# Atividades - 03

* 1) Os trechos de códigos a seguir violam princípios SOLID.
* 2) Deve-se identificar para cada trecho qual(is) princípio(s) está(ão) sendo violado(s).
* 3) Refatore o código para resolver essas violações.
* 4) Entregue uma ***solution*** do Visual Studio contendo os arquivos originais (*antes*) e

os refatorados (*depois*), com comentários explicativos.

* 5) Gerar o “***Main***”, de forma que se tenha um executável funcionando do exemplo



**Código 01**

Refatore **NotificationManager** para que dependa de abstrações, não de classes concretas.

Permita adicionar novos canais de notificação sem modificar *Notify*.