# S.O.L.I.D.

GCES - Gerência de Configuração e Evolução de Software

Prof. Renato Sampaio

# SOLID



Robert C. Martin - princípios de OO

**S** - Single-responsiblity principle

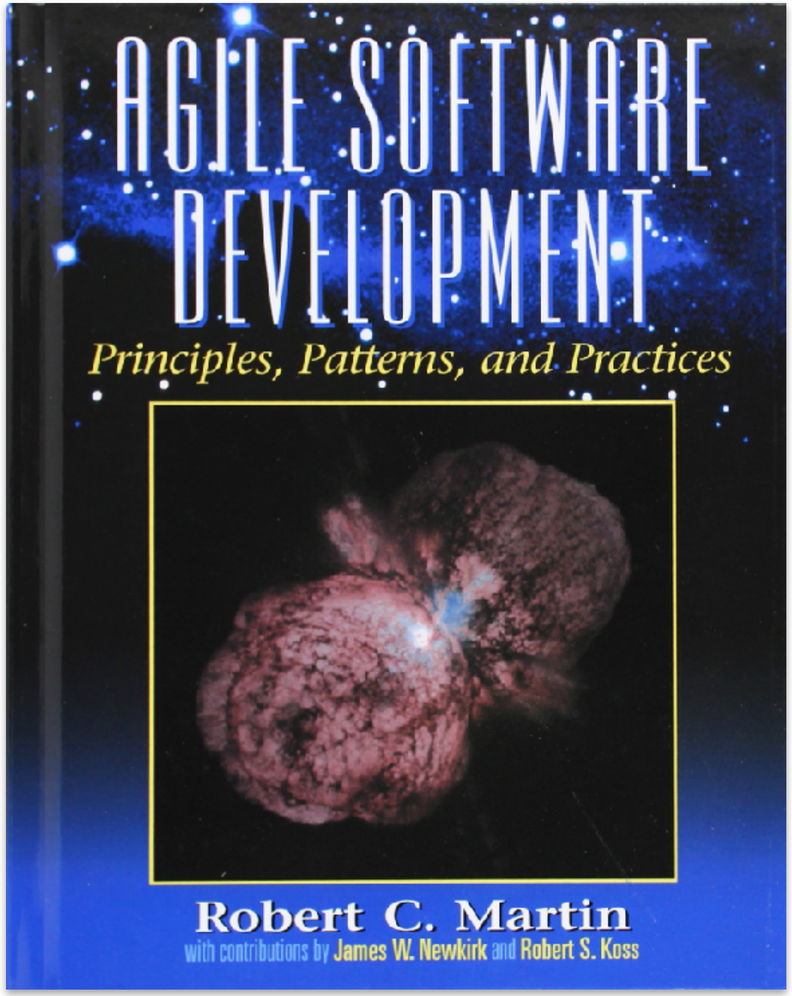
**O** - Open-closed principle

**L** - Liskov substitution principle

**I** - Interface segregation principle

**D** - Dependency Inversion Principle

Michael Feathers - nomeou o SOLID



# Objetivos de O.O.



- Ser fácil de se manter, adaptar e ajustar à alterações de escopo;
- Ser testável e de fácil entendimento;
- Ser extensível para alterações com o mínimo de esforço;
- Fornecer o máximo de reaproveitamento;
- Ser utilizável pelo máximo tempo possível.

# SOLID



- **Evita problemas como:**
  - dificuldade na testabilidade (criação de testes unitários);
  - código sem estrutura padronizada (macarrão);
  - dificuldade de isolar funcionalidades (acoplamento alto);
  - duplicação de código (não ter que mudar algo em vários lugares distintos na hora da manutenção);
  - fragilidade de código (uma pequena mudança que gera grandes efeitos colaterais).

# SOLID



**S**ingle Responsibility Principle

**O**pen-Closed Principle

**L**iskov Substitution Principle

**I**nterface Segregation Principle

**D**ependency Inversion Principle

# SOLID



**S**ingle Responsibility Principle

**O**pen-Closed Principle

**L**iskov Substitution Principle

**I**nterface Segregation Principle

**D**ependency Inversion Principle

# SRP - Single Responsibility Principle



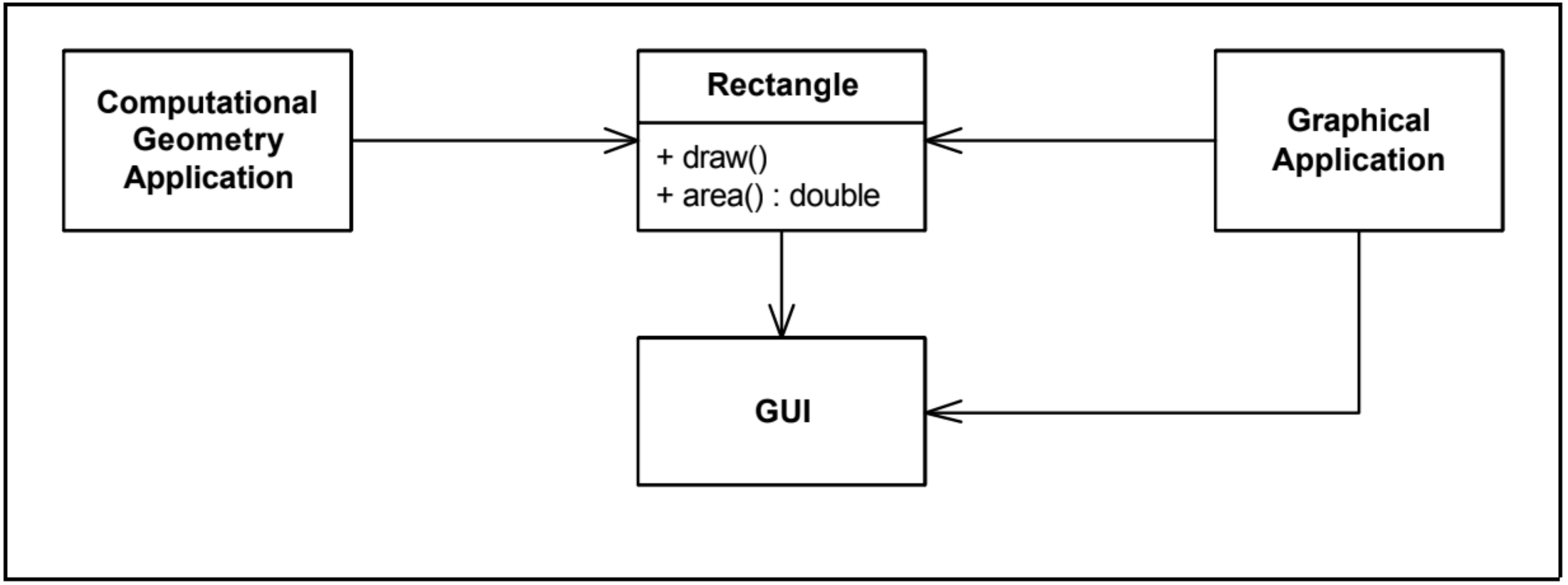
**"A class should have one, and only one, reason to change."**

Martin Fowler

- A classe deve ter uma única responsabilidade.
- Pode ser aplicado a classe, métodos, arquivos (um por classe, por exemplo).
- Responsabilidade não é função. Significa pontos de mudança.

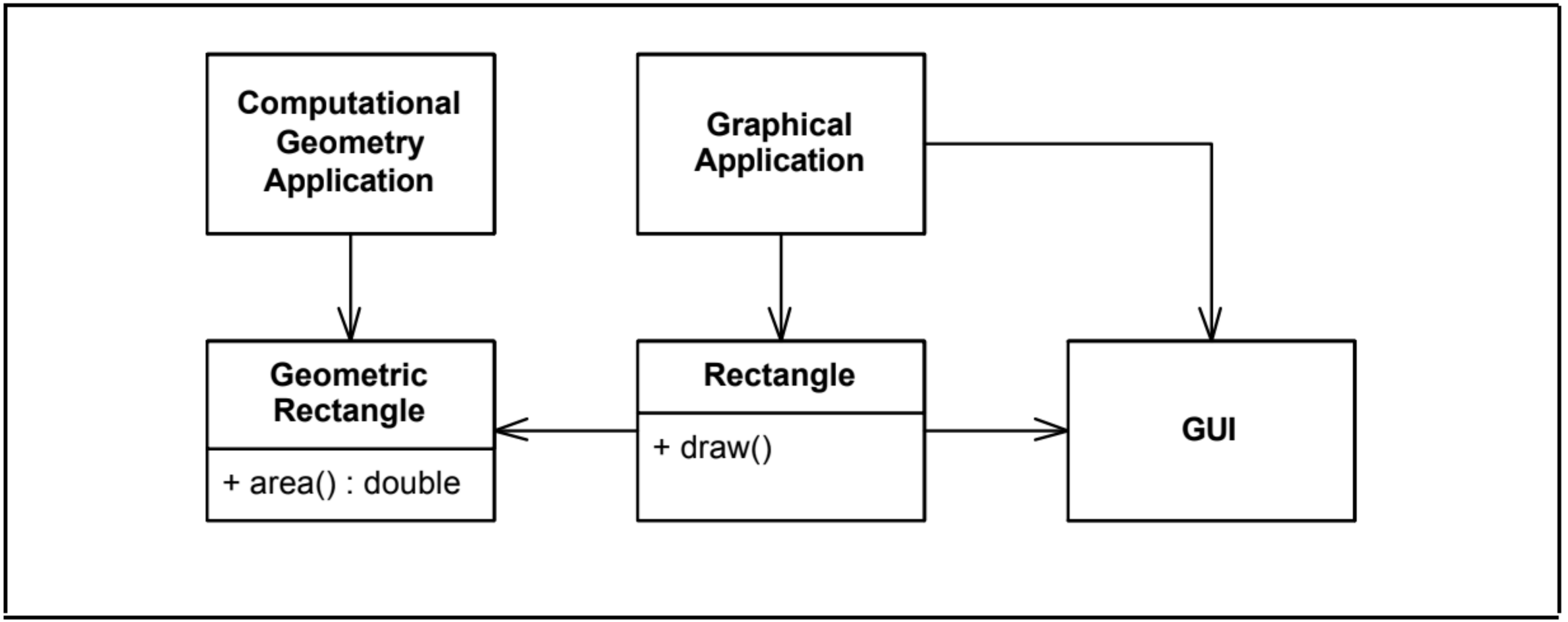
# SRP - Single Responsibility Principle





# SRP - Single Responsibility Principle





# SRP - Single Responsibility Principle

❌ Violação do SRP

```python
class Invoice:
    def __init__(self, customer, items):
        self.customer = customer
        self.items = items  # list of (description, quantity, price)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item["quantity"] * item["price"]
        return total

    def print_invoice(self):
        print("Invoice for:", self.customer)
        for item in self.items:
            print(f"{item['description']} x {item['quantity']} \
                    = ${item['price'] * item['quantity']}")
        print("Total:", self.calculate_total())

    def save_to_db(self):
        # Escreve no Banco de Dados
        print(f"Saving invoice for {self.customer} to database...")
```

A classe Invoice tem muitas responsabilidades:

- Lógica de negócio (cálculo dos totais)
- Lógica de apresentação (print)
- Lógica de persistencia (Banco de Dados)

**Qualquer mudança** nas lógicas de apresentação ou banco de dados irão **demandar modificações na mesma classe**.

**Dificulta os testes**: qualquer teste da lógica de negócio irá exigir impressão e interação com o Banco de Dados.

# SRP - Single Responsibility Principle



❌ Violação do SRP

```python
class Invoice:
    def __init__(self, customer, items):
        self.customer = customer
        self.items = items  # list of (description, quantity, price)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item["quantity"] * item["price"]
        return total

    def print_invoice(self):
        print("Invoice for:", self.customer)
        for item in self.items:
            print(f"{item['description']} x {item['quantity']} \
                    = ${item['price'] * item['quantity']}")
        print("Total:", self.calculate_total())

    def save_to_db(self):
        # Escreve no Banco de Dados
        print(f"Saving invoice for {self.customer} to database...")
```

✅ Aplicando o SRP

```python
class Invoice:
    def __init__(self, customer, items):
        self.customer = customer
        self.items = items

    def calculate_total(self):
        return sum(item["quantity"] * item["price"] for item in self.items)


class InvoicePrinter:
    def print(self, invoice):
        print("Invoice for:", invoice.customer)
        for item in invoice.items:
            print(f"{item['description']} x {item['quantity']} \
                    = ${item['price'] * item['quantity']}")
        print("Total:", invoice.calculate_total())


class InvoiceRepository:
    def save(self, invoice):
        # Código de interação com o Banco de dados
        print(f"Saving invoice for {invoice.customer} to database...")
```

# SOLID



Single Responsibility Principle

**O**pen-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

# OCP - Open Closed Principle

**"Modules (classes, functions, etc.) should be open for extension, but closed for modification."**
**(Bertrand Meyer)**

- Uma classe após pronta não deve ser mexida. Deve ser extensível.
- O polimorfismo é a base da extensão. Você cria novo código mas não mexe no código anterior.
- Ex: classe abstrata com herança e sobrescrita de métodos. Se a classe muda, seus testes terão que ser modificados. Se é estendida, cria-se um novo teste.

*Shape.h*

```
enum ShapeType {circle, square};
struct Shape
     {enum ShapeType itsType;};
```

*Circle.h*

```
struct Circle
{
  enum ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};
void DrawCircle(struct Circle*)
```

*Square.h*

```
struct Square
{
  enum ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};
void DrawSquare(struct Square*)
```

*DrawAllShapes.c*

```
#include <Shape.h>
#include <Circle.h>
#include <Square.h>

typedef struct Shape* ShapePtr;

void
DrawAllShapes(ShapePtr list[], int n)
{
  int i;
  for( i=0; i< n, i++ )
  {
    ShapePtr s = list[i];
    switch ( s->itsType )
    {
    case square:
      DrawSquare((struct Square*)s);
      break;
    case circle:
      DrawCircle((struct Circle*)s);
      break;
    }
  }
}
```

# OCP - Open Closed Principle

- A tentativa de inserir uma nova classe, por exemplo um círculo ou uma elipse tem um grande efeito colateral.
- Para adicionar uma nova forma geométrica temos que inserí-la em Shape.h. Tudo terá que ser compilado.
- Todos as ocorrências do s*witch statement* terão que ser re-escritas, etc.

*Shape.h*

```
Class Shape
{
public:
  virtual void Draw() const=0;
};
```

*Square.h*

```
Class Square: public Shape
{
public:
  virtual void Draw() const;
};
```

*Circle.h*

```
Class Circle: public Shape
{
public:
  virtual void Draw() const;
};
```

*DrawAllShapes.cpp*

```
#include <Shape.h>

void
DrawAllShapes(Shape* list[],int n)
{
  for(int i=0; i< n; i++)
    list[i]->draw();
}
```

# OCP - Open Closed Principle



- Com a herança de classes e o polimorfismo, atendemos o OCP.
- Não temos mais um código tão rígido.
- Nada precisa ser recompilado caso seja adicionada uma nova classe com uma nova forma geométrica.

# SOLID

**S**ingle Responsibility Principle

**O**pen-Closed Principle

**L**iskov Substitution Principle

**I**nterface Segregation Principle

**D**ependency Inversion Principle

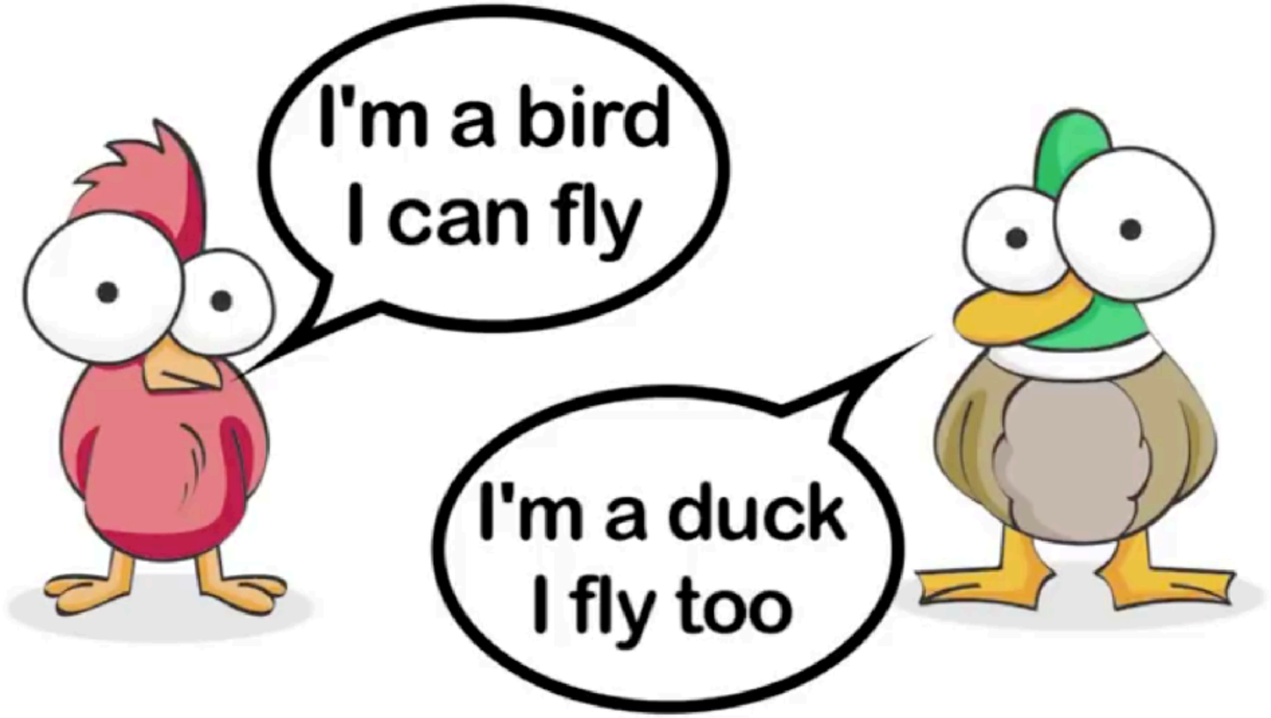
# LSP - Liskov Substitution Principle

**“Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S, where S is a subtype of T.”** Barbara Liskov - 1988 (MIT)

- Uma classe base deve poder ser substituída por sua classe derivada.
- Classes derivadas devem ser utilizáveis através da interface da classe base, sem a necessidade do usuário saber a diferença.

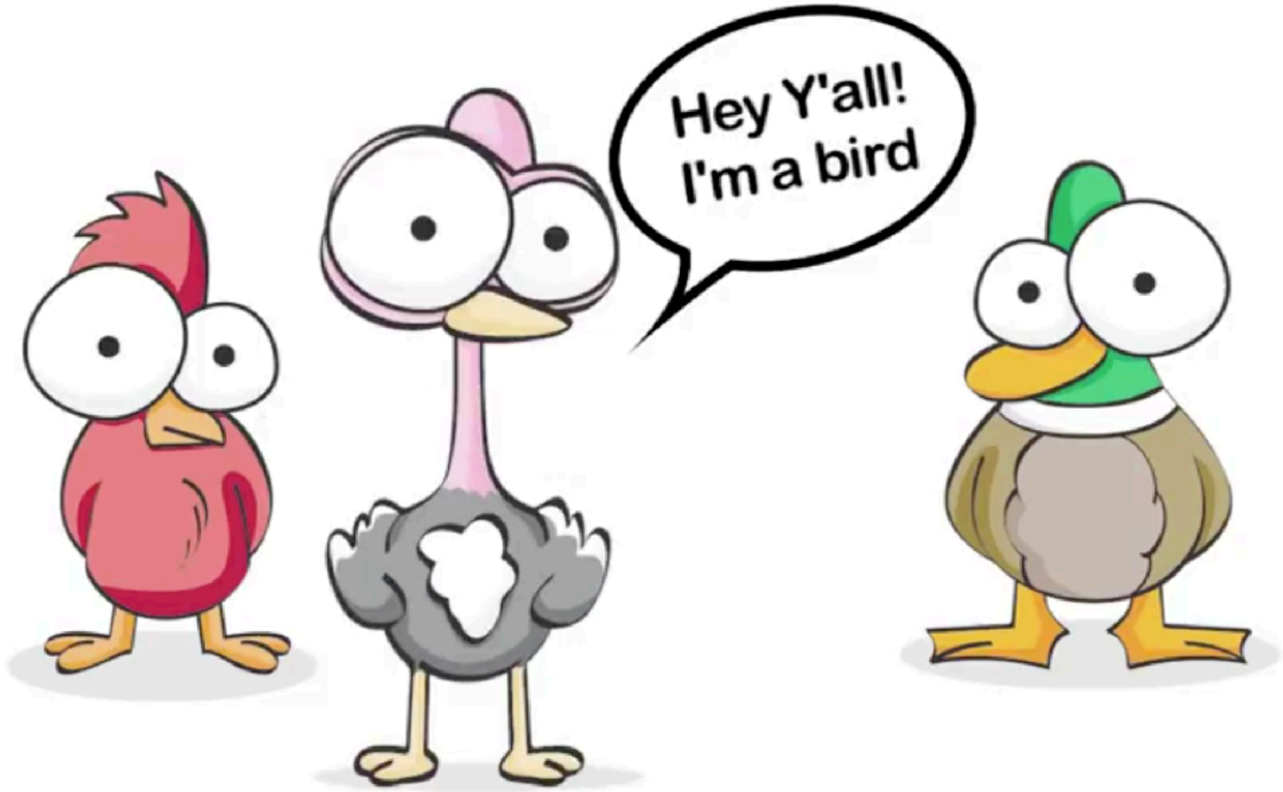
# LSP - Liskov Substitution Principle

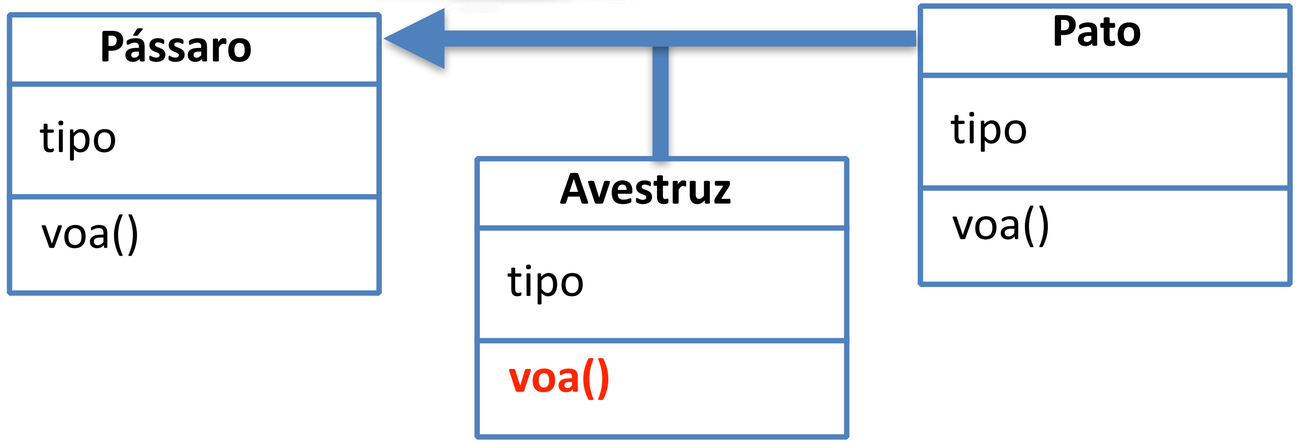





# LSP - Liskov Substitution Principle



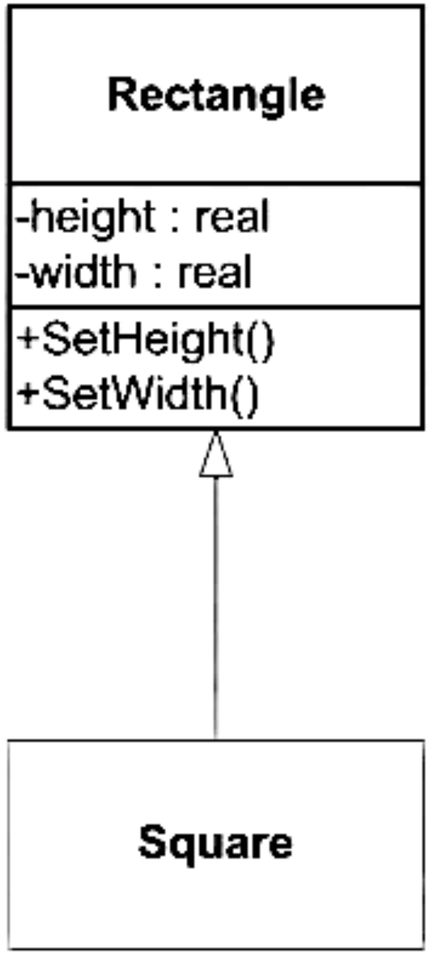


# LSP - Liskov Substitution Principle



❌ Violação do LSP




```python
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    def set_width(self, width):
        self._width = width

    def set_height(self, height):
        self._height = height

    def get_area(self):
        return self._width * self._height

class Square(Rectangle):
    def set_width(self, width):
        self._width = width
        self._height = width

    def set_height(self, height):
        self._width = height
        self._height = height
```

```python
def print_area(rectangle):
    rectangle.set_width(5)
    rectangle.set_height(10)
    print("Área esperada:", 5 * 10)
    print("Área calculada:", rectangle.get_area())


r = Rectangle(2, 3)
s = Square(2, 3)

print_area(r)
print_area(s)
```

**Saída**

```
Área esperada: 50
Área calculada: 50
Área esperada: 50
Área calculada: 100
```

# LSP - Liskov Substitution Principle



✅ Aplicando o LSP

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def get_area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height


class Square(Shape):
    def __init__(self, side):
        self.side = side

    def get_area(self):
        return self.side * self.side
```

```python
def print_area(shape: Shape):
    print(f"A área é: {shape.get_area()}")


shapes = [
    Rectangle(5, 10),
    Square(5),
]

for shape in shapes:
    print_area(shape)
```

**Saída**

```
A área é: 50
A área é: 25
```

# LSP - Liskov Substitution Principle



- Exemplo de solução: perguntar o tipo do objeto antes de usá-lo. (**IF**) Instanceof(), is(), as(), etc.
- Já quebrou o OCP. Criou uma dependência.
- Relação **É UM** (is a) nem sempre funciona.
- Qual seria a relação entre retângulo e quadrado? Podem ambos derivar de uma classe base. Porém, elas não tem relação direta. É uma violação do LSP.
- Se você viola o LSP eventualmente você irá quebrar a regra e usar um IF em algum lugar!

# LSP - Liskov Substitution Principle



- Respeitar o LSP implica em respeitar o OCP

**LSP => OCP**

# LSP - Liskov Substitution Principle



❌ Violação do LSP

```python
class Invoice:
    def __init__(self, customer, items):
        self.customer = customer
        self.items = items  # list of (description, quantity, price)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item["quantity"] * item["price"]
        return total

    def print_invoice(self):
        print("Invoice for:", self.customer)
        for item in self.items:
            print(f"{item['description']} x {item['quantity']} \
                    = ${item['price'] * item['quantity']}")
        print("Total:", self.calculate_total())

    def save_to_db(self):
        # Escreve no Banco de Dados
        print(f"Saving invoice for {self.customer} to database...")
```

A classe Invoice tem muitas responsabilidades:

- Lógica de negócio (cálculo dos totais)
- Lógica de apresentação (print)
- Lógica de persistencia (Banco de Dados)

**Qualquer mudança** nas lógicas de apresentação ou banco de dados irão **demandar modificações na mesma classe**.

**Dificulta os testes**: qualquer teste da lógica de negócio irá exigir impressão e interação com o Banco de Dados.

# LSP - Liskov Substitution Principle



❌ Violação do SRP

```python
class Invoice:
    def __init__(self, customer, items):
        self.customer = customer
        self.items = items  # list of (description, quantity, price)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item["quantity"] * item["price"]
        return total

    def print_invoice(self):
        print("Invoice for:", self.customer)
        for item in self.items:
            print(f"{item['description']} x {item['quantity']} \
                    = ${item['price'] * item['quantity']}")
        print("Total:", self.calculate_total())

    def save_to_db(self):
        # Escreve no Banco de Dados
        print(f"Saving invoice for {self.customer} to database...")
```

✅ Aplicando o SRP

```python
class Invoice:
    def __init__(self, customer, items):
        self.customer = customer
        self.items = items

    def calculate_total(self):
        return sum(item["quantity"] * item["price"] for item in self.items)


class InvoicePrinter:
    def print(self, invoice):
        print("Invoice for:", invoice.customer)
        for item in invoice.items:
            print(f"{item['description']} x {item['quantity']} \
                    = ${item['price'] * item['quantity']}")
        print("Total:", invoice.calculate_total())


class InvoiceRepository:
    def save(self, invoice):
        # Código de interação com o Banco de dados
        print(f"Saving invoice for {invoice.customer} to database...")
```

# SOLID

**S**ingle Responsibility Principle

**O**pen-Closed Principle

**L**iskov Substitution Principle

**I**nterface Segregation Principle

**D**ependency Inversion Principle

# ISP - Interface Segregation Principle



**“States that no client should be forced to depend on methods it does not use.”**

- Clientes (classes) não devem ser forçados a depender de métodos que não usam.
- Muitas interfaces simples é melhor do que uma única interface genérica.

# ISP - Interface Segregation Principle



```
interface StreamIO {
  void reset();
  void read( … );
  void write( … );
  }
```

Como usar esta interface para um sensor? (Read-only)

Ou para uma impressora? (Write-only)

# ISP - Interface Segregation Principle



```
interface ReadableStream {
  void reset();
  void read( … );
  }
interface WritableStream {
  void write( … );
  }
```

# ISP - Interface Segregation Principle

❌ Violação do ISP

```python
# Interface
class WorkerInterface:
    def work(self):
        pass

    def eat(self):
        pass

    def sleep(self):
        pass
```

```python
class HumanWorker(WorkerInterface):
    def work(self):
        print("Human working...")

    def eat(self):
        print("Human eating lunch...")

    def sleep(self):
        print("Human sleeping...")



class RobotWorker(WorkerInterface):
    def work(self):
        print("Robot working tirelessly...")

    def eat(self):
        raise NotImplementedError("Robots don't eat!")

    def sleep(self):
        raise NotImplementedError("Robots don't sleep!")
```

A classe **RobotWorker** é forçada a implementar os métodos eat() e sleep()

- Isso torna a interface incoerente já que nem todos que implementam a interface dão suporte a todos os seus métodos.

- Violação do ISP já que clientes que usem a **WorkerInterface** podem chamar métodos não suportados

# ISP - Interface Segregation Principle



✅ Aplicando o ISP

```python
# Interfaces
class Workable:
    def work(self):
        pass

class Eatable:
    def eat(self):
        pass

class Sleepable:
    def sleep(self):
        pass
```

```python
class HumanWorker(Workable, Eatable, Sleepable):
    def work(self):
        print("Human working...")

    def eat(self):
        print("Human eating lunch...")

    def sleep(self):
        print("Human sleeping...")

class RobotWorker(Workable):
    def work(self):
        print("Robot working 24/7...")
```

**Solução**:

Organização em interfaces mais granulares que definam somente os métodos necessários.

Desse modo, as classes definem quais interfaces irão implementar.

# ISP - Interface Segregation Principle



❌ Violação do ISP

```python
class Invoice:
    def __init__(self, customer, items):
        self.customer = customer
        self.items = items  # list of (description, quantity, price)

    def calculate_total(self):
        total = 0
        for item in self.items:
            total += item["quantity"] * item["price"]
        return total

    def print_invoice(self):
        print("Invoice for:", self.customer)
        for item in self.items:
            print(f"{item['description']} x {item['quantity']} \
                    = ${item['price'] * item['quantity']}")
        print("Total:", self.calculate_total())

    def save_to_db(self):
        # Escreve no Banco de Dados
        print(f"Saving invoice for {self.customer} to database...")
```

✅ Aplicando o ISP

```python
class Invoice:
    def __init__(self, customer, items):
        self.customer = customer
        self.items = items

    def calculate_total(self):
        return sum(item["quantity"] * item["price"] for item in self.items)


class InvoicePrinter:
    def print(self, invoice):
        print("Invoice for:", invoice.customer)
        for item in invoice.items:
            print(f"{item['description']} x {item['quantity']} \
                    = ${item['price'] * item['quantity']}")
        print("Total:", invoice.calculate_total())


class InvoiceRepository:
    def save(self, invoice):
        # Código de interação com o Banco de dados
        print(f"Saving invoice for {invoice.customer} to database...")
```

# SOLID

**S**ingle Responsibility Principle

**O**pen-Closed Principle

**L**iskov Substitution Principle

**I**nterface Segregation Principle

**D**ependency Inversion Principle

# DIP - Dependency Inversion Principle

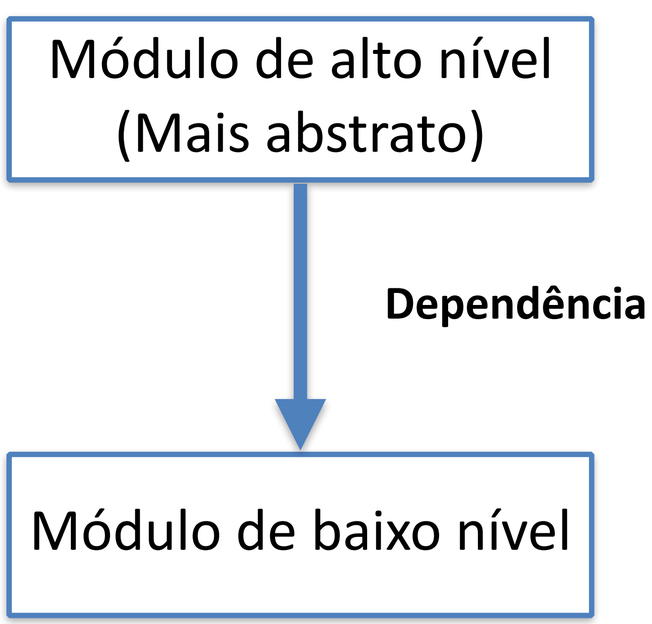


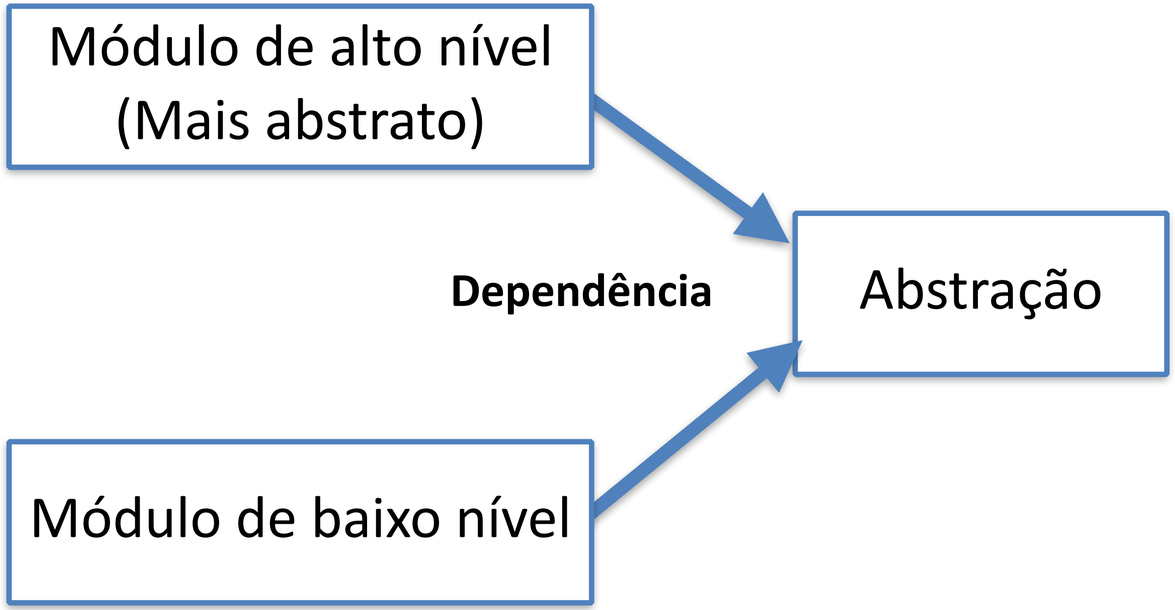
**“High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions”**

- 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.
- Dependa de uma abstração e não de uma implementação.

# DIP - Dependency Inversion Principle





# DIP - Dependency Inversion Principle

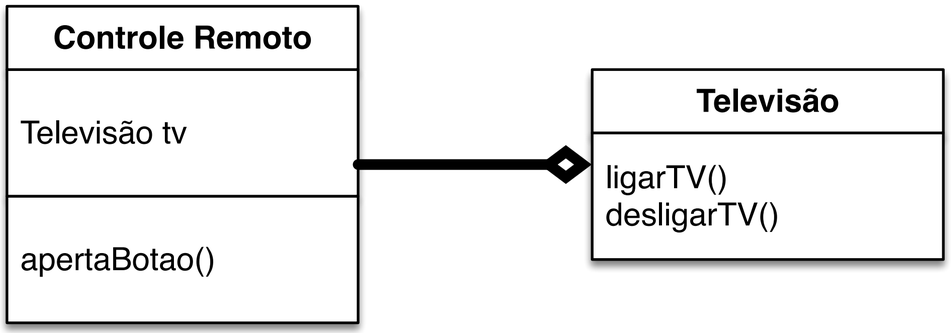


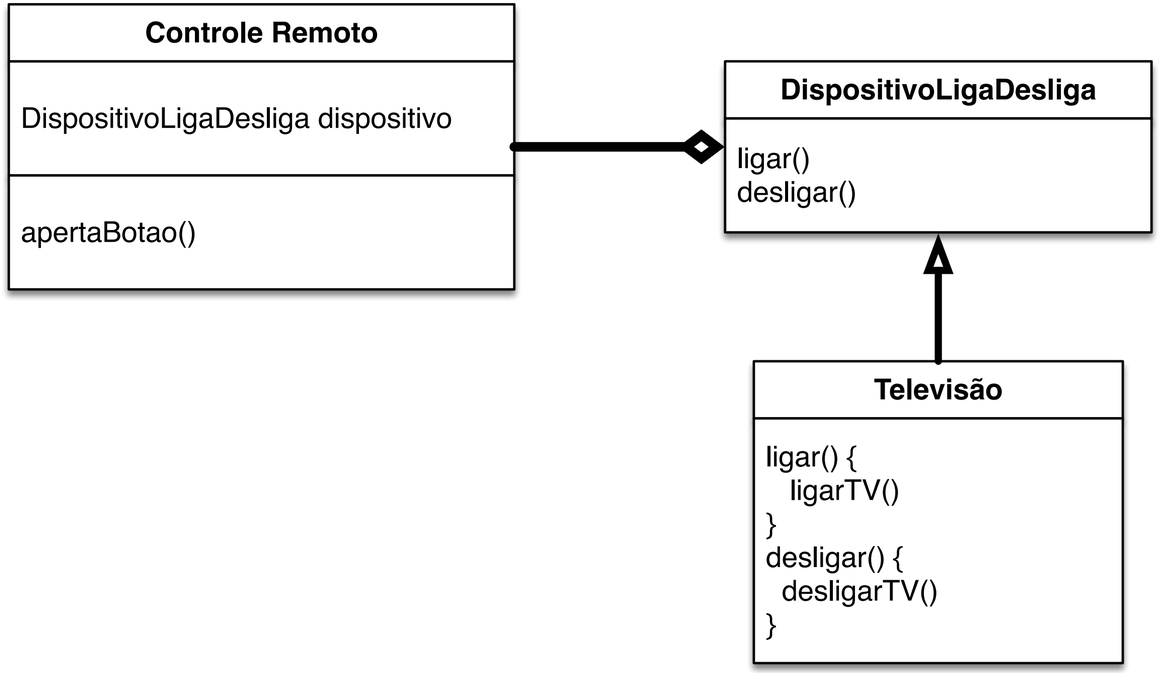
- Ex: a classe de cadastro de cliente e a classe de cadastro de banco de dados. A classe deveria depender da abstração da classe de banco de dados e não da implementação.

- Se eu implemento uma classe diretamente no código de outra eu estou criando um **acoplamento**.

- Qualquer mudança na classe, vai impactar as classes que a utilizam.

# DIP - Dependency Inversion Principle



# DIP - Dependency Inversion Principle

❌ Violação do DIP

```python
class EmailService:
    def send_email(self, to, subject, body):
        print(f"Enviando e-mail para {to}: {subject} - {body}")


class PaymentProcessor:
    def __init__(self):
        self.email_service = EmailService()  # dependência concreta!

    def process_payment(self, customer_email, amount):
        print(f"Processando pagamento de R${amount}...")
        # pagamento feito
        self.email_service.send_email(customer_email,
                "Pagamento recebido", f"Valor: R${amount}")
```

A classe **PaymentProcessor** depende diretamente da implementação concreta EmailService.

- Se amanhã quisermos enviar SMS ou mensagem pelo WhatsApp, teremos que editar **PaymentProcessor**, o que quebra o Princípio Aberto/Fechado e o DIP.

- É **difícil testar** PaymentProcessor isoladamente (precisamos de um EmailService real).

# DIP - Dependency Inversion Principle



✅ Aplicando o DIP

```python
class Notificador:
    def enviar(self, destinatario, mensagem):
        raise NotImplementedError


class EmailNotificador(Notificador):
    def __init__(self, servidor, porta, usuario, senha):
        self.servidor = servidor
        self.porta = porta
        self.usuario = usuario
        self.senha = senha

    def enviar(self, destinatario, mensagem):
        print(f"[EMAIL] Enviando para {destinatario}: {mensagem}")
        # Código real de envio (SMTP, API etc.)


class PagamentoService:
    def __init__(self, notificador: Notificador):
        self.notificador = notificador

    def aprovar_pagamento(self, pedido):
        pedido.status = "aprovado"
        print("Pagamento aprovado.")

        mensagem = f"Olá {pedido.cliente_nome}, seu pedido #{pedido.id} foi aprovado!"
        self.notificador.enviar(pedido.cliente_email, mensagem)
```

A classe **PaymentProcessor** dependia diretamente de EmailService. Agora depende da abstração Notifier.

Para **adicionar um novo canal** (ex: WhatsApp), basta criar uma nova classe que implemente Notifier.

**Fácil de testar** — podemos usar um mock de Notifier.

# Referências


- **Principles of OOD - Robert C. Martin** http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- **Alan Barber** - https://alanbarber.com/2015/08/06/solid-principles-five-principles-of-objectoriented-programming-and-design/
- **Eduardo Pires** - http://eduardopires.net.br/2015/01/solid-teoria-e-pratica/
- Samuel Oloruntoba, **S.O.L.I.D: The First 5 Principles of Object Oriented Design**: https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- http://www.cvc.uab.es/shared/teach/a21291/temes/object_oriented_design/materials_adicionals/principles_and_patterns.pdf

# Outros Princípios Comuns



- Encapsular o que varia muito das área de código mais permanentes.
- Dar preferência a **Composição** ao invés de **Herança**
- Reduzir acoplamentos (***Loose Coupling***)
- Abstração através de **interfaces**.

# Mais Referências

- **Martin Fowler**: https://refactoring.com
- **Filipe Deschamps:**
  **Princípios Sólid:** https://www.youtube.com/watch?v=6SfrO3D4dHM