Design de Software

Princípios SOLID

Baseado no material de Lucas Souto Maior

Habib Asseiss Neto

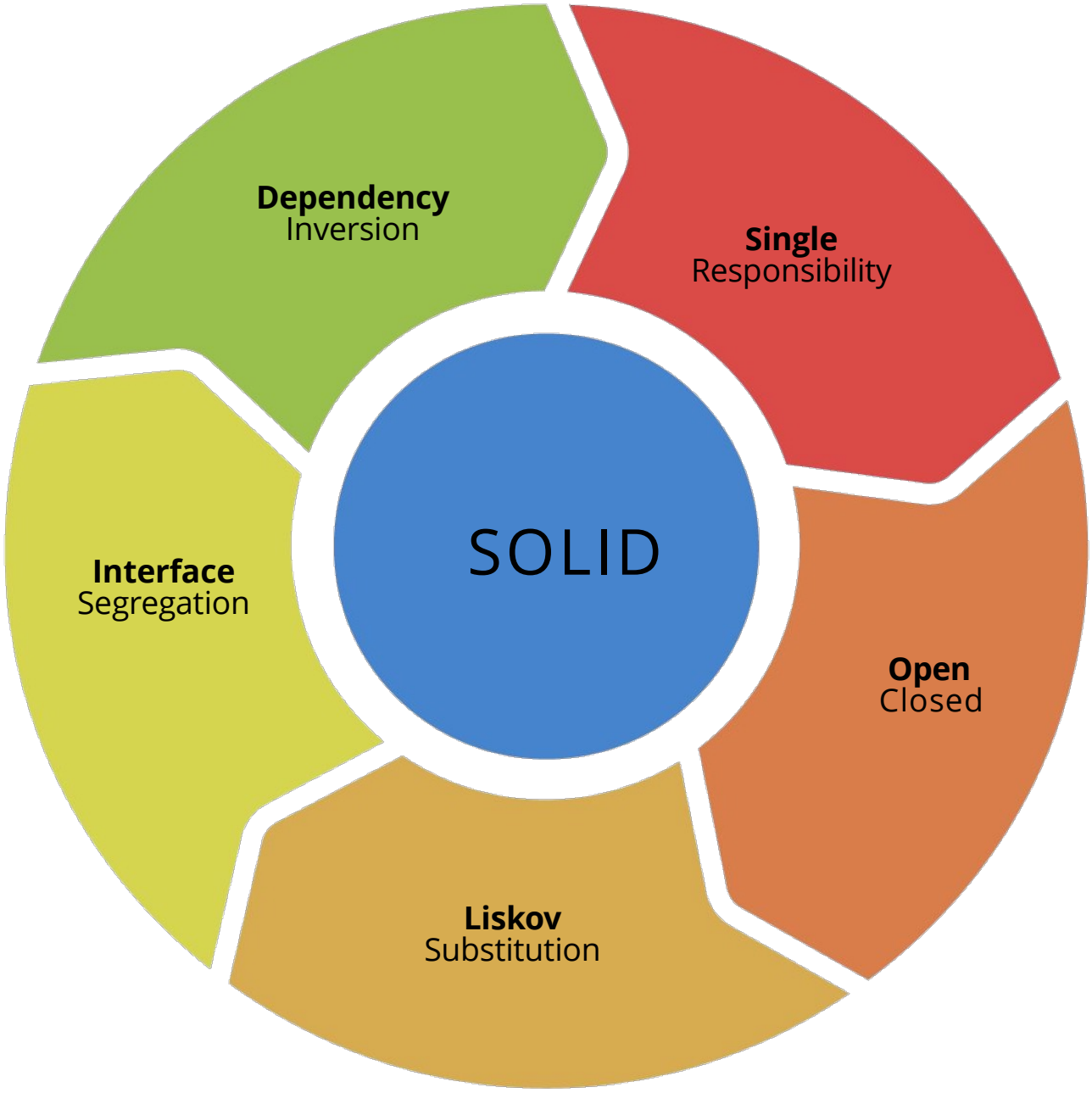
INSTITUTO FEDERAL
Mato Grosso do Sul

Nesta aula

- Design estratégico
- Contextos delimitados
- Microsserviços

SOLID

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


Dependency
Inversion
Single
Responsibility
SOLID
Interface
Segregation
Open
Closed
Liskov
Substitution

Benefícios

- Facilidade em manter, estender, ajustar, testar
- Usar todo o potencial da Programação Orientada a Objetos

Benefícios

- **Manutenção mais fácil**: Código mais modular e independente facilita alterações e correções
- **Flexibilidade**: Facilita a extensão do sistema sem a necessidade de modificar o código existente
- **Reutilização de código**: Componentes mais modulares e focados permitem maior reutilização
- **Redução de bugs**: Menos risco de introduzir problemas ao adicionar novas funcionalidades
- **Facilidade de teste**: A inversão de dependências e a segregação de interfaces facilitam a criação de testes unitários.

Princípios

S - Single responsibility principle

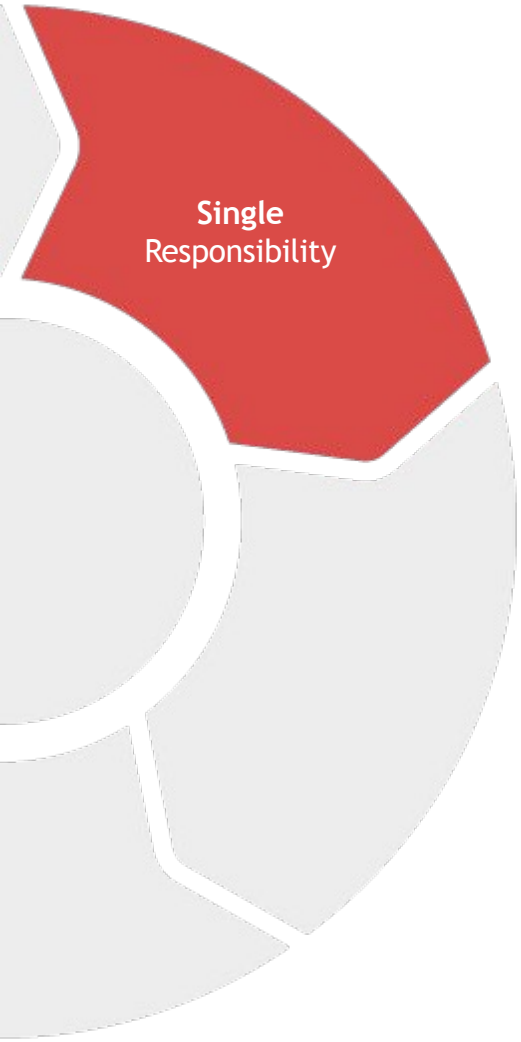
O - Open/closed principle

L - Liskov substitution principle

I - Interface segregation principle

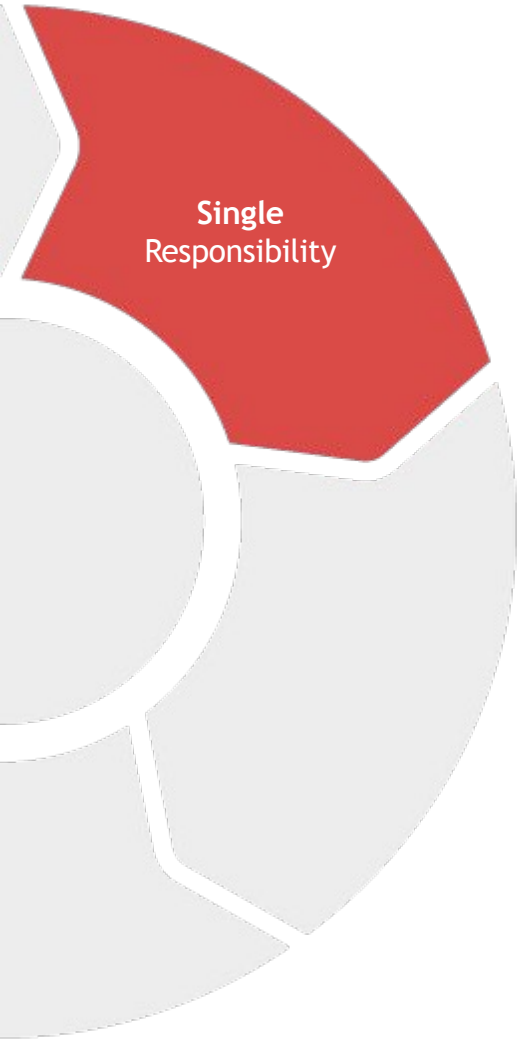
D - Dependency inversion principle

Single responsibility principle



- **“Uma classe deve ter somente uma razão para mudar”**
- Isso se aplica não apenas a classes, mas também a funções, arquivos, etc

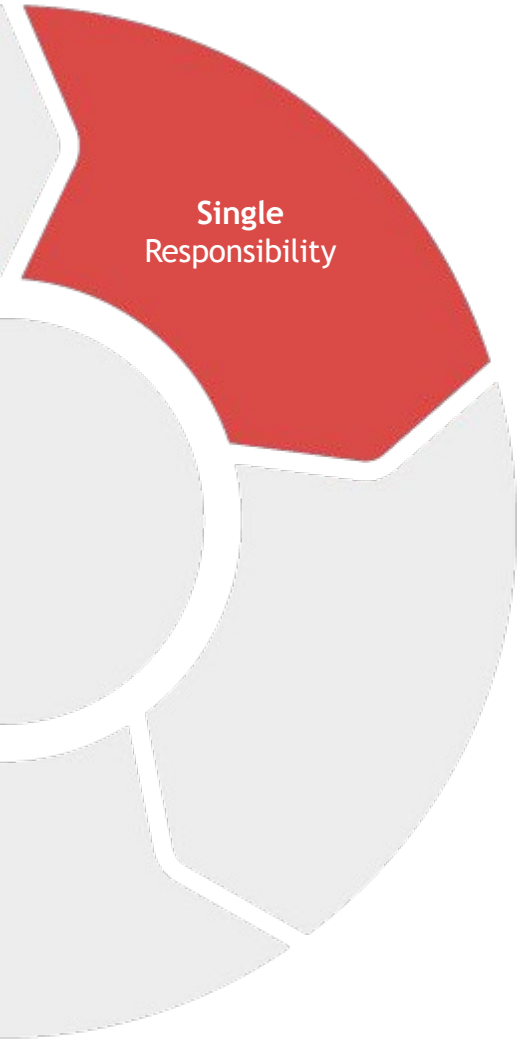
Single responsibility principle




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

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

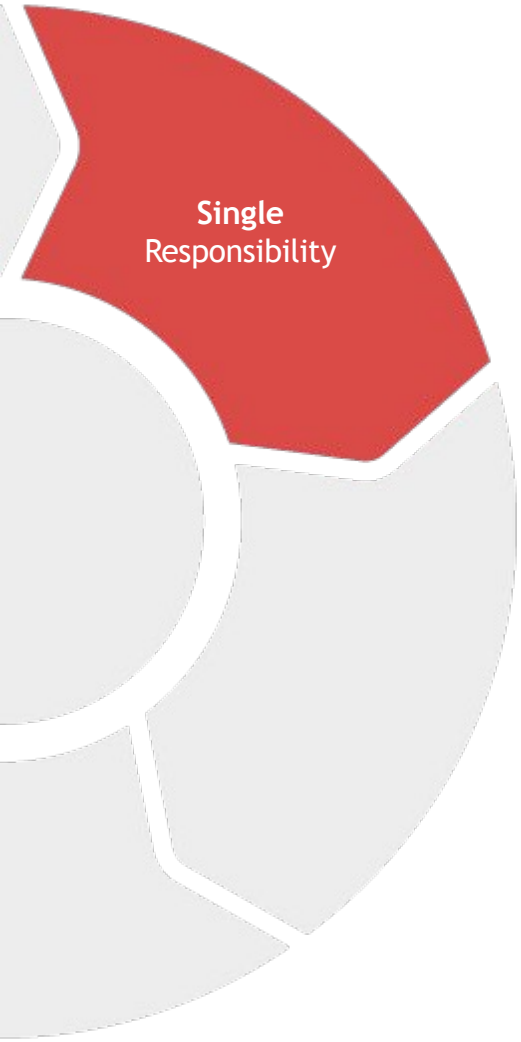
```
public class Aircraft
{
    public void SpeedUp() { /* Code to speed up */ }
    public void Brake() { /* Code to brake */ }
    public void FuelUp() { /* Code to fuel up */ }
}
```

Single responsibility principle



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

Single responsibility principle




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

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

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

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

Exemplo – Antes

Problema: A classe Order tem duas responsabilidades: gerenciar o pedido e salvá-lo em um arquivo. Isso viola o SRP.

```python
class Order:
  def __init__(self, items, total):
    self.items = items
    self.total = total

  def calculate_total(self):
    self.total = sum(item['price'] for item in self.items)

  def save_to_file(self, filename):
    with open(filename, 'w') as file:
      file.write(f"Items: {self.items}\n")
      file.write(f"Total: {self.total}\n")

# Exemplo de uso:
order = Order([{'item': 'Apple', 'price': 5}, {'item': 'Banana', 'price': 3}], 0)
order.calculate_total()
order.save_to_file('order.txt')
```

Exemplo – Depois

Agora, a classe Order é responsável apenas por gerenciar os dados do pedido, enquanto a classe OrderRepository cuida da persistência do pedido. Cada classe tem uma única responsabilidade, respeitando o SRP.

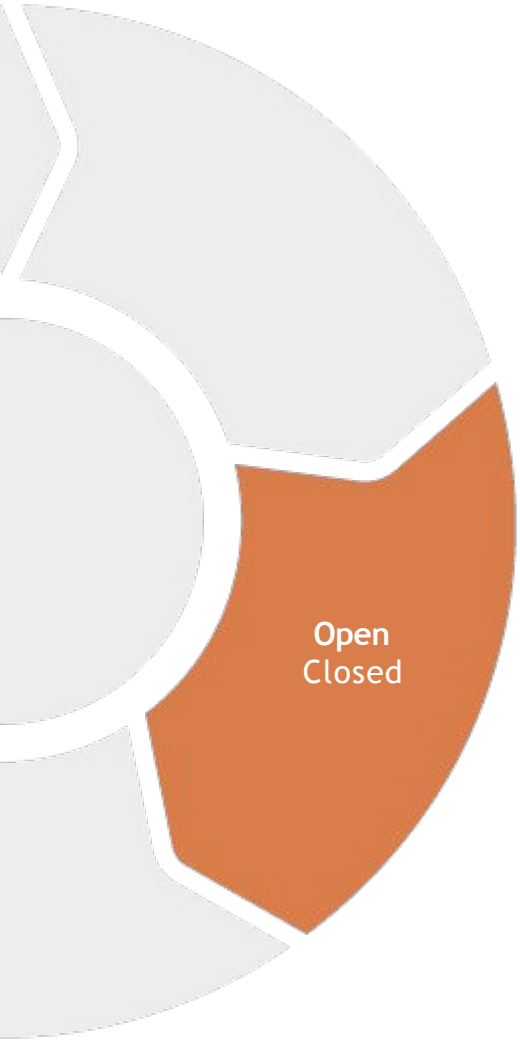
```python
class Order:
  def __init__(self, items, total):
    self.items = items
    self.total = total

  def calculate_total(self):
    self.total = sum(item['price'] for item in self.items)

class OrderRepository:
  def save_to_file(self, order, filename):
    with open(filename, 'w') as file:
      file.write(f"Items: {order.items}\n")
      file.write(f"Total: {order.total}\n")

# Exemplo de uso:
order = Order([{'item': 'Apple', 'price': 5}, {'item': 'Banana', 'price': 3}], 0)
order.calculate_total()

repository = OrderRepository()
repository.save_to_file(order, 'order.txt')
```

Exercício - SRP

A classe `User` a seguir apresenta um problema relacionado ao princípio Single Responsibility. Identifique o problema e modifique o código para que o princípio seja respeitado.

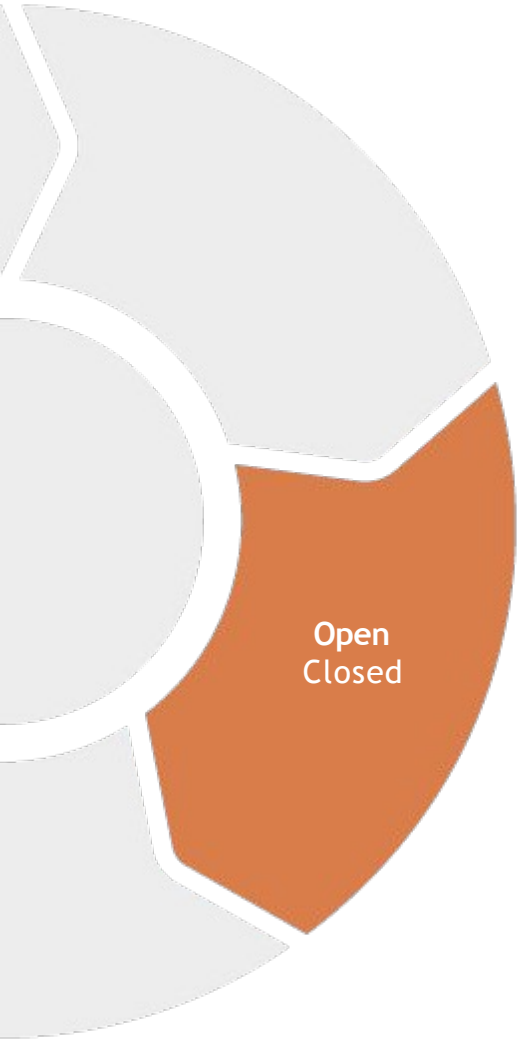
```python
class User:
  def __init__(self, name, email):
    self.name = name
    self.email = email

  def save(self):
    print(f"Saving user {self.name} to the database.")

  def send_welcome_email(self):
    print(f"Sending welcome email to {self.email}.")

# Exemplo de uso:
user = User("Alice", "alice@example.com")
user.save()
user.send_welcome_email()
```

Open/Closed Principle (OCP)



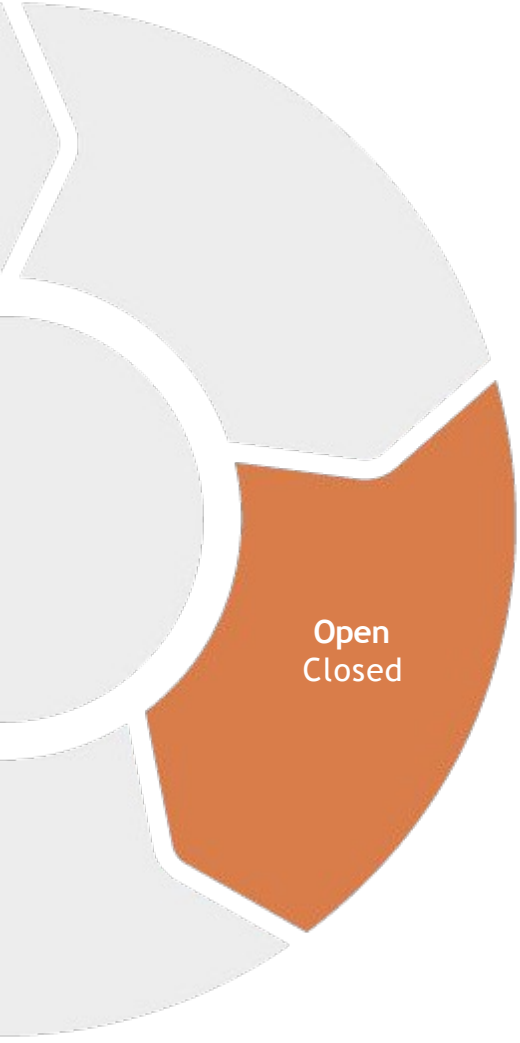
"Entidades de software (classes, módulos, funções etc) devem ser abertas para extensão, mas fechadas para modificação"

Open/Closed Principle (OCP)



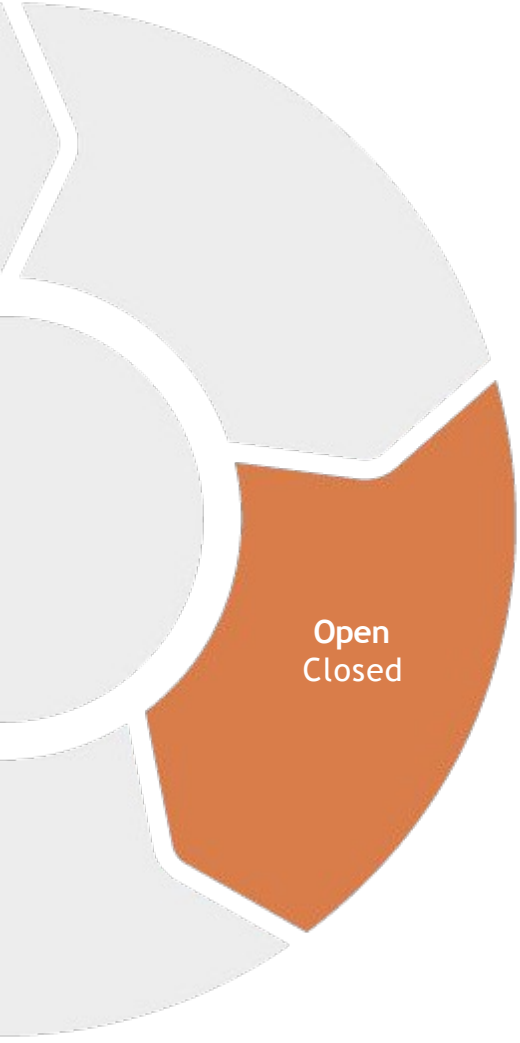
- Problema: **Gostaria de uma classe que efetue pagamentos**
- Preciso checar se há saldo antes de executar o pagamento
- O pagamento pode ser realizado em dinheiro, cartão ou boleto bancário

Open/Closed Principle (OCP)



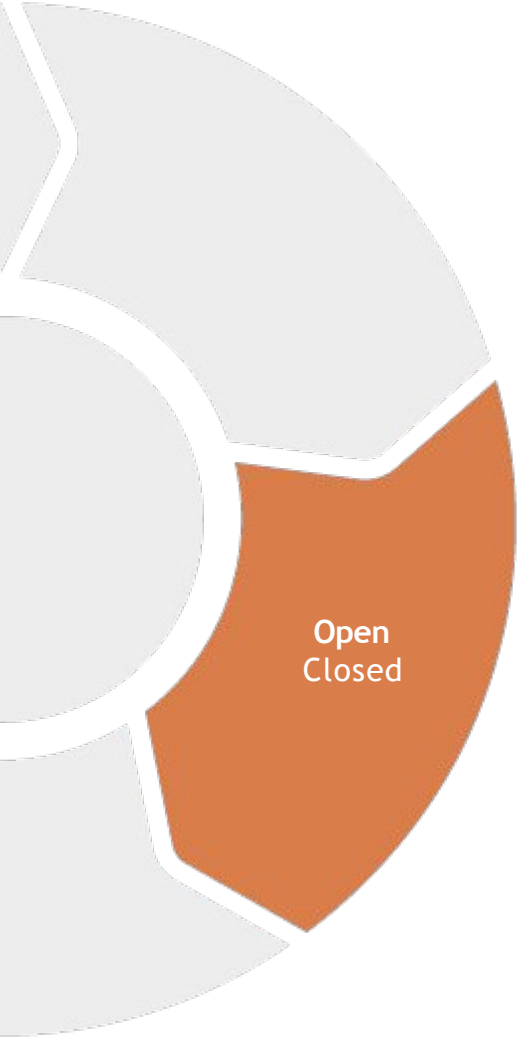
```
public class Payment
{
    private readonly PaymentType paymentType;

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

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

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

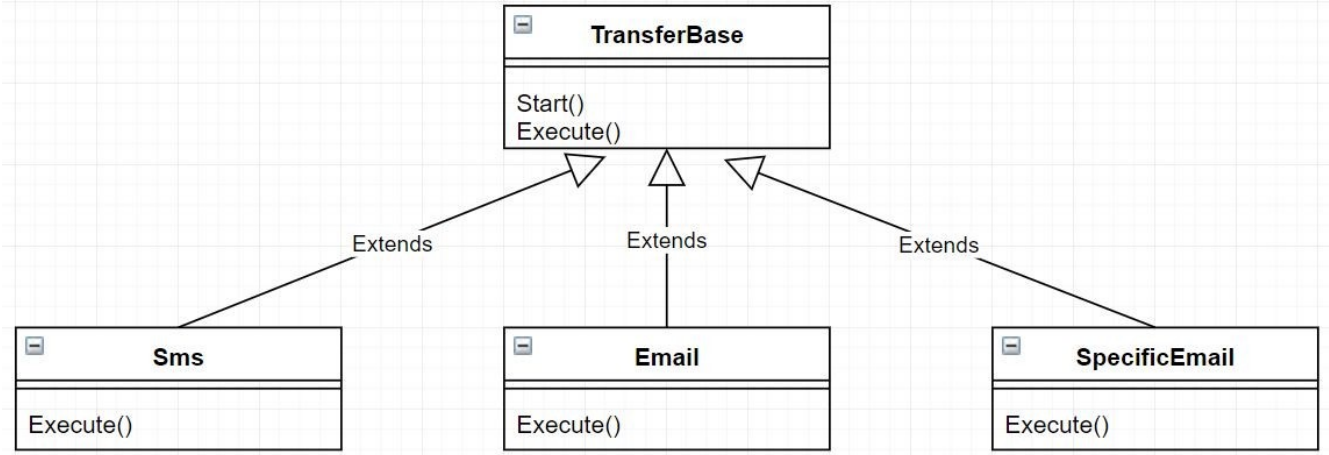
Open/Closed Principle (OCP)



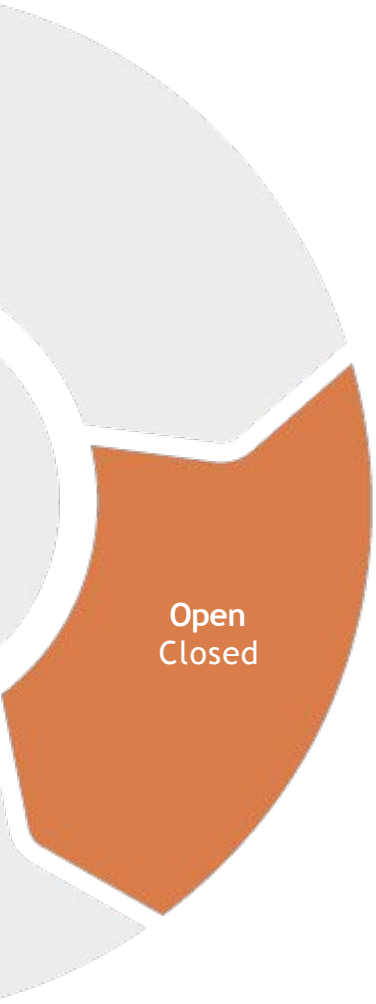
```
public abstract class Payment
{
    protected abstract void Execute();

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

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

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

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

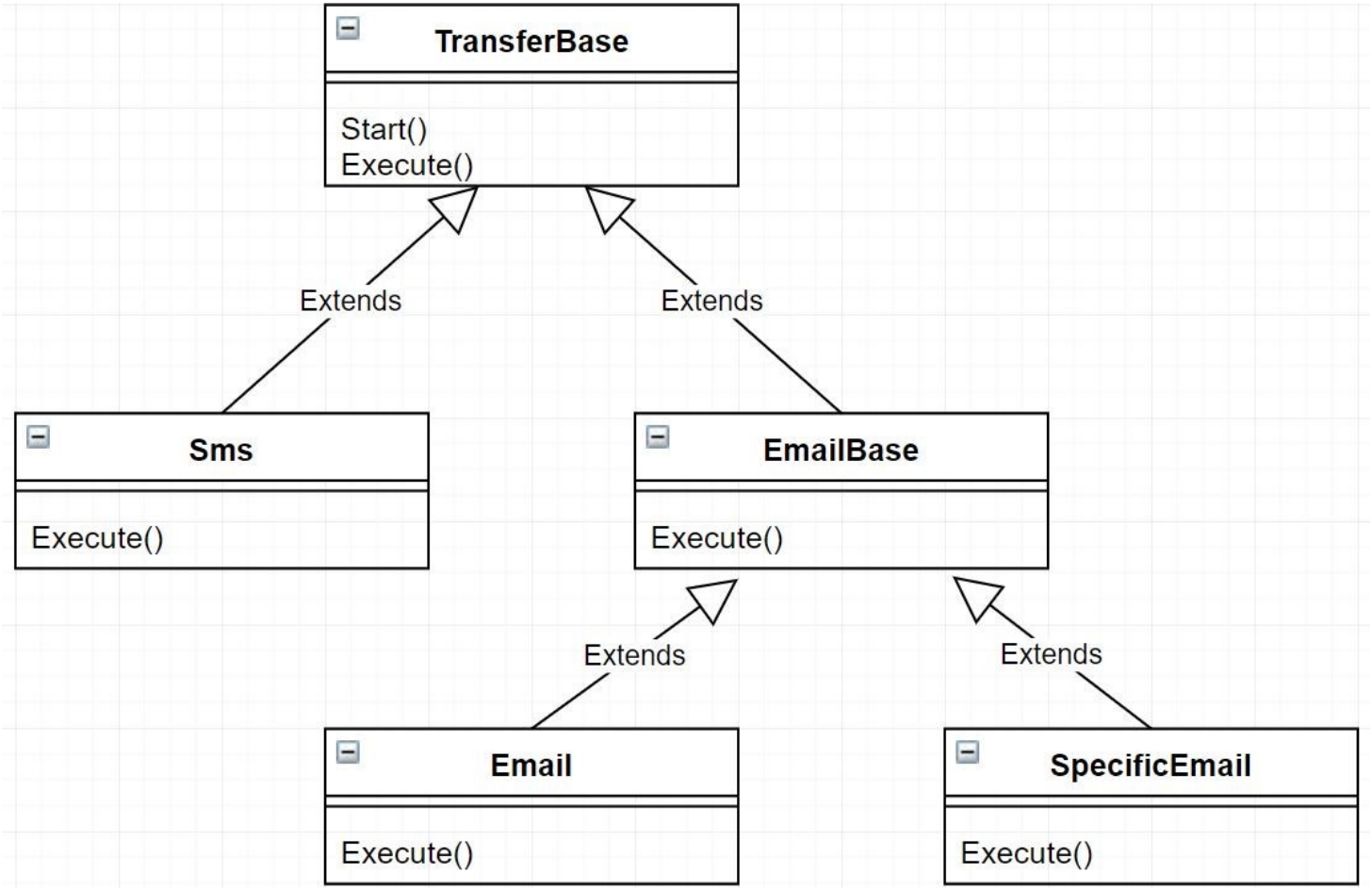
Open/Closed Principle (OCP)





Open/Closed Principle (OCP)





Exemplo – Antes

```python
class Discount:
  def __init__(self, customer_type, price):
    self.customer_type = customer_type
    self.price = price

  def calculate(self):
    if self.customer_type == 'regular':
      return self.price * 0.9
    elif self.customer_type == 'vip':
      return self.price * 0.8
    elif self.customer_type == 'employee':
      return self.price * 0.7
    else:
      return self.price

# Exemplo de uso:
discount = Discount('vip', 100)
print(discount.calculate()) # Saída: 80.0
```

A classe Discount precisa ser modificada sempre que um novo tipo de cliente é adicionado, violando o OCP.

Exemplo – Depois

```python
from abc import ABC, abstractmethod

class Discount(ABC):
  def __init__(self, price):
    self.price = price

  @abstractmethod
  def calculate(self):
    pass

class RegularCustomerDiscount(Discount):
  def calculate(self):
    return self.price * 0.9

class VIPCustomerDiscount(Discount):
  def calculate(self):
    return self.price * 0.8

class EmployeeDiscount(Discount):
  def calculate(self):
    return self.price * 0.7

# Exemplo de uso:
discount = VIPCustomerDiscount(100)
print(discount.calculate()) # Saída: 80.0
```

Agora, o código pode ser estendido para novos tipos de clientes, sem alterar a implementação das classes existentes. Basta criar uma nova classe que herda de Discount e implementar o método calculate.

Exercício - OCP

A classe `ShapeAreaCalculator` a seguir apresenta um problema relacionado ao princípio Open/Closed. Identifique o problema e modifique o código para que o princípio seja respeitado.

```python
class ShapeAreaCalculator:
  def calculate_area(self, shape):
    if shape['type'] == 'rectangle':
      return shape['width'] * shape['height']
    elif shape['type'] == 'circle':
      return 3.14 * shape['radius'] ** 2
    else:
      return None

# Exemplo de uso:
rectangle = {'type': 'rectangle', 'width': 5, 'height': 10}
circle = {'type': 'circle', 'radius': 7}

calculator = ShapeAreaCalculator()
print(calculator.calculate_area(rectangle)) # Saída: 50
print(calculator.calculate_area(circle)) # Saída: 153.86
```

Exercício - OCP

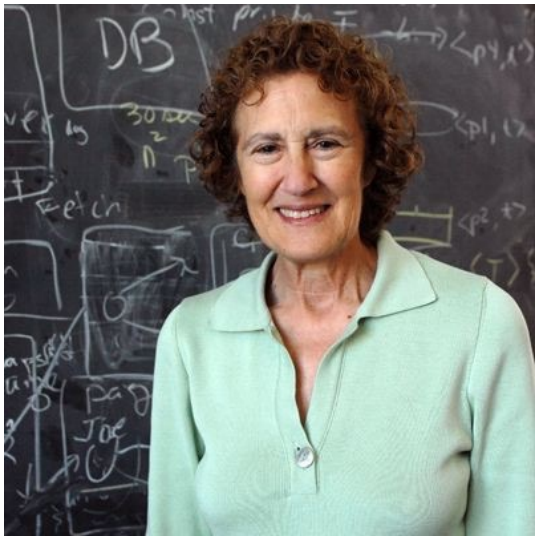
A classe `RelatorioDeVendas` a seguir apresenta um problema relacionado ao princípio Open/Closed. Identifique o problema e modifique o código para que o princípio seja respeitado.

```python
class RelatorioDeVendas:
  def gerar(self, tipo_relatorio):
    if tipo_relatorio == "pdf":
      return "Gerando relatório PDF"
    elif tipo_relatorio == "csv":
      return "Gerando relatório CSV"
    elif tipo_relatorio == "html":
      return "Gerando relatório HTML"
    elif tipo_relatorio == "xlsx":
      return "Gerando relatório em Excel"
```

Liskov Substitution Principle (LSP)

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

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



Liskov Substitution Principle (LSP)

Se algo se parece um pato, nada como um pato e grasna como um pato, então provavelmente é um pato.

O problema do pato...

Liskov Substitution Principle (LSP)

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

    public void Fly() { }
}

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

Liskov Substitution Principle (LSP)

- Poderíamos criar uma classe para patos que voam e outra para patos que não voam, ambas herdando de Pato

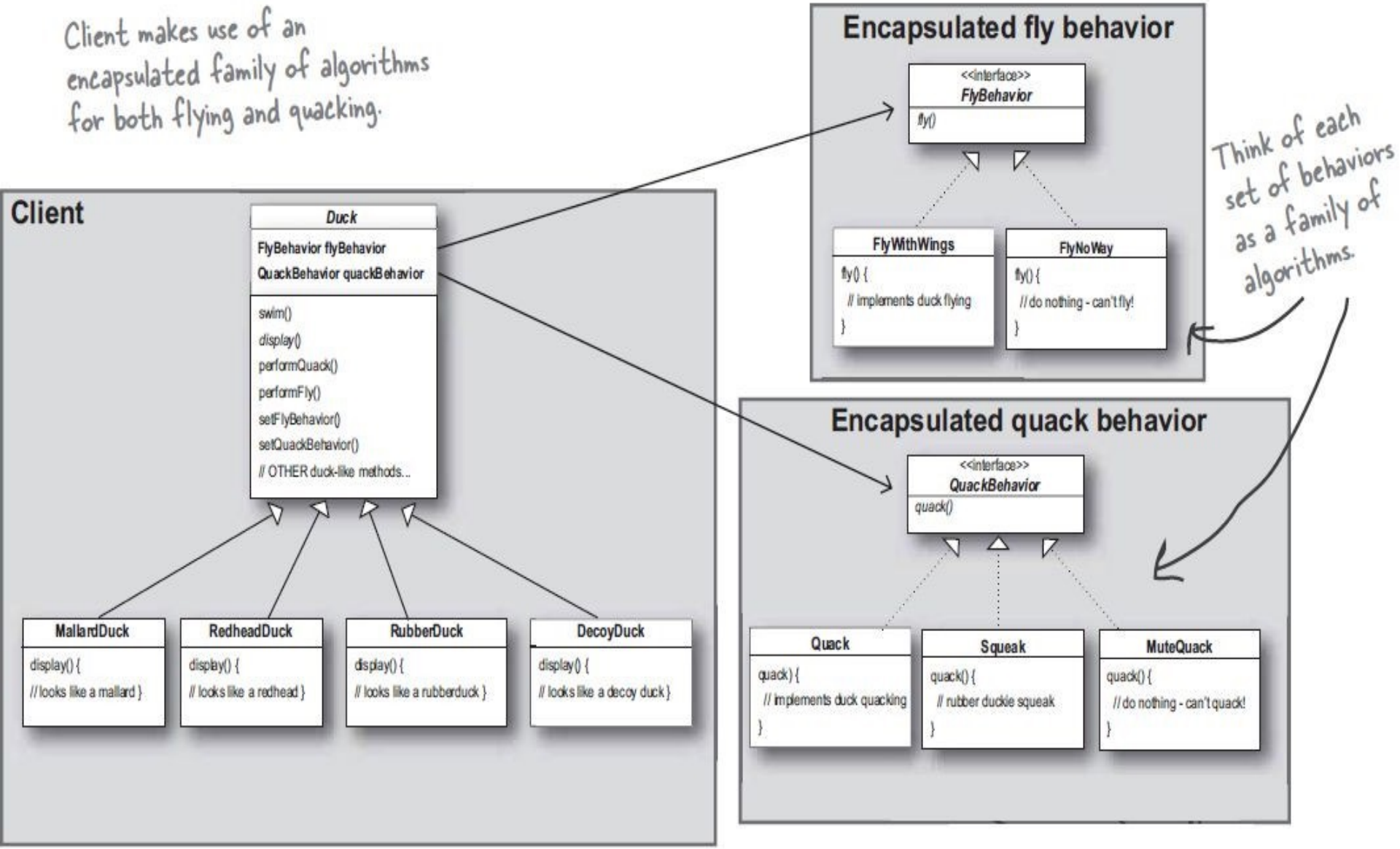
- A classe RubberDuck herdaria da classe “PatosQueNaoVoam” e as demais aves herdariam da classe “PatosQueVoam”

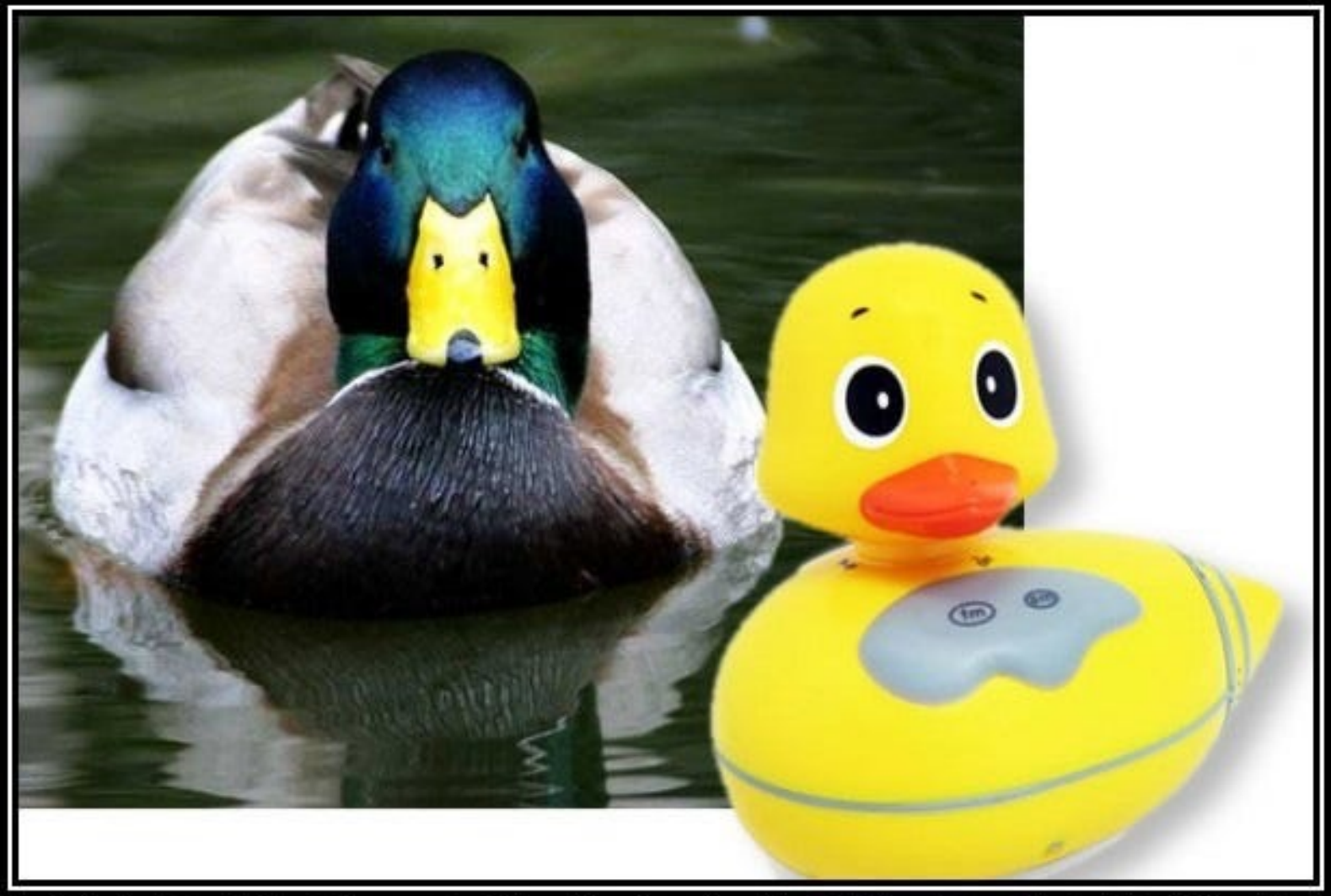
Liskov Substitution Principle (LSP)

- **“Crie suas classes pensando em herança, ou então proíba-a”** (Joshua Bloch)
- No começo das linguagens orientadas a objeto, a herança era usada para vender a ideia
- Mas, utilizar herança pode não ser tão simples
  - É fácil cair em armadilhas criadas por hierarquias de classes mais complexas

Liskov Substitution Principle (LSP)

...surge um pato de madeira.


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Exemplo – Antes

```python
class Bird:
  def fly(self):
    return "I'm flying!"

class Penguin(Bird):
  def fly(self):
    raise Exception("Penguins can't fly")

# Exemplo de uso:
def make_bird_fly(bird: Bird):
  return bird.fly()

sparrow = Bird()
penguin = Penguin()

print(make_bird_fly(sparrow)) # Saída: "I'm flying!"
print(make_bird_fly(penguin)) # Exceção: "Penguins can't fly"
```

A classe Bird é estendida por Penguin, mas a implementação do pinguim não pode voar, o que pode causar problemas se o código depender do método fly de Bird.

Problema: O método fly da classe Penguin quebra o contrato da classe Bird, pois não respeita a expectativa de que todos os pássaros podem voar. Isso viola o LSP.

Exemplo – Depois

```python
from abc import ABC, abstractmethod

class Bird(ABC):
  @abstractmethod
  def move(self):
    pass

class FlyingBird(Bird):
  def move(self):
    return "I'm flying!"

class Penguin(Bird):
  def move(self):
    return "I'm swimming!"

# Exemplo de uso:
def make_bird_move(bird: Bird):
  return bird.move()

sparrow = FlyingBird()
penguin = Penguin()

print(make_bird_move(sparrow)) # Saída: "I'm flying!"
print(make_bird_move(penguin)) # Saída: "I'm swimming!"
```

Para aplicar o LSP, devemos garantir que as classes derivadas possam ser usadas no lugar de suas classes base sem alterar o comportamento esperado do programa. Neste caso, separamos a habilidade de voar em uma interface separada.

Agora, o método move da classe Bird foi generalizado para diferentes tipos de movimento, e o contrato é respeitado em todas as subclasses. O código é mais robusto, pois Penguin pode substituir Bird sem problemas, e não há expectativas quebradas.

Exercício - LSP

Imagine um sistema de pagamento que suporta diferentes métodos, como cartão de crédito e PIX. A classe a seguir quebra o LSP. Identifique o motivo e reescreva o código para que o princípio seja respeitado.

```python
class PaymentProcessor:
  def process_payment(self, amount):
    pass

class CreditCardProcessor(PaymentProcessor):
  def __init__(self, card_number):
    self.card_number = card_number

  def process_payment(self, amount):
    return f"Processed {amount} using Credit Card {self.card_number}"

class PixProcessor(PaymentProcessor):
  def __init__(self, chave):
    self.chave = chave

  def process_payment(self, amount):
    raise Exception("PIX processing is not supported yet")
```

```python
class Notificacao:
  def enviar(self, mensagem):
    pass

class NotificacaoEmail(Notificacao):
  def __init__(self, email):
    self.email = email

  def enviar(self, mensagem):
    return f"Enviado '{mensagem}' via Email para {self.email}"

class NotificacaoSMS(Notificacao):
  def __init__(self, numero_telefone):
    self.numero_telefone = numero_telefone

  def enviar(self, mensagem):
    if len(mensagem) > 160:
      raise ValueError("Mensagens SMS não podem exceder 160 caracteres")
    return f"Enviado '{mensagem}' via SMS para {self.numero_telefone}"


# Exemplo de uso:
def notificar_usuario(notificacao: Notificacao, mensagem):
  return notificacao.enviar(mensagem)

notificacao_email = NotificacaoEmail("usuario@exemplo.com")
notificacao_sms = NotificacaoSMS("555-1234")

print(notificar_usuario(notificacao_email, "Olá via Email"))
print(notificar_usuario(notificacao_sms, "Olá via SMS"))
print(notificar_usuario(notificacao_sms, "A" * 200)) # Exceção: ValueError
```