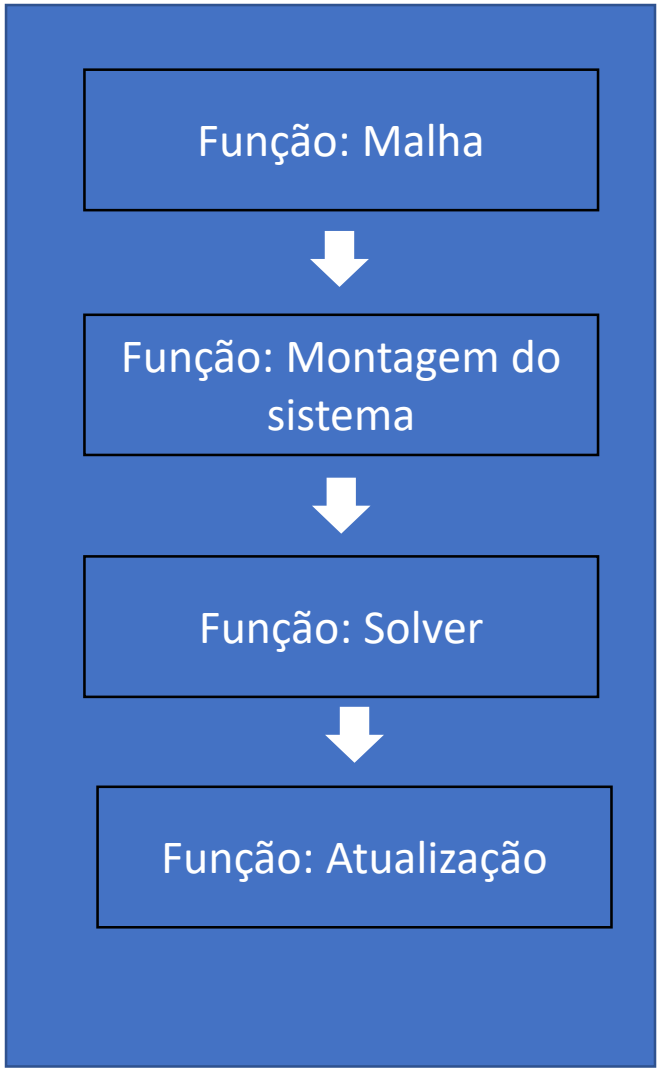
Estrutura de classes (OOP)

Numerical
Research
Academy

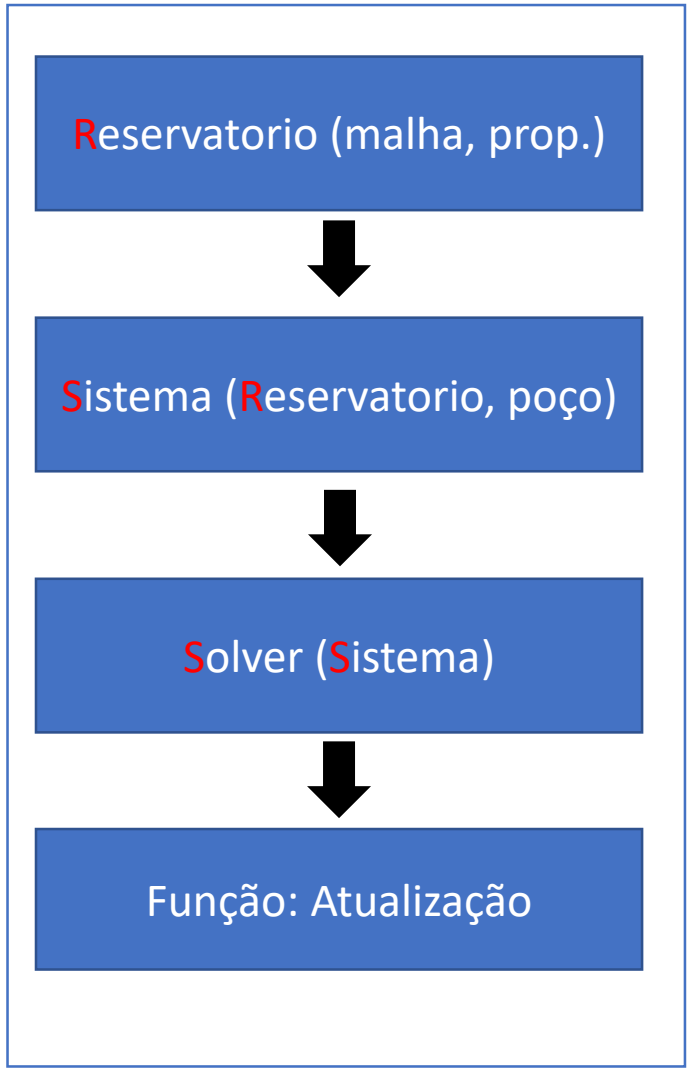
Qual a diferença de um código orientado a objeto?

Programação Estruturada



X

Programação Orientada a Objeto



Benefícios da orientação a objeto

1) Reutilização do código;

É possível utilizar códigos anteriores produzidos por outros colaboradores no intuito de poupar tempo e melhorar a eficiência.

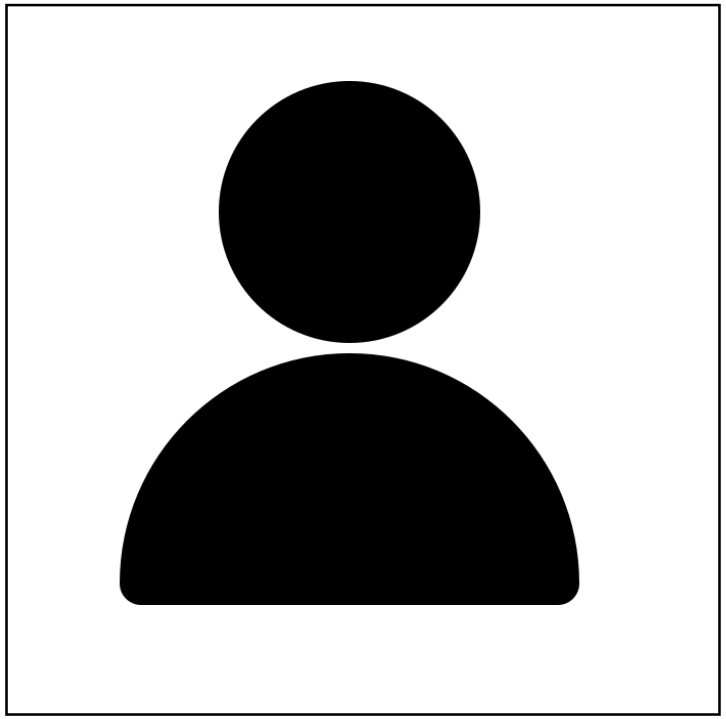
2) Modularidade;

É possível dividir o programa em pequenos módulos facilitando o trabalho em equipe.

3) Organização do programa;

É possível dividir em classes que possuam representatividade com o mundo real, por exemplo: Reservatorio, Equipe.

Criando minha primeira classe em Python



Método construtor

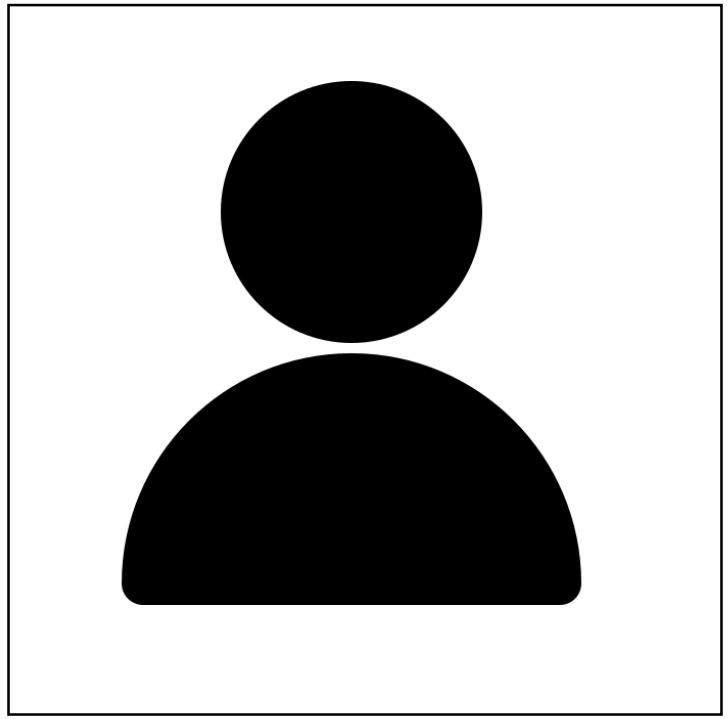
```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
```

Atributos

Instanciando a minha primeira classe em Python


```
cidadao1 = Pessoa( 'Gabi', 26 )
cidadao2 = Pessoa( 'Pedro', 25 )
cidadao3 = Pessoa( 'Giovana', 33 )
```

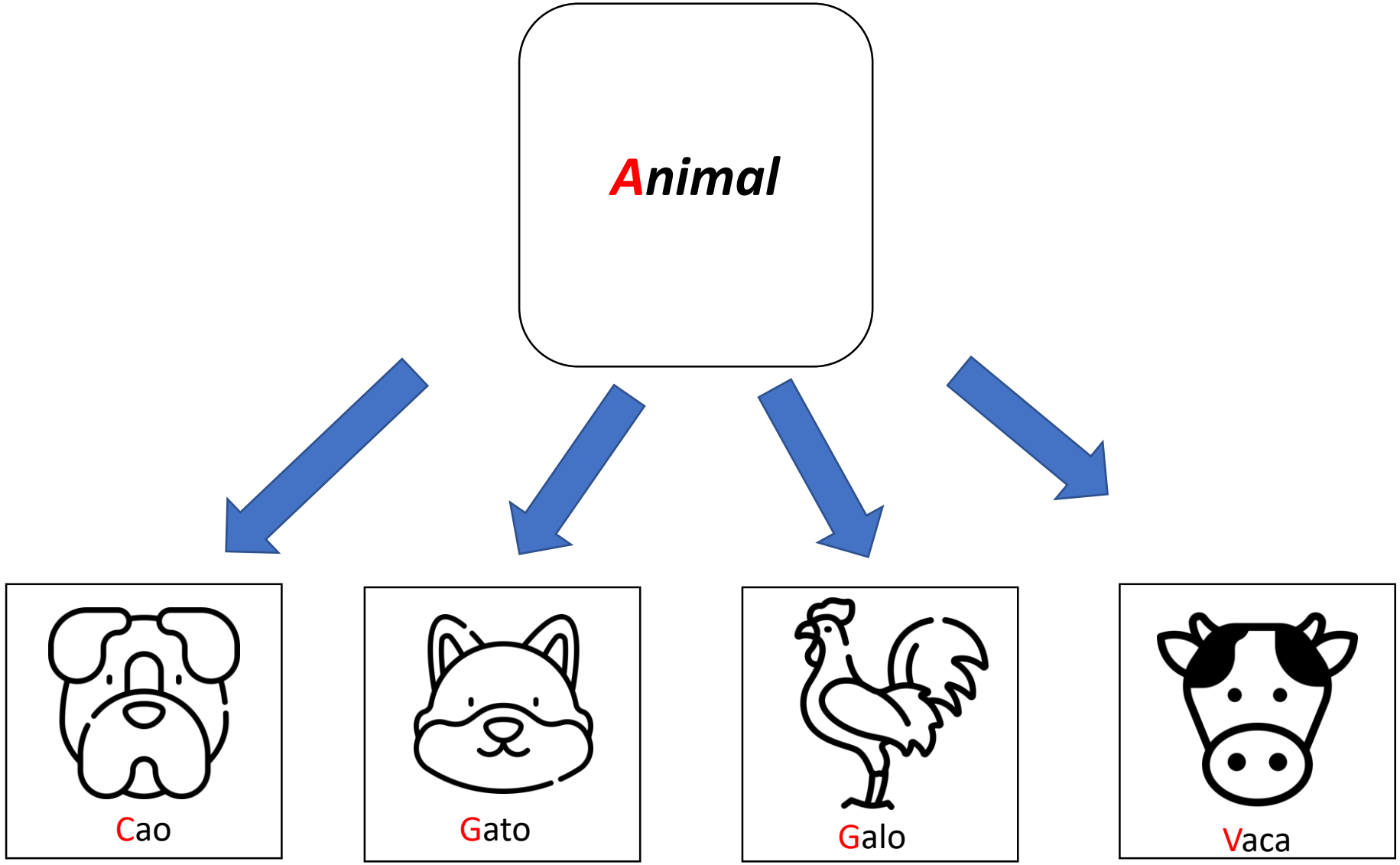
Criando métodos na minha classe



métodos

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def setProfissao(self, profissao):
        self. profissao = profissao

    def get profissao(self):
        return  self.profissão

    def updateProfissao(self, profissao):
        self. profissao = profissao
```

Estrutura de herança na orientação a objetos



Estrutura de herança na orientação a objetos

```
class Animal:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def fazer_barulho(self):
        print("Algum som")
```

```
class Cachorro(Animal):
    def __init__(self, nome, idade, raca):
        super().__init__(nome, idade)
        self.raca = raca

    def fazer_barulho(self):
        print("Au au!")
```

```
class Gato(Animal):
    def __init__(self, nome, idade, cor):
        super().__init__(nome, idade)
        self.cor = cor

    def fazer_barulho(self):
        print("Miau!")
```

```
class Galo(Animal):
    def __init__(self, nome, idade, cor):
        super().__init__(nome, idade)
        self.cor = cor

    def fazer_barulho(self):
        print("Cócóricó!")
```

```
class Vaca(Animal):
    def __init__(self, nome, idade, raca):
        super().__init__(nome, idade)
        self.raca = raca

    def fazer_barulho(self):
        print("Muuuu!")
```

Estrutura de herança na orientação a objetos

```
rex = Cachorro("Rex", 3, "Labrador")
print(rex.nome)
print(rex.idade)
print(rex.raca)
rex.fazer_barulho()
```

```
garfield = Gato("Garfield", 5, "Laranja")
print(garfield.nome)
print(garfield.idade)
print(garfield.cor)
garfield.fazer_barulho()
```

Estrutura de herança na orientação a objetos

```
zeca = Galo(“Zeca", 5, "Laranja")
print(zeca.nome)
print(zeca.idade)
print(zeca.cor)
zeca.fazer_barulho()
```

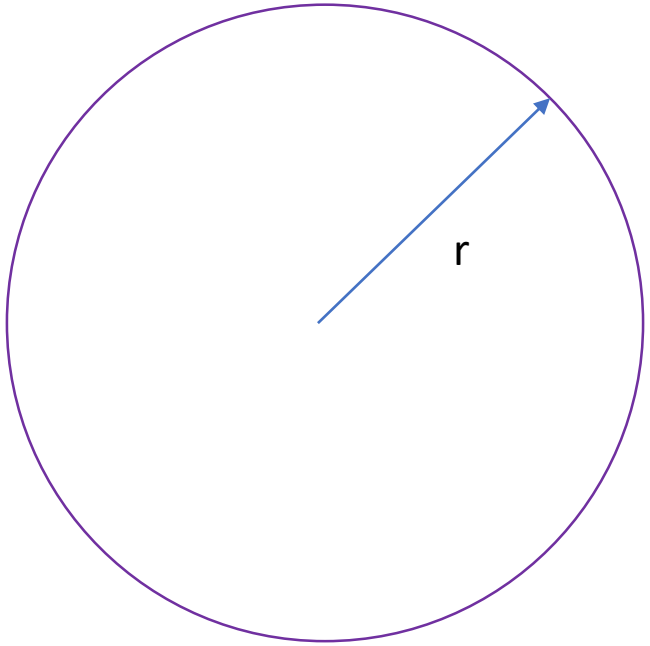
```
mimosa = Vaca(“Mimosa", 5, “Holandesa")
print(mimosa.nome)
print(mimosa.idade)
print(mimosa.cor)
garfield.fazer_barulho()
```

Estrutura de herança na orientação a objetos

```
class Retangulo:
    def __init__(self, altura, comprimento):
        self.altura = altura
        self.comprimento = comprimento

    def calculoArea(self):
        self.area = self.altura * self.comprimento
        return self.area

def diagonal (self):
            self.diagonal = √ (self.altura **2 +  self.comprimento **2)


class Quadrado(Retangulo):
    def __init__(self, lado):
        super().__init__(lado, lado)
```

Utilização do decorador *@staticmethod*

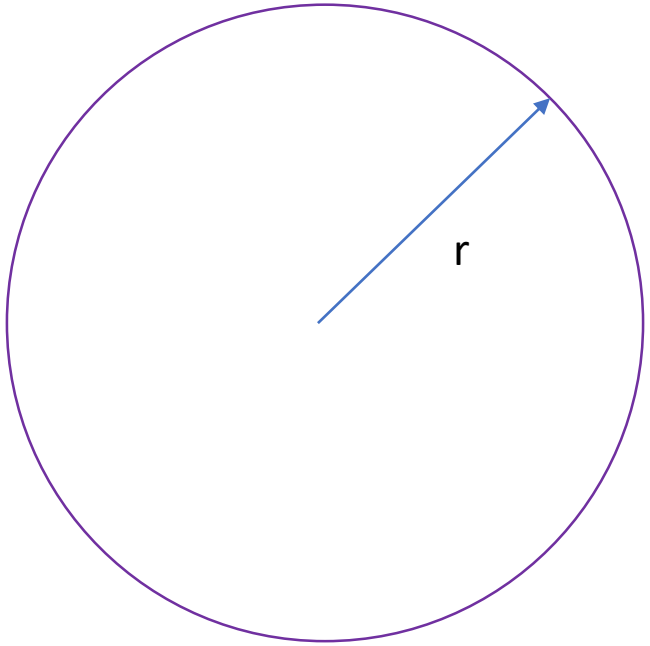
Com a presença desse decorador em uma classe, podemos utilizar uma operação a ele associado sem precisar instanciar toda a classe.




```
class Circulo:
    PI = 3.14159

    def __init__(self, raio):
        self.raio = raio

    def calcular_area(self):
        return self.PI * self.raio ** 2
```

Utilização do decorador *@staticmethod*

Com a presença desse decorador em uma classe, podemos utilizar uma operação a ele associado sem precisar instanciar toda a classe.

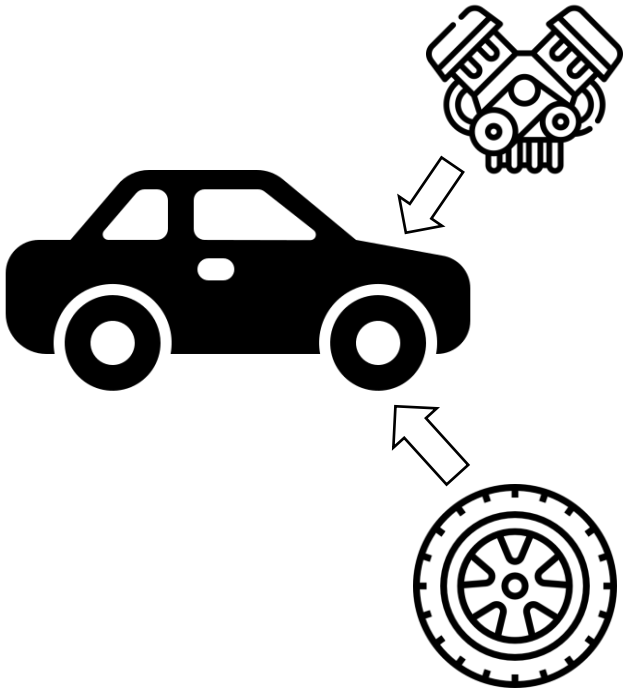



```
class Circulo:
    PI = 3.14159

    def __init__(self, raio):
        self.raio = raio

    @staticmethod
    def obter_pi():
        return Circulo.PI

    def calcular_area(self):
        return obter_pi* self.raio ** 2
```

```
print(Circulo.obter_pi())

# Saída: 3.14159
```

Desse modo a classe pode retornar o valor de Pi sem a necessidade de ser instanciada.

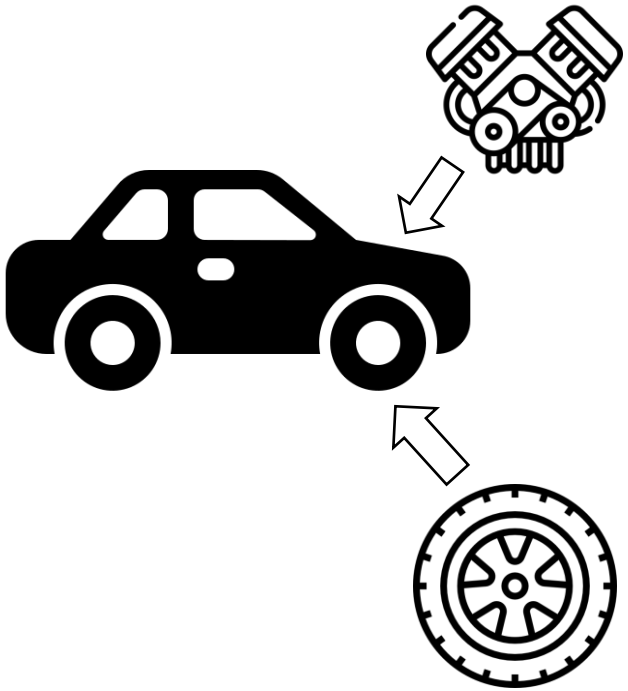
Utilização da técnica de composição



```
class Motor:
    def __init__(self, tipo, potencia):
        self.tipo = tipo
        self.potencia = potencia

class Pneu:
    def __init__(self, largura, tipo):
        self.tipo = tipo
        self.largura = largura

class Carro:
    def __init__(self, modelo, motor , pneu):
        self.modelo = modelo
        self.motor = motor
        self.pneu = pneu

    def acelerar(self):
        print("O carro está acelerando com um motor movido a{} e usando um pneu
para {}.".format(self.motor.tipo, self.pneu.tipo))
```

Os atributos motor e pneu são instancias de outas classes.

Utilização da técnica de composição



```
motor1 = Motor("gasolina", 150)

pneu1 = Pneu('chuva',60)

carro1 = Carro("Fusca", motor1, pneu1)

carro1.acelerar()

# Saída: O carro está acelerando com um motor movido a gasolina e usando um pneu para chuva.
```

Utilizando a estrutura de classes

Suponha que você está desenvolvendo um sistema de gerenciamento de contas bancárias. O sistema deve ser capaz de criar e gerenciar contas correntes e contas poupanças. Ambos os tipos de conta devem ter um número de conta, um titular, um saldo e um limite de saque. As contas correntes devem ter um atributo adicional de "cheque especial", que indica o limite de crédito que o titular pode usar em caso de saldo insuficiente. As contas poupanças devem ter um atributo adicional de "taxa de juros", que representa a taxa de juros anual aplicada ao saldo da conta.

Para implementar isso, você pode usar herança em classes. A classe **ContaBancaria** pode ser a classe pai e ter os atributos comuns de todas as contas bancárias. As classes filhas **ContaCorrente** e **ContaPoupanca** herdam os atributos de **ContaBancaria** e adicionam os atributos específicos de cada tipo de conta.

Utilizando a estrutura de classes

1. Crie uma classe **ContaBancaria** com os atributos **numero_conta**, **titular**, **saldo** e **limite_saque**. Implemente os métodos **depositar(valor)** e **sacar(valor)** para adicionar e remover dinheiro da conta. O método sacar(valor) deve verificar se o saldo é suficiente para o saque e se o valor do saque é menor ou igual ao limite de saque.

2. Crie uma classe **ContaCorrente** que herda de **ContaBancaria** e adiciona o atributo **cheque_especial**. Implemente o método **usar_cheque_especial(valor)** para permitir que o titular da conta use o limite de crédito caso o saldo seja insuficiente para o saque. Esse método deve verificar se o valor do saque mais o limite de crédito é menor ou igual ao limite de saque.

3. Crie uma classe **ContaPoupanca** que herda de **ContaBancaria** e adiciona o atributo **taxa_juros**. Implemente o método **calcular_juros()** para calcular o juros da conta poupança. Esse método deve adicionar ao saldo o resultado da multiplicação do saldo pela taxa de juros anual dividido por 12.

4. Crie objetos de cada tipo de conta e teste os métodos implementados.

Utilizando a estrutura de classes (GABARITO)

```
class ContaBancaria:
    def __init__(self, numero_conta, titular, saldo, limite_saque):
        self.numero_conta = numero_conta
        self.titular = titular
        self.saldo = saldo
        self.limite_saque = limite_saque

    def depositar(self, valor):
        self.saldo += valor

    def sacar(self, valor):
        if valor <= self.limite_saque and valor <= self.saldo:
            self.saldo -= valor
            print("Saque realizado.")
        else:
            print("Saldo insuficiente ou limite de saque excedido.")
```

Utilizando a estrutura de classes (GABARITO)

```
class ContaCorrente(ContaBancaria):
    def __init__(self, numero_conta, titular, saldo, limite_saque, cheque_especial):
        super().__init__(numero_conta, titular, saldo, limite_saque)
        self.cheque_especial = cheque_especial

    def usar_cheque_especial(self, valor):
        if valor <= self.limite_saque + self.cheque_especial and valor <= self.saldo + self.cheque_especial:
            self.saldo -= valor
            print("Saque realizado com cheque especial.")
        else:
            print("Limite de cheque especial excedido.")

class ContaPoupanca(ContaBancaria):
    def __init__(self, numero_conta, titular, saldo, limite_saque, taxa_juros):
        super().__init__(numero_conta, titular, saldo, limite_saque)
        self.taxa_juros = taxa_juros

    def calcular_juros(self):
        juros = self.saldo * (self.taxa_juros / 12)
        self.saldo += juros
```

Utilizando a estrutura de classes (GABARITO)

```
# Teste das classes
conta_corrente = ContaCorrente(12345, "Fulano de Tal", 1000.00, 500.00, 2000.00)
conta_corrente.usar_cheque_especial(2000.00)
conta_corrente.sacar(600.00)
print("Saldo conta corrente:", conta_corrente.saldo)

conta_poupanca = ContaPoupanca(67890, "Ciclano de Tal", 5000.00, 1000.00, 0.05)
conta_poupanca.depositar(1000.00)
conta_poupanca.calcular_juros()
print("Saldo conta poupança:", conta_poupanca.saldo)
```