UNIVERSIDADE FEDERAL DO CEARÁ
Departamento de Computação
Prof. Lincoln Souza Rocha
CK0442- Técnicas de Programação para Ciência de Dados - T02

TP04
2024.2

1. **Implementando a classe ContaImposto via Herança**

   Descrição: Altere o módulo `sisbanco.py` (melhorado no TP03) para incluir a classe `ContaImposto` seguindo a especificação abaixo. Lembre-se, o método `debitar()` da classe `Conta` deve ser redefinido na classe `ContaImposto` para implementar o comportamento de taxar a operação de débito. (**Dica**. A Nota de Aula 07 descreve a implementação completa da classe `ContaImposto`).

```
class ContaImposto(Conta):
    def __init__(self, numero:str):
        pass

    def debitar(self, valor:float) -> None:
        pass

    def get_taxa(self) -> float:
        pass

    def set_taxa(self, taxa:float) -> None:
        pass
```

2. **Implementando a classe ContaAbstrata**

   Descrição: Altere o módulo `sisbanco.py` (melhorado no TP03) para incluir a classe `ContaAbstrata` seguindo a especificação abaixo. Lembre-se que a classe `ContaAbstrata` possui um método abstrato, `debitar()`. (**Dica**. A Nota de Aula 07 descreve a implementação completa da classe `ContaAbstrata`).

```
class ContaAbstrata(ABC):
    def __init__(self, numero:str):
        pass

    def creditar(self, valor:float) -> None:
        pass

    @abstractmethod
    def debitar(self, valor:float) -> None:
        pass

    def get_numero(self) -> str:
        return self.__numero

    def get_saldo(self) -> float:
        return self.__saldo
```

3. **Redefinindo a Hierarquia de Herança das classes Conta e ContaImposto**

Descrição: No módulo `sisbanco.py`, altere as classes `Conta` e `ContaImposto` para que eleas herdem da classe `ContaAbstrata` seguindo as especificações abaixo. Lembre-se que ambas as classes devem prover uma implementação para o método abstrato `debitar()` da classe `ContaAbstrata`. (**Dica**. A Nota de Aula 07 descreve a implementação completa das classes `Conta` e `ContaImposto`).

```
class Conta(ContaAbstrata):
    def __init__(self, numero:str):
        pass

    def debitar(self, valor:float) -> None:
        pass

class ContaImposto(ContaAbstrata):
    def __init__(self, numero:str):
        pass

    def debitar(self, valor:float) -> None:
        pass

    def get_taxa(self) -> float:
        pass

    def set_taxa(self, taxa:float) -> None:
        pass
```

4. **Aplicando Taxa de Imposto no SisBanco**

Descrição: No módulo `sisbanco.py`, modifique a classe Banco para que seja possível alterar a taxa de imposto para operações de débito em objetos da classe `ContaImposto`. A taxa de imposto do Banco deve deve ser um atributo privado da classe Banco. Além disso, crie métodos para alterar e recuperar o valor da taxa de imposto (veja a descrição abaixo). Observe que houveram alterações em partes do código relativo relativo a manipulação da taxa de correção da poupança para evitar confusão entre a taxa de correção da poupança e a taxa de imposto.

```
class Banco:
    def __init__(self, taxa_poupanca:float=0.001, taxa_imposto:float=0.001):
        pass
    (...)

    def get_taxa_poupanca(self) -> float:
        pass

    def set_taxa_poupanca(self, taxa:float) -> None:
        pass

    def get_taxa_imposto(self) -> float:
        pass

    def set_taxa_imposto(self, taxa:float) -> None:
        pass
```

5. **Interagindo com o Sistema Bancário**

Descrição: Altere o módulo `terminal_atendimento.py` para trabalhar com a classe `ContaImposto` no SisBanco. A descrição abaixo fornece os detalhes das novas funcionalidades que devem ser implementadas (i.e., opções 0 e 8 do menu principal).

```
def terminal():
    sisbanco = Banco()
    while(True):
        print("SisBanco::Bem-Vindo!")
        print(".::Opcoes::.")
        print("[0]-Cadastrar_Conta")
        print("[1]-Creditar")
        print("[2]-Debitar")
        print("[3]-Transferir")
        print("[4]-Consultar_Saldo")
        print("[5]-Render_Juros")
        print("[6]-Render_Bonus")
        print("[7]-Alterar_Taxa_Juros")
        print("[8]-Alterar_Taxa_Imposto")
        print("[9]-Sair")
        opcao = input("Digite:")

        if opcao == 0:
            #qual tipo de conta a ser criada:
            #S - Simples  | P - Poupanca | E - Especial | I - Imposto
            #solicite o numero da conta a ser criada
            #instancie uma conta do tipo selecionado com esse numero
            #cadastre a conta no sisbanco

        elif opcao == 1:
            #solicite o numero da conta alvo
            #solicite o valor a ser creditado
            #realize a operacao de credito no sisbanco

        elif opcao == 2:
            #solicite o numero da conta alvo
            #solicite o valor a ser debitado
            #realize a operacao de debito no sisbanco

        elif opcao == 3:
            #solicite o numero da conta origem
            #solicite o numero da conta destino
            #solicite o valor a ser transferido
            #realize a operacao de transferencia no sisbanco

        elif opcao == 4:
            #solicite o numero da conta alvo
            #realize a operacao de obtencao de saldo no sisbanco
            #exiba o saldo na tela

        elif opcao == 5:
            #solicite o numero da conta alvo
            #realize a operacao correcao da poupanca no sisbanco

        elif opcao == 6:
            #solicite o numero da conta alvo
            #realize a operacao conversao/rendimento de bonus no sisbanco

        elif opcao == 7:
```

```
            #solicite a nova taxa de correcao da poupanca
            #realize a operacao de alteracao da taxa no sisbanco

        elif opcao == 8:
            #solicite a nova taxa de imposto
            #realize a operacao de alteracao da taxa no sisbanco

        elif opcao == 9:
            print("SisBanco::Bye!")
            break

if __name__ == "__main__":
    terminal()
```

6. **Decorando a Operação Dividir**

Descrição: Utilizando o conceito de decoradores, crie o módulo `play_decorador.py` e implemente o que se pede. Considere uma operação de divisão como descrita na especificação abaixo (`dividir()`). Agora, considere a função `divisao_inteligente()` que recebe como argumento outra função (`func`). A função `divisao_inteligente()` funcionará como um decorador para a função `dividir()`. Assim, dentro da função interna `wrapper(x,y)`, onde `x` casa com `operando_a` e `y` casa com `operando_b`, você deve implementar a seguinte lógica: (i) inicialmente `print("Dividindo:{}/{}".format(x,y))` – (ii) verifique se `y == 0.0`, em caso verdadeiro retorne `"ERRO_DIV_POR_ZERO"`, caso contrário retorne `func(x,y)`. Lembre-se de anotar a função `dividir()` com o decorador (`@divisao_inteligente`).

```
def divisao_inteligente(func):
    def wrapper(x,y):
        (...)
    return wrapper

def dividir(operando_a:float, operando_b:float) -> None:
  return operando_a/operando_b

if __name__ == "__main__":
    print("->Resultado: {}\n".format(dividir(3,3)))
#saida esperada:
#Dividindo:3/3
#->Resultado: 1.0

    print("->Resultado: {}\n".format(dividir(3,0)))
#saida esperada:
#Dividindo:3/0
#->Resultado: ERRO_DIV_POR_ZERO
```