Boas Práticas de Programação e Escrita de Documentação Técnica

1. Introdução

Nesta atividade laboratorial, o foco será a adoção e aplicação de boas práticas de programação escrita de documentação técnica. Estes aspetos são cruciais para garantir que o código desenvolvido é de alta qualidade, fácil de manter e compreensível por outros programadores e utilizadores. A atividade permitirá aos estudantes aplicar convenções de codificação, apreender a elaborar documentação de código e APIs, utilizar comentários eficazes para manter o código legível, e desenvolver documentação técnica e manuais de utilizador de forma estruturada e detalhada.

O cenário proposto é um fragmento de código já existente que necessita de ser melhorado e devidamente documentado. Os estudantes irão trabalhar em várias etapas, desde a organização do código segundo convenções estabelecidas, até à criação de documentação técnica completa. O trabalho será desenvolvido em grupo (3 elementos), de modo a promover a colaboração e o alinhamento das práticas de programação.

2. Objetivos e competências a desenvolver

Ao concluir esta atividade laboratorial, os estudantes deverão ser capazes de:

- Aplicar convenções de codificação para melhorar a clareza e consistência do código.
- Elaborar documentação de código eficaz, incluindo a documentação de APIs, que permita a outros programadores compreender e utilizar o sistema.
- Utilizar comentários úteis e estratégicos para explicar lógica complexa, sem sobrecarregar o código com informações desnecessárias.
- Criar uma documentação técnica clara a concisa, incluindo um manual de utilizador, para facilitar a utilização e manutenção futura do sistema.
- Refletir sobre a importância das boas práticas de programação e documentação no contexto de desenvolvimento de software.

3. Instruções

A atividade será divida em cinco partes:

Parte 1: Convenções de codificação

Objetivo: Garantir que o código segue as boas práticas de codificação, com nomeação adequada de variáveis, funções e classes, e com formatação e estruturação consistentes, assegurando a legibilidade e a uniformidade.

Tarefas:

1. Analisar e reestruturar o fragmento de código fornecido, aplicando convenções de codificação apropriadas. Estas incluem:

1.1. Utilização de boas práticas na nomeação de variáveis e funções (e.g., camelCase, PascalCase).
1.2. Correta indentação, espaçamento e estrutura de blocos de código.
1.3. Organização adequada de funções e métodos.

Parte 2: Documentação de código e de API

Objetivo: Criar documentação de código clara e eficaz, facilitando o entendimento por parte de outros programadores. Os estudantes devem documentar as funções e classes de uma aplicação, explicando a sua lógica, retornos e possíveis exceções.

Tarefas:

1. Documentar o código fornecido, utilizando ferramentas como Doxygen, Sphinx ou outra de documentação automática.
2. Cada função ou classe deve incluir:
   2.1. Descrição clara da funcionalidade.
   2.2. Explicação de parâmetros de entrada e tipos de dados.
   2.3. Valor de retorno.
   2.4. Erros ou exceções que podem ser gerados.
   2.5. Exemplos de uso (se aplicável).

Parte 3: Comentários úteis e manutenção de código legível

Objetivo: Melhorar a legibilidade do código ao adicionar comentários úteis, que expliquem o raciocínio por trás de decisões técnicas ou lógicas complexas, facilitando a manutenção do código.

Tarefas:

1. Analisar o código fornecido e adicionar comentários úteis que expliquem:
   1.1. Lógica de loops ou condições complexas.
   1.2. Justificações para escolhas de algoritmos ou estruturas de dados.
   1.3. Comportamento especial ou exceções de código.
2. Garantir que os comentários são concisos e relevantes, evitando o excesso de comentários desnecessários.

Parte 4: Elaboração de documentação técnica e manual de utilizador

Objetivo: Criar a documentação técnica do sistema e um manual de utilizador. A documentação técnica deve descrever a arquitetura, o fluxo de dados e a interação entre componentes, enquanto o manual de utilizador deve fornecer instruções claras sobre como utilizar a aplicação.

Tarefas:

1. Documentação Técnica: Descrever a arquitetura do sistema, incluindo:
   1.1. Componentes principais (e.g., módulos, classes).
   1.2. Fluxos de dados entre componentes.

1.3. Principais interações e dependências.

2. Manual de utilizador: Elabora um guia de utilização para o sistema, incluindo:
   2.1. Visão geral das funcionalidades da aplicação.
   2.2. Instruções passo a passo para executar as principais funções.
   2.3. Capturas de ecrã, se aplicável, para ilustrar as etapas.
   2.4. Resolução de problemas comuns.

Parte 5: Conclusão e reflexão

Objetivo: Refletir sobre o impacto das boas práticas de programação e documentação na manutenção e longevidade do código.

Tarefas:

1. Redigir uma breve reflexão (máx. 1 página) sobre a importância das boas práticas de codificação, documentação e a sua aplicação na atividade. Discutir os desafios enfrentados durante o processo e como estas práticas podem melhorar o trabalho em equipa e a qualidade do código.

Bom trabalho!

Anexo A – Código-fonte do cenário

O código abaixo implementa um sistema de reservas e gestão de stock para um restaurante. O sistema prevê gestão de múltiplas reservas por cliente, atualizações automáticas de inventário após consumo, e verificação de disponibilidade de mesas.

```
from datetime import datetime, timedelta

class C:
 def __init__(self, n, e):
  self.n = n
  self.e = e
  self.r = []  # A lista de reservas (isto poderia ser útil mais tarde)

 def addR(self, r): # Adiciona uma reserva ao cliente
    self.r.append(r) # Adiciona à lista de reservas

 def lR(self): # Lista todas as reservas de um cliente
  if not self.r: # Se não houver reservas
            print(f"{self.n} não tem reservas.") # Imprime que o cliente não tem reservas
            return # Sai da função
  print(f"Reservas de {self.n}:") # Imprime o nome do cliente
  for r in self.r: # Para cada reserva na lista de reservas
      print(f"- {r.num_pessoas} pessoas no dia {r.data_reserva}, mesa {r.mesa.numero_mesa}") # Im-
prime os detalhes da reserva

class M:
 def __init__(self, nm, c):
  self.nm = nm
  self.c = c
  self.r = False # Indica se a mesa está reservada ou não

 def reservar(self): # Função para reservar uma mesa
        if self.r: # Se a mesa já estiver reservada
         raise Exception(f"A mesa {self.nm} já está reservada!") # Levanta uma exceção
        self.r = True # Reserva a mesa

 def l(self): # Função para libertar uma mesa
            self.r = False # Define a reserva como falsa

class R:
 def __init__(self, c, np, d, m):
     self.c = c # Cliente
     self.num_pessoas = np # Número de pessoas
     self.data_reserva = d # Data da reserva
     self.mesa = m # Mesa reservada
```

```
class P: # Produto
 def __init__(self, n, q):
  self.n = n # Nome do produto
  self.q = q # Quantidade do produto

 def consume(self, q):
  if q > self.q: # Verifica se há quantidade suficiente
   raise Exception(f"Quantidade insuficiente de {self.n}!") # Levanta exceção
  self.q -= q # Subtrai a quantidade consumida

class G: # Classe que representa o gestor do restaurante
 def __init__(self):
   self.r = [] # Lista de reservas
   self.i = {} # Dicionário de inventário
   self.m = [M(i, 4) for i in range(1, 11)] # 10 mesas de 4 pessoas

 def aR(self, c, np, d): # Função para adicionar uma reserva
       m = self.fM(np) # Encontra uma mesa disponível
       if not m: # Se não houver mesa disponível
        print("Não há mesas disponíveis.") # Informa que não há mesas
        return # Sai da função

    r = R(c, np, d, m) # Cria uma nova reserva
    c.addR(r) # Adiciona a reserva ao cliente
    m.reservar() # Reserva a mesa
    self.r.append(r) # Adiciona a reserva à lista
    print(f"Reserva feita para {c.n}: {np} pessoas no dia {d}, mesa {m.nm}.") # Confirma a reserva

 def fM(self, np): # Função para encontrar uma mesa
     for m in self.m: # Para cada mesa
      if not m.r and m.c >= np: # Se a mesa não estiver reservada e tiver capacidade suficiente
       return m # Retorna a mesa disponível
     return None # Se não encontrar mesa, retorna None

 def vR(self, nc): # Função para verificar reserva de cliente
  for r in self.r: # Para cada reserva na lista
   if r.c.n == nc: # Se o nome do cliente corresponder
        print(f"Reserva encontrada para {nc}: {r.num_pessoas} pessoas no dia {r.data_reserva}, mesa
{r.mesa.nm}.") # Detalhes da reserva
        return r # Retorna a reserva encontrada
  print(f"Nenhuma reserva encontrada para {nc}.") # Se não encontrar reserva, informa
  return None # Retorna None se não encontrar nada

 def cR(self, c, d): # Função para cancelar reserva
  for r in self.r: # Para cada reserva
   if r.c == c and r.data_reserva == d: # Se encontrar a reserva
    r.mesa.l() # Liberta a mesa
```

```
    self.r.remove(r) # Remove a reserva da lista
    c.r.remove(r) # Remove a reserva da lista do cliente
    print(f"Reserva de {c.n} no dia {d} foi cancelada.") # Confirma o cancelamento
    return # Sai da função
  print(f"Nenhuma reserva encontrada para {c.n} no dia {d}.") # Se não encontrar, informa

 def aP(self, np, q): # Função para adicionar produto ao inventário
  if np in self.i: # Se o produto já existir
    self.i[np].q += q # Atualiza a quantidade do produto
  else: # Caso contrário
    p = P(np, q) # Cria um novo produto
    self.i[np] = p # Adiciona ao inventário
  print(f"{q} unidades de {np} adicionadas ao inventário.") # Informa o usuário

 def cp(self, np, q): # Função para consumir produto do inventário
    if np in self.i: # Se o produto estiver no inventário
     self.i[np].consume(q) # Consome a quantidade
     print(f"{q} unidades de {np} consumidas.") # Informa que a quantidade foi consumida
    else: # Se o produto não estiver no inventário
     print(f"Produto {np} não encontrado.") # Informa que o produto não foi encontrado

 def vi(self, np): # Função para verificar inventário
  if np in self.i: # Se o produto estiver no inventário
   print(f"{self.i[np].q} unidades de {np} disponíveis.") # Informa a quantidade disponível
   return self.i[np].q # Retorna a quantidade
  else: # Se o produto não estiver no inventário
   print(f"Produto {np} não encontrado no inventário.") # Informa que o produto não foi encontrado
   return 0 # Retorna 0

# Exemplo de uso da aplicação
def main():
 g = G() # Inicializa o gestor do restaurante

 # Cria dois clientes
 c1 = C("João", "joao@email.com")
 c2 = C("Maria", "maria@email.com")

 # Adiciona duas reservas para os clientes
 d1 = datetime.now() + timedelta(days=1)
 d2 = datetime.now() + timedelta(days=2)
 g.aR(c1, 4, d1)
 g.aR(c2, 3, d2)

 # Verifica as reservas
 g.vR("João")
 g.vR("Maria")
```

```
 # Adiciona produtos ao inventário
 g.aP("Arroz", 50)
 g.aP("Massa", 30)

 # Consome produtos do inventário
 g.cp("Arroz", 10)
 g.cp("Massa", 5)

 # Verifica inventário
 g.vi("Arroz")
 g.vi("Massa")
 g.vi("Feijão")

 # Lista as reservas dos clientes
 c1.lR()
 c2.lR()

 # Cancela uma reserva
 g.cR(c1, d1)

 # Verifica a reserva de João após o cancelamento
 g.vR("João")

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