Atividade Prática: Refatoração e Padrões de Projeto

1. Objetivo

Em dupla, refatore três sistemas legados (código "espaguete") aplicando padrões de projeto criacionais e arquiteturais para garantir escalabilidade, organização e economia de recursos.

2. Contexto Teórico (Revisão)

Conforme discutido em nossos slides:

- **MVC:** Separação de responsabilidades (Dados, Interface e Lógica).
- **Singleton:** Garantia de uma única instância para recursos críticos (ex: conexão com banco de dados) [Gamma, 429].
- **Factory:** Centralização da lógica de criação de objetos, desacoplando o "como criar" do "como usar" [Gamma, 388].

---

3. O Desafio: As Três Massas de Macarrão 🍝

Abaixo, apresento três cenários de códigos mal estruturados. Sua missão é refatorá-los.

Cenário A: O Gerenciador de Tarefas Monolítico (Foco em MVC)

Este código mistura lógica de armazenamento, regras de negócio e exibição no terminal em um único bloco.

```
# CODIGO MACARRÔNICO A
tarefas = []
while True:
    op = input("1-Add, 2-List, 3-Sair: ")
    if op == "1":
        t = input("Tarefa: ")
        tarefas.append(t) # Lógica de dados misturada com UI
        print("Salvo!")
    elif op == "2":
        for t in tarefas: print(f"- {t}") # Exibição misturada
    else: break
```

Sua tarefa: Refatore este código aplicando o padrão **MVC**.

- **Model:** Gerencia a lista de tarefas.
- **View:** Cuida apenas dos `inputs` e `prints`.
- **Controller:** Faz a ponte entre os dois.

Cenário B: O Caos das Conexões de API (Foco em Singleton)

Neste cenário, cada vez que o sistema precisa logar uma atividade ou consultar um recurso REST (como vimos no Slide 2), ele cria uma nova "conexão" pesada, desperdiçando memória.

```
# CODIGO MACARRÔNICO B
class APIConnection:
    def __init__(self):
        print("⚠ Conexão pesada aberta com a API REST...")

def salvar_log(mensagem):
    conn = APIConnection() # Cria uma nova conexão toda vez!
    print(f"Log: {mensagem}")
```

```
salvar_log("Usuario logou")
salvar_log("Usuario clicou em salvar")
```

Sua tarefa: Aplique o padrão **Singleton** [Danjou, 1483] para garantir que a classe `APIConnection` tenha apenas **uma instância** em todo o ciclo de vida da aplicação, evitando o overhead de múltiplas conexões.

Cenário C: A Fábrica de Notificações (Foco em Factory)

O sistema abaixo envia notificações para diferentes tipos de usuários, mas o código principal está "sujo" com muitos `if/else`, dificultando a adição de novos métodos (como SMS ou Push).

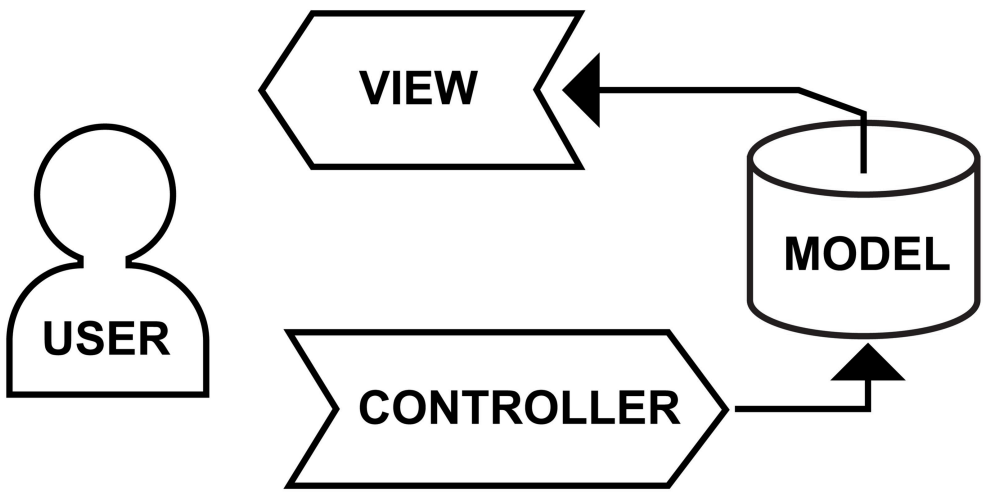
```
# CODIGO MACARRÔNICO C
class EmailNotificacao:
    def enviar(self, msg): print(f"Enviando Email: {msg}")

class WhatsAppNotificacao:
    def enviar(self, msg): print(f"Enviando Zap: {msg}")

# Código cliente cheio de acoplamento
tipo = input("Tipo: ")
if tipo == "email":
    notificador = EmailNotificacao()
elif tipo == "zap":
    notificador = WhatsAppNotificacao()
notificador.enviar("Olá!")
```

Sua tarefa: Implemente o padrão **Factory**. Crie uma função ou classe `NotificacaoFactory` que receba o tipo e retorne o objeto correto [Gamma, 392]. O código principal não deve saber *qual* classe está instanciando.

---

4. Instruções de Entrega

1. **Modularização:** Cada cenário deve ser resolvido em arquivos separados ou pastas organizadas.
2. **Padrão Pythonico:** Utilize as práticas de **Danjou**, como o método `__new__` para o Singleton.
3. **JSON & REST:** No cenário B, simule que a resposta da conexão Singleton retorna um **JSON** (como visto no Slide 4).
4. Faça a prática usando o google colab e compartilhe o link do notebook colab com todos os códigos.



MODEL - VIEW - CONTROLLER PATTERN