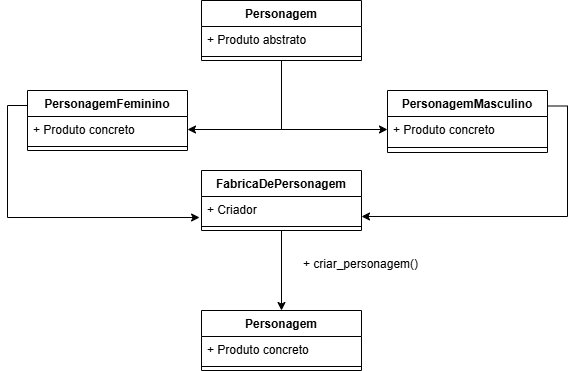
Padrões de projeto

### **1. Factory Method**

#### **Descrição:**

O **Factory Method** permite a criação dinâmica de personagens (como masculino ou feminino) e cenários, sem que o código principal precise saber as classes específicas. Isso facilita a inclusão de novos tipos de personagens ou elementos de jogo, sem modificar a lógica do sistema. Com isso, o código fica mais flexível, permitindo futuras expansões e personalizações sem impactar outras partes do jogo. Esse padrão também centraliza a criação de objetos, tornando o código mais organizado e de fácil manutenção.



# Produto Abstrato

class Personagem:

def exibir(self):

pass

# Produto Concreto 1

class PersonagemMasculino(Personagem):

def exibir(self):

return "Personagem: Masculino"

# Produto Concreto 2

class PersonagemFeminino(Personagem):

def exibir(self):

return "Personagem: Feminino"

# Fábrica

class FabricaDePersonagem:

def criar\_personagem(self, genero):

if genero == 'masculino':

return PersonagemMasculino()

elif genero == 'feminino':

return PersonagemFeminino()

else:

raise ValueError("Gênero inválido")

# Código do Cliente

fabrica = FabricaDePersonagem()

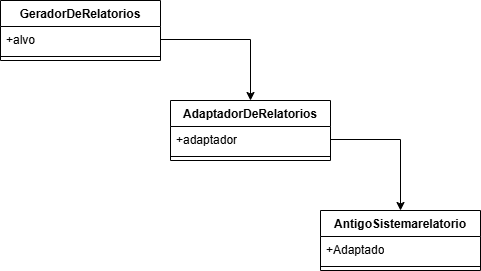
personagem = fabrica.criar\_personagem("masculino")

print(personagem.exibir()) # Saída: Personagem: Masculino

### **2. Adapter**

#### **Descrição:**

O **Adapter** é um padrão de design estrutural que permite que interfaces incompatíveis trabalhem juntas. No meu jogo, ele pode ser utilizado para integrar um sistema de relatórios antigo (por exemplo, para gerar relatórios de progresso) com uma nova interface mais moderna que o jogo utiliza.



# Alvo

class GeradorDeRelatorio:

def gerar\_relatorio(self):

pass

# Adaptee

class SistemaAntigoDeRelatorio:

def gerar\_relatorio\_antigo(self):

return "Relatório no formato antigo"

# Adaptador

class AdaptadorDeRelatorio(GeradorDeRelatorio):

def \_\_init\_\_(self, sistema\_antigo):

self.sistema\_antigo = sistema\_antigo

def gerar\_relatorio(self):

return self.sistema\_antigo.gerar\_relatorio\_antigo()

# Código do Cliente

sistema\_antigo = SistemaAntigoDeRelatorio()

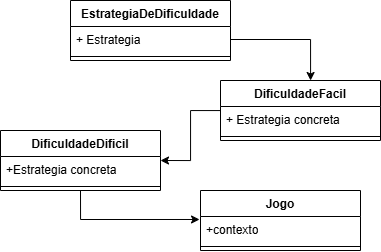
adaptador = AdaptadorDeRelatorio(sistema\_antigo)

print(adaptador.gerar\_relatorio()) # Saída: Relatório no formato antigo

### **3. Strategy**

#### **Descrição:**

O **Strategy** é um padrão comportamental que permite que o comportamento de uma classe seja alterado em tempo de execução. Pode ser usado no meu jogo para definir diferentes níveis de dificuldade que podem ser alterados durante o jogo.



from abc import ABC, abstractmethod

# Estratégia

class EstrategiaDeDificuldade(ABC):

@abstractmethod

def executar(self):

pass

# Estratégia Concreta 1

class DificuldadeFacil(EstrategiaDeDificuldade):

def executar(self):

return "Dificuldade fácil selecionada"

# Estratégia Concreta 2

class DificuldadeDificil(EstrategiaDeDificuldade):

def executar(self):

return "Dificuldade difícil selecionada"

# Contexto

class Jogo:

def \_\_init\_\_(self, estrategia: EstrategiaDeDificuldade):

self.\_estrategia = estrategia

def definir\_estrategia(self, estrategia: EstrategiaDeDificuldade):

self.\_estrategia = estrategia

def iniciar\_jogo(self):

print(self.\_estrategia.executar())

# Código do Cliente

jogo = Jogo(DificuldadeFacil())

jogo.iniciar\_jogo() # Saída: Dificuldade fácil selecionada

jogo.definir\_estrategia(DificuldadeDificil())

jogo.iniciar\_jogo() # Saída: Dificuldade difícil selecionada