Untitled

““” SISTEMA DE NOTIFICACIONES DE RED SOCIAL Ejemplo que combina los patrones Factory y Observer

Contexto del problema: - Una red social donde los usuarios pueden seguir diferentes tipos de contenido - Cuando se publica nuevo contenido, los seguidores reciben notificaciones - Diferentes tipos de contenido requieren diferentes formas de creación y notificación ““”

from abc import ABC, abstractmethod from typing import List, Dict from datetime import datetime import uuid

# =============================================

# PATRÓN OBSERVER: Definición de Observadores

# =============================================

class Observer(ABC): “““Interfaz Observer - Los usuarios observan el contenido”“”

@abstractmethod  
def update(self, contenido: 'Contenido'):  
 """Método llamado cuando se publica nuevo contenido"""  
 pass

class Usuario(Observer): “““Usuario concreto que observa el contenido”“”

def \_\_init\_\_(self, nombre: str, email: str):  
 self.nombre = nombre  
 self.email = email  
 self.id = str(uuid.uuid4())[:8]  
 self.notificaciones = []  
  
def update(self, contenido: 'Contenido'):  
 """Implementación del método update - Recibe notificación"""  
 mensaje = f"Nuevo {contenido.tipo.upper()}: '{contenido.titulo}' por {contenido.autor}"  
 self.notificaciones.append({  
 'fecha': datetime.now(),  
 'mensaje': mensaje,  
 'contenido\_id': contenido.id  
 })  
 print(f"📧 Notificación para {self.nombre}: {mensaje}")  
  
def mostrar\_notificaciones(self):  
 """Muestra el historial de notificaciones del usuario"""  
 print(f"\n=== Notificaciones de {self.nombre} ===")  
 for notif in self.notificaciones[-5:]: # Últimas 5 notificaciones  
 print(f"[{notif['fecha'].strftime('%H:%M')}] {notif['mensaje']}")

# =============================================

# PATRÓN OBSERVER: Sujeto Observable

# =============================================

class SujetoObservable(ABC): “““Interfaz para sujetos observables (contenido)”“”

def \_\_init\_\_(self):  
 self.\_observadores: List[Observer] = []  
  
def agregar\_observador(self, observador: Observer):  
 """Agrega un observador a la lista"""  
 if observador not in self.\_observadores:  
 self.\_observadores.append(observador)  
  
def eliminar\_observador(self, observador: Observer):  
 """Elimina un observador de la lista"""  
 self.\_observadores.remove(observador)  
  
def notificar\_observadores(self):  
 """Notifica a todos los observadores"""  
 for observador in self.\_observadores:  
 observador.update(self)

# =============================================

# PATRÓN FACTORY: Productos (Tipos de Contenido)

# =============================================

class Contenido(SujetoObservable, ABC): “““Clase abstracta para diferentes tipos de contenido”“”

def \_\_init\_\_(self, titulo: str, autor: str):  
 super().\_\_init\_\_()  
 self.id = str(uuid.uuid4())[:8]  
 self.titulo = titulo  
 self.autor = autor  
 self.fecha\_publicacion = datetime.now()  
 self.tipo = self.\_\_class\_\_.\_\_name\_\_.lower()  
  
@abstractmethod  
def publicar(self):  
 """Método abstracto para publicar contenido"""  
 pass  
  
def \_\_str\_\_(self):  
 return f"{self.tipo.upper()}: {self.titulo} por {self.autor}"

class Post(Contenido): “““Post de texto - Producto concreto del Factory”“”

def \_\_init\_\_(self, titulo: str, autor: str, texto: str, etiquetas: List[str] = None):  
 super().\_\_init\_\_(titulo, autor)  
 self.texto = texto  
 self.etiquetas = etiquetas or []  
 self.likes = 0  
  
def publicar(self):  
 """Publica el post y notifica a los observadores"""  
 print(f"📝 Publicando POST: '{self.titulo}'")  
 self.notificar\_observadores()  
  
def dar\_like(self):  
 """Incrementa el contador de likes"""  
 self.likes += 1  
 print(f"❤️ Post '{self.titulo}' tiene {self.likes} likes")

class Video(Contenido): “““Video - Producto concreto del Factory”“”

def \_\_init\_\_(self, titulo: str, autor: str, url: str, duracion: int):  
 super().\_\_init\_\_(titulo, autor)  
 self.url = url  
 self.duracion = duracion # en segundos  
 self.reproducciones = 0  
  
def publicar(self):  
 """Publica el video y notifica a los observadores"""  
 print(f"🎥 Publicando VIDEO: '{self.titulo}' ({self.duracion}s)")  
 self.notificar\_observadores()  
  
def reproducir(self):  
 """Simula la reproducción del video"""  
 self.reproducciones += 1  
 print(f"▶️ Reproduciendo '{self.titulo}' - Total reproducciones: {self.reproducciones}")

class Evento(Contenido): “““Evento - Producto concreto del Factory”“”

def \_\_init\_\_(self, titulo: str, autor: str, fecha\_evento: datetime, ubicacion: str):  
 super().\_\_init\_\_(titulo, autor)  
 self.fecha\_evento = fecha\_evento  
 self.ubicacion = ubicacion  
 self.asistentes = []  
  
def publicar(self):  
 """Publica el evento y notifica a los observadores"""  
 print(f"📅 Publicando EVENTO: '{self.titulo}' en {self.ubicacion}")  
 self.notificar\_observadores()  
  
def registrar\_asistente(self, usuario: Usuario):  
 """Registra un usuario como asistente al evento"""  
 if usuario not in self.asistentes:  
 self.asistentes.append(usuario)  
 print(f"✅ {usuario.nombre} se registró para el evento '{self.titulo}'")

# =============================================

# PATRÓN FACTORY: Fábrica de Contenido

# =============================================

class FabricaContenido: “““Fábrica para crear diferentes tipos de contenido”“”

@staticmethod  
def crear\_contenido(tipo: str, \*\*kwargs) -> Contenido:  
 """  
 Método factory que crea diferentes tipos de contenido  
 basado en el parámetro 'tipo'  
 """  
 tipo = tipo.lower()  
   
 if tipo == "post":  
 return Post(  
 titulo=kwargs['titulo'],  
 autor=kwargs['autor'],  
 texto=kwargs['texto'],  
 etiquetas=kwargs.get('etiquetas', [])  
 )  
   
 elif tipo == "video":  
 return Video(  
 titulo=kwargs['titulo'],  
 autor=kwargs['autor'],  
 url=kwargs['url'],  
 duracion=kwargs['duracion']  
 )  
   
 elif tipo == "evento":  
 return Evento(  
 titulo=kwargs['titulo'],  
 autor=kwargs['autor'],  
 fecha\_evento=kwargs['fecha\_evento'],  
 ubicacion=kwargs['ubicacion']  
 )  
   
 else:  
 raise ValueError(f"Tipo de contenido no soportado: {tipo}")

# =============================================

# EJEMPLO DE USO - SIMULACIÓN DEL SISTEMA

# =============================================

def demostrar\_sistema(): “““Función principal que demuestra el sistema en acción”“”

print("=" \* 60)  
print("🚀 INICIANDO SIMULACIÓN: SISTEMA DE RED SOCIAL")  
print("=" \* 60)  
  
# Crear usuarios (observadores)  
print("\n👥 CREANDO USUARIOS...")  
usuario1 = Usuario("Ana García", "ana@email.com")  
usuario2 = Usuario("Carlos López", "carlos@email.com")  
usuario3 = Usuario("María Rodríguez", "maria@email.com")  
  
# Crear contenido usando la fábrica  
print("\n🏭 CREANDO CONTENIDO CON FACTORY PATTERN...")  
  
# Crear un post  
post = FabricaContenido.crear\_contenido(  
 tipo="post",  
 titulo="Introducción a Python",  
 autor="Profesor Python",  
 texto="Python es un lenguaje de programación versátil...",  
 etiquetas=["python", "programación", "tutorial"]  
)  
  
# Crear un video  
video = FabricaContenido.crear\_contenido(  
 tipo="video",  
 titulo="Tutorial de Django",  
 autor="Web Developer Pro",  
 url="https://example.com/video123",  
 duracion=600  
)  
  
# Crear un evento  
from datetime import datetime, timedelta  
fecha\_evento = datetime.now() + timedelta(days=7)  
evento = FabricaContenido.crear\_contenido(  
 tipo="evento",  
 titulo="Conferencia de Inteligencia Artificial",  
 autor="AI Experts",  
 fecha\_evento=fecha\_evento,  
 ubicacion="Centro de Convenciones"  
)  
  
# Los usuarios siguen el contenido (se suscriben como observadores)  
print("\n🔔 USUARIOS SIGUIENDO CONTENIDO...")  
post.agregar\_observador(usuario1)  
post.agregar\_observador(usuario2)  
  
video.agregar\_observador(usuario2)  
video.agregar\_observador(usuario3)  
  
evento.agregar\_observador(usuario1)  
evento.agregar\_observador(usuario2)  
evento.agregar\_observador(usuario3)  
  
# Publicar contenido (notifica automáticamente a los observadores)  
print("\n📢 PUBLICANDO CONTENIDO...")  
print("-" \* 40)  
post.publicar()  
print("-" \* 40)  
video.publicar()  
print("-" \* 40)  
evento.publicar()  
  
# Interacciones adicionales  
print("\n💫 INTERACCIONES ADICIONALES...")  
post.dar\_like()  
post.dar\_like()  
video.reproducir()  
evento.registrar\_asistente(usuario1)  
  
# Mostrar notificaciones de usuarios  
print("\n📋 RESUMEN DE NOTIFICACIONES...")  
usuario1.mostrar\_notificaciones()  
usuario2.mostrar\_notificaciones()  
usuario3.mostrar\_notificaciones()  
  
# Demostrar flexibilidad del Factory  
print("\n🎯 DEMOSTRANDO FLEXIBILIDAD DEL FACTORY...")  
try:  
 nuevo\_contenido = FabricaContenido.crear\_contenido(  
 tipo="podcast", # Tipo no soportado  
 titulo="Test",  
 autor="Test"  
 )  
except ValueError as e:  
 print(f"❌ Error esperado: {e}")

# =============================================

# EXPLICACIÓN PARA PARTICIPACIÓN EN CLASE

# =============================================

def explicacion\_patrones(): ““” Función con explicaciones para la participación en clase ““” print(“” + “=” \* 70) print(“💡 EXPLICACIÓN PARA PARTICIPACIÓN EN CLASE”) print(“=” \* 70)

explicaciones = {  
 "FACTORY PATTERN": [  
 "✓ Propósito: Crear objetos sin especificar la clase exacta",  
 "✓ Beneficio: Encapsula la lógica de creación",  
 "✓ Escalabilidad: Fácil agregar nuevos tipos de contenido",  
 "✓ En nuestro ejemplo: FabricaContenido crea Posts, Videos, Eventos"  
 ],  
   
 "OBSERVER PATTERN": [  
 "✓ Propósito: Notificar cambios a múltiples objetos",  
 "✓ Beneficio: Acoplamiento loose entre sujeto y observadores",   
 "✓ Escalabilidad: Fácil agregar/remover observadores",  
 "✓ En nuestro ejemplo: Usuarios reciben notificaciones de contenido nuevo"  
 ],  
   
 "COMBINACIÓN DE PATRONES": [  
 "✓ Factory crea el contenido (Productos)",  
 "✓ Content implementa Observable (Sujeto)",  
 "✓ Usuarios implementan Observer",  
 "✓ Cuando se publica contenido, se notifica automáticamente"  
 ],  
   
 "BENEFICIOS EN EL MUNDO REAL": [  
 "✅ Mantenibilidad: Código organizado y extensible",  
 "✅ Escalabilidad: Fácil agregar nuevos tipos de contenido",  
 "✅ Reusabilidad: Patrones aplicables a otros contextos",  
 "✅ Testabilidad: Cada componente puede probarse por separado"  
 ]  
}  
  
for patron, puntos in explicaciones.items():  
 print(f"\n{patron}:")  
 for punto in puntos:  
 print(f" {punto}")

# Ejecutar la demostración

if **name** == “**main**”: demostrar\_sistema() explicacion\_patrones()

print("\n" + "=" \* 60)  
print("🎉 DEMOSTRACIÓN COMPLETADA - PATRONES FACTORY Y OBSERVER")  
print("=" \* 60)