Practica 6. Factory y Observer

José Fernández Quintana

Introducción

En esta práctica se presentan dos patrones de diseño muy utilizados en la programación orientada a objetos: Factory y Observer.
Ambos permiten resolver problemas comunes de creación y comunicación de objetos en proyectos de software, facilitando la reutilización y el mantenimiento del código.

Patrón Factory

Antecedentes

El patrón Factory Method pertenece a los patrones creacionales.
Su propósito es delegar la creación de objetos a una clase fábrica, evitando que el código cliente tenga que saber los detalles de construcción.

Se usa cuando: - Hay que crear múltiples objetos relacionados.
- Queremos centralizar y simplificar la lógica de creación.

Ejemplo: Fábrica de vehículos

```
# Patrón Factory aplicado a una fábrica de vehículos

class Vehiculo:
    def mover(self):
        pass

class Coche(Vehiculo):
    def mover(self):
        return "El coche circula por la carretera."
```

```python
class Bicicleta(Vehiculo):
    def mover(self):
        return "La bicicleta avanza por el carril bici."

class Camion(Vehiculo):
    def mover(self):
        return "El camión transporta mercancía por la autopista."

# Clase Factory
class VehiculoFactory:
    @staticmethod
    def crear_vehiculo(tipo):
        if tipo == "coche":
            return Coche()
        elif tipo == "bicicleta":
            return Bicicleta()
        elif tipo == "camion":
            return Camion()
        else:
            raise ValueError("Tipo de vehículo no válido.")

# Uso del patrón Factory
tipos = ["coche", "bicicleta", "camion"]

for t in tipos:
    vehiculo = VehiculoFactory.crear_vehiculo(t)
    print(f"{t.title()}: {vehiculo.mover()}")
```

```
Coche: El coche circula por la carretera.
Bicicleta: La bicicleta avanza por el carril bici.
Camion: El camión transporta mercancía por la autopista.
```

Patrón Observer

Antecedentes

El patrón Observer pertenece a los patrones de comportamiento. Su propósito es mantener la consistencia entre objetos relacionados, de modo que si uno cambia de estado, los demás son notificados automáticamente.

Se usa cuando: - Hay múltiples objetos que dependen del estado de otro. - Se necesita comunicación desacoplada entre clases.

Ejemplo: Sistema de noticias

```
# Patrón Observer aplicado a un sistema de noticias

# Sujeto
class Periodico:
    def __init__(self):
        self.suscriptores = []
        self.noticias = []

    def agregar_suscriptor(self, suscriptor):
        self.suscriptores.append(suscriptor)

    def eliminar_suscriptor(self, suscriptor):
        self.suscriptores.remove(suscriptor)

    def nueva_noticia(self, noticia):
        self.noticias.append(noticia)
        print(f"\n[Periódico] Nueva noticia publicada: {noticia}")
        self.notificar_suscriptores(noticia)

    def notificar_suscriptores(self, noticia):
        for suscriptor in self.suscriptores:
            suscriptor.actualizar(noticia)

# Observadores
class Suscriptor:
    def __init__(self, nombre):
        self.nombre = nombre

    def actualizar(self, noticia):
        print(f"{self.nombre} recibió la noticia: {noticia}")

# Uso del patrón Observer
periodico = Periodico()

s1 = Suscriptor("Ana")
s2 = Suscriptor("Luis")
s3 = Suscriptor("María")
```

```
periodico.agregar_suscriptor(s1)
periodico.agregar_suscriptor(s2)
periodico.agregar_suscriptor(s3)

periodico.nueva_noticia("Nuevo descubrimiento científico sobre IA")
periodico.nueva_noticia("Resultados de las elecciones 2025")
```

```
[Periódico] Nueva noticia publicada: Nuevo descubrimiento científico sobre IA
Ana recibió la noticia: Nuevo descubrimiento científico sobre IA
Luis recibió la noticia: Nuevo descubrimiento científico sobre IA
María recibió la noticia: Nuevo descubrimiento científico sobre IA

[Periódico] Nueva noticia publicada: Resultados de las elecciones 2025
Ana recibió la noticia: Resultados de las elecciones 2025
Luis recibió la noticia: Resultados de las elecciones 2025
María recibió la noticia: Resultados de las elecciones 2025
```

Conclusión

El Factory simplifica la creación de objetos y centraliza la lógica de instanciación. El Observer permite la comunicación automática entre objetos sin acoplarlos directamente.

Ambos patrones son fundamentales en el desarrollo de software, especialmente en sistemas grandes donde se requiere mantener el código limpio y flexible.