Justificación de Patrones de Diseño

Este documento presenta los patrones de diseño aplicados en la solución del

Ejercicio 1: Gestión de Elementos y Kits y justifica su uso en el contexto del problema.

1. Pattern Composite

Contexto: Un kit agrupa elementos simples y, recursivamente, otros kits. Necesitamos tratar ambos tipos de forma uniforme.

Solución: Aplicamos el patrón Composite creando una jerarquía donde:

- Elemento (componente común) define la interfaz getPrecio() y getCodigo().
- ElementoSimple (hoja) implementa getPrecio() devolviendo su precio.
- `Kit` (compuesto) implementa `getPrecio()` sumando los precios de sus componentes y aplicando un descuento del 10%.

Justificación: Composite nos permite:

- Gestionar de forma uniforme objetos individuales y compuestos.
- Añadir o eliminar recursivamente componentes sin cambiar el cliente.
- Escalar la estructura sin límite de anidamiento.

2. Pattern Builder

Contexto: La construcción de un kit puede implicar añadir múltiples componentes y configurar su código antes de crear el objeto final.

Solución: Implementamos KitBuilder con métodos encadenados:

java

```
KitBuilder builder = new KitBuilder()
   .setCodigo(1001)
```

```
    .addComponente(new Mesa(...))
    .addComponente(new Silla(...));
Kit miKit = builder.build();
```

Justificación: Builder es útil cuando:

- Hay muchos parámetros opcionales (lista de componentes).
- Se desea claridad en el proceso de construcción.
- Se necesita inmutabilidad o validación previa al crear el objeto.

3. Pattern Factory Method (opcional)

Contexto: Si preferimos centralizar la creación de kits estándar o preconfigurados.

Solución: Un KitFactory con métodos estáticos:

java

```
public class KitFactory {
  public static Kit crearKitOficina() { ... }
  public static Kit crearKitJardin() { ... }
}
```

Justificación: Factory Method permite:

- Encapsular la lógica de creación de kits comunes.
- Facilitar la extensión añadiendo nuevos métodos de fábrica sin modificar el cliente.

Conclusión

La combinación de Composite y Builder cubre los requisitos de recursividad y claridad en la construcción de kits, mientras que Factory aporta flexibilidad para kits

predefinidos. Estos patrones optimizan la mantenibilidad, escalabilidad y legibilidad del código.