Guía de Patrones de Diseño en Java

Introducción

Los **Patrones de Diseño** son soluciones probadas a problemas comunes en el desarrollo de software. No son código reutilizable directamente, sino **modelos conceptuales** que ayudan a diseñar sistemas más flexibles, escalables y mantenibles.

Se dividen en tres grandes grupos:

- **Creacionales** → Cómo crear objetos.
- **Estructurales** → Cómo organizar clases y objetos.
- **Comportamiento** → Cómo los objetos interactúan entre sí.

Patrones Creacionales

Factory Method
Abstract Factory
Builder
Prototype
Singleton

Patrones Estructurales

Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Patrones de Comportamiento

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

Patrones Creacionales

1. Singleton

Definición: Garantiza que una clase tenga una única instancia en todo el sistema y proporciona un punto global de acceso.
Uso: Manejo de configuraciones globales, conexiones a BD, logs.

```
class Singleton {
    private static Singleton instancia;
    private Singleton() {}
    public static Singleton getInstancia() {
        if (instancia == null) {
            instancia = new Singleton();
        }
        return instancia;
    }
}
```

```
// Ejemplo: Conexión a la base de datos
class ConexionBD {
    private static ConexionBD instancia;

    private ConexionBD() {
        System.out.println("Conectando a la base de datos...");
    }

    public static ConexionBD getInstancia() {
        if (instancia == null) {
            instancia = new ConexionBD();
        }
        return instancia;
    }

    public void ejecutarConsulta(String sql) {
        System.out.println("Ejecutando: " + sql);
    }
}

public class SingletonEjemplo {
    public static void main(String[] args) {
        ConexionBD c1 = ConexionBD.getInstancia();
        ConexionBD c2 = ConexionBD.getInstancia();

        c1.ejecutarConsulta("SELECT * FROM usuarios");
        System.out.println(c1 == c2); // true (misma instancia)
    }
}
```

Singleton en ambiente concurrente

¿Cómo funciona el Singleton?

El patrón Singleton asegura que **solo exista una única instancia de una clase en la JVM** y provee un punto global de acceso a ella.

- Esa **única instancia se comparte entre todos los hilos y usuarios** que acceden a la aplicación.
- No importa cuántos clientes se conecten, **no se crea una instancia por usuario**, sino que hay **una sola en memoria** para toda la aplicación (salvo que el desarrollador implemente variantes particulares, como un "singleton por sesión").

El problema aparece cuando **múltiples hilos intentan crear la instancia al mismo tiempo**.
Ejemplo clásico sin sincronización:

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton(); // peligro en concurrencia
        }
        return instance;
    }
}
```

Si **dos hilos** entran al getInstance() al mismo tiempo, ambos pueden evaluar instance == null como verdadero y terminar creando **dos instancias distintas**, rompiendo la idea del Singleton.

Soluciones thread-safe

Para hacerlo seguro en un ambiente multiusuario concurrente, hay varias estrategias:

Synchronized (simple pero costoso)

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Garantiza que solo un hilo cree la instancia.
Puede tener impacto en performance bajo mucha concurrencia, porque sincroniza cada llamada.

Double-Checked Locking (más eficiente)

Idea básica:
Solo crear la instancia del Singleton **una vez**, incluso si hay múltiples hilos intentando acceder al mismo tiempo.

Cómo funciona paso a paso:

1. Declaramos la instancia como volatile.
   Esto garantiza que todos los hilos vean el valor actualizado de la variable, evitando errores de "lectura sucia" (cuando un hilo ve un objeto parcialmente creado).
2. Cuando un hilo llama a getInstance(), primero verifica (if (instance == null)) **sin bloquear**.
   - Si ya existe, la devuelve directamente → rápido.
   - Si no existe, entra en un synchronized.
3. Dentro del bloque sincronizado, vuelve a verificar (if (instance == null)).
   - Esto es necesario porque otro hilo pudo haber creado la instancia entre la primera verificación y el momento de adquirir el lock.
4. Solo el primer hilo que llega crea la instancia. Los demás entran al if y la ven ya creada.

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) { // Primera verificación (sin lock)
            synchronized (Singleton.class) {
                if (instance == null) { // Segunda verificación (con lock)
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Sincroniza solo la primera vez, luego acceso rápido.
volatile evita problemas de visibilidad de memoria.
Muy usado en ambientes concurrentes.

Ventaja: eficiente, no se sincroniza siempre, solo al crear la primera vez.
Cuidado: si olvidas volatile, podrías tener un objeto en estado inconsistente (visible para otros hilos antes de terminar de construirse).

Singleton con Holder (lazy, thread-safe y simple)

```
public class SingletonHolder {
    private SingletonHolder () {}
```

```
    private static class Holder {
        private static final SingletonHolder INSTANCE = new SingletonHolder ();
    }

    public static SingletonHolder getInstance() {
        return Holder.INSTANCE;
    }
}
```

Creación diferida (lazy).
Thread-safe sin sincronización explícita.
Recomendado en la mayoría de los casos.

Ventajas:

- Súper simple.
- No necesita volatile.
- 100% thread-safe gracias a la especificación de inicialización de clases de la JVM.
- Lazy: solo se crea cuando se usa.

Entonces, en ambiente multiusuario:

- No se crea una instancia por usuario.
- Se comparte una única instancia entre todos los clientes y todos los hilos del backend.
- Esto es útil, por ejemplo, para manejar un pool de conexiones, un caché global, un manejador de logs, configuraciones compartidas, etc.

2. Factory Method

Definición: Define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar.
Uso: Frameworks que deben crear objetos sin acoplarse a clases concretas.

La idea principal es:
una clase define el contrato (interfaz/método abstracto) de creación,
pero **las subclases deciden qué objeto concreto instanciar**.

```
abstract class Transporte {
   abstract void entregar();
}

class Camion extends Transporte {
   void entregar() { System.out.println("Entrega por carretera"); }
}

class Barco extends Transporte {
```

```
    void entregar() { System.out.println("Entrega por mar"); }
}

class Logistica {
    static Transporte crearTransporte(String tipo) {
        if (tipo.equals("camion")) return new Camion();
        else return new Barco();
    }
}
```

---------- ejemplo 2 ----------

```
// Producto
abstract class Documento {
    public abstract void imprimir();
}

// Concretos
class PDF extends Documento {
    @Override
    public void imprimir() {
        System.out.println("Imprimiendo PDF...");
    }
}

class Word extends Documento {
    @Override
    public void imprimir() {
        System.out.println("Imprimiendo Word...");
    }
}

// Factory
class DocumentoFactory {
    public static Documento crearDocumento(String tipo) {
        switch (tipo) {
            case "PDF": return new PDF();
            case "Word": return new Word();
            default: throw new IllegalArgumentException("Tipo desconocido");
        }
    }
}

public class FactoryEjemplo {
    public static void main(String[] args) {
        Documento doc1 = DocumentoFactory.crearDocumento("PDF");
        Documento doc2 = DocumentoFactory.crearDocumento("Word");

        doc1.imprimir();
        doc2.imprimir();
```

```
  }
}
```

¿Influye la concurrencia en Factory Method?

Generalmente **NO afecta directamente la concurrencia**, porque:

1. **Cada llamada al Factory Method crea un nuevo objeto**, independiente del resto.
   → No hay un único recurso compartido como en Singleton.
2. **Los objetos retornados pueden ser usados en paralelo** por múltiples hilos sin interferir, *siempre que los objetos creados sean thread-safe o inmutables*.
3. El Factory Method solo tendría problemas si:
   - Usa **recursos compartidos estáticos** dentro del método fábrica.
   - Implementa **caché de instancias** (por ejemplo, un "pool" de objetos).
     En esos casos sí habría que sincronizar.

3. Abstract Factory

Definición: Proporciona una interfaz para crear **familias de objetos relacionados** sin especificar clases concretas.
Uso: GUI multiplataforma (Windows/Linux/Mac).

Un ejemplo de la vida real puede ser el desarrollo de una **interfaz gráfica multiplataforma**: tu aplicación debe funcionar en **Windows** y en **Linux**, pero cada sistema operativo tiene su propia implementación de botones, checkboxes, menús, etc.

En vez de llenar el código con if (windows) ... else (linux)..., usamos un **Abstract Factory** que nos da los objetos correctos según el entorno.

```
// Abstract Factory
interface UIFactory {
    Boton crearBoton();
    Checkbox crearCheckbox();
}

// Productos
interface Boton { void pintar(); }
interface Checkbox { void seleccionar(); }

// Implementaciones Windows
class WindowsBoton implements Boton {
    public void pintar() { System.out.println("Botón estilo Windows"); }
}
class WindowsCheckbox implements Checkbox {
    public void seleccionar() { System.out.println("Checkbox Windows seleccionado");
}
}

// Implementaciones Mac
class MacBoton implements Boton {
    public void pintar() { System.out.println("Botón estilo Mac"); }
```

```
}
class MacCheckbox implements Checkbox {
    public void seleccionar() { System.out.println("Checkbox Mac seleccionado"); }
}

// Factories concretas
class WindowsFactory implements UIFactory {
    public Boton crearBoton() { return new WindowsBoton(); }
    public Checkbox crearCheckbox() { return new WindowsCheckbox(); }
}
class MacFactory implements UIFactory {
    public Boton crearBoton() { return new MacBoton(); }
    public Checkbox crearCheckbox() { return new MacCheckbox(); }
}

public class AbstractFactoryEjemplo {
    public static void main(String[] args) {
        UIFactory factory = new WindowsFactory(); // Podría ser MacFactory
        Boton boton = factory.crearBoton();
        Checkbox checkbox = factory.crearCheckbox();

        boton.pintar();
        checkbox.seleccionar();
    }
}
```

En la vida real este patrón se usa, por ejemplo:

- Frameworks de **UI multiplataforma** (Java Swing, Qt, Flutter, etc.).
- Motores de base de datos (cambiar de MySQL a PostgreSQL sin reescribir la lógica de acceso).
- Motores de renderizado en videojuegos (OpenGL, DirectX, Vulkan).

4. Builder

Definición: Separa la construcción de un objeto complejo de su representación final.
Uso: Creación de objetos con muchos parámetros opcionales.

El **Builder** es un patrón creacional que se utiliza cuando queremos **construir objetos complejos paso a paso**.
En lugar de tener un **constructor gigante con muchos parámetros**, usamos un *Builder* que nos permite ir configurando cada parte de manera clara y flexible.

Problema que resuelve

Cuando una clase tiene **muchos atributos opcionales y obligatorios**, el constructor tradicional se vuelve difícil de usar y de mantener.
Ejemplo: un Auto que puede tener color, motor, aire acondicionado, GPS, llantas especiales, etc.
Un constructor con todos los parámetros sería confuso:

```
Auto auto = new Auto("Toyota", "Rojo", 2025, true, false, "V8", "Michelin");
```

¿Entiendes qué significa cada true o false sin mirar la documentación? Muy poco claro.

Idea principal

- Se crea una clase interna o separada llamada **Builder**.
- El Builder tiene **métodos encadenados** (setColor(), setMotor(), etc.) que devuelven this.
- Al final, se llama a .build() para construir el objeto completo.

```
class Pizza {
  private String masa, salsa, relleno;
  static class Builder {
    private String masa, salsa, relleno;
    Builder masa(String m) { this.masa = m; return this; }
    Builder salsa(String s) { this.salsa = s; return this; }
    Builder relleno(String r) { this.relleno = r; return this; }
    Pizza build() {
      Pizza p = new Pizza();
      p.masa = masa; p.salsa = salsa; p.relleno = relleno;
      return p;
    }
  }
}
```

```
class Hamburguesa {
    private String pan;
    private String carne;
    private boolean queso;
    private boolean bacon;

    private Hamburguesa(Builder builder) {
        this.pan = builder.pan;
        this.carne = builder.carne;
        this.queso = builder.queso;
        this.bacon = builder.bacon;
    }

    public void mostrar() {
        System.out.println("Hamburguesa con: " + pan + ", " + carne +
            (queso ? ", queso" : "") + (bacon ? ", bacon" : ""));
    }

    static class Builder {
        private String pan;
        private String carne;
        private boolean queso;
        private boolean bacon;

        public Builder setPan(String pan) { this.pan = pan; return this; }
```

```
        public Builder setCarne(String carne) { this.carne = carne; return this; }
        public Builder addQueso() { this.queso = true; return this; }
        public Builder addBacon() { this.bacon = true; return this; }

        public Hamburguesa build() {
            return new Hamburguesa(this);
        }
    }
}

public class BuilderEjemplo {
    public static void main(String[] args) {
        Hamburguesa h = new Hamburguesa.Builder()
                .setPan("Integral")
                .setCarne("Vacuna")
                .addQueso()
                .addBacon()
                .build();
        h.mostrar();
    }
}
```

5. Prototype

Definición: Permite crear objetos clonando instancias existentes.
Uso: Cuando la creación de objetos es costosa.

El **Prototype** es un patrón creacional que permite **crear nuevos objetos copiando un objeto existente** (clonación), en lugar de instanciarlos desde cero con new.

Idea clave:

- Tienes un **objeto "prototipo"** que sirve como modelo.
- Cada vez que necesitas un objeto igual o similar, **lo clonas**.
- Útil cuando:
    - La creación de un objeto es **costosa** (muchos cálculos o consultas a base de datos).
    - Necesitas **copiar objetos complejos** sin acoplarte a su clase concreta.

Ejemplo Clonar contratos

```
class Contrato implements Cloneable {
    private String cliente;
    private String tipo;

    public Contrato(String cliente, String tipo) {
        this.cliente = cliente;
        this.tipo = tipo;
    }

    @Override
    public Contrato clone() {
        try {
            return (Contrato) super.clone();
```

```
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException(e);
        }
    }

    public void mostrar() {
        System.out.println("Contrato de " + cliente + " - Tipo: " + tipo);
    }
}

public class PrototypeEjemplo {
    public static void main(String[] args) {
        Contrato base = new Contrato("Empresa X", "Licencia Software");
        Contrato copia = base.clone();

        copia.mostrar();
    }
}
```

Caso del mundo real: Juegos de Video – Personajes con configuración compleja

Imaginá que estás desarrollando un **videojuego de estrategia** y tenés un personaje base "soldado" con muchas configuraciones:

- Armas, armaduras, habilidades, puntos de vida, velocidad, posición inicial, inventario, efectos visuales, etc.

Cada soldado que aparece en el juego **tiene la misma base**, pero puede tener algunas modificaciones mínimas (por ejemplo, color de uniforme diferente o arma especial).

Si tuvieras que **crear cada soldado desde cero** (instanciar y setear todas las propiedades), sería muy lento y engorroso.

Solución: **Prototype**

- Tenés un objeto "prototipo" de soldado base.
- Cada vez que necesitas un nuevo soldado, **lo clonas**.
- Solo modificas las propiedades que cambian (ej: color, arma).

Así, **ahorras tiempo de creación y memoria**, y mantenés consistencia en los atributos base.

Beneficio real

- **Velocidad**: No hay que inicializar manualmente todas las propiedades.
- **Consistencia**: Todos los clones tienen los atributos base iguales.
- **Flexibilidad**: Solo cambias lo necesario en cada clon.

Otros casos de uso similares:

- **Documentos predefinidos** (plantillas Word/PDF).
- **Configuraciones de productos** en e-commerce (ej: un producto base con distintas combinaciones).
- **Objetos de red** en simuladores (routers, switches con configuraciones iniciales).