## Guia de Estudio: Patrones Singleton y Factory Method

**1. Introducción a los Patrones de Diseño**

Los patrones de diseño son soluciones probadas a problemas comunes en el desarrollo de software. Entre los patrones creacionales más utilizados encontramos:

* **Singleton:** Garantiza una única instancia de una clase en toda la aplicación.
* **Factory Method:** Facilita la creación de objetos sin especificar sus clases concretas.

**2. Patrón Singleton**

**a) Definición**

El patrón Singleton asegura que una clase tenga una única instancia y proporciona un punto de acceso global a ella.

**b) Caso de Uso: Logger en Singleton**

**Escenario:** Una aplicación bancaria necesita registrar eventos de usuario (inicio de sesión, transacciones, errores). Para evitar múltiples instancias del Logger y asegurar un control centralizado, se implementa el patrón Singleton.

**Implementación en Java:**

class Logger {

private static Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("[LOG] " + message);

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Inicio de sesión");

Logger logger2 = Logger.getInstance();

logger2.log("Transacción realizada");

}

}

**c) Cuándo Usarlo**

* Gestión de recursos compartidos (conexiones a BD, logs).
* Configuración centralizada en la aplicación.

**d) Casos donde NO usar Singleton**

* Aplicaciones altamente concurrentes (puede generar bloqueos).
* Sistemas distribuidos donde cada nodo podría tener una instancia distinta.

**3. Patrón Factory Method**

**a) Definición**

Permite la creación de objetos sin especificar su clase concreta, delegando la lógica de instanciación a una **fábrica**.

**b) Caso de Uso: Creación de Vehículos**

**Escenario:** Una empresa automotriz fabrica autos eléctricos y a gasolina. Se requiere una fábrica que genere instancias sin especificar directamente la clase concreta.

**Implementación en Java:**

interface Vehiculo {

void conducir();

}

class CarroElectrico implements Vehiculo {

public void conducir() {

System.out.println("Conduciendo un carro eléctrico");

}

}

class CarroGasolina implements Vehiculo {

public void conducir() {

System.out.println("Conduciendo un carro a gasolina");

}

}

class FabricaVehiculos {

public static Vehiculo crearVehiculo(String tipo) {

if (tipo.equalsIgnoreCase("electrico")) return new CarroElectrico();

else if (tipo.equalsIgnoreCase("gasolina")) return new CarroGasolina();

return null;

}

}

public class Main {

public static void main(String[] args) {

Vehiculo v1 = FabricaVehiculos.crearVehiculo("electrico");

v1.conducir();

}

}

**c) Cuándo Usarlo**

* Cuando el código necesita flexibilidad en la creación de objetos.
* Para evitar dependencias directas entre clases y sus implementaciones.

**4. Conclusión**

* **Singleton** es ideal para instancias únicas pero puede ser un problema en aplicaciones multihilo.
* **Factory Method** es útil para la flexibilidad en la creación de objetos, reduciendo el acoplamiento.

**Ejemplos Prácticos**

**Ejemplo 1: Implementar Singleton en un servicio de base de datos**

**Historia de usuario:** Un sistema de gestión de pedidos necesita que la conexión a la base de datos sea única para evitar conflictos y mejorar el rendimiento.

**Solución:**

class Database {

private static Database instance;

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

public static Database getInstance() {

if (instance == null) {

instance = new Database();

}

return instance;

}

public void ejecutarConsulta(String consulta) {

System.out.println("Ejecutando: " + consulta);

}

}

public class Main {

public static void main(String[] args) {

Database db1 = Database.getInstance();

db1.ejecutarConsulta("SELECT \* FROM pedidos");

}

}

**Ejemplo 2: Crear una fábrica de figuras geométricas**

**Historia de usuario:** Un software de diseño gráfico necesita una fábrica que genere distintos tipos de figuras sin definir explícitamente sus clases.

**Solución:**

interface Figura {

void dibujar();

}

class Circulo implements Figura {

public void dibujar() {

System.out.println("Dibujando un círculo");

}

}

class Cuadrado implements Figura {

public void dibujar() {

System.out.println("Dibujando un cuadrado");

}

}

class FiguraFactory {

public static Figura crearFigura(String tipo) {

if (tipo.equalsIgnoreCase("circulo")) return new Circulo();

else if (tipo.equalsIgnoreCase("cuadrado")) return new Cuadrado();

return null;

}

}

public class Main {

public static void main(String[] args) {

Figura f1 = FiguraFactory.crearFigura("circulo");

f1.dibujar();

}

}

## Ejercicio: Refactorizar código para usar Factory Method

**Historia de usuario 1:** Un sistema de vehículos necesita eliminar la creación manual de objetos y aplicar Factory Method.

**Código actual:**

Vehiculo v1 = new CarroElectrico();

Vehiculo v2 = new CarroGasolina();

**Historia de usuario 2:** Un sistema de liquidación de pensiones requiere procesar varias leyes de liquidación de pensión para realizar el pago mensual, para ello se necesita que el salario de pensión sea liquidado con base en la ley a la que pertenece, con los siguientes datos:

Ley 78: 15 salarios

Ley 86: 5 salarios

Ley 98: 2.5 salarios

Ley 100: promedio últimos 10 años

Ley Petro: tope máximo 3.5 salarios y menores solo el 70%

Escenario Población:

Ley 78: 3 personas salario $850mil

Ley 86: 2 personas salario $930mil

Ley 98: 3 personas salario $1020mil, 2 personas salario doble

Ley 100: 2 personas salario $1200mil últimos 10 años, 1 persona $1230mil últimos 5 años +$1300 ultimos 10-5 años

Ley Petro: 4 personas 4 salarios mínimos de $1400mil, 3 personas $1400mil