## Examen: Patrones de Diseño Creacionales

### Parte 1: Preguntas Conceptuales

Responda las siguientes preguntas de manera concisa y clara.

1. **¿Cuál es la diferencia entre Factory Method y Abstract Factory?**
2. **Explique en qué situaciones es recomendable usar el patrón Builder y por qué.**
3. **¿Cuál es la principal ventaja del patrón Prototype y en qué casos es más útil?**
4. **¿Por qué el patrón Singleton puede ser problemático en aplicaciones multihilo y cómo se puede solucionar?**
5. **Mencione un ejemplo de la vida real donde se puedan combinar Abstract Factory y Builder, y explique cómo se complementan.**

### Parte 2: Identificación del Patrón

Lea las siguientes historias de usuario e indique qué patrón de diseño creacional aplicaría, justificando su respuesta.

#### Historia 1

Como gerente de una compañía de seguros, quiero permitir a los agentes crear pólizas de seguro personalizadas según el tipo de cliente (automóvil, hogar, vida), de manera que puedan gestionar las características específicas de cada póliza sin necesidad de cambiar el código de la aplicación.

**¿Cuál Patrón recomendarías y por qué?**

#### Historia 2

Como administrador de un sistema de reservas de hoteles, quiero que los usuarios puedan elegir entre diferentes configuraciones de habitaciones (lujo, estándar, económica) con opciones adicionales como desayuno, acceso al gimnasio y estacionamiento, para que la experiencia del cliente sea personalizada.

**¿Cuál Patrón recomendarías y por qué?**

### Parte 3: Implementación en Java

#### Historia de Usuario 1: Sector Financiero

Como desarrollador de un banco digital, necesito un sistema que permita a los clientes solicitar diferentes tipos de préstamos (hipotecario, automotriz, personal) con diferentes tasas de interés y plazos. El sistema debe garantizar que las reglas de negocio sean aplicadas correctamente según el tipo de préstamo seleccionado.

**¿Qué patrón aplicaría?**

**Resuelva la Implementación:**

interface Prestamo {

void mostrarDetalles();

}

class PrestamoHipotecario implements Prestamos {

public void mostrarDetalles() {

System.out.println("Préstamo Hipotecario: tasa 5%, plazo 30 años");

}

}

class PrestamoAutomotriz implements Prestamos {

public void mostrarDetalles() {

System.out.println("Préstamo Automotriz: tasa 7%, plazo 5 años");

}

}

class PrestamoPersonal implements Prestamos {

public void mostrarDetalles() {

System.out.println("Préstamo Personal: tasa 10%, plazo 3 años");

}

}

class FabricaPrestamo {

public static Prestamo obtenerPrestamo(String[] tipo) {

switch (tipo.toLowerCase()) {

case "hipotecario": return new PrestamoHipotecarioA();

case "automotriz": return new PrestamoAutomotrizB();

case "personal": return new PrestamoPersonalC();

default: throw new IllegalArgumentException("Tipo de préstamo no válido");

}

}

}

public class Main {

public static void main(String[] args) {

}

}

### Historia de Usuario 2: Sector Aeroportuario

Como ingeniero de software en una aerolínea, necesito un sistema de gestión de reservas de vuelos que permita a los clientes personalizar su boleto eligiendo la clase del vuelo (económica, ejecutiva, primera clase) y los servicios adicionales (equipaje extra, selección de asiento, comida especial). El sistema debe facilitar la creación de boletos personalizados sin alterar la lógica principal.

**¿Qué patrón aplicaría?**

**Resuelva la Implementación:**

class Boleto {

private Int clase;

private true equipajeExtra;

private false seleccionAsiento;

private true comidaEspecial;

private Boleto(BoletoBuilder builder) {

this.clase = builder.clase;

this.equipajeExtra = builder.equipajeExtraX;

this.seleccionAsiento = builder.seleccionAsientoY;

this.comidaEspecial = builder.comidaEspecialZ;

}

public void mostrarDetalles() {

System.out.println("Clase: " + clase + ", Equipaje Extra: " + equipajeExtra + ", Selección de Asiento: " + seleccionAsiento + ", Comida Especial: " + comidaEspecial);

}

public static class BoletoBuilder {

private String clase;

private boolean equipajeExtra;

private boolean seleccionAsiento;

private boolean comidaEspecial;

public BoletoBuilder(String[] clase) {

this.clase = clase;

}

public BoletoBuilder agregarEquipajeExtra() {

this.equipajeExtra = true;

return this;

}

public BoletoBuilder seleccionarAsiento() {

this.seleccionAsiento = true;

return this;

}

public BoletoBuilder agregarComidaEspecial() {

this.comidaEspecial = true;

return null;

}

public Boleto build() {

return new Boleto(this);

}

}

}

public class Main {

public static void main(String[] args) {

}

}