## Guia de Estudio: Abstract Factory, Builder y Prototype

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

Los patrones de diseño creacionales ayudan a gestionar la creación de objetos de manera eficiente y estructurada. En esta guía exploraremos:

* **Abstract Factory:** Creación de familias de objetos relacionados sin especificar sus clases concretas.
* **Builder:** Construcción de objetos complejos paso a paso.
* **Prototype:** Creación de nuevos objetos mediante la clonación de objetos existentes.

**2. Patrón Abstract Factory**

**a) Definición**

El patrón **Abstract Factory** proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas. Es útil cuando se necesita garantizar la compatibilidad entre productos de una misma familia.

**b) Cuándo utilizarlo**

* Cuando se necesita garantizar la compatibilidad entre productos relacionados.
* Cuando se desea encapsular la lógica de creación de objetos.
* Cuando el código debe ser extensible para admitir nuevas familias de productos sin modificar el código cliente.

**c) Cuándo NO utilizarlo**

* Cuando solo se necesita crear un objeto simple sin dependencias con otros.
* Cuando la cantidad de productos no está claramente definida, lo que haría que la fábrica se vuelva difícil de mantener.

**d) Caso de Uso: Creación de una Interfaz Gráfica Adaptable**

**Escenario:** Un sistema operativo permite cambiar el tema de su interfaz gráfica (Light Mode y Dark Mode). Cada tema debe proporcionar sus propios botones y ventanas con estilos coherentes.

**Implementación en Java:**

interface Boton {

void render();

}

// Interfaz común para Ventana

interface Ventana {

void mostrar();

}

// Implementaciones para Light Mode

class BotonLight implements Boton {

public void render() { System.out.println("Renderizando botón Light"); }

}

class VentanaLight implements Ventana {

public void mostrar() { System.out.println("Mostrando ventana Light"); }

}

// Implementaciones para Dark Mode

class BotonDark implements Boton {

public void render() { System.out.println("Renderizando botón Dark"); }

}

class VentanaDark implements Ventana {

public void mostrar() { System.out.println("Mostrando ventana Dark"); }

}

// Abstract Factory

interface GUIFactory {

Boton crearBoton();

Ventana crearVentana();

}

// Fábricas concretas

class LightFactory implements GUIFactory {

public Boton crearBoton() { return new BotonLight(); }

public Ventana crearVentana() { return new VentanaLight(); }

}

class DarkFactory implements GUIFactory {

public Boton crearBoton() { return new BotonDark(); }

public Ventana crearVentana() { return new VentanaDark(); }

}

// Cliente

public class Main {

public static void main(String[] args) {

GUIFactory factory = new DarkFactory(); // Puede cambiarse por LightFactory

Boton boton = factory.crearBoton();

Ventana ventana = factory.crearVentana();

boton.render();

ventana.mostrar();

}

}

**3. Patrón Builder**

**a) Definición**

El patrón **Builder** permite la construcción paso a paso de objetos complejos. Es útil cuando un objeto tiene múltiples atributos opcionales o configuraciones variadas.

**b) Cuándo utilizarlo**

* Cuando se deben construir objetos con múltiples configuraciones opcionales.
* Cuando el proceso de creación del objeto es complejo y requiere varios pasos.
* Cuando se necesita evitar múltiples constructores con diferentes combinaciones de parámetros.

**c) Cuándo NO utilizarlo**

* Cuando los objetos a construir son simples y no requieren pasos intermedios.
* Cuando el número de atributos del objeto es fijo y no varía en diferentes casos de uso.

**d) Caso de Uso: Creación de un Usuario con Múltiples Atributos**

**Escenario:** En una aplicación, los usuarios pueden configurarse con diferentes atributos como nombre, edad, dirección y número de teléfono. No todos los atributos son obligatorios, por lo que es conveniente usar **Builder** para evitar múltiples constructores.

**Implementación en Java:**

class Usuario {

private String nombre;

private int edad;

private String direccion;

private String telefono;

private Usuario(UsuarioBuilder builder) {

this.nombre = builder.nombre;

this.edad = builder.edad;

this.direccion = builder.direccion;

this.telefono = builder.telefono;

}

public static class UsuarioBuilder {

private String nombre;

private int edad;

private String direccion;

private String telefono;

public UsuarioBuilder(String nombre) {

this.nombre = nombre;

}

public UsuarioBuilder setEdad(int edad) {

this.edad = edad;

return this;

}

public UsuarioBuilder setDireccion(String direccion) {

this.direccion = direccion;

return this;

}

public UsuarioBuilder setTelefono(String telefono) {

this.telefono = telefono;

return this;

}

public Usuario build() {

return new Usuario(this);

}

}

public String toString() {

return "Usuario: " + nombre + ", Edad: " + edad + ", Dirección: " + direccion + ", Teléfono: " + telefono;

}

}

public class Main {

public static void main(String[] args) {

Usuario usuario = new Usuario.UsuarioBuilder("Juan Pérez")

.setEdad(30)

.setDireccion("Av. Principal 123")

.setTelefono("123456789")

.build();

System.out.println(usuario);

}

}

**4. Patrón Prototype**

**a) Definición**

El patrón **Prototype** permite la creación de nuevos objetos mediante la clonación de una instancia existente, en lugar de crear una nueva desde cero. Es útil cuando el costo de crear un objeto desde cero es elevado o cuando los objetos tienen muchas configuraciones similares.

**b) Cuándo utilizarlo**

* Cuando la creación de un objeto es costosa en términos de tiempo o recursos.
* Cuando se necesita evitar la sobrecarga de múltiples inicializaciones similares.
* Cuando se desea mantener la independencia de la implementación en la clonación de objetos.

**c) Cuándo NO utilizarlo**

* Cuando los objetos son simples y pueden ser creados fácilmente sin necesidad de clonación.
* Cuando los objetos contienen referencias complejas a otros objetos, lo que puede complicar la clonación profunda.

**d) Caso de Uso: Clonación de Documentos**

**Escenario:** Un sistema de gestión documental permite a los usuarios crear documentos a partir de una plantilla preexistente en lugar de configurarlos manualmente desde cero.

**Implementación en Java:**

1. class Documento implements Cloneable {

2. private String contenido;

3.

4. public Documento(String contenido) {

5. this.contenido = contenido;

6. }

7.

8. public void mostrar() {

9. System.out.println("Contenido del documento: " + contenido);

10. }

11.

12. @Override

13. protected Documento clone() throws CloneNotSupportedException {

14. return (Documento) super.clone();

15. }

16. }

17.

18. public class Main {

19. public static void main(String[] args) {

20. try {

21. Documento docOriginal = new Documento("Plantilla Base");

22. Documento docClonado = docOriginal.clone();

23.

24. docOriginal.mostrar();

25. docClonado.mostrar();

26. } catch (CloneNotSupportedException e) {

27. e.printStackTrace();

28. }

29. }

30. }

31.

**5. Conclusión**

* **Abstract Factory** permite la creación de familias de objetos sin acoplarse a implementaciones concretas.
* **Builder** simplifica la creación de objetos complejos de forma estructurada.
* **Prototype** permite la clonación eficiente de objetos, reduciendo el costo de creación.
* Todos estos patrones mejoran la flexibilidad y mantenibilidad del código.