**PATRONES DE DISEÑO CREACIONALES**

Los patrones creacionales proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente. Estos nos proveen soluciones para la creación de objetos, permitiéndonos hacer un sistema independiente de cómo sus objetos son creados.

**Abstract Factory**

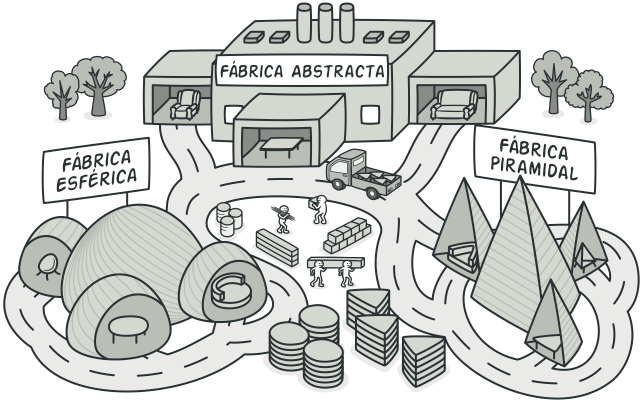
Propósito: Proveer una interfaz para la creación de familias u objetos dependientes relacionados, sin especificar sus clases concretas.

Es una jerarquía que encapsula muchas familias posibles y la creación de un conjunto de productos. El objeto "fábrica" tiene la responsabilidad de proporcionar servicios de creación para toda una familia de productos. Los "clientes" nunca crean directamente los objetos de la familia, piden la fábrica que los cree por ellos.



**Aplicación:**

* Cuando tenemos una o múltiples familias de productos.
* Cuando tenemos muchos objetos que pueden ser cambiados o agregados durante el tiempo de ejecución.
* Cuando queremos obtener un objeto compuesto de otros objetos, los cuales desconocemos a que clase pertenecen.
* Para encapsular la creación de muchos objetos.

[](https://camo.githubusercontent.com/bdf9a043c0bca98cad86df3dbb602c1a02955e0c9e635402e15f8f9154e96f7c/68747470733a2f2f7265666163746f72696e672e677572752f696d616765732f7061747465726e732f636f6e74656e742f61627374726163742d666163746f72792f61627374726163742d666163746f72792d65732e706e673f69643d3033373863396661636133396166613230653431613464333765376533383238)

**Implementación**

Queremos crear una serie de vehículos, específicamente automóviles y motos, que se pueden fabricar tanto en versión estándar como deportiva. Utilizaremos el patrón Abstract Factory para crear estos vehículos sin especificar directamente las clases concretas de los vehículos.

**Ejemplo de vehículos:**

1. Definimos las interfaces para los vehículos.

// Interfaz para los automóviles

interface Car {

void drive();

}

// Interfaz para las motos

interface Motorcycle {

void ride();

}

1. Creamos las implementaciones concretas para la versión estándar.

// Implementación concreta de un automóvil estándar

class StandardCar implements Car {

@Override

public void drive() {

System.out.println("Driving a standard car.");

}

}

// Implementación concreta de una moto estándar

class StandardMotorcycle implements Motorcycle {

@Override

public void ride() {

System.out.println("Riding a standard motorcycle.");

}

}

1. Creamos las implementaciones concretas para la versión deportiva.

// Implementación concreta de un automóvil deportivo

class SportsCar implements Car {

@Override

public void drive() {

System.out.println("Driving a sports car.");

}

}

// Implementación concreta de una moto deportiva

class SportsMotorcycle implements Motorcycle {

@Override

public void ride() {

System.out.println("Riding a sports motorcycle.");

}

}

1. Definimos la fábrica abstracta.

// Fábrica abstracta que crea automóviles y motos

interface VehicleFactory {

Car createCar();

Motorcycle createMotorcycle();

}

1. Creamos las fábricas concretas para los vehículos estándar y deportivos.

// Fábrica concreta para vehículos estándar

class StandardVehicleFactory implements VehicleFactory {

@Override

public Car createCar() {

return new StandardCar();

}

@Override

public Motorcycle createMotorcycle() {

return new StandardMotorcycle();

}

}

// Fábrica concreta para vehículos deportivos

class SportsVehicleFactory implements VehicleFactory {

@Override

public Car createCar() {

return new SportsCar();

}

@Override

public Motorcycle createMotorcycle() {

return new SportsMotorcycle();

}

}

1. Utilizamos las fábricas en el cliente.

public class Main {

public static void main(String[] args) {

// Elegir la fábrica concreta

VehicleFactory factory = new SportsVehicleFactory(); // O puede ser StandardVehicleFactory

// Crear vehículos usando la fábrica

Car car = factory.createCar();

Motorcycle motorcycle = factory.createMotorcycle();

// Usar los vehículos

car.drive();

motorcycle.ride();

}

}

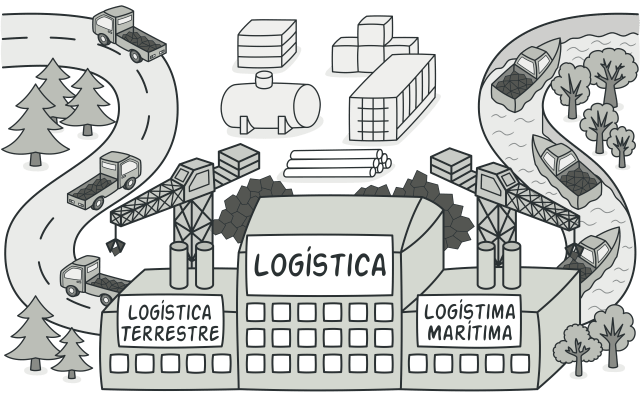
**Factory Method**

Propósito: Definir una interface para crear un objeto, dejando a las subclases decidir de que tipo de clase se realizará la instancia en tiempo de ejecución. Reducir el uso del operador new.

Crear objetos en una clase usando un método factory es más flexible que crear un objeto directamente. Es posible conectar la generación de familias de clases que tienen comportamientos en común. Elimina la necesidad de estar haciendo binding (casting) hacia clases específicas dentro del código, ya que éste sólo se entiende con las clases abstractas.

Aplicación:

* Cuando una clase no puede anticipar que clase de objetos debe crear, esto se sabe durante el tiempo de ejecución.
* Cuando un método regresa una de muchas posibles clases que comparten carecterísticas comunes a través de una superclase.
* Para encapsular la creación de objetos.

[](https://camo.githubusercontent.com/db5b9f68c18f96904207fbd503bde5de47a331e06c7c76a4c9b58796a746392b/68747470733a2f2f7265666163746f72696e672e677572752f696d616765732f7061747465726e732f636f6e74656e742f666163746f72792d6d6574686f642f666163746f72792d6d6574686f642d65732e706e67)

**Implementación**

**Ejemplo jerarquía de vehículos:**

1. Define una interfaz o clase abstracta para los productos.

// Vehículo.java

public interface Vehiculo {

void conducir();

}

1. Crea implementaciones concretas de la interfaz.

// Coche.java

public class Coche implements Vehiculo {

@Override

public void conducir() {

System.out.println("Conduciendo un coche.");

}

}

// Motocicleta.java

public class Motocicleta implements Vehiculo {

@Override

public void conducir() {

System.out.println("Conduciendo una motocicleta.");

}

}

1. Define la clase creatora abstracta.

// FabricaVehiculos.java

public abstract class FabricaVehiculos {

public abstract Vehiculo crearVehiculo();

}

1. Implementa las fábricas concretas.

// FabricaCoche.java

public class FabricaCoche extends FabricaVehiculos {

@Override

public Vehiculo crearVehiculo() {

return new Coche();

}

}

// FabricaMotocicleta.java

public class FabricaMotocicleta extends FabricaVehiculos {

@Override

public Vehiculo crearVehiculo() {

return new Motocicleta();

}

}

1. Utiliza el patrón Factory Method.

// Main.java

public class Main {

public static void main(String[] args) {

FabricaVehiculos fabricaCoche = new FabricaCoche();

Vehiculo coche = fabricaCoche.crearVehiculo();

coche.conducir();

FabricaVehiculos fabricaMotocicleta = new FabricaMotocicleta();

Vehiculo motocicleta = fabricaMotocicleta.crearVehiculo();

motocicleta.conducir();

}

}

**Builder**

Propósito: Separar la construcción de un objeto complejo de su representación para que el mismo proceso de construcción puede crear diferentes representaciones.

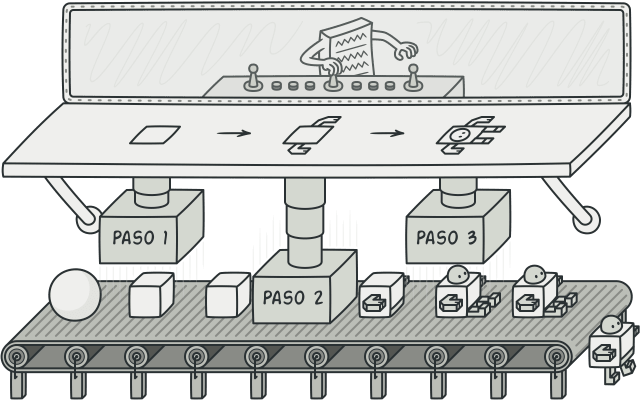
Nos permite crear un objeto que está compuesto por muchos otros objetos. Sólo el "Builder" conoce a detalle las clases concretas de los objetos que serán creados, nadie más.

En este patrón intervienen un "Director" y un "Builder". El "Director" invoca los servicios del "Builder" el cual va creando las partes de un objeto complejo y al mismo tiempo guardo un estado intermedio de la construcción del objeto. Cuando el producto se ha construido por completo el cliente recupera el resultado.

A diferencia de otros patrones creacionales que construyen productos de una sola vez, el patrón "Builder" construye paso a paso los productos bajo el control del "Director".

Aplicación:

* Construir un objeto compuesto de otros objetos.
* Que la creación de las partes de un objeto sea independiente del objeto principal.
* Ocultar la creación de las partes de un objeto del cliente, de esta manera no existe dependencia entre el cliente y las partes.

[](https://camo.githubusercontent.com/c3122987412e6616c578265d7853593523c073450c37bf151452e887070fe229/68747470733a2f2f7265666163746f72696e672e677572752f696d616765732f7061747465726e732f636f6e74656e742f6275696c6465722f6275696c6465722d65732e706e67)

**Implementación**

Imaginemos que estamos construyendo una clase Pizza que tiene varios atributos opcionales como el tipo de masa, el tamaño, los ingredientes, etc. Vamos a usar el patrón Builder para crear objetos Pizza.

**Ejemplo Pizza:**

1. Crear la clase Pizza con los atributos que queremos construir.

public class Pizza {

private String masa;

private String tamaño;

private String queso;

private String ingredientes;

private Pizza(PizzaBuilder builder) {

this.masa = builder.masa;

this.tamaño = builder.tamaño;

this.queso = builder.queso;

this.ingredientes = builder.ingredientes;

}

@Override

public String toString() {

return "Pizza{" +

"masa='" + masa + '\'' +

", tamaño='" + tamaño + '\'' +

", queso='" + queso + '\'' +

", ingredientes='" + ingredientes + '\'' +

'}';

}

public static class PizzaBuilder {

private String masa;

private String tamaño;

private String queso;

private String ingredientes;

public PizzaBuilder setMasa(String masa) {

this.masa = masa;

return this;

}

public PizzaBuilder setTamaño(String tamaño) {

this.tamaño = tamaño;

return this;

}

public PizzaBuilder setQueso(String queso) {

this.queso = queso;

return this;

}

public PizzaBuilder setIngredientes(String ingredientes) {

this.ingredientes = ingredientes;

return this;

}

public Pizza build() {

return new Pizza(this);

}

}

}

2: Usar el Builder para construir una Pizza paso a paso.

public class Main {

public static void main(String[] args) {

Pizza pizza = new Pizza.PizzaBuilder()

.setMasa("delgada")

.setTamaño("grande")

.setQueso("mozzarella")

.setIngredientes("pepperoni, champiñones")

.build();

System.out.println(pizza);

}

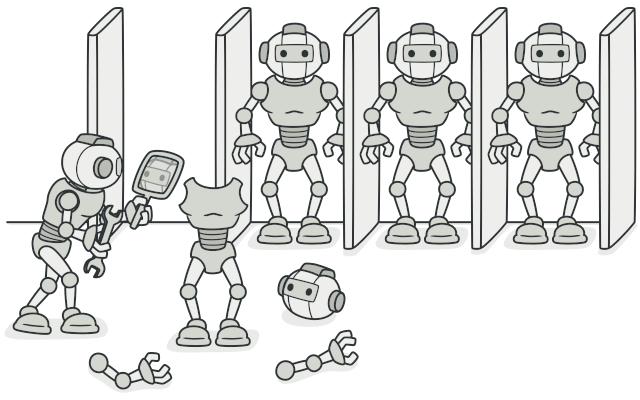
}

**Prototype**

Propósito: Especificar varios tipos de objetos que pueden ser creados en un prototipo para crear nuevos objetos copiando ese prototipo. Reduce la necesidad de crear subclases.

Aplicación:

* Queremos crear nuevos objetos mediante la clonación o copia de otros.
* Cuando tenemos muchas clases potenciales que queremos usar sólo si son requeridas durante el tiempo de ejecución.

[](https://camo.githubusercontent.com/4118eeb4f3352df75b3a9e8d6cd45cb6951d3d5fd8261b236d8eb082250c4f3e/68747470733a2f2f7265666163746f72696e672e677572752f696d616765732f7061747465726e732f636f6e74656e742f70726f746f747970652f70726f746f747970652e706e67)

**Implementación**

**Ejemplo Prototipo:**

1. Define la interfaz Prototype: Esta interfaz debe declarar un método clone que permita copiar el objeto.

public interface Prototype {

Prototype clone();

}

1. Crea una clase concreta que implemente Prototype: En este caso, vamos a crear una clase ConcretePrototype que implementa la interfaz Prototype.

public class ConcretePrototype implements Prototype {

private String field1;

private int field2;

public ConcretePrototype(String field1, int field2) {

this.field1 = field1;

this.field2 = field2;

}

public String getField1() {

return field1;

}

public void setField1(String field1) {

this.field1 = field1;

}

public int getField2() {

return field2;

}

public void setField2(int field2) {

this.field2 = field2;

}

@Override

public Prototype clone() {

// Aquí se realiza una copia superficial del objeto.

return new ConcretePrototype(this.field1, this.field2);

}

@Override

public String toString() {

return "ConcretePrototype{" +

"field1='" + field1 + '\'' +

", field2=" + field2 +

'}';

}

}

1. Utiliza el patrón Prototype en una clase cliente: Puedes usar el método clone para crear copias del objeto en lugar de construir nuevos.

public class PrototypeDemo {

public static void main(String[] args) {

// Crear un objeto original

ConcretePrototype original = new ConcretePrototype("Original", 123);

// Clonar el objeto

ConcretePrototype clone = (ConcretePrototype) original.clone();

// Mostrar los resultados

System.out.println("Original: " + original);

System.out.println("Clone: " + clone);

// Modificar el clone para demostrar que es una copia independiente

clone.setField1("Clone");

clone.setField2(456);

// Mostrar los resultados después de la modificación

System.out.println("Original después de modificar el clone: " + original);

System.out.println("Clone después de la modificación: " + clone);

}

}

**Singleton**

Propósito: Asegurar que una clase tenga una única instancia y proporcionar un punto de acceso global a la misma. El cliente llama a la función de acceso cuando se requiere una referencia a la instancia única.

Aplicación:

* La aplicación necesita una, y sólo una, instancia de una clase, además está instancia requiere ser accesible desde cualquier punto de la aplicación.
* Tipicamente para:
  + Manejar conexiones con la base de datos.
  + La clase para hacer Login.

[](https://camo.githubusercontent.com/f0db7b0d952e54cb79064879eb748de18d8e369a52528ff7bcb52b768f8f55ed/68747470733a2f2f7265666163746f72696e672e677572752f696d616765732f7061747465726e732f636f6e74656e742f73696e676c65746f6e2f73696e676c65746f6e2e706e67)

**Implementación**

Supongamos que queremos modelar una configuración global para una aplicación, como una configuración de registro (logging). Usaremos el patrón Singleton para asegurarnos de que solo haya una instancia de la configuración en toda la aplicación.

**Ejemplo configuración aplicación:**

Imagina que queremos tener una clase Configuracion que almacena la configuración global para la aplicación y queremos asegurarnos de que solo haya una instancia de esta configuración.

// Clase Configuracion con el patrón Singleton

public class Configuracion {

// Instancia privada estática de Configuracion

private static Configuracion instancia;

// Propiedad de configuración

private String nivelDeRegistro;

// Constructor privado para evitar instanciación externa

private Configuracion() {

// Inicializar la configuración con un valor por defecto

nivelDeRegistro = "INFO";

}

// Método público estático para obtener la instancia única

public static Configuracion getInstancia() {

if (instancia == null) {

// Sincronización para manejar entornos multihilo

synchronized (Configuracion.class) {

if (instancia == null) {

instancia = new Configuracion();

}

}

}

return instancia;

}

// Método para obtener el nivel de registro

public String getNivelDeRegistro() {

return nivelDeRegistro;

}

// Método para establecer el nivel de registro

public void setNivelDeRegistro(String nivelDeRegistro) {

this.nivelDeRegistro = nivelDeRegistro;

}

}

// Clase principal para probar la configuración

public class Main {

public static void main(String[] args) {

// Obtener la instancia única de Configuracion

Configuracion config1 = Configuracion.getInstancia();

Configuracion config2 = Configuracion.getInstancia();

// Comprobar que ambas instancias son la misma

if (config1 == config2) {

System.out.println("Las dos instancias son iguales.");

} else {

System.out.println("Las instancias son diferentes.");

}

// Usar la instancia única para configurar y obtener el nivel de registro

config1.setNivelDeRegistro("DEBUG");

System.out.println("Nivel de registro: " + config2.getNivelDeRegistro());

}

}