



Factory Pattern

**La clave para un código más limpio,
flexible y fácil de mantener**



PABLO DEL ÁLAMO

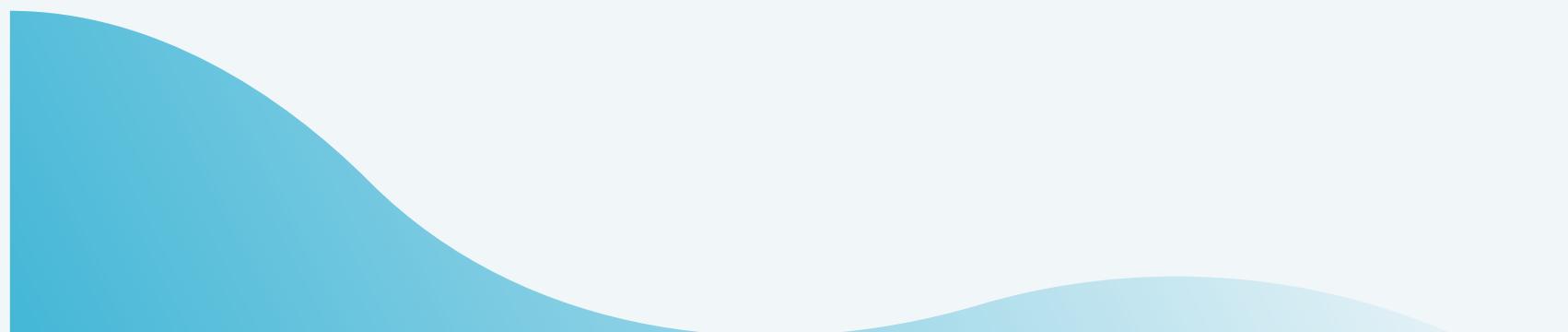


Introducción

El Patrón Factory es como un chef privado, crea objetos (platos) para ti sin mostrarte la receta exacta.

Úsalo para mantener tu código limpio, organizado y optimizado para escalar.

Vamos a profundizar en cómo funciona este patrón.



PABLO DEL ÁLAMO



¿Por qué usar Factory?

¿Cuántas veces has tenido que modificar tu instanciación de objetos en mil lugares después de un cambio?

El patrón Factory encapsula este proceso, permitiendo un único punto de modificación.

Te da la flexibilidad de cambiar la lógica de creación sin tocar el resto de tu código.



PABLO DEL ÁLAMO

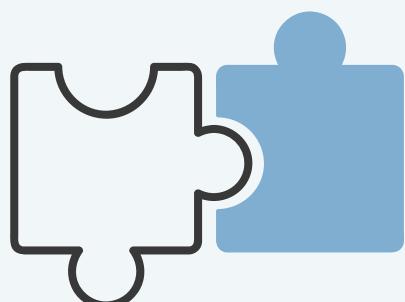


Concepto Básico

Piensa en una fábrica que recibe órdenes y te entrega productos terminados, sin que tengas que entender cómo son creados.

Deja de lado el código enrevesado con múltiples instancias directas de una clase.

Eso es lo que Factory Pattern trae a la mesa: simplicidad y organización.



PABLO DEL ÁLAMO



¿Cómo funciona el Patrón Factory?

El patrón utiliza una interfaz o una clase abstracta para especificar el tipo de objetos que deben ser creados.

La responsabilidad de la instanciación se delega a subclases concretas que conocen el tipo de objeto a crear.

Esto permite que el código que usa estos objetos no conozca detalles de implementación.

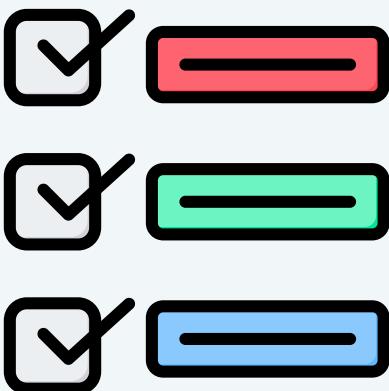


PABLO DEL ÁLAMO



Elementos Clave

1. Interfaz General: Define el tipo de objeto a crear.
2. Clases Concretas: Implementan la interfaz para crear el objeto 'real'.
3. Una fábrica/factoría: Es la encargada de decidir qué clase concreta debe instanciar, basándose en alguna lógica de negocio.



PABLO DEL ÁLAMO

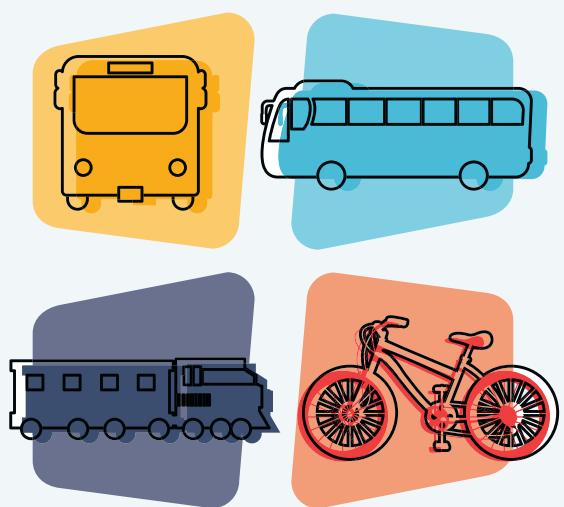


Ejemplo Real - Vehículos

Imagínate una fábrica que produce diferentes tipos de vehículos: coches, bicicletas y motos.

La fábrica recibe un tipo y decide qué vehículo devolver, según sea necesario.

¿Necesitas un coche o una moto? La fábrica se encarga, tú solo esperas el producto.



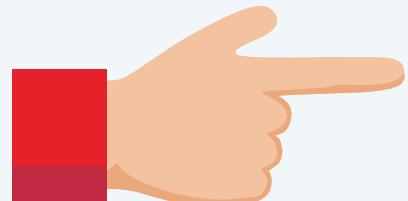
PABLO DEL ÁLAMO



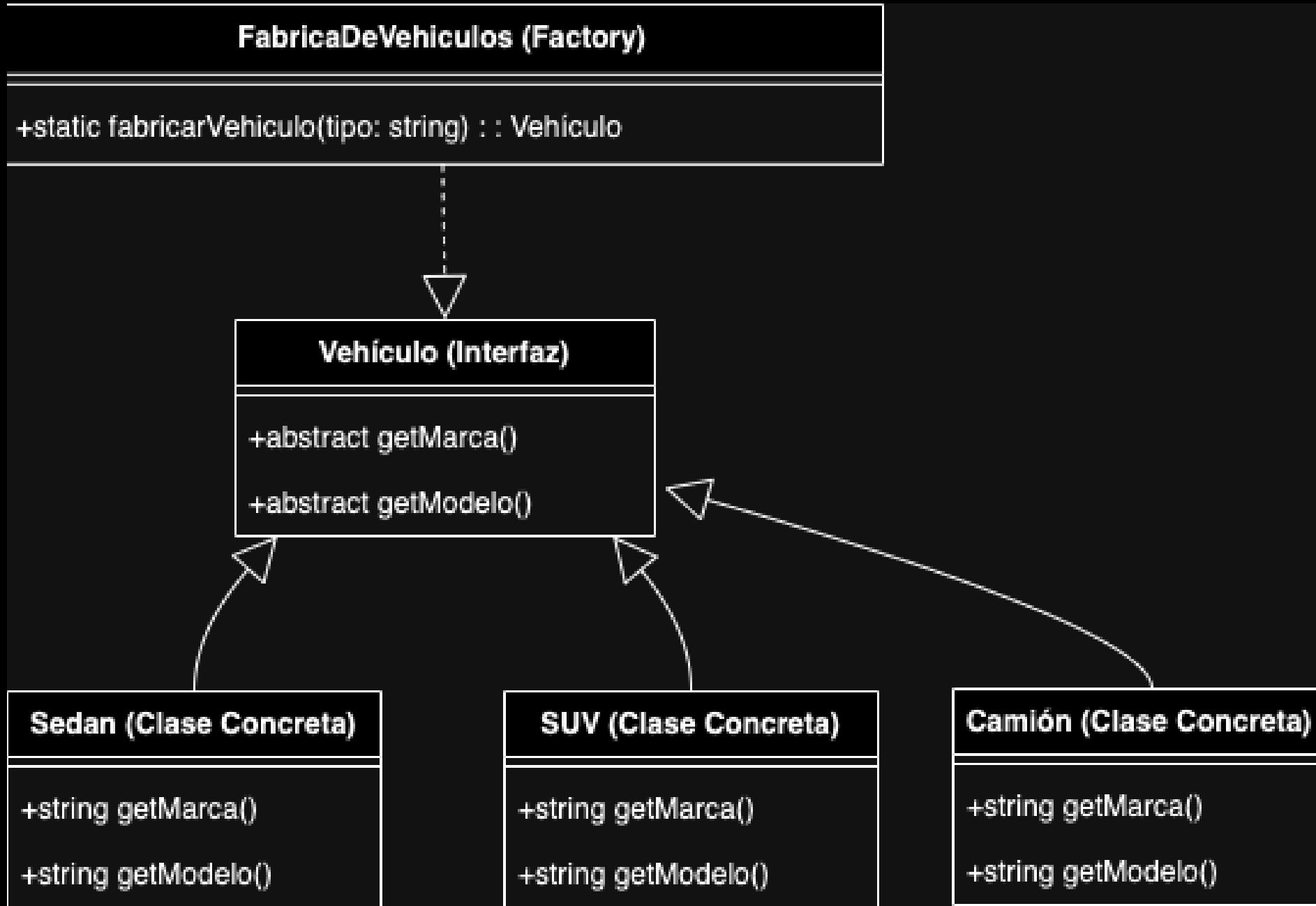
Diagrama Ejemplo

Este diagrama ilustra cómo una simple Factory decide entre crear instancias de Sedan, SUV o Camión.

Cualquier cambio en las clases concretas no afecta al cliente, mantiene tu código desacoplado.



PABLO DEL ÁLAMO





¿Qué es una Fábrica?

Una fábrica en el contexto del diseño de software es una clase especial que simplemente crea instancias de otras clases.

Su meta es aislar la lógica de instanciación y administrar cómo y cuándo ciertas clases complejas deberían ser creadas.

De este modo, facilita el manejo de la complejidad y promueve el reciclaje de código.



PABLO DEL ÁLAMO



Pseudocódigo

```
interface Vehículo {  
    conducir();  
}
```

```
class Sedan implements Vehículo {  
    conducir() {  
        /* Conduce un sedán */  
    }  
}
```

```
class Factory {  
    Vehículo creaVehículo(String tipo) {  
        // Lógica para decidir qué Vehículo instanciar  
    }  
}
```



PABLO DEL ÁLAMO



Ventajas del Patrón Factory

1. Desacoplamiento: Mantiene separados el código que usa los objetos y el que los crea.
2. Escalabilidad: Añadir nuevas clases de productos es sencillo, solo ajustas la factory.
3. Mantenimiento: Facilita una modificación más fácil y económica si algo cambia.

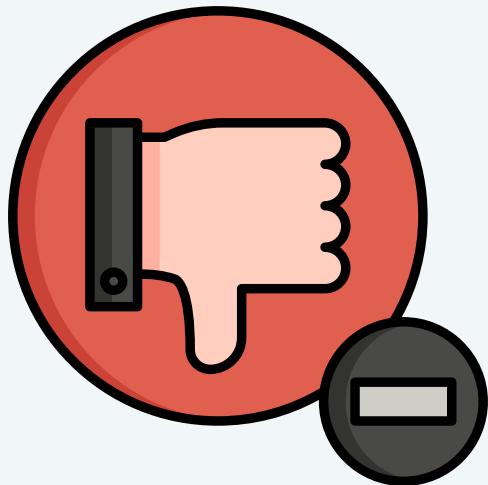


PABLO DEL ÁLAMO



Desventajas

1. Complejidad añadida: Puede complicar la solución si no es necesario usarlo.
2. Difícil trazabilidad de la lógica si está demasiado dividida.



PABLO DEL ÁLAMO



Casos de Uso Comunes

El Factory Pattern es ideal cuando:

- Tienes un sistema que trabaja con muchos tipos de objetos interconectados.
- Quieres centralizar la lógica de creación evitando duplicación de código.
- Es posible que las clases concretas cambien o evolucionen con el tiempo.



PABLO DEL ÁLAMO



¿Cuándo NO usarlo?

Este patrón no tiene por qué ser siempre la mejor solución.

Evítalo si tienes una estructura de clase simple, y tu aplicación no requiere flexibilidad.

Usa 'new' directamente si el patrón añade más complejidad que soluciones a problemas reales.



PABLO DEL ÁLAMO



Código Ejemplo en Java

Vamos a ver cómo esto se traduce en un código práctico usando Java.

La Factory crea diferentes tipos de vehículos según el tipo pasado.

Este ejemplo nos ayuda a ver cómo centralizar y simplificar la creación de instancias.



PABLO DEL ÁLAMO

```
1 public interface Vehiculo {  
2     void conducir();  
3     String getDescripcion();  
4     int getCapacidadCarga();  
5 }  
6
```

```
1 public class Sedan implements Vehiculo {
2     private String descripcion = "Sedán de 4 puertas";
3     private int capacidadCarga = 500; // capacidad en kg
4
5     @Override
6     public void conducir() {
7         System.out.println("Conduciendo un " + descripcion);
8     }
9
10    @Override
11    public String getDescripcion() {
12        return descripcion;
13    }
14
15    @Override
16    public int getCapacidadCarga() {
17        return capacidadCarga;
18    }
19 }
20
21 public class SUV implements Vehiculo {
22     private String descripcion = "SUV todoterreno";
23     private int capacidadCarga = 1000;
24
25     @Override
26     public void conducir() {
27         System.out.println("Conduciendo un " + descripcion);
28     }
29
30     @Override
31     public String getDescripcion() {
32         return descripcion;
33     }
34
35     @Override
36     public int getCapacidadCarga() {
37         return capacidadCarga;
38     }
39 }
40
41 public class Camion implements Vehiculo {
42     private String descripcion = "Camión de carga";
43     private int capacidadCarga = 5000;
44
45     @Override
46     public void conducir() {
47         System.out.println("Conduciendo un " + descripcion);
48     }
49
50     @Override
51     public String getDescripcion() {
52         return descripcion;
53     }
54
55     @Override
56     public int getCapacidadCarga() {
57         return capacidadCarga;
58     }
59 }
60
```

```
3  public static Vehiculo crearVehiculo(String tipo) {  
4      switch (tipo.toLowerCase()) {  
5          case "sedan":  
6              return new Sedan();  
7          case "suv":  
8              return new SUV();  
9          case "camion":  
10             return new Camion();  
11         default:  
12             throw new IllegalArgumentException("Tipo de vehículo desconocido");  
13     }  
14 }  
15  
16
```

```
2 public class Main {
3     public static void main(String[] args) {
4         Vehiculo sedan = VehiculoFactory.crearVehiculo("sedan");
5         Vehiculo suv = VehiculoFactory.crearVehiculo("suv");
6         Vehiculo camion = VehiculoFactory.crearVehiculo("camion");
7
8         sedan.conducir();
9         suv.conducir();
10        camion.conducir();
11
12        System.out.println("Capacidad de carga del " + sedan.getDescripcion() + ": " + sedan.getCapacidadCarga() + " kg");
13        System.out.println("Capacidad de carga del " + suv.getDescripcion() + ": " + suv.getCapacidadCarga() + " kg");
14        System.out.println("Capacidad de carga del " + camion.getDescripcion() + ": " + camion.getCapacidadCarga() + " kg");
15    }
16
17 }
```



Ventajas de usar el patrón Factory en este caso

- Desacoplamiento: El cliente (en este caso, la clase Main) no necesita conocer las clases concretas (Sedan, SUV, Camion) para crear una instancia de Vehiculo. Solo necesita saber el tipo de vehículo en forma de String y utilizar la VehiculoFactory para obtener la instancia correspondiente.



PABLO DEL ÁLAMO



- **Facilidad de mantenimiento y escalabilidad:**
Si se necesita añadir un nuevo tipo de vehículo, como Moto o Bicicleta, solo es necesario crear una nueva clase que implemente la interfaz Vehiculo y actualizar la fábrica VehiculoFactory para reconocer el nuevo tipo. No hay necesidad de modificar el código del cliente, ya que seguirá llamando al método crearVehiculo.
- **Centralización:** Al usar una fábrica, la lógica de creación de objetos se centraliza en una sola clase (VehiculoFactory). Esto facilita la aplicación de reglas adicionales en el momento de la creación (por ejemplo, si quisiéramos añadir validaciones).



PABLO DEL ÁLAMO



- **Flexibilidad frente a cambios en las clases concretas:** Si en el futuro queremos modificar la implementación de SUV (por ejemplo, aumentando la capacidad de carga), solo es necesario cambiar la clase SUV, sin afectar la lógica en Main ni en otras clases que utilicen VehiculoFactory. Esto asegura que el sistema sea más resistente a los cambios internos.



PABLO DEL ÁLAMO



Comparación con una implementación sin Factory Pattern

Si no usáramos el Factory Pattern, el código en Main sería algo como esto:

```
1 Vehiculo sedan = new Sedan();
2 Vehiculo suv = new SUV();
3 Vehiculo camion = new Camion();
```



PABLO DEL ÁLAMO



Este enfoque presenta varios inconvenientes:

- **Falta de flexibilidad:** Cada vez que se añade un nuevo tipo de vehículo, habría que modificar el código del cliente (Main) para incluir la nueva clase concreta.



PABLO DEL ÁLAMO



- Mayor acoplamiento: Main debe conocer las clases concretas (Sedan, SUV, Camion) en lugar de solo interactuar con Vehiculo. Esto implica que cualquier cambio en las clases concretas (nombre de la clase, ubicación) obligaría a modificar Main.
- Dificultad para gestionar dependencias: Si los constructores de Sedan, SUV, o Camion requieren parámetros adicionales en el futuro, tendríamos que actualizar el código en cada lugar donde se instancian estos objetos.



PABLO DEL ÁLAMO



Factory vs Abstract Factory

El Abstract Factory Pattern permite crear familias de objetos relacionados, donde cada fábrica concreta produce un conjunto de objetos que están relacionados o son dependientes entre sí.

Este patrón es útil cuando se necesita flexibilidad para crear diferentes familias de productos de manera independiente, especialmente si estas familias deben mantenerse coherentes.



PABLO DEL ÁLAMO



- Ejemplo: En lugar de una única fábrica **VehiculoFactory**, podríamos tener varias fábricas específicas como **DeportivoFactory** y **TodoTerrenoFactory**. Cada una de estas fábricas abstractas puede crear tipos de vehículos específicos (como Sedan, SUV, Camion) adaptados a las características de cada familia (deportivo o todoterreno).

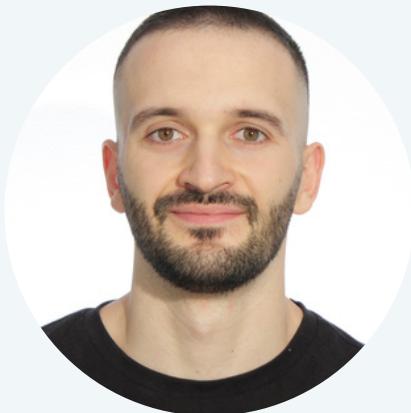
Si veo interés lo explicaré en futuros posts más en detalle 😊.



PABLO DEL ÁLAMO



¿Te ha resultado útil?



- Comparte esta guía con tu equipo o amigos desarrolladores.
- Guárdala para tenerla siempre a mano.
- ¡Dale un like o comenta si tienes preguntas!

