

# Patrones de Diseño de Software

Los patrones de diseño son soluciones reutilizables para problemas de diseño de software que se presentan comúnmente. Proporcionan una forma de organizar y estructurar el código, haciéndolo más fácil de entender, mantener y extender.



**3 Contributors**



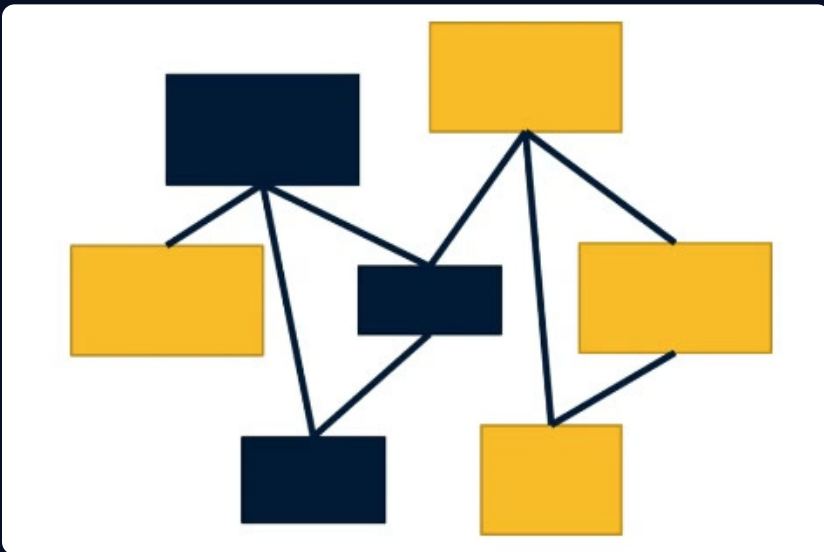
# Acoplamiento

Refiere al grado en que un componente o módulo depende de otros componentes o módulos en un sistema

El acoplamiento estrecho o alto acoplamiento, que el componente o módulo(**clases, métodos, bibliotecas...**) está muy interdependiente de otros componentes o módulos.

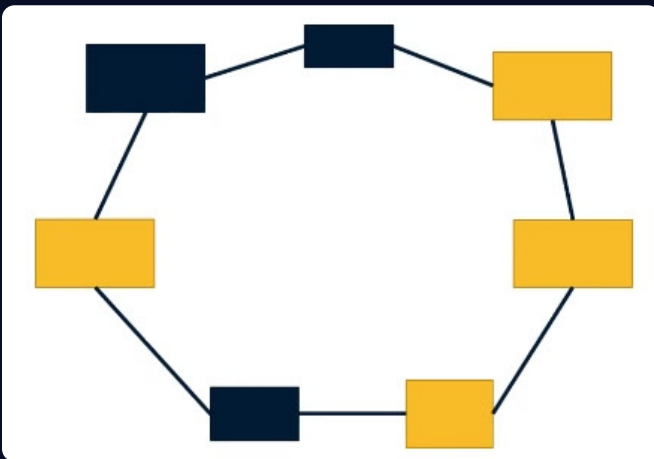
```
public class Cliente {  
    public String nombre;  
  
    public Cliente(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
}
```

```
public class Pedido {  
    private Cliente cliente;  
  
    public Pedido(Cliente cliente)  
    {  
        this.cliente = cliente;  
    }  
  
    public void procesarPedido() {  
  
        System.out.println("Procesando  
pedido para: " +  
cliente.getNombre());  
    }  
}
```



# Acoplamiento

Bajo acoplamiento, que las clases o módulos tienen poca dependencia o ninguna del funcionamiento interno de la otra clase.



```
public interface PaymentProcessor {  
    void procesarPago(double  
    cantidad);  
}
```

```
public class PayPalProcessor  
implements PaymentProcessor {  
    @Override  
    public void procesarPago(double  
    cantidad) {  
  
        System.out.println("Procesando  
        pago con PayPal: $" + cantidad);  
    }  
}
```

```
public class CreditCardProcessor  
implements PaymentProcessor {  
    @Override  
    public void procesarPago(double  
    cantidad) {  
  
        System.out.println("Procesando  
        pago con tarjeta de crédito: $" +  
        cantidad);  
    }  
}
```

# Desacoplamiento

La reducción de la dependencia entre componentes del sistema.

En un sistema DESACOPLADO, los componentes interactúan entre sí de manera que los cambios en uno no requieren cambios en los demás

**fuertemente desacoplado** (bajo acoplamiento) , depender de cuán independientes sean sus componentes.

**débilmente desacoplado** (alto acoplamiento) depender de cuán independientes sean sus componentes.

# Cohesión

Refiere al grado en que las responsabilidades y funciones de un módulo o componente están relacionadas entre sí

**Alta cohesión** significa que un módulo tiene una sola responsabilidad bien definida

**baja cohesión** indica que un módulo realiza una variedad de tareas que no están estrechamente relacionadas entre sí

# Explicación del Patrón Factory Method

El patrón Factory Method es un patrón de diseño creacional que define una interfaz para crear objetos, pero delega la instancia específica a subclases.

## 1 Abstracción de la Creación

Permite crear objetos de diferentes tipos sin especificar explícitamente la clase concreta.

## 2 Flexibilidad

Facilita la introducción de nuevas subclases de objetos sin modificar el código existente.

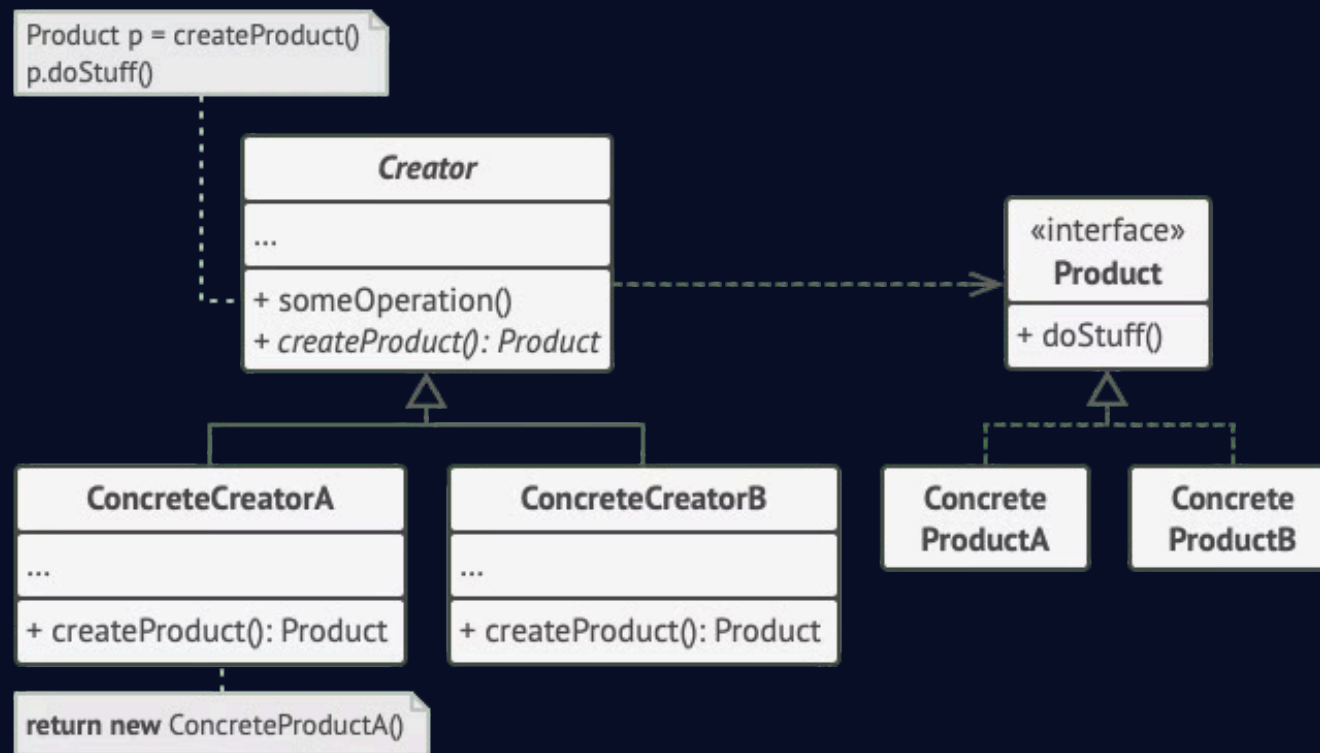
## 3 Reutilización

Promueve la reutilización de código al centralizar la lógica de creación en una sola clase.



# Estructura Básica del patrón en código

La estructura básica del patrón Factory Method en código consta de una clase abstracta o una interfaz que define el método de fábrica, y las subclases concretas que implementan dicho método y crean instancias de objetos específicos. El método de fábrica devuelve un objeto de la interfaz o clase base, lo que permite la creación flexible de objetos sin conocer las implementaciones específicas.






```
public interface Product {  
    void use();  
}
```

Clases Concretas de Producto (ConcreteProductA y ConcreteProductB)

```
public class ConcreteProductA implements Product {  
    @Override  
    public void use() {  
        System.out.println("Usando ConcreteProductA");  
    }  
}  
  
public class ConcreteProductB implements Product {  
    @Override  
    public void use() {  
        System.out.println("Usando ConcreteProductB");  
    }  
}
```



## Clase Abstracta de Creador (Creator):




```
public abstract class Creator {  
    public abstract Product factoryMethod();  
  
    public void someOperation() {  
        Product product = factoryMethod();  
        product.use();  
    }  
}
```

## Clases Concretas de Creador (ConcreteCreatorA y ConcreteCreatorB):

```
public class ConcreteCreatorA extends Creator {  
    @Override  
    public Product factoryMethod() {  
        return new ConcreteProductA();  
    }  
}  
  
public class ConcreteCreatorB extends Creator {  
    @Override  
    public Product factoryMethod() {  
        return new ConcreteProductB();  
    }  
}
```

**Cliente** (Client):



```
public class Client {  
    public static void main(String[] args) {  
  
        Creator creatorA = new ConcreteCreatorA();  
        creatorA.someOperation();  
  
        Creator creatorB = new ConcreteCreatorB();  
        creatorB.someOperation();  
    }  
}
```

# Problema que soluciona el patrón

El patrón Factory Method resuelve el problema de **crear objetos de diferentes tipos sin necesidad de especificar explícitamente la clase concreta del objeto en el código del cliente.**

## Complejidad

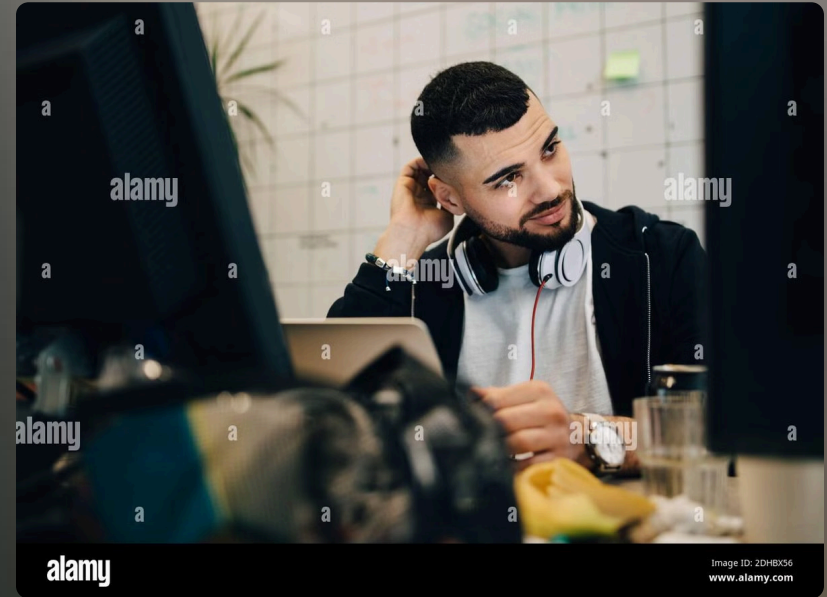
Reduce la complejidad del código al abstraer la lógica de creación.

## Mantenimiento

Facilita el mantenimiento del código al centralizar la lógica de creación en una sola clase.

## Extensibilidad

Permite añadir nuevas clases de objetos sin modificar el código existente.



# Componentes del patrón Factory Method

El patrón Factory Method se compone de los siguientes componentes:

## Producto Abstracto

Define la interfaz o clase abstracta que todos los productos deben implementar.

## Producto Concreto

Implementaciones específicas del producto abstracto.

## Creador Abstracto

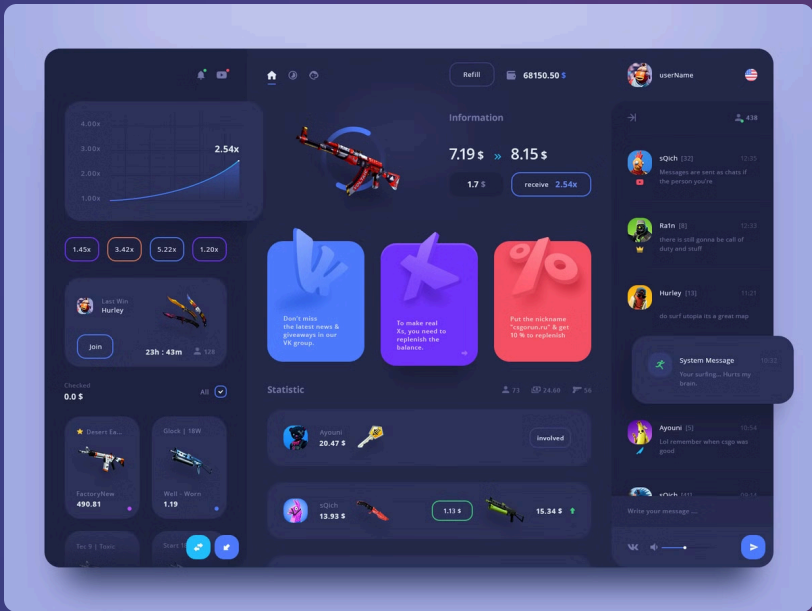
Declara el método fábrica para la creación de objetos.

## Creador Concreto

Sobrescribe el método fábrica para devolver instancias de productos concretos.

# Ejemplo práctico completo de la aplicación del patrón

Un ejemplo práctico podría ser la creación de un sistema de comercio electrónico donde se necesitan diferentes tipos de productos: libros, ropa, electrónica, etc.



```
<code>
// Clase abstracta Product
abstract class Product {
  abstract getName(): string;
}

// Clase concreta Book
class Book extends Product {
  getName(): string {
    return "Libro";
  }
}

// Clase concreta Clothing
class Clothing extends Product {
  getName(): string {
    return "Prenda de vestir";
  }
}

// Clase abstracta Factory
abstract class Factory {
  abstract createProduct(): Product;
}

// Clase concreta BookFactory
class BookFactory extends Factory {
  createProduct(): Product {
    return new Book();
  }
}

// Clase concreta ClothingFactory
class ClothingFactory extends Factory {
  createProduct(): Product {
    return new Clothing();
  }
}

// Ejemplo de uso
const bookFactory = new BookFactory();
const book = bookFactory.createProduct();
console.log(book.getName()); // Salida: "Libro"

const clothingFactory = new ClothingFactory();
const clothing = clothingFactory.createProduct();
console.log(clothing.getName()); // Salida: "Prenda de vestir"
</code>
```

# Ventajas del uso del patrón Factory Method

El uso del patrón Factory Method tiene varias ventajas, como la flexibilidad, la extensibilidad y la reducción de la complejidad.



1

## Flexibilidad

Permite cambiar el tipo de objeto creado sin modificar el código cliente.

2

## Extensibilidad

Facilita la adición de nuevas clases de objetos sin afectar el código existente.

3

## Complejidad

Reduce la complejidad del código al centralizar la lógica de creación en una sola clase.



# Desventajas del patrón Factory Method

Aunque el patrón Factory Method tiene ventajas, también tiene algunas desventajas, como la posible complejidad en casos simples y la necesidad de definir subclases para cada tipo de objeto.

## Complejidad

Puede introducir complejidad en casos simples donde la creación de objetos es sencilla.

## Subclases

Requiere definir una subclase para cada tipo de objeto, lo que puede aumentar el número de clases en el proyecto.

# Conclusión y recomendaciones

El patrón Factory Method es una herramienta poderosa para la creación de objetos de forma flexible y extensible. Es recomendable usarlo cuando se necesita crear objetos de diferentes tipos sin necesidad de especificar la clase concreta en el código.



1

## Análisis

Evaluar si el patrón Factory Method es adecuado para el problema específico.

2

## Diseño

Definir las clases Factory y Product, y las subclases concretas para cada tipo de objeto.

3

## Implementación

Implementar el método factory en cada subclase concreta.

4

## Pruebas

Realizar pruebas exhaustivas para garantizar el correcto funcionamiento del patrón.