



UNIVERSIDAD  
Popular del cesar

# Ingeniería de Sistemas

**ESPECIALIZACION EN INGENIERIA DE SOFTWARE**  
**MODULO PATRONES DE DISEÑO DE SOFTWARE**



## EL DOCENTE



**JAIRO FRANCISCO  
SEOANES LEON**

[jairoseoanes@unicesar.edu.co](mailto:jairoseoanes@unicesar.edu.co)  
(300) 600 06 70



## Educación formal

- ✓ **Ingeniero de sistemas**, Universidad Popular del Cesar sede Valledupar, Feb 2002 – Jun 2009.
- ✓ **MsC en Ingeniería de Sistemas y Computación**, Universidad Nacional de Colombia, Bogotá, Feb 2011 – Mar 2015
- ✓ **PhD Ciencia, Tecnología e innovación**, Urbe, Venezuela, Mayo 2024

## Formación complementaria

- ✓ **AWS Academy Graduate** - AWS Academy Cloud Foundations, 2022  
<https://www.credly.com/go/p3Uwht36>
- ✓ **Associate Cloud Engineer Path** - Google Cloud Academy, 2022  
[https://www.cloudskillsboost.google/public\\_profiles/c7e7936c-3e37-4bad-b822-74d40c49d0db](https://www.cloudskillsboost.google/public_profiles/c7e7936c-3e37-4bad-b822-74d40c49d0db)
- ✓ **Fundamentos De Programación Con Énfasis En Cloud Computing** – AWS Academy y Misión Tic 2022
- ✓ **Google Cloud Computing Foundations** – Google Academy, 2022
- ✓ **Aplicación de cloud: retos y oportunidades de mejora para las empresas de software gestionando la computación en la nube** – Fedesoft, 2023
- ✓ **Desarrollo De Aplicaciones Web En Angular, Para El Nivel Frontend** – Universidad EAFIT, 2023
- ✓ **Microsoft Scrum Foundations** – Intelligent Training - MinTic , 2023

## Experiencia profesional

- ✓ **Docente Universitario**, Universidad Popular del Cesar sede Valledupar, marzo del 2013.
- ✓ **Técnico de Sistemas Grado 11**, Rama judicial Seccional Cesar, SRPA Valledupar, Junio del 2009

# MODULO DE PATRONES DE DISEÑO DE SOFTWARE



# MODULO DE PATRONES DE DISEÑO DE SOFTWARE



## Unidad 2. Patrones de diseño creacionales

**2.1** Factory Method

**2.2** Abstract Factory

**2.3** Singleton

**2.4** Builder

**2.5** Prototype

**2.6** Inyección de dependencias

**2.7** Uso de patrones en Spring y otros frameworks



**Factory Method**

Proporciona una interfaz para la creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



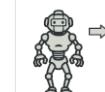
**Abstract Factory**

Permite producir familias de objetos relacionados sin especificar sus clases concretas.



**Builder**

Permite construir objetos complejos paso a paso. Este patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.



**Prototype**

Permite copiar objetos existentes sin que el código dependa de sus clases.



**Singleton**

Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.



## Patrones creacionales - Generales

Los **patrones de diseño creacionales** se enfocan en los mecanismos de creación de objetos.

Su propósito principal es abstraer el proceso de instantiación, proporcionando flexibilidad en la decisión de **qué objetos crear, cómo crearlos y cuándo crearlos**.

Clase objeto = **new Clase();**



Razones para que  
los objetos se  
creen de forma  
controlada



- ✓ Necesidad de existencia de una sola instancia de un tipo de clase
- ✓ No se sabe qué tipo de objeto requiero utilizar hasta en tiempo de ejecución
- ✓ La dificultad es todavía mayor cuando hay que construir objetos compuestos cuyos componentes pueden instanciarse mediante clases diferentes.



# Patrones creacionales – Principios fundamentales

Resuelven el problema de cómo crear objetos de manera que se logre una **arquitectura flexible, reutilizable y mantenible**

## Facilitan el cumplimiento de principios SOLID

### SRP

Al delegar la creación de objetos a una clase específica.



### OCP

Es fácil agregar nuevas formas de crear objetos sin modificar las clases existentes.

### DIP

Facilitan la inyección de dependencias sin acoplarse a implementaciones concretas.

### Encapsulación de la creación:

Separan la lógica de creación del uso de objetos. Promueve baja dependencia entre las clases, lo cual facilita la extensión y el mantenimiento del sistema.

### Flexibilidad compositiva:

Permiten cambiar las clases concretas sin modificar el código cliente

### Reutilización de código:

Evitan la duplicación del código de instancia en distintas partes del sistema, promoviendo así la **reutilización de componentes** y la **reducción de errores**

### Desacoplamiento:

Reducen las dependencias entre componentes

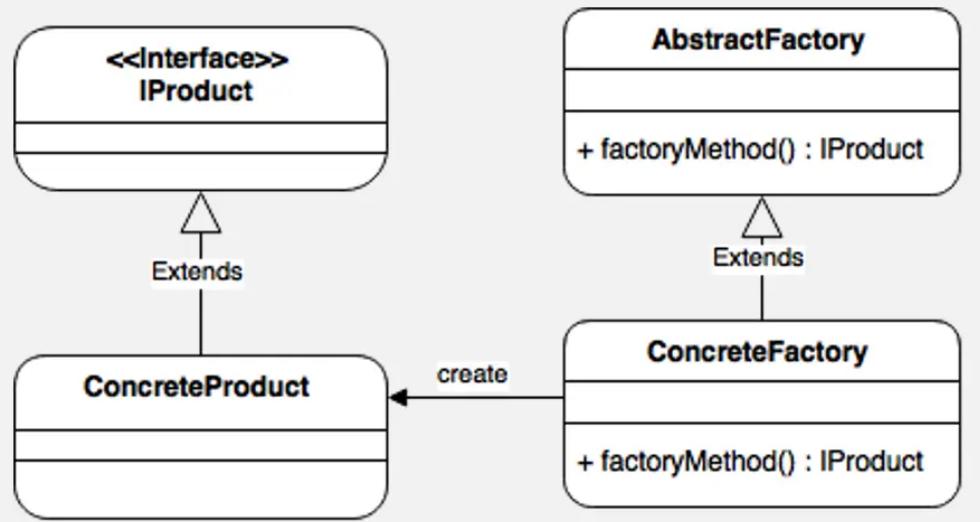
### Interoperabilidad con frameworks:

Muchos frameworks modernos como Spring, Hibernate o Angular internamente usan y facilitan el uso de patrones creacionales.

# Patrones creacionales – Factory Method

El patrón **Factory Method** define una interfaz para crear objetos, pero permite a las subclases decidir qué clase instanciar. Este patrón **delega la responsabilidad de instantiación a las subclases**, promoviendo el principio de inversión de dependencias.

## Factory Method – Class diagram



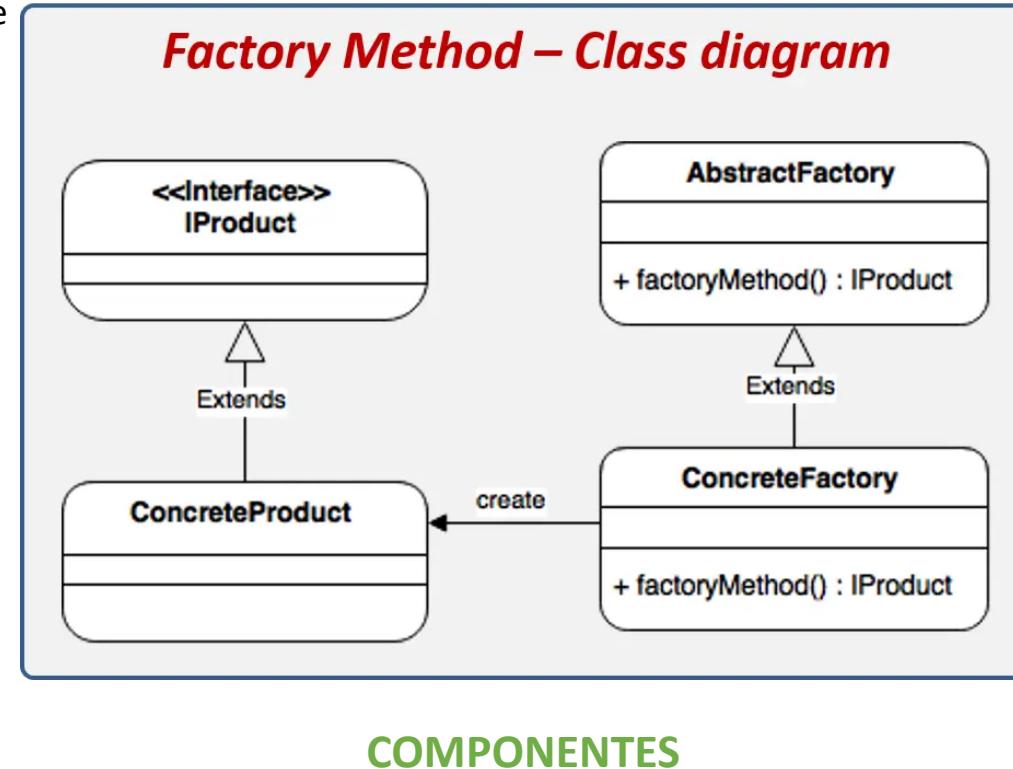
## Problemas que Resuelve

- **Alto acoplamiento** entre clases y sus dependencias concretas (uso excesivo de new).
- Dificultad para cambiar la lógica de creación de objetos sin modificar clases existentes.
- Violación del principio de **abierto/cerrado (OCP)**: el sistema no puede ser extendido sin modificar código fuente.
- Necesidad de **crear familias de objetos relacionados sin conocer sus clases concretas**.

# Patrones creacionales – Factory Method

**IProduct:** forma abstracta del objeto que de desea crear, define la estructura que tendrá el objeto creado

**AbstractFactory:** puede ser opcional, define el comportamiento por default de los **ConcreteFactory**



**ConcreteProduct:** representa una implementación concreta de IProduct

**ConcreteFactory:** representa una fabrica concreta, utilizada para la creación de los ConcreteProduct. Implementa el comportamiento básico del AbstractFactory.



# Patrones creacionales – Factory Method

## Cuándo Utilizar Factory Method

No conoces por adelantado las clases exactas de los objetos que debes crear

Delegar la responsabilidad de creación de objetos a subclases

Permitir la extensión del sistema mediante nuevas implementaciones sin modificar el código base

Seguir el principio de inversión de dependencias, inyectando abstracciones en lugar de implementaciones concretas

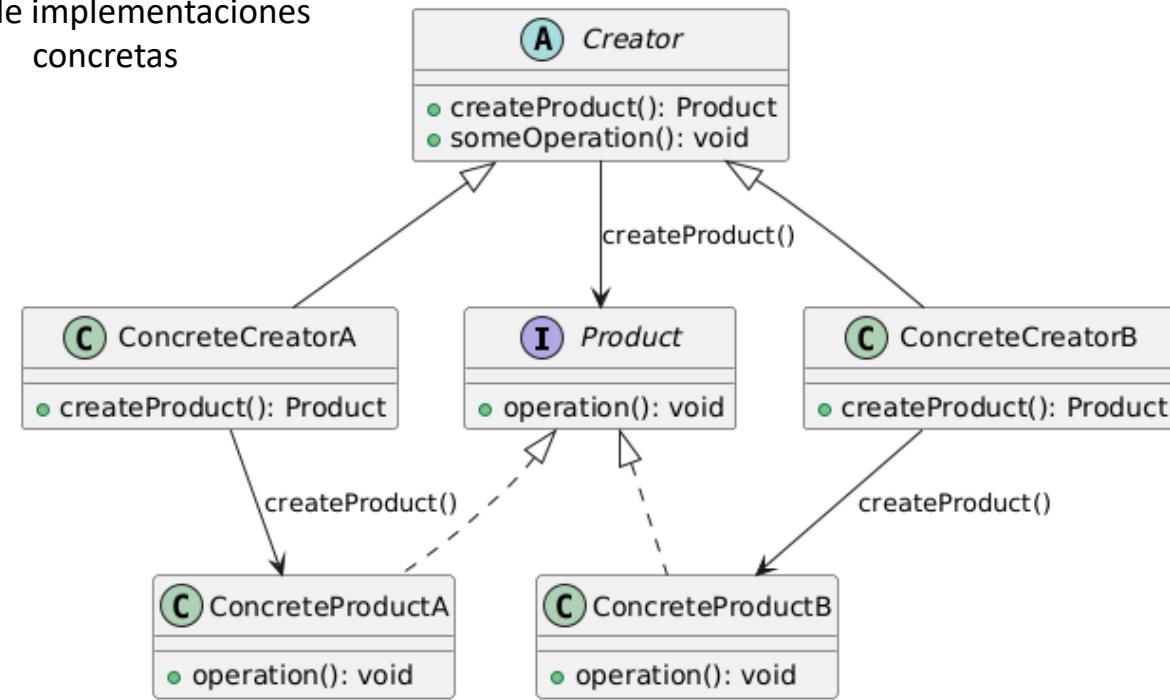
### Casos de Uso Comunes y Realistas

**Frameworks web como Spring Framework:** mecanismo de creación de beans y objetos por medio del BeanFactory.

**Sistemas de notificaciones:** puedes tener una Factory para la creación de diferentes tipos de notificación (NotificadorFactory)

**Procesadores de archivos:** puede usar un factor que cree el tipo de FileParser adecuado según la extensión del archivo (CSVParser, JSONParser, XMLParser).

**Aplicaciones bancarias o financieras:** un factory para crear distintos productos financieros CuentaCorriente, CuentaAhorros.



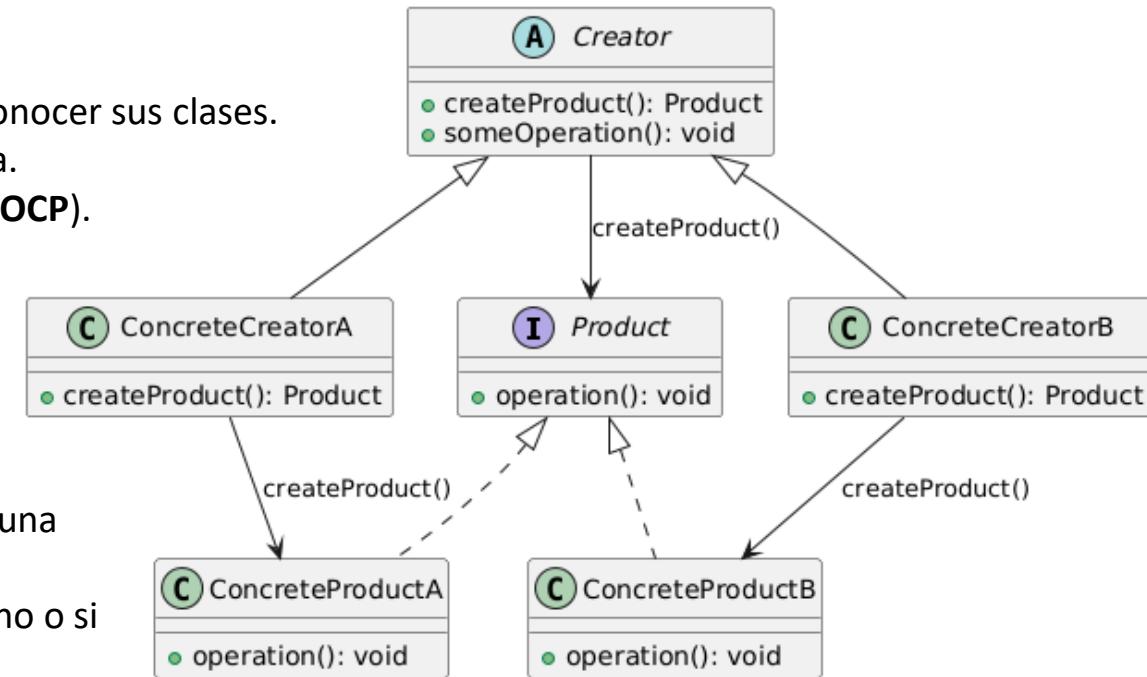
# Patrones creacionales – Factory Method

## Ventajas

- Desacopla la creación de objetos del código que los utiliza.
- Facilita el uso de **polimorfismo**, permitiendo instanciar objetos concretos sin conocer sus clases.
- Cumple con el principio SRP: cada clase tiene una responsabilidad bien definida.
- Favorece la extensión del sistema sin necesidad de modificar código existente (**OCP**).

## Desventajas

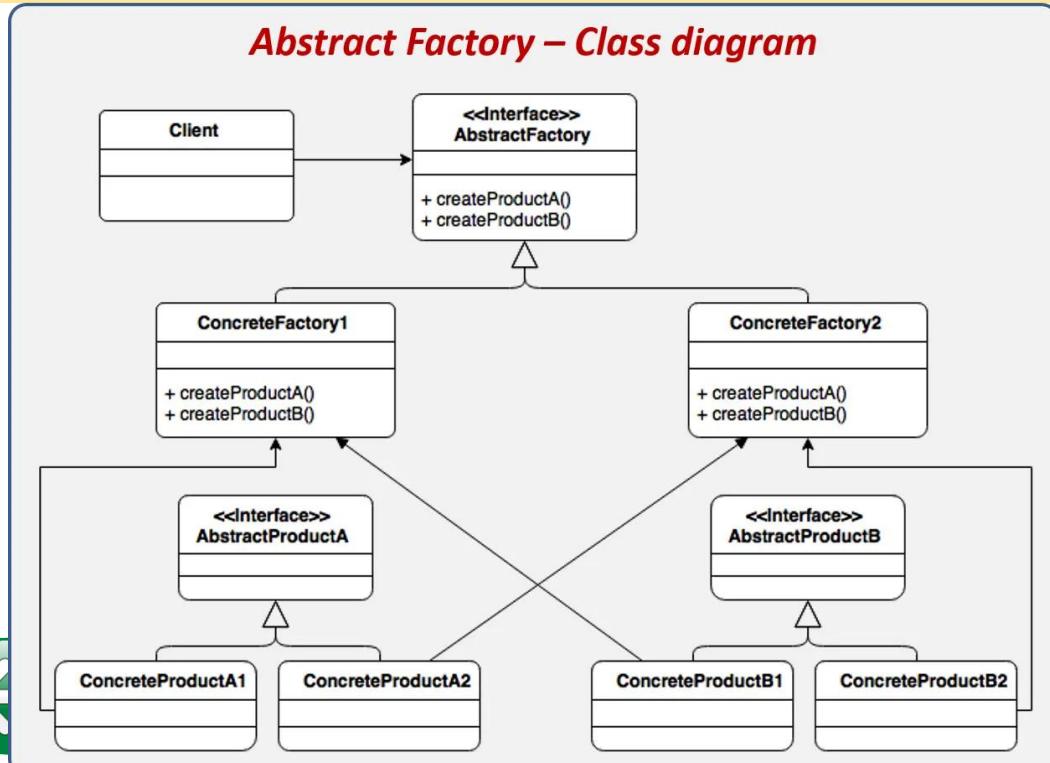
- Mayor número de **clases** en el sistema, ya que cada tipo de producto requiere una subclase creadora.
- Puede introducir **complejidad innecesaria** si la lógica de creación no varía mucho o si solo hay un tipo de producto.
- En algunos casos, puede dificultar el **debugging y trazabilidad** al abstraer demasiado el proceso de instantiación.



# Patrones creacionales – Abstract Factory

El patrón **Abstract Factory** proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas. Este patrón es especialmente útil cuando el sistema debe ser independiente de cómo se crean, componen y representan sus productos.

**Abstract Factory – Class diagram**



## PROBLEMAS QUE RESOLVE

- ✓ Evita la dependencia directa de clases concretas, promoviendo **bajo acoplamiento**.
- ✓ Permite mantener la **consistencia entre múltiples objetos** relacionados.
- ✓ Facilita la **sustitución de familias de productos completas** (por ejemplo, cambiar temas visuales o estilos de UI) sin modificar el código cliente.
- ✓ Resuelve el problema de instanciación cuando hay **familias de objetos interdependientes**.

# Patrones creacionales – Abstract Factory

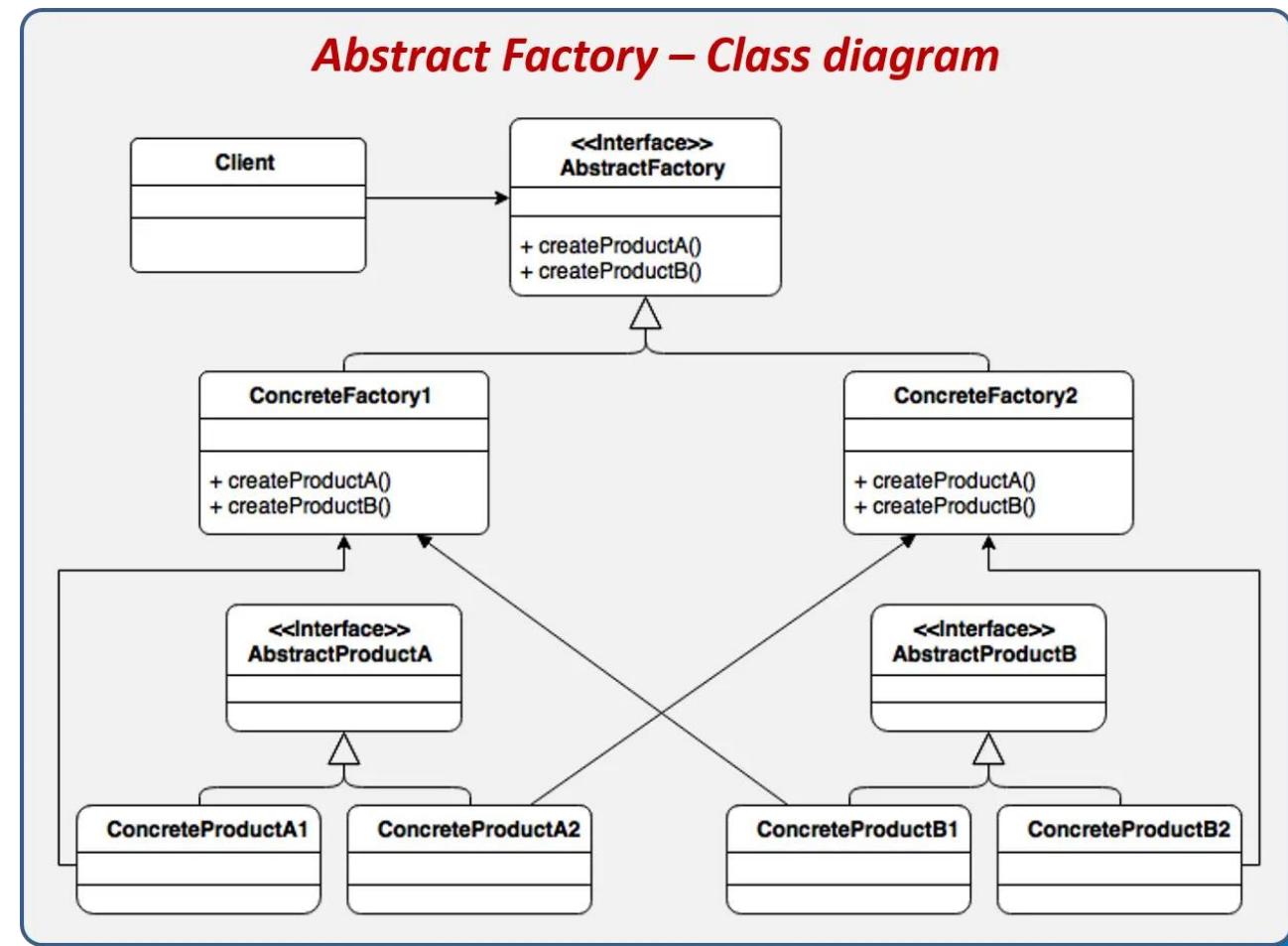
**Client:** Quien dispara la ejecución del patrón.

**AbstractProduct (A, B):** Interfaces que definen la estructura de los objetos para crear familias.

**ConcreteProduct (A, B):** Clases que heredan de **AbstractProduct** con el fin de implementar familias de objetos concretos.

**AbstractFactory:** Define la estructura de las fábricas y deben proporcionar un método para cada clase de la familia

**ConcreteFactory:** Representan las fábricas concretas que servirán para crear las instancias de todas las clases de la familia. En esta clase debe existir un método para crear cada una de las clases de la familia.

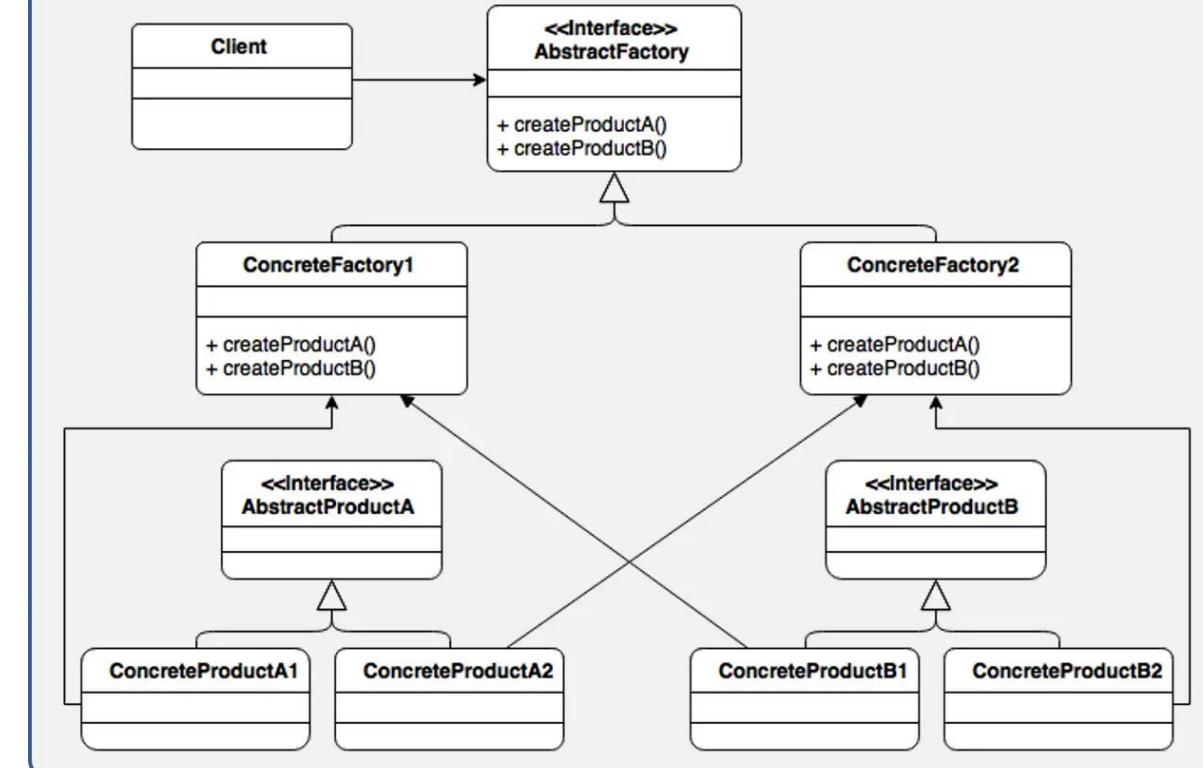


# Patrones creacionales – Abstract Factory

## Cuando Utilizar Abstract Factory

- El sistema debe ser **independiente de cómo se crean, componen y representan sus productos**.
- Se necesita trabajar con **familias de productos relacionados** que deben ser utilizados juntos (por ejemplo, GUI multiplataforma, temas de una aplicación, servicios REST vs SOAP).
- Deseas **encapsular la creación lógica de múltiples objetos**.
- Quieres aplicar el **principio de inversión de dependencias** (DIP), programando contra interfaces en lugar de implementaciones.

## Abstract Factory – Class diagram

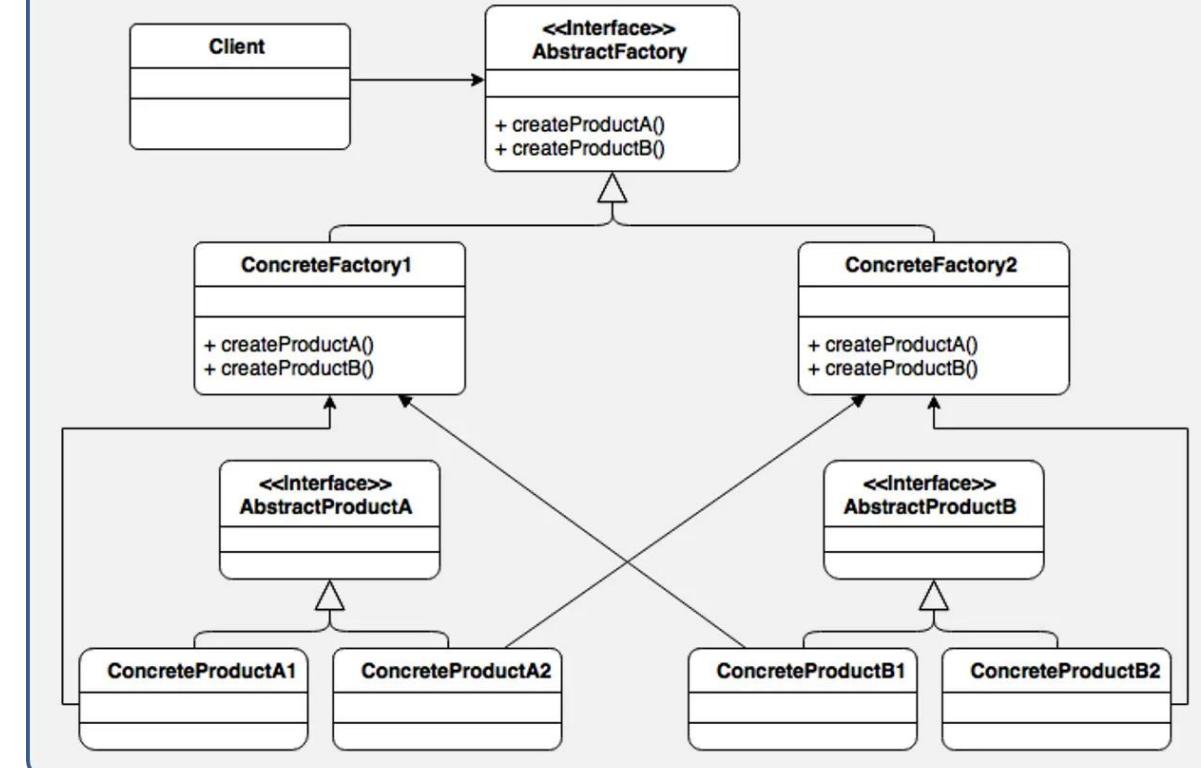


# Patrones creacionales – Abstract Factory

## Cuando Utilizar Abstract Factory

- El sistema debe ser **independiente de cómo se crean, componen y representan sus productos**.
- Se necesita trabajar con **familias de productos relacionados** que deben ser utilizados juntos (por ejemplo, GUI multiplataforma, temas de una aplicación, servicios REST vs SOAP).
- Deseas **encapsular la creación lógica de múltiples objetos**.
- Quieres aplicar el **principio de inversión de dependencias (DIP)**, programando contra interfaces en lugar de implementaciones.

## Abstract Factory – Class diagram



# Patrones creacionales – Abstract Factory

## Casos de uso comunes

### Interfaces gráficas de usuario (GUI)

Swing, JavaFX o Android permiten cambiar el aspecto completo (Dark/Light Theme) usando diferentes fábricas para crear botones, campos de texto y ventanas.

### Aplicaciones multiplataforma (Windows, Linux, macOS)

Cada sistema puede tener una implementación distinta de ventanas, botones o controles.

### Sistemas de persistencia

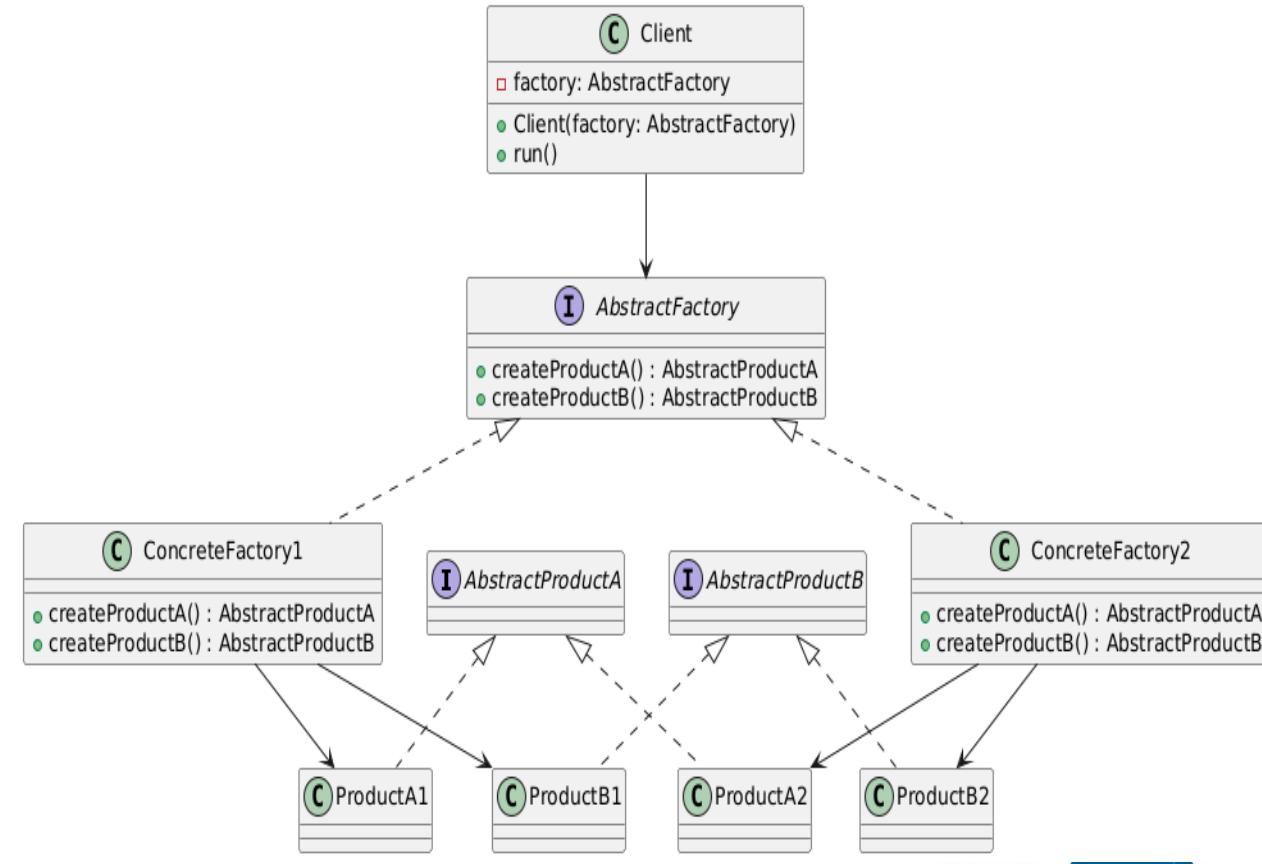
Crear DAOs o repositorios distintos para diferentes motores (MySQL, PostgreSQL, MongoDB), sin cambiar el código de negocio.

### Aplicaciones e-commerce con canales de pago múltiples

Crear fábricas para PayPal, Stripe o tarjetas, que producen objetos relacionados como autorizadores, capturadores y notificadores.

### Frameworks que permiten plug-ins intercambiables

Donde se requiere que un conjunto completo de componentes sea reemplazado dinámicamente por una variante distinta pero coherente.



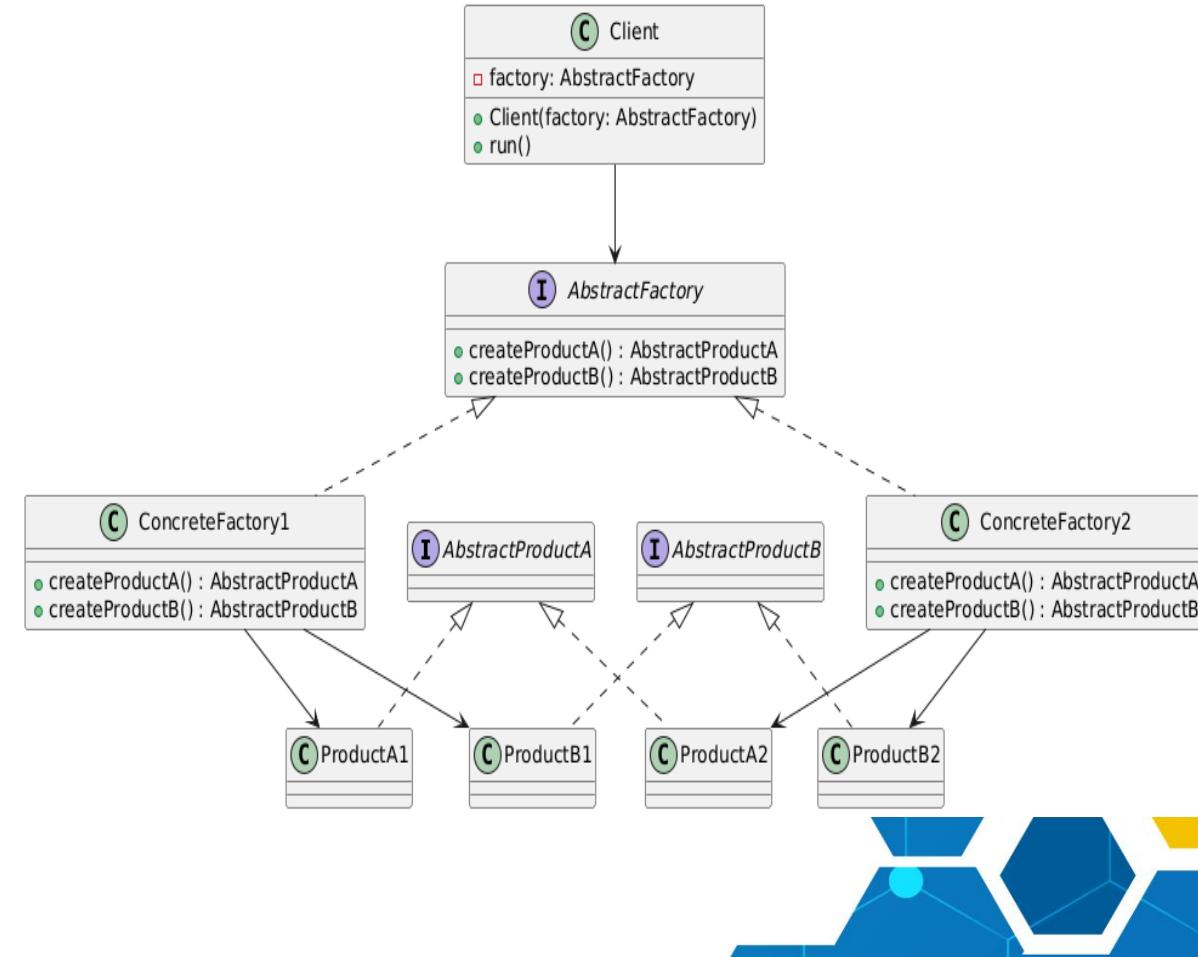
# Patrones creacionales – Abstract Factory

## Ventajas:

- ✓ Bajo acoplamiento entre código cliente y clases concretas.
- ✓ Consistencia entre productos relacionados, evitando errores de combinación.
- ✓ Facilita la sustitución de familias de productos.
- ✓ Cumple con principios SOLID: especialmente OCP (cerrado para modificación) y DIP (inversión de dependencias).

## Desventajas:

- ✗ Puede incrementar la complejidad del diseño, especialmente si se abusa cuando no se requieren familias de productos.
- ✗ Añade múltiples interfaces y clases que pueden parecer innecesarias en sistemas pequeños.
- ✗ Difícil extender con nuevas clases de producto, ya que hay que modificar todas las fábricas.

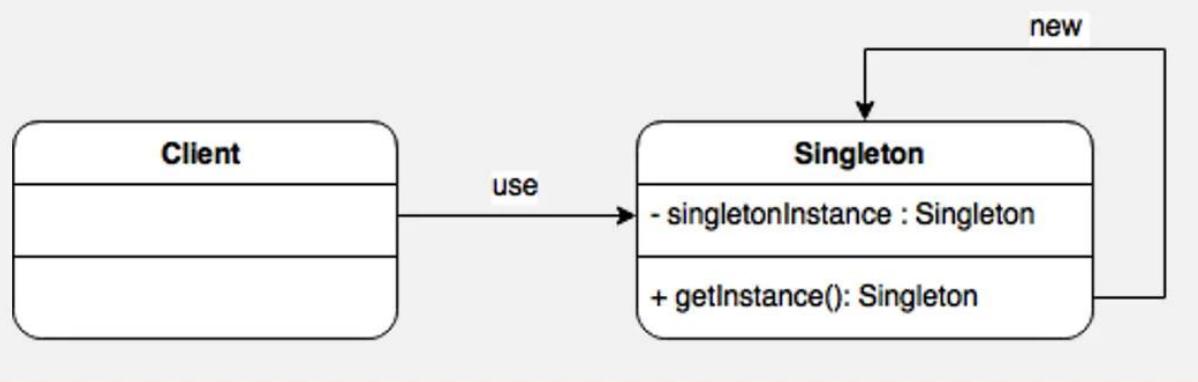


# Patrones creacionales – Singleton

El patrón **Singleton** garantiza que una clase tenga una única instancia y proporciona un punto global de acceso a ella.

En arquitecturas modernas presenta desafíos significativos relacionados **con testing, concurrencia y principios SOLID**.

## *Singleton pattern – Class diagram*



## PROBLEMAS QUE RESUELVE

### Gestión de recursos compartidos:

Evita la creación redundante de objetos costosos (como conexiones a bases de datos, gestores de configuración, o caches), garantizando una instancia única.

### Consistencia del estado global:

Asegura que distintos componentes del sistema accedan al mismo estado compartido sin duplicación.

### Control de acceso centralizado:

Proporciona un único punto de coordinación y control para operaciones distribuidas o concurrencia.

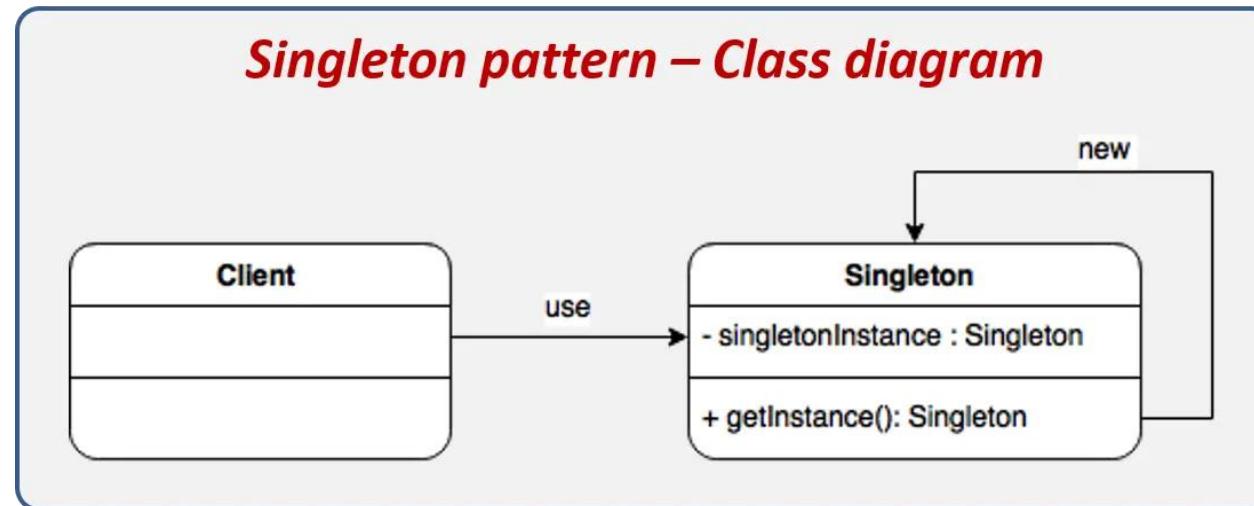
## Patrones creacionales – Singleton

### Client:

Componente en el cual se desea obtener la instancia de la clase Singleton

### Singleton:

Clase que implementa el patrón singleton. De ella solo se permitirá la creación de una única instancia.



Este patrón **encapsula** la lógica para crear y mantener una sola instancia, gestionando así el ciclo de vida del objeto a lo largo de toda la aplicación.



# Patrones creacionales – Singleton

## Casos de uso comunes (ajustados a la realidad)

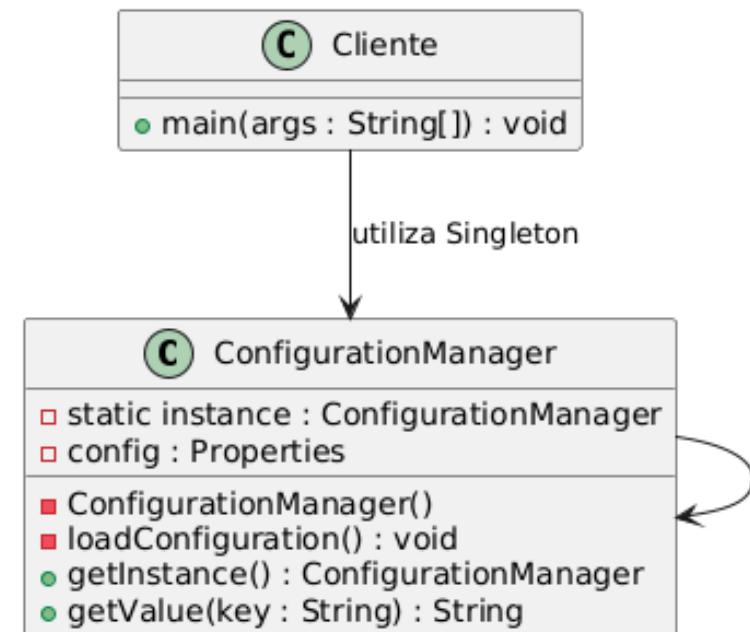
**Logging:** Un sistema de logs debe escribir en una única fuente (archivo, consola, etc.). Usar un Singleton evita múltiples instancias y garantiza la consistencia del log.

**Gestor de configuración (config manager):** Las aplicaciones cargan parámetros desde un archivo .properties o desde variables de entorno. El Singleton asegura que la configuración se lee una vez y se reutiliza.

**Pool de conexiones:** En sistemas que acceden a una base de datos, un Singleton gestiona el pool de conexiones, evitando múltiples creaciones costosas.

**Cache global:** Servicios distribuidos como microservicios pueden usar un Singleton para mantener una cache local de respuestas recientes o tokens de autenticación.

**Gestor de contexto en frameworks:** Frameworks como Spring mantienen un ApplicationContext que es instanciado una vez, funcionando como un Singleton centralizado.



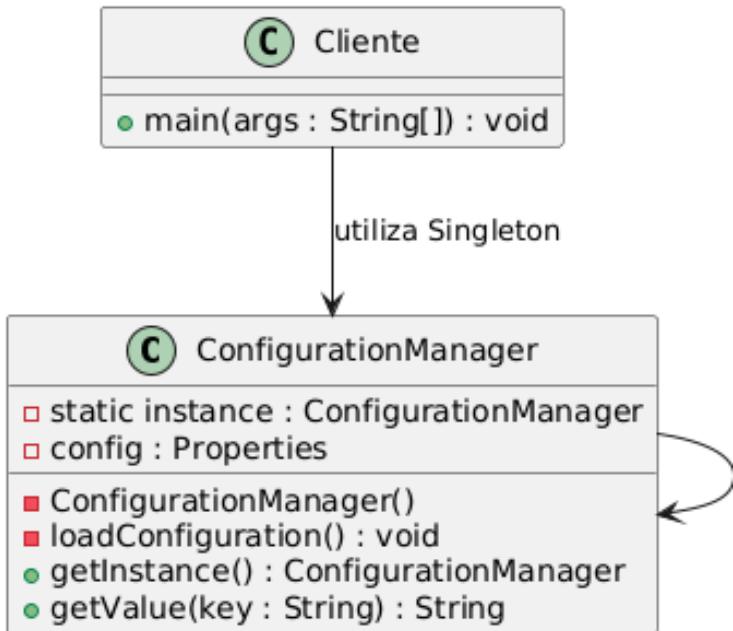
# Patrones creacionales – Singleton

## Ventajas

- Control de acceso centralizado:** Permite controlar desde un único punto las operaciones críticas o compartidas.
- Reducción de consumo de memoria:** Solo se crea una instancia, evitando la duplicación de objetos pesados.
- Mejor integración con arquitecturas globales:** Útil para servicios compartidos como autenticación, logs o métricas.
- Fácil acceso global:** Cualquier clase puede acceder a la instancia, facilitando la integración.

## Desventajas:

- Oculta dependencias:** Su acceso global puede romper el principio de **DIP** y dificultar la prueba unitaria.
- Problemas con pruebas (testing):** Introduce estado global, lo que puede generar efectos secundarios en pruebas concurrentes o secuenciales.
- Difícil de escalar en entornos distribuidos:** En sistemas multi-node, cada nodo podría tener su propia instancia local, rompiendo la unicidad.
- Dificulta la extensibilidad:** Al ser final o tener constructores privados, el Singleton es difícil de extender o reutilizar por herencia.
- Violación del principio de responsabilidad única (SRP):** La clase Singleton maneja su lógica y además controla su instancia, violando el SRP de SOLID.



# Patrones creacionales – Singleton

```
public class Logger {

    private static volatile Logger instance;

    private Logger() {
        // Simular conexión o archivo
    }

    public static Logger getInstance() {
        if (instance == null) {
            synchronized (Logger.class) {
                if (instance == null) {
                    instance = new Logger();
                }
            }
        }
        return instance;
    }

    public void log(String message) {
        System.out.println("[LOG]: " + message);
    }
}
```

Inicialización perezosa (lazy) y segura para hilos

```
public class Logger {

    // Instancia única creada en tiempo de carga de clase
    private static final Logger instance = new Logger();

    // Constructor privado para evitar instanciación externa
    private Logger() {
        System.out.println("Instancia de Logger creada");
    }

    // Método público para obtener la instancia
    public static Logger getInstance() {
        return instance;
    }

    // Método de utilidad
    public void log(String message) {
        System.out.println("[LOG]: " + message);
    }
}
```

Inicialización temprana

## Buenas prácticas para uso avanzado

```
public class Logger {
    private Logger() {}

    private static class Holder {
        private static final Logger INSTANCE = new Logger();
    }

    public static Logger getInstance() {
        return Holder.INSTANCE;
    }
}
```

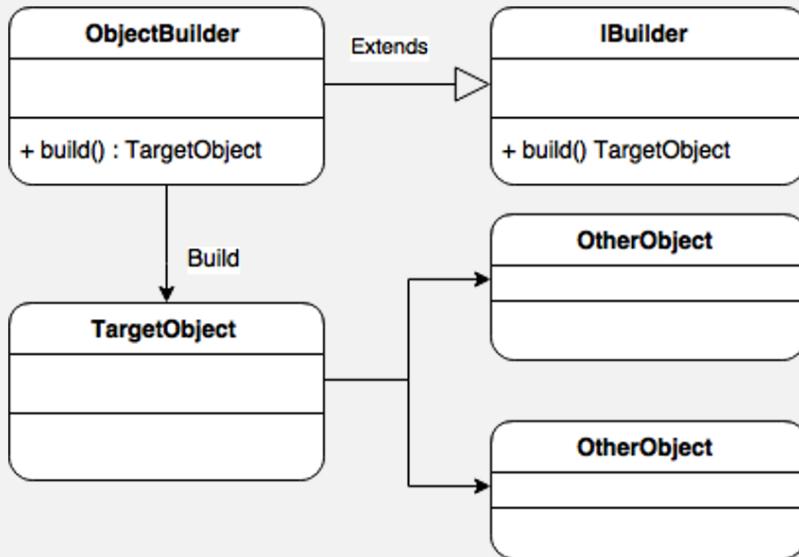
Initialization-on-demand holder idiom, es thread-safe

Inicialización de clases en la JVM

# Patrones creacionales – Builder

El patrón **Builder** separa la construcción de objetos complejos de su representación, permitiendo crear diferentes representaciones usando el mismo proceso de construcción

## *Builder pattern – Class diagram*



## PROBLEMAS QUE RESUELVE

```

public class Usuario {
    private String nombre;
    private String apellido;
    private String email;
    private String telefono;
    private String direccion;
    private int edad;

    public Usuario(String nombre) {
        this(nombre, null, null, null, null, 0);
    }

    public Usuario(String nombre, String apellido) {
        this(nombre, apellido, null, null, null, 0);
    }

    public Usuario(String nombre, String apellido, String email) {
        this(nombre, apellido, email, null, null, 0);
    }

    // ... y así sucesivamente
}
  
```

Constructores con demasiados parámetros (problema del "**telescoping constructor**").

Configuraciones opcionales para un objeto.

Objetos **inmutables** con muchas propiedades.

**Variantes de representación** con diferentes configuraciones, pero mismo proceso de construcción.

Encapsular la lógica de construcción de un objeto complejo, permitiendo una flexibilidad total en su configuración, sin obligar a tener constructores con muchos parámetros o múltiples sobrecargas.

## Patrones creacionales – Builder

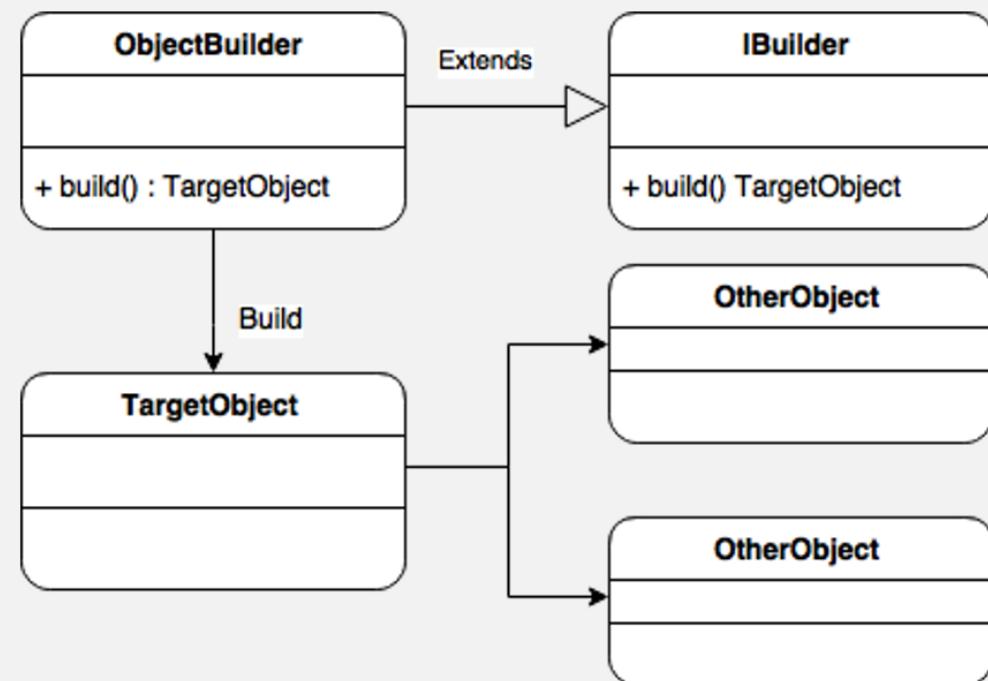
**IBuilder:** Componente opcional, especifica una interface común que tendrán todos los Builder, puede ser una interface que defina únicamente el método build.

**ObjectBuilder:** Implementación de **IBuilder**, es la clase que utilizaremos para crear los **TarjetObjet**. Como regla general todos los métodos de esta clase retornan a si mismo con la finalidad de agilizar la creación, esta clase por lo general es creada como una clase interna del **TargetObject**.

**TarjetObjet:** Representa el objeto que deseamos crear mediante el **ObjectBuilder**, ésta puede ser una clase simple o puede ser una clase muy compleja que tenga dentro más objetos.

**OtherObjs:** Representa los posibles objetos que deberán ser creados cuando el **TarjetObject** sea construido por el **ObjectBuilder**. Son objetos que hacen parte del **TarjetObject**

### *Builder pattern – Class diagram*



# Patrones creacionales – Builder

## CASOS DE USO REALES

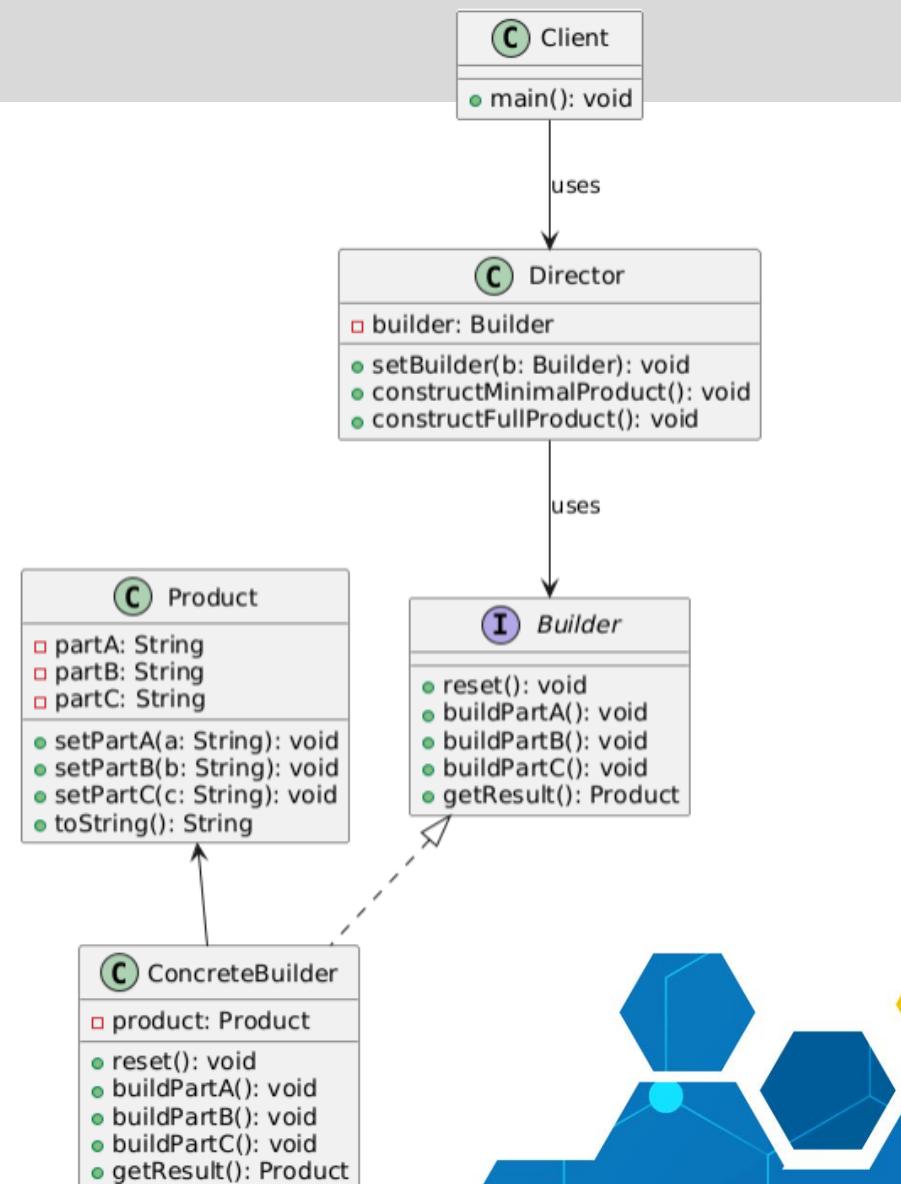
**Construcción de configuraciones complejas:** configuración de clientes HTTP, conexiones de base de datos, clientes de APIs.

**Creación de UIs:** construir componentes con múltiples atributos opcionales (color, tamaño, fuente, etc.).

**Generación de documentos o reportes:** donde el contenido es el mismo, pero las representaciones cambian (HTML, PDF, JSON).

**ORMs:** objetos entidad con múltiples atributos opcionales.

**APIs de construcción de solicitudes:** por ejemplo, Request.Builder en OkHttp, StringBuilder, o DocumentBuilder en Java XML.



# Patrones creacionales – Builder

## ✓ Ventajas

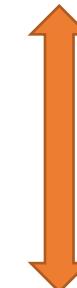
- ✓ Mejora la legibilidad y mantenimiento: Al evitar constructores con muchos parámetros.
- ✓ Soporta inmutabilidad: Ya que el objeto final puede no exponer setters.
- ✓ Permite validaciones durante la construcción: Puedes aplicar lógica antes de finalizar el objeto.
- ✓ Fomenta reutilización y desacoplamiento: Separa el cómo del qué.
- ✓ Variantes de producto sin alterar el cliente: Se puede cambiar el builder sin cambiar el cliente.

## ✗ Desventajas

- ! Aumento en el número de clases: Requiere crear muchas clases para cada variante.
- ! Complejidad innecesaria para objetos simples: Si el objeto no es complejo, puede sobre ingenierizarse.
- ! Posible sobrecarga si hay muchas variantes: Se pueden multiplicar las combinaciones.

```
public Libro(String titulo, String autor, int anioPublicacion, String editorial, String isbn,
  this.titulo = titulo;
  this.autor = autor;
  this.anioPublicacion = anioPublicacion;
  this.editorial = editorial;
  this.isbn = isbn;
  this.paginas = paginas;
}
```

```
Libro libro = new Libro("Design Patterns", "Gamma", 1994, "Addison-Wesley", "0-201-63361-2", 395)
```



- Lectura más clara.
- Solo se construyen los campos que el cliente necesita.
- Inmutabilidad del objeto final (Objeto no tiene setters).
- Evitamos errores por orden incorrecto de parámetros.

```
Libro libro = new Libro.Builder("Effective Java", "Joshua Bloch")
  .anioPublicacion(2008)
  .editorial("Addison-Wesley")
  .isbn("978-0321356680")
  .paginas(416)
  .build();
```

# Patrones creacionales – Builder

## Builder con clase interna

```
public class Person {
    private final String name;
    private final int age;

    private Person(Builder builder) {
        this.name = builder.name;
        this.age = builder.age;
    }

    public static class Builder {
        private String name;
        private int age;

        public Builder name(String name) { this.name = name; return this; }
        public Builder age(int age) { this.age = age; return this; }
        public Person build() { return new Person(this); }

        public String toString() { return name + " - " + age; }
    }
}
```

```
Person person = new Person.Builder()
    .name("Ana")
    .age(28)
    .address("Calle 10 #20-30")
    .phone("3001234567")
    .build();
```

## Builder separado

```
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() { return name + " - " + age; }
}

class PersonBuilder {
    private String name;
    private int age;

    public PersonBuilder name(String name) { this.name = name; return this; }
    public PersonBuilder age(int age) { this.age = age; return this; }
    public Person build() { return new Person(name, age); }
}
```

```
Person p = new PersonBuilder()
    .name("Luis")
    .age(40)
    .build();
```

```
class Person {
    String name;
    int age;

    public String toString() { return name + " - " + age; }
}
```

```
interface Builder {
    void setName();
    void setAge();
    Person build();
}
```

```
class AdultBuilder implements Builder {
    private final Person person = new Person();

    public void setName() { person.name = "Carlos"; }
    public void setAge() { person.age = 45; }
    public Person build() { return person; }
}
```

```
class Director {
    private Builder builder;

    public void setBuilder(Builder builder) { this.builder = builder; }

    public Person construct() {
        builder.setName();
        builder.setAge();
        return builder.build();
    }
}
```

```
Director director = new Director();
director.setBuilder(new AdultBuilder());
Person p = director.construct();
```

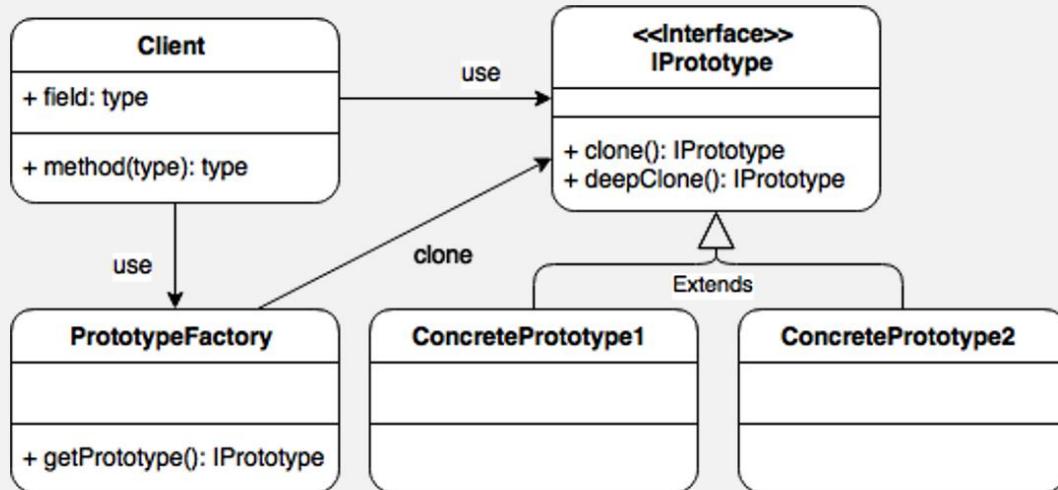
## Builder con director

# Patrones creacionales – Prototype

El **Patrón Prototipo** tiene como objetivo crear nuevos objetos copiando (clonando) instancias existentes, en lugar de crearlos desde cero (con **new**).

Estas instancias llamadas prototipos disponen de la capacidad de clonación.

## *Prototype pattern – Class diagram*



## PROBLEMAS QUE RESUELVE

**Evitar instanciaciones costosas:** Cuando crear un objeto desde cero es costoso en tiempo o recursos.

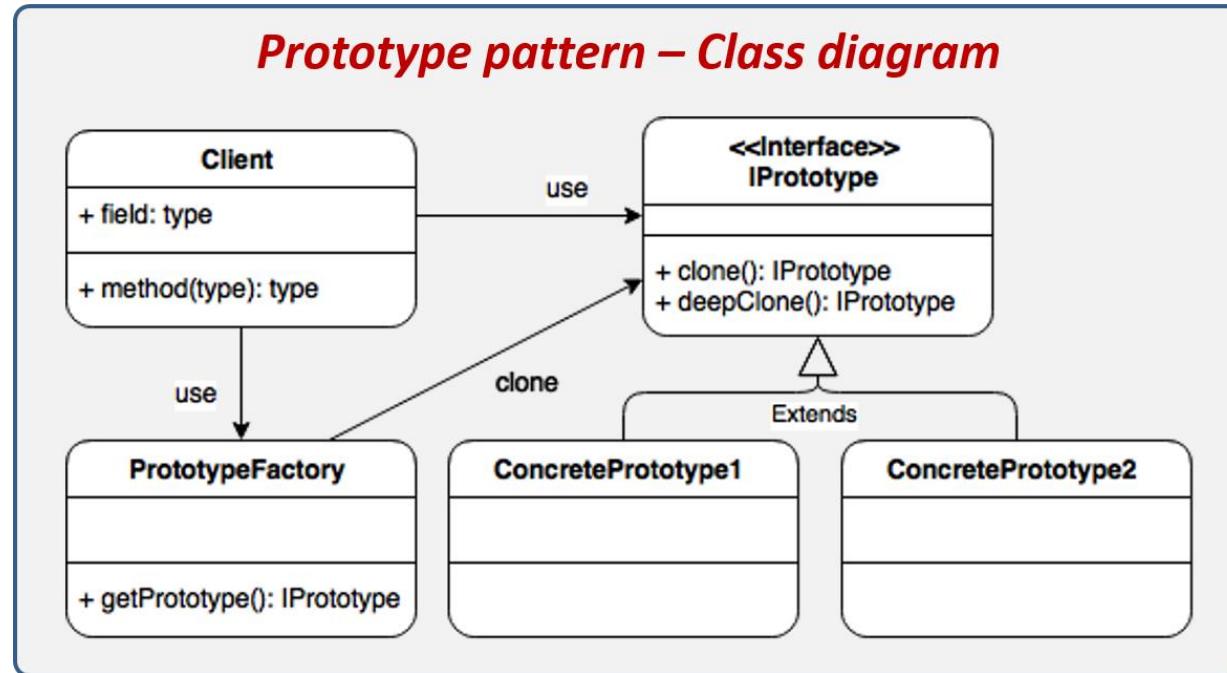
**Evitar dependencias con clases concretas:** Permite al cliente trabajar con interfaces en lugar de usar **new**.

**Crear familias de objetos** con configuraciones similares.

**Clonar objetos** con estructuras complejas (gráficos, árboles, configuraciones, etc.).

# Patrones creacionales – Prototype

**Client:** Componente que interactúa con los prototipos.



**PrototypeFactory:** Mantiene el cache de los prototipos existentes, y permite para crear clonaciones de los mismos.

**ConcretePrototype:** Implementaciones concretas de IPrototype los cuales podrán ser clonados.

**IPrototype:** Interface que define el comportamiento del prototipo, debe contar por lo menos con alguno de los dos tipos de clonación, superficial (clone) o profunda(deepClone)

# Patrones creacionales – Prototype

```
public class Pokemon {  
    private String nombre;  
    private String tipo;  
    private int nivel;  
    private int puntosDeVida;  
    private String ataqueEspecial;  
  
    public Pokemon(String nombre, String tipo, int nivel, int puntosDeVida, String ataqueEspecial){  
        this.nombre = nombre;  
        this.tipo = tipo;  
        this.nivel = nivel;  
        this.puntosDeVida = puntosDeVida;  
        this.ataqueEspecial = ataqueEspecial;  
    }  
  
    public void mostrarInfo(){  
        System.out.println("Pokemon: " + nombre + ", Tipo: " + tipo +  
            ", Nivel: " + nivel + ", Vida: " + puntosDeVida + ", Ataque: " + ataqueEspecial);  
    }  
}
```

## Problemas detectados:

- 🚫 Repetición de lógica para crear múltiples Pokémons similares.
- 🚫 Si cambia el formato del Pikachu inicial (ej. vida = 120), hay que cambiarlo en todos los lugares manualmente.
- 🚫 Escalabilidad limitada si agregamos más tipos de Pokémon con variaciones mínimas.

```
public class JuegoPokemon {  
    public static void main(String[] args) {  
        // Repetición innecesaria para pokemons similares  
        Pokemon pikachu1 = new Pokemon("Pikachu", "Eléctrico", 5, 100, "Impactrueno");  
        Pokemon pikachu2 = new Pokemon("Pikachu", "Eléctrico", 5, 100, "Impactrueno");  
        Pokemon pikachu3 = new Pokemon("Pikachu", "Eléctrico", 5, 100, "Impactrueno");  
  
        pikachu1.mostrarInfo();  
        pikachu2.mostrarInfo();  
        pikachu3.mostrarInfo();  
    }  
}
```

# Patrones creacionales – Prototype

```

public class Pokemon implements PrototypePokemon {
    private String nombre;
    private String tipo;
    private int nivel;
    private int puntosDeVida;
    private String ataqueEspecial;

    // Clonador
    public Pokemon(Pokemon original) {
        this.nombre = original.nombre;
        this.tipo = original.tipo;
        this.nivel = original.nivel;
        this.puntosDeVida = original.puntosDeVida;
        this.ataqueEspecial = original.ataqueEspecial;
    }

    @Override
    public PrototypePokemon clonar() {
        return new Pokemon(this);
    }

    public void mostrarInfo() {
        System.out.println("Pokemon: " + nombre + ", Tipo: " + tipo +
            ", Nivel: " + nivel + ", Vida: " + puntosDeVida + ", Ataque: " + ataqueEspecial);
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}

```

```

public interface PrototypePokemon {
    PrototypePokemon clonar();
}

import java.util.HashMap;
import java.util.Map;

public class RegistroDePokemons {
    private static Map<String, PrototypePokemon> prototipos = new HashMap<>();

    public static void registrar(String clave, PrototypePokemon pokemon) {
        prototipos.put(clave, pokemon);
    }

    public static PrototypePokemon obtenerClon(String clave) {
        return prototipos.get(clave).clonar();
    }
}

```

```

public class JuegoPokemon {
    public static void main(String[] args) {
        // Se define solo una vez el prototipo de Pikachu
        Pokemon pikachuBase = new Pokemon("Pikachu", "Eléctrico", 5, 100, "Impactrueno");
        RegistroDePokemons.registrar("pikachu", pikachuBase);

        // Se clonian fácilmente
        Pokemon pikachu1 = (Pokemon) RegistroDePokemons.obtenerClon("pikachu");
        Pokemon pikachu2 = (Pokemon) RegistroDePokemons.obtenerClon("pikachu");
        Pokemon pikachu3 = (Pokemon) RegistroDePokemons.obtenerClon("pikachu");

        // Pueden personalizarse después del clon
        pikachu2.setNombre("Pikachu Alfa");
        pikachu3.setNombre("Pikachu Beta");

        pikachu1.mostrarInfo();
        pikachu2.mostrarInfo();
        pikachu3.mostrarInfo();
    }
}

```

- ✓ **Reutilización:** No se repite lógica al crear objetos similares.
- ✓ **Facilidad de mantenimiento:** Cambiar el "modelo base" en un solo lugar afecta a todos los clones futuros.
- ✓ **Bajo costo:** No requiere construcción pesada desde cero para cada instancia.
- ✓ **Flexible:** Puedes clonar y luego personalizar cada clon fácilmente.

# Patrones creacionales – Prototype

## Clonación superficial:

Se crea una copia del objeto principal, pero todos los objetos internos no se clonian. Si no que son compartidos.

```
public class Pokemon implements Cloneable {
    private String nombre;
    private String tipo;
    private List<String> habilidades;

    public Pokemon(String nombre, String tipo, List<String> habilidades) {
        this.nombre = nombre;
        this.tipo = tipo;
        this.habilidades = habilidades;
    }

    @Override
    public Pokemon clone() throws CloneNotSupportedException {
        return (Pokemon) super.clone(); // clonación superficial
    }
}
```



Las referencias internas (como listas, arrays, u otros objetos) siguen apuntando a las mismas instancias que el original.

## Clonación profunda:

Se realiza una copia idéntica del prototipo, incluyendo todos los objetos que este contenga

```
public class Pokemon implements Cloneable {
    private String nombre;
    private String tipo;
    private List<String> habilidades;

    public Pokemon(String nombre, String tipo, List<String> habilidades) {
        this.nombre = nombre;
        this.tipo = tipo;
        this.habilidades = habilidades;
    }

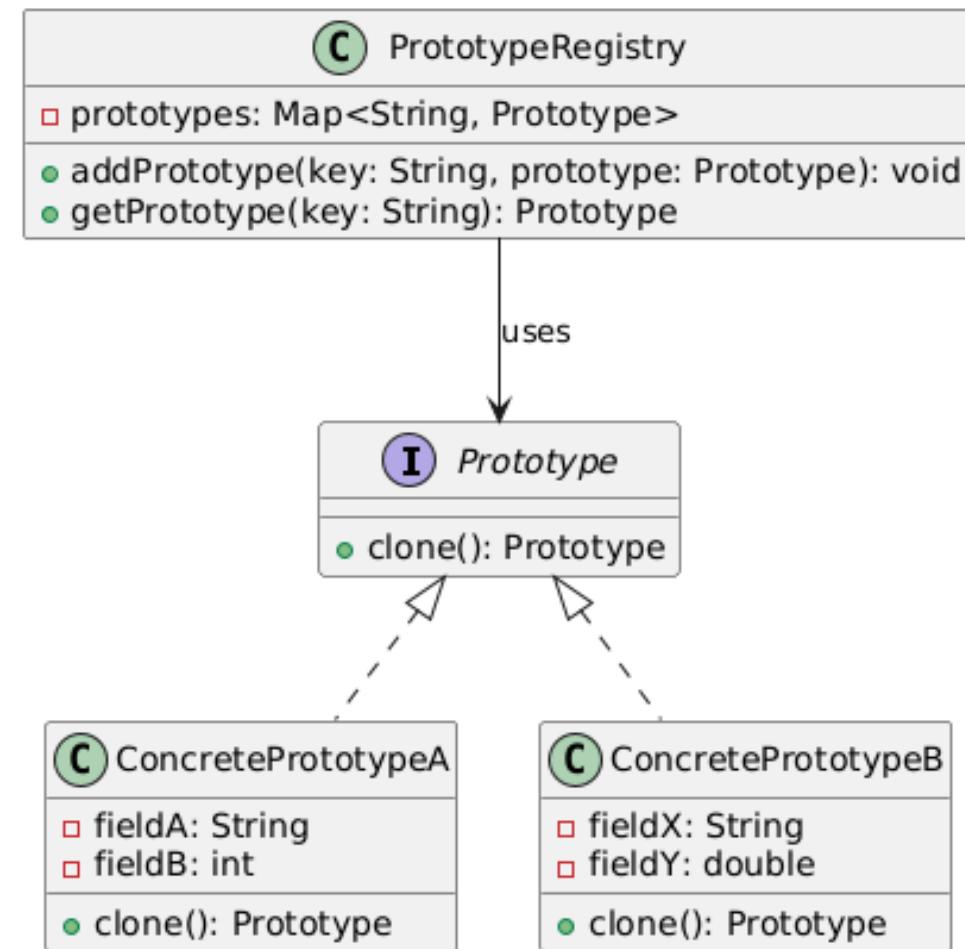
    @Override
    public Pokemon clone() throws CloneNotSupportedException {
        List<String> habilidadesClonadas = new ArrayList<>(this.habilidades); // Se clona La Lista
        return new Pokemon(this.nombre, this.tipo, habilidadesClonadas); // Clon profundo
    }
}
```

Cada Objeto interno también clonado y no compartido.

# Patrones creacionales – Prototype

## CUÁNDO IMPLEMENTARLO

- Crear un objeto es **costoso** (cálculos pesados, consultas, I/O).
- Necesitas **crear múltiples objetos similares**.
- Quieres mantener la **lógica de creación fuera del cliente**.
- Necesitas **independencia de clases** concretas en tiempo de ejecución.
- Necesitas **clonar estructuras de datos complejas**.



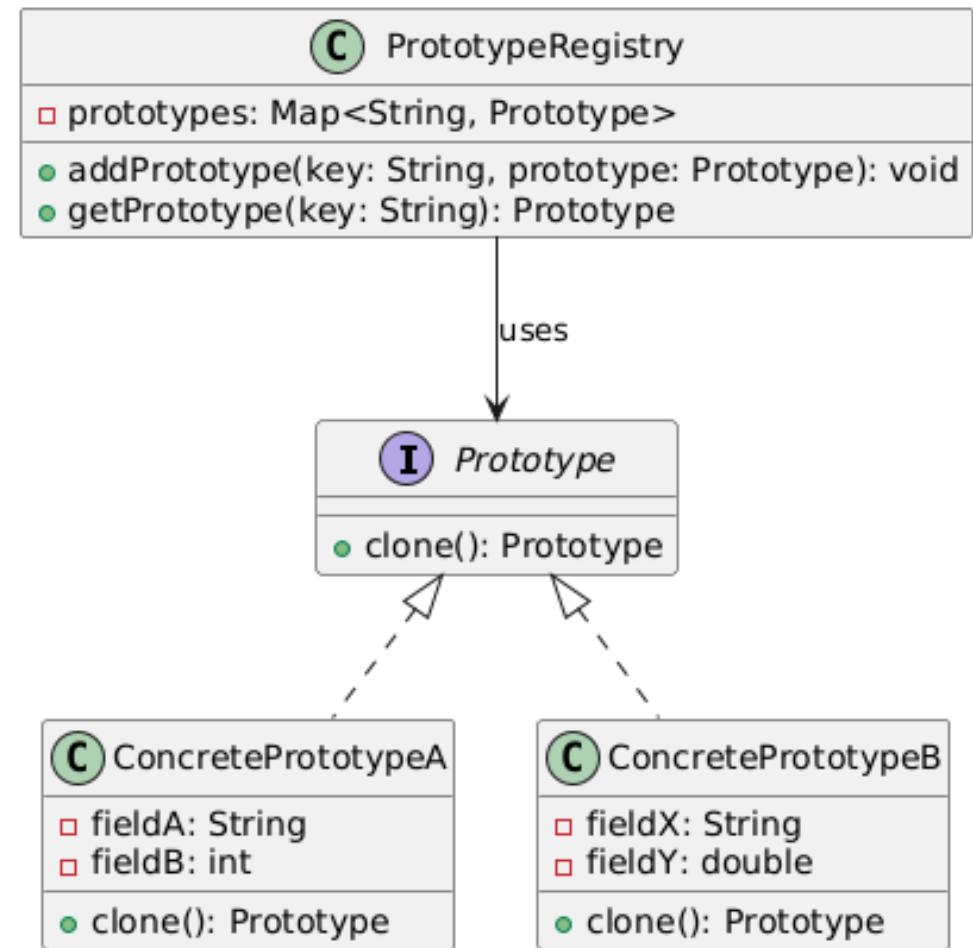
# Patrones creacionales – Prototype

## Ventajas

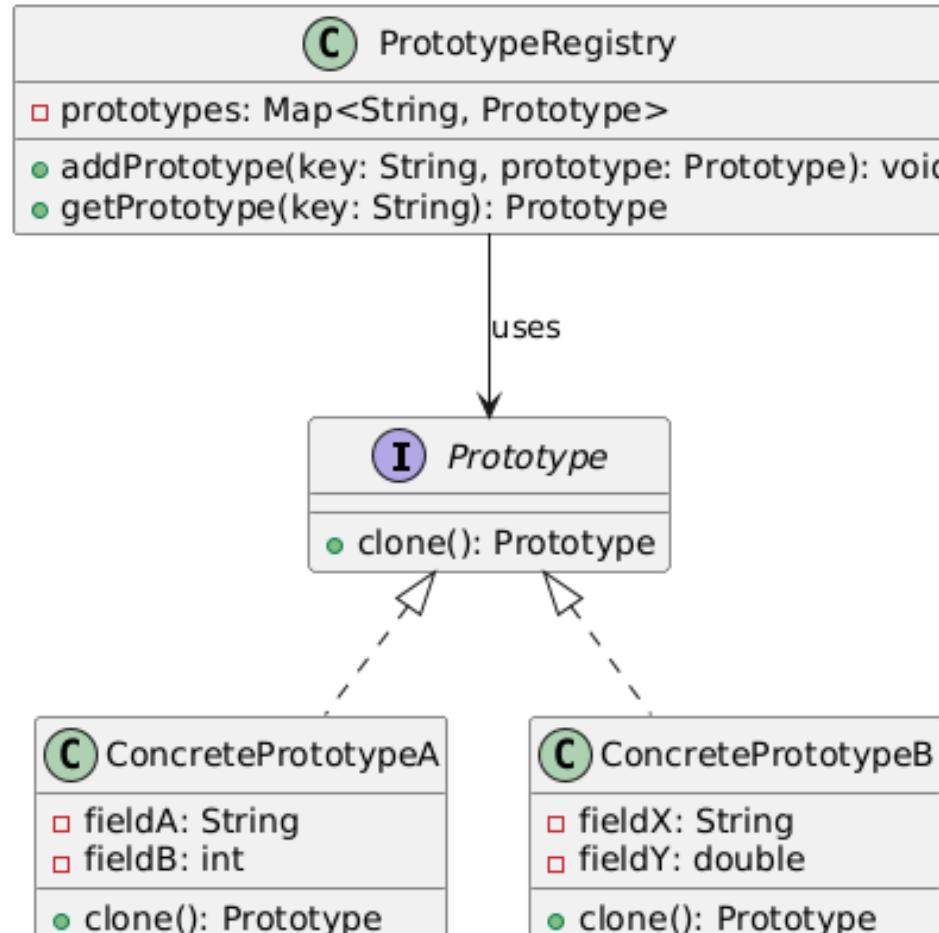
- Desacopla de las clases concretas.** El cliente no necesita saber la clase exacta. Solo usa `clone()`.
- Creación eficiente de objetos.** Muy útil para evitar procesos costosos de creación.
- Permite la clonación de estructuras complejas.** Ideal para grafos, árboles, configuraciones anidadas.
- Compatible con el registro de prototipos.** Puedes registrar prototipos y clonarlos dinámicamente.

## Desventajas

- Clonación superficial vs profunda:** Java por defecto usa clonación superficial (`Object.clone()`), y puede ser problemático si hay referencias mutables.
- Mantenimiento de lógica de clonación.** Cada clase debe implementar su propia lógica de `clone()`, lo cual puede ser complejo.
- Manejo de errores.** El uso de `Cloneable` y `Object.clone()` puede ser frágil si no se implementa correctamente.
- No compatible con todas las estructuras:** Algunos objetos no son clonables por naturaleza (e.g., objetos que dependen de recursos únicos).



# Patrones creacionales – Prototype



## Beneficios obtenidos con Prototype

### Problema original

Creación costosa en cada instancia

Repetición de lógica

Acoplamiento a clases concretas

Difícil mantenimiento

### Solución con Prototype

Se clona un prototipo ya configurado

Centralización de lógica en la clase prototipo

Cliente accede mediante el registro dinámico

Agregar nuevos tipos no afecta al cliente



# Patrones creacionales – Integración con frameworks

## SpringBoot

### Singleton

#### Uso:

- Todos los beans Spring por defecto son Singletons.
- Permite que haya una única instancia gestionada por el IoC container.

```
@Service
public class ServicioUnico {
    public void operar() {
        System.out.println("Operación única");
    }
}
```

@Service, @Component, @Repository, @Controller  
**(singleton por defecto)**

### Factory Method

#### Uso típico:

- En la creación de beans personalizados.
- También se usa cuando implementas lógicas de negocio que cambian el tipo de objeto retornado en tiempo de ejecución.

```
@Configuration
public class ConfiguracionApp {

    @Bean
    public Servicio servicio() {
        if (modo.equals("desarrollo")) {
            return new ServicioDesarrollo();
        } else {
            return new ServicioProduccion();
        }
    }
}
```

**@Configuration** Actúa como la clase fábrica concreta

**@Bean** Actúa como el método fábrica, Spring lo registra en su contenedor de IoC.



# Patrones creacionales – Integración con frameworks

## SpringBoot

### Builder

#### Uso típico:

- Cuando necesitas crear objetos con múltiples parámetros opcionales (por ejemplo, DTOs, respuestas de API).

```
@Data
@Builder
public class UsuarioResponse {
    private String nombre;
    private String email;
    private String telefono;
}
```

DTOs en servicios REST, Kafka, bases de datos, etc

#### Usa Lombok con **@Builder**.

(controladores, servicios o integraciones REST para construir objetos de salida o entrada)

```
@GetMapping("/usuario")
public ResponseEntity<UsuarioResponse> obtenerUsuario() {
    UsuarioResponse response = UsuarioResponse.builder()
        .nombre("Ana")
        .email("ana@mail.com")
        .telefono("+57 3000000000")
        .build();

    return ResponseEntity.status(HttpStatus.OK).body(response);
}
```

### Prototype

#### Uso típico:

- Cuando se requiere clonar objetos complejos configurados.
- En Spring, se puede definir el scope como **@Scope("prototype")** para que se cree una nueva instancia en cada solicitud.

```
@Component
@Scope("prototype")
public class DocumentoTemporal {
    private UUID id = UUID.randomUUID();

    public UUID getId() {
        return id;
    }
}
```



# Patrones creacionales – Integración con Arquitecturas

## Integración con Arquitecturas

### Microservicios

**Singleton** para servicios sin estado (por defecto).

**Prototype o Builder** para objetos que se construyen en tiempo de solicitud (por ejemplo, DTOs complejos, validaciones).

**Factory** para seleccionar estrategias de integración (por ejemplo, REST, SOAP, Kafka, etc.).

### Hexagonal / Ports and Adapters

**Factory** para separar implementación de puertos (interfaces).

**Builder** para mapear modelos entre capas (adapter ↔ domain).

**Abstract Factory** si se necesita seleccionar familias completas de implementaciones en tiempo de ejecución (por ejemplo, varias bases de datos o servicios externos).

### Clean Architecture

**Patrón Builder** para construir DTOs y respuestas de manera ordenada.

**Factory Method** para crear instancias de casos de uso o estrategias.

**Singleton** para clases de infraestructura compartidas



## Patrón de diseño – Recursos

### Libros:

- ***“Patrones de diseño. Elementos de software orientado a objetos reutilizables”***  
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- ***“Sumérgete en los patrones de diseño”***  
Alexander Shvets
- ***“Introducción a los patrones de diseño. Un enfoque práctico”***  
Oscar Blancharte
- ***“Patrones de diseño en java. Los 23 modelos de diseño: descripción y soluciones ilustradas”***  
Lauren Debrauwer



**UNIVERSIDAD**  
Popular del cesar