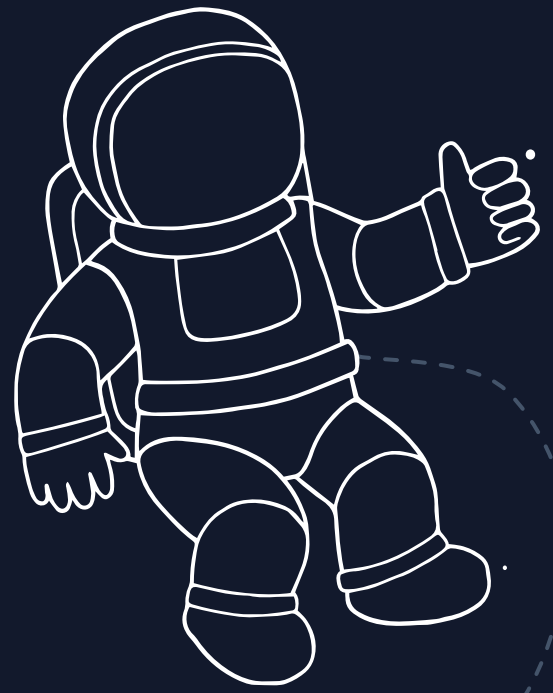
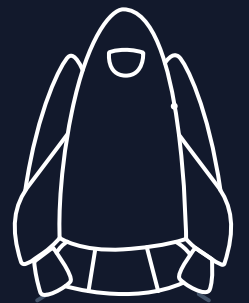


Programa académico CAMPUS

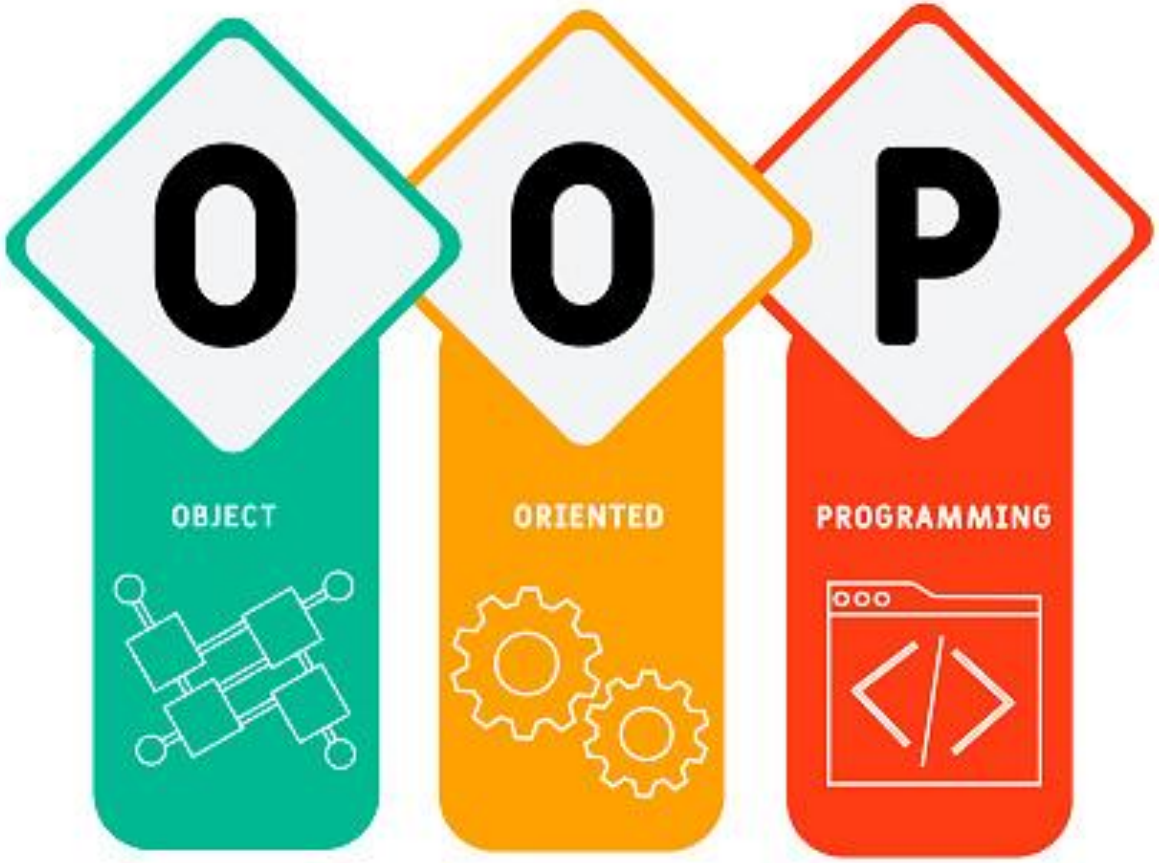
MODULO JAVA
Sesión 11



Patrones de Diseño (Creacionales)

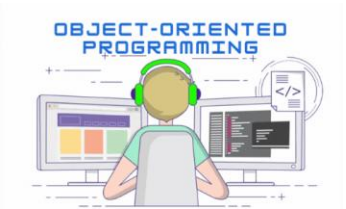
Trainer Carlos H. Rueda C.


Java™


O
O
P
OBJECT
ORIENTED
PROGRAMMING

INTRODUCCIÓN A LOS PATRONES

“ Un patrón describe primero un problema que ocurre una y otra vez en nuestro entorno, y a continuación describe el núcleo de la solución al problema, de tal manera que puede usar la solución millones de veces sin repetir la solución específica una sola vez”



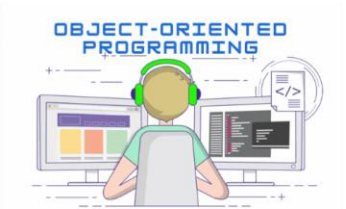
INTRODUCCIÓN A LOS PATRONES

Idiomas: patrones de bajo nivel específico a un lenguaje de programación; tales como Java y C++ (Coplien, 1992).

Patrones de Diseño: se enfoca en un pequeño grupo de de objetos que colaboran entre sí. Gamma, Helm, Johnson, Vlissides (1995)

Patrones Arquitectónicos: se enfoca en la estructura de subsistemas de un sistema. Buschmann Meunier, Rohnert, Sommerlad, Stal (1996)

Patrones de Análisis: se enfoca en patrones encontrados durante el análisis OO. Fowler (2002)

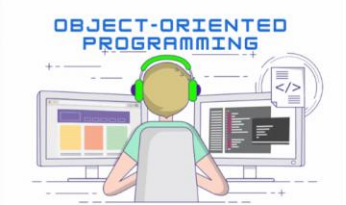


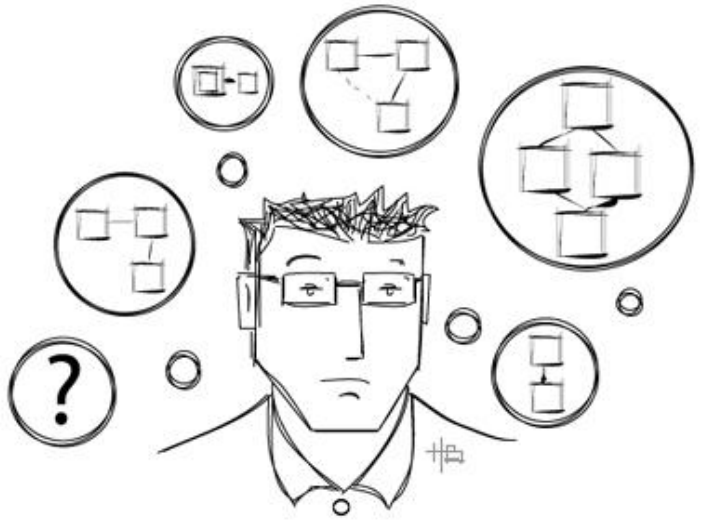
PATRONES DE DISEÑO

Los patrones de diseño son **soluciones habituales** a problemas que ocurren con **frecuencia** en el diseño de software. **Son como planos prefabricados** que se pueden personalizar para resolver un problema de diseño recurrente en el código.

Un patrón es **una solución probada** que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún campo.

Son un **esqueleto básico** que cada diseñador **adapta** a las peculiaridades de su aplicación



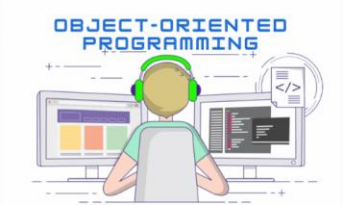


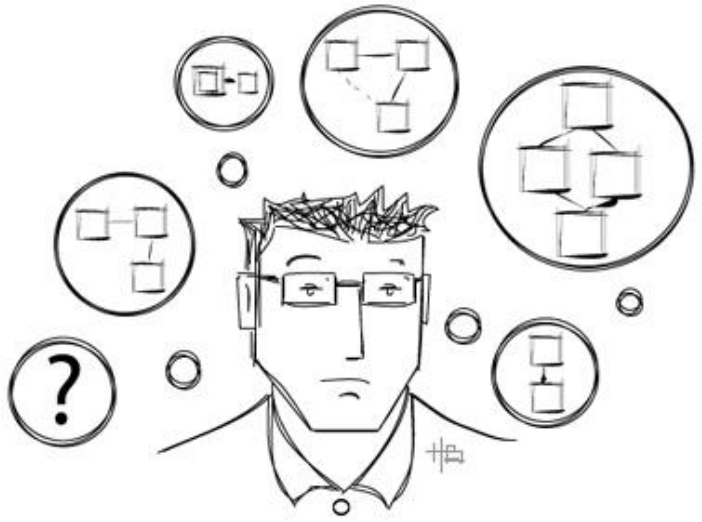
PATRONES DE DISEÑO

Los patrones de diseño son **soluciones habituales** a problemas que ocurren con **frecuencia** en el diseño de software. **Son como planos prefabricados** que se pueden personalizar para resolver un problema de diseño recurrente en el código.

Un patrón es **una solución probada** que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún campo.

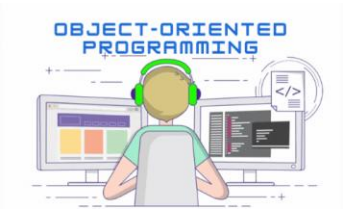
Son un **esqueleto básico** que cada diseñador **adapta** a las peculiaridades de su aplicación

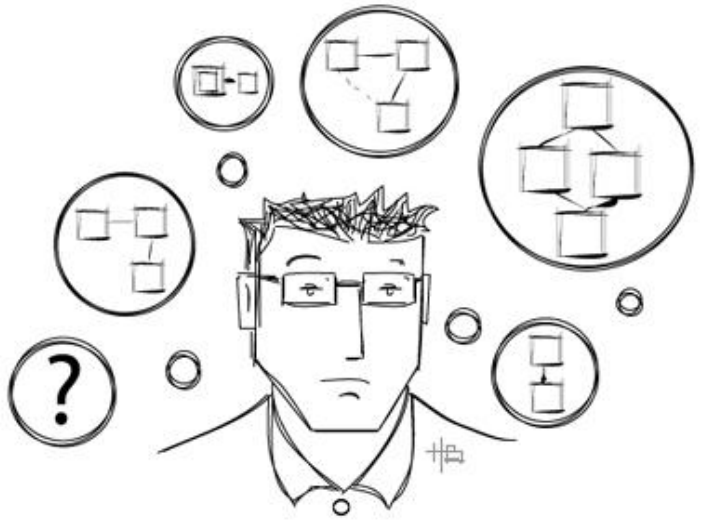




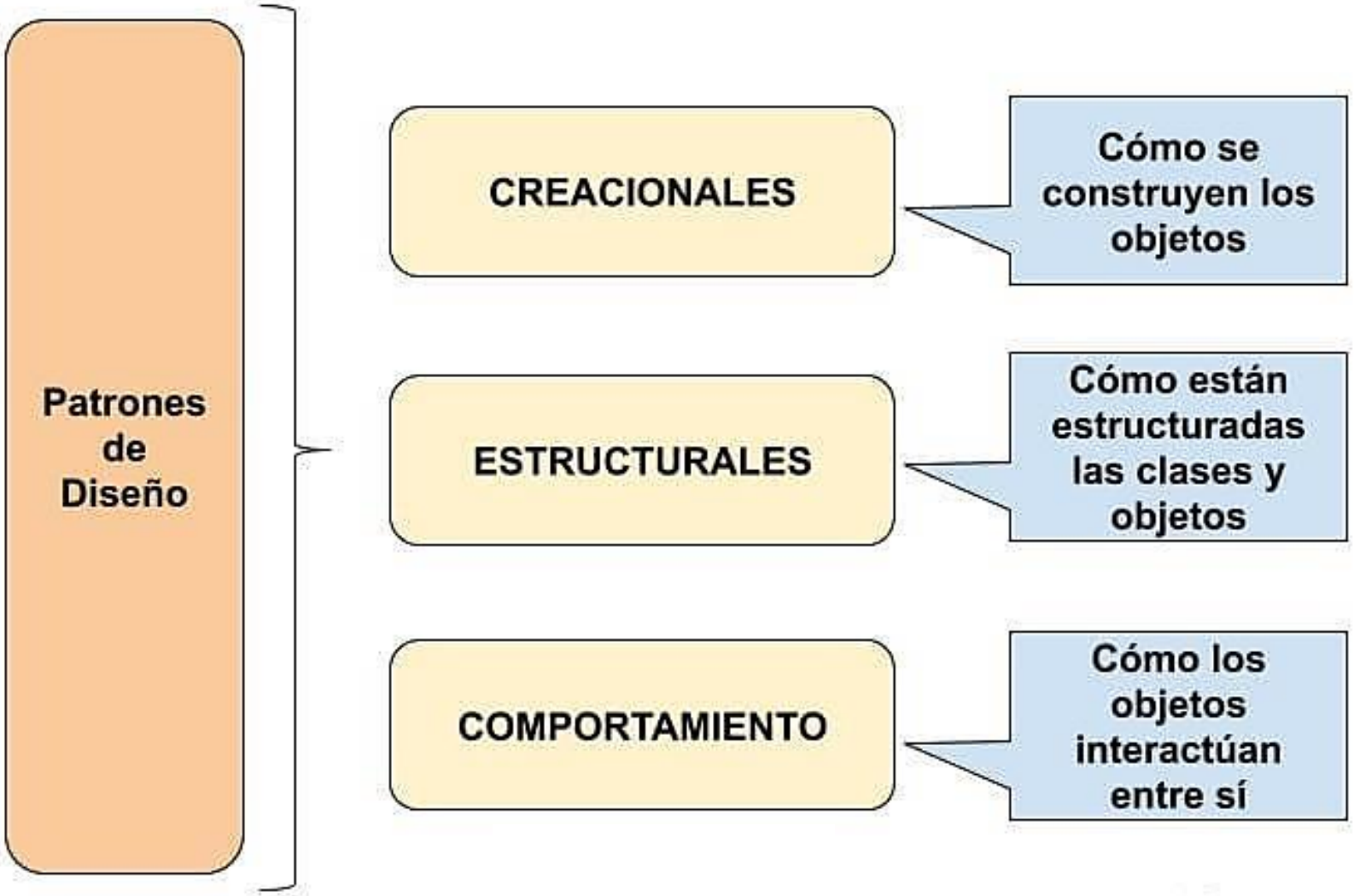
PATRONES DE DISEÑO

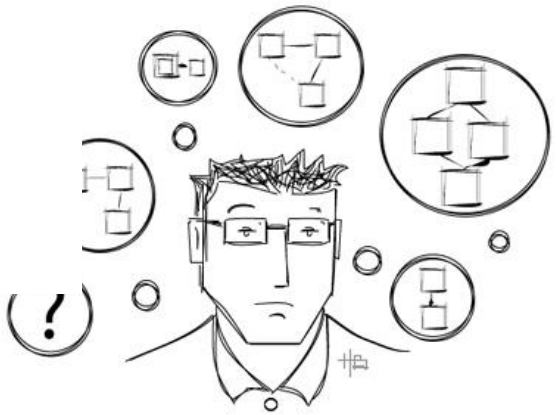
Los patrones de diseño son soluciones a nivel de diseño para problemas recurrentes que nosotros como ingenieros de software enfrentamos frecuentemente. No es código - Repito ❌ **CÓDIGO**. Es como una descripción sobre como atacar estos problemas y diseñar una solución.





PATRONES DE DISEÑO





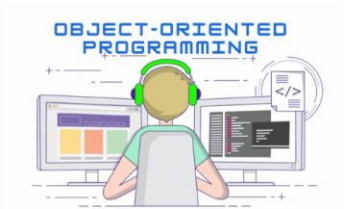
PATRONES DE DISEÑO - SINGLETON

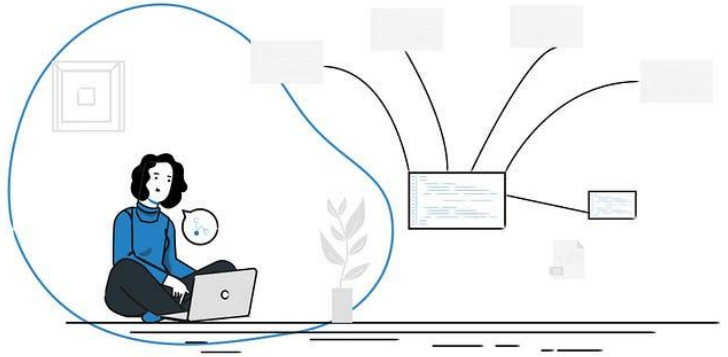
El patrón Singleton es un patrón de diseño creacional que garantiza que **una clase *solo tenga una única instancia*** y proporciona un punto de acceso global a esa instancia.

Esto se logra mediante la creación de un ***constructor privado*** en la clase y ***un método estático*** para obtener la única instancia de la clase.

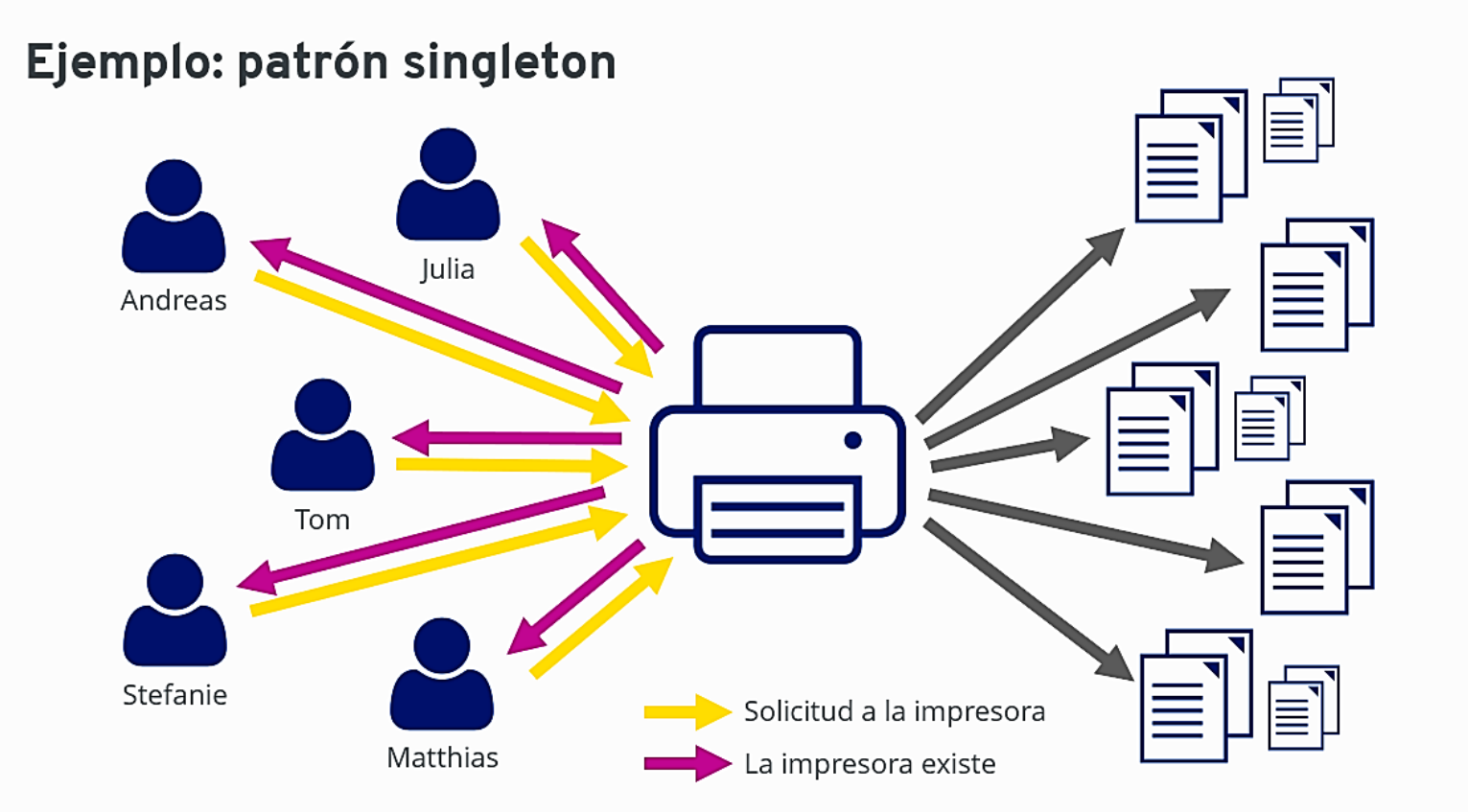
El propósito de este patrón es ***evitar que sea creado más de un objeto por clase***.

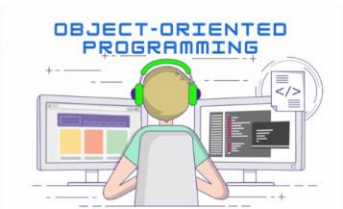
El singleton es uno de los patrones más simples, pero ***más poderosos*** en el desarrollo de software





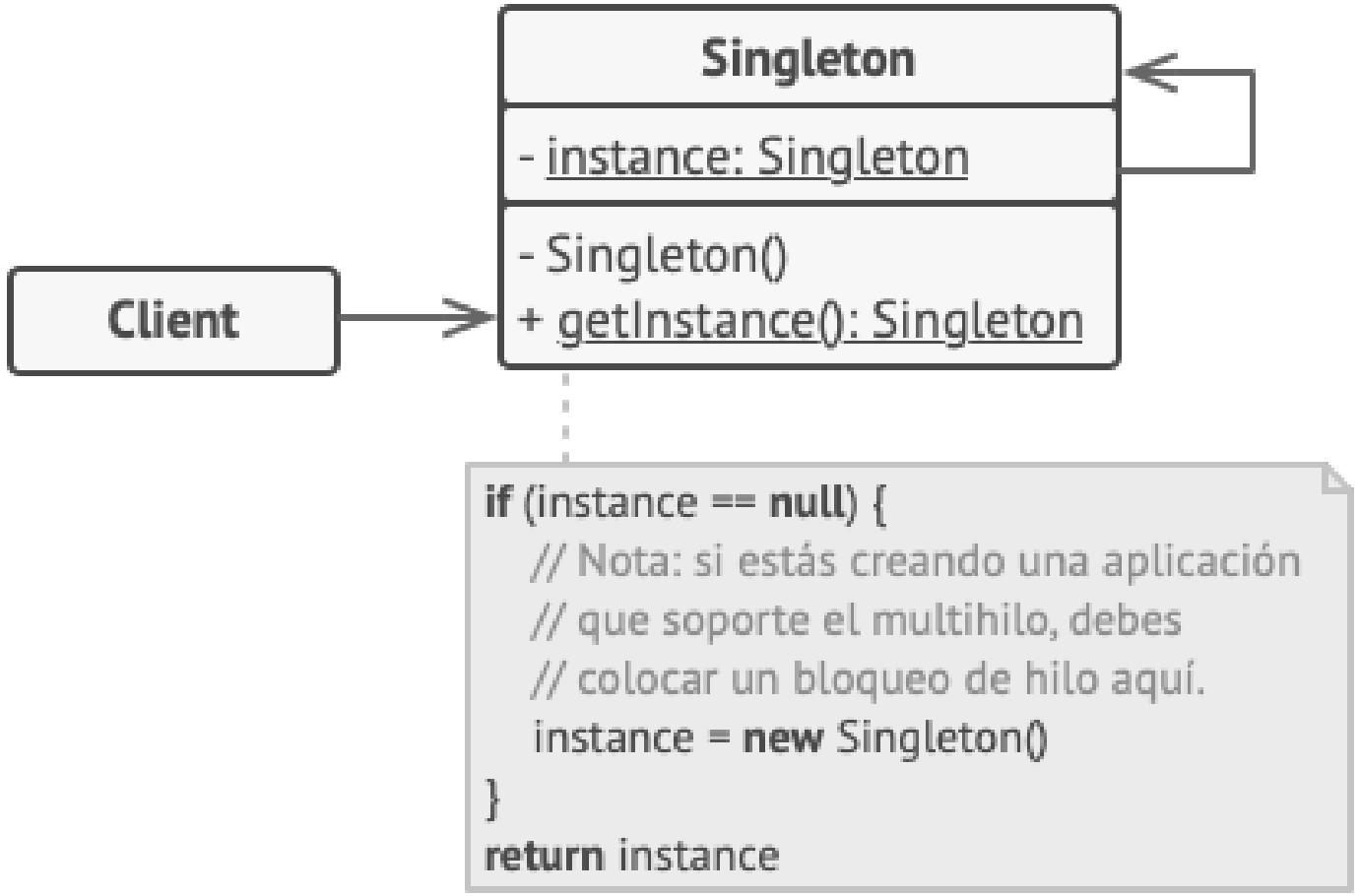
PATRONES DE DISEÑO - SINGLETON

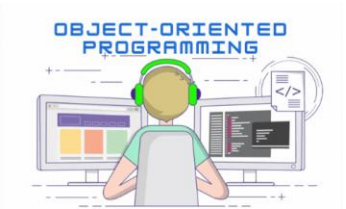




Si un usuario envía una solicitud a la impresora, el patrón singleton hace la "pregunta": "¿Ya hay un objeto de la impresora? Si no, entonces crea uno." Esto se resuelve con un if/then-statement (*devolver impresora == cero ?*).

PATRONES DE DISEÑO - SINGLETON





PATRONES DE DISEÑO - SINGLETON

Supongamos que se tiene una clase Configuracion que almacena la configuración de una aplicación y se requiere que solo haya una instancia de esta clase en toda la aplicación.

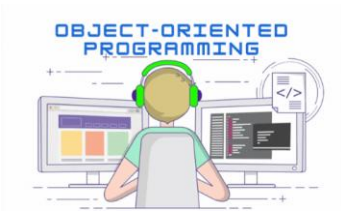
```java
public class Configuracion {
    private String servidor;
    private String puerto;
    private String usuario;
    private String contrasena;


    public Configuracion(String servidor, String puerto, String usuario,
        String contrasena) {
        this.servidor = servidor;
        this.puerto = puerto;
        this.usuario = usuario;
        this.contrasena = contrasena;
    }
    //... otros atributos y métodos getter/setter
}
```

```java
Configuracion config1 = new Configuracion(
    "servidor1",
    "8080",
    "usuario1",
    "contrasena1");

Configuracion config2 = new Configuracion(
    "servidor2",
    "9090",
    "usuario2",
    "contrasena2");
```

Se pueden crear dos o más instancias de Configuracion, lo que significa que es posible tener múltiples instancias de la misma configuración. Esto puede llevar a problemas de inconsistencia y duplicación de datos.

PATRONES DE DISEÑO – SINGLETON

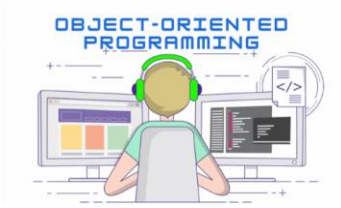
```java
public class Configuracion {
    private String servidor;
    private String puerto;
    private String usuario;
    private String contrasena;

    // Instancia única de Configuración
    private static Configuracion instancia;

    // Constructor privado para evitar la creación de nuevas instancias
    private Configuracion(String servidor, String puerto, String usuario, String contrasena) {
        this.servidor = servidor;
        this.puerto = puerto;
        this.usuario = usuario;
        this.contrasena = contrasena;
    }

    // Métodos estáticos para obtener la instancia única de Configuración
    public static Configuracion obtenerInstancia() {
        if(instancia == null) {
            instancia = new Configuracion("servidor1",
                "8080",
                "usuario1",
                "contrasena1");
        }
        return instancia;
    }
    // ... otros métodos getter/setter
}
```



PATRONES DE DISEÑO – SINGLETON

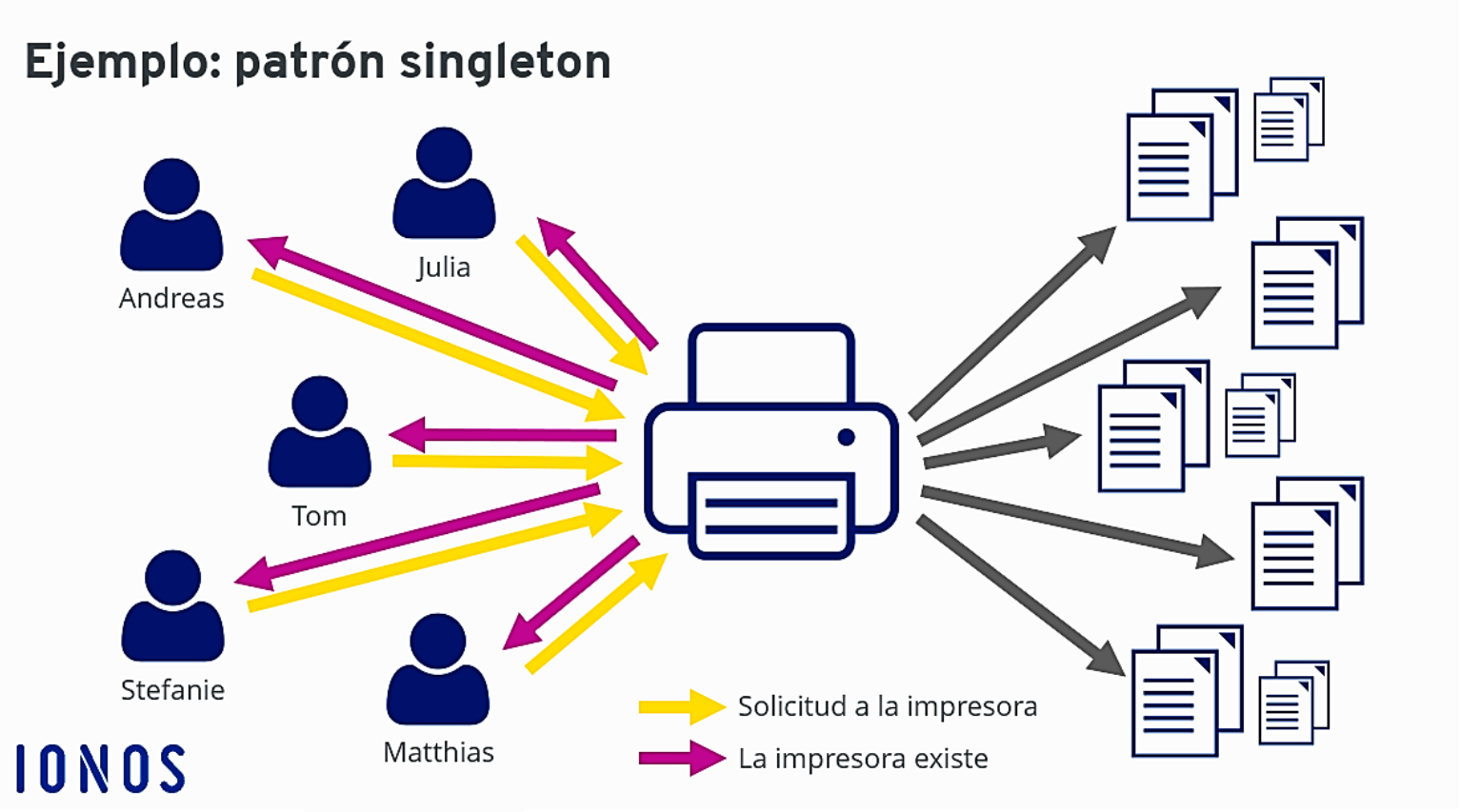
Al llamar al método obtenerInstancia(), siempre se obtendrá la misma instancia de Configuracion, lo que garantiza que solo haya una instancia de la clase.

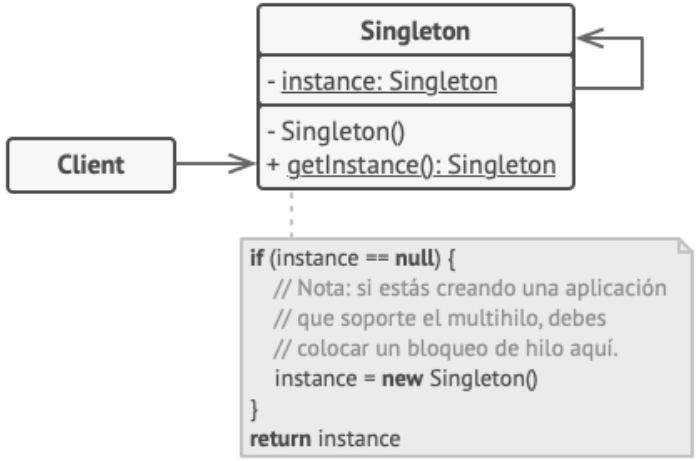
```
// Se obtiene la instancia única de Configuración
Configuracion config1 = Configuracion.obtenerInstancia();
Configuracion config2 = Configuracion.obtenerInstancia();

// Ahora config1 y config2 apuntan a la misma instancia de
Configuración
```

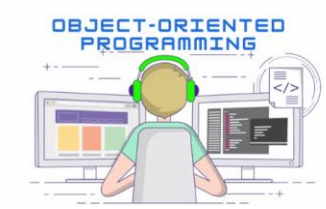


Ejercicio

Los empleados de ACME Corporation: Julia, Andreas, Tom, Stefanier, Mathias necesitan usar la impresora compartida que hay en la oficina para enviar sus documentos.
Escriba un programa en Java que implemente la situación de la oficina.



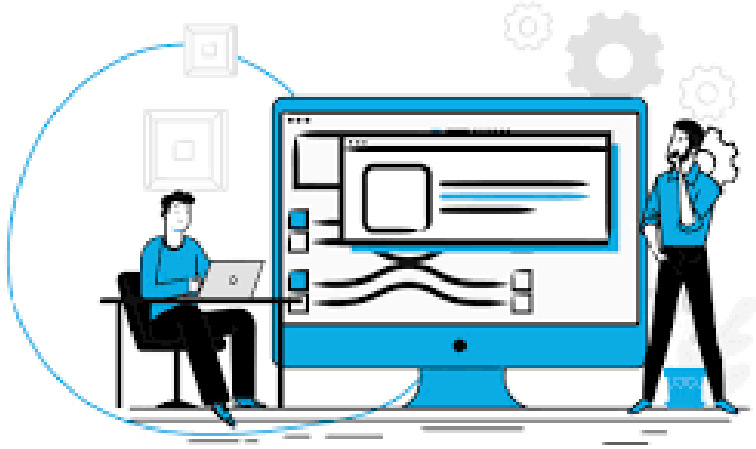




PATRONES DE DISEÑO – FACTORY METHOD

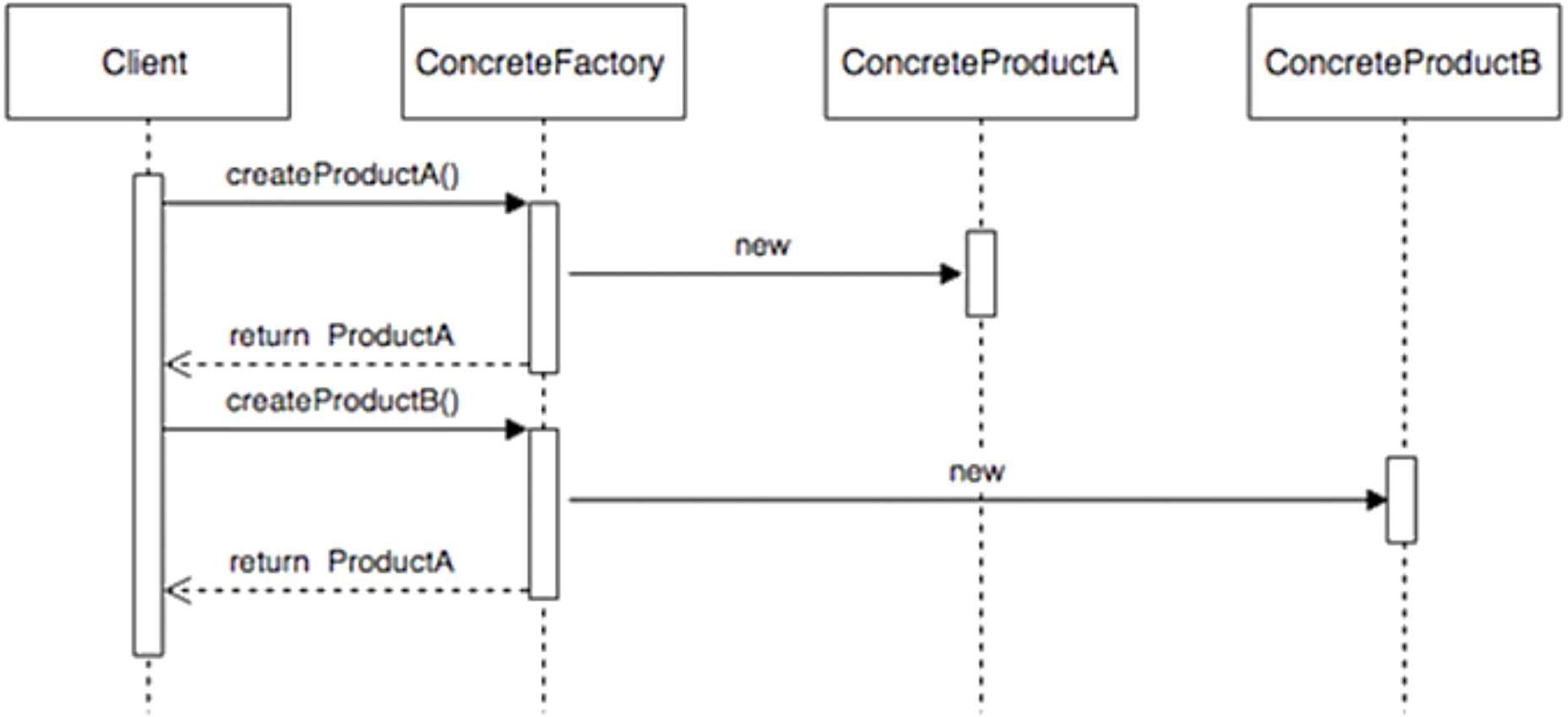
El patrón **Factory**, o patrón de diseño **Método Fact*oría***, describe un enfoque de programación que sirve para ***crear objetos sin tener que especificar su clase exacta***. Esto quiere decir que el objeto creado *puede intercambiarse con flexibilidad y facilidad*.

El patrón de Diseño **Factory Method** tiene como función abordar el problema de crear objetos sin tener que especificar la clase exacta a la que han de pertenecer y *sin tener que acceder directamente a la lógica de creación*.

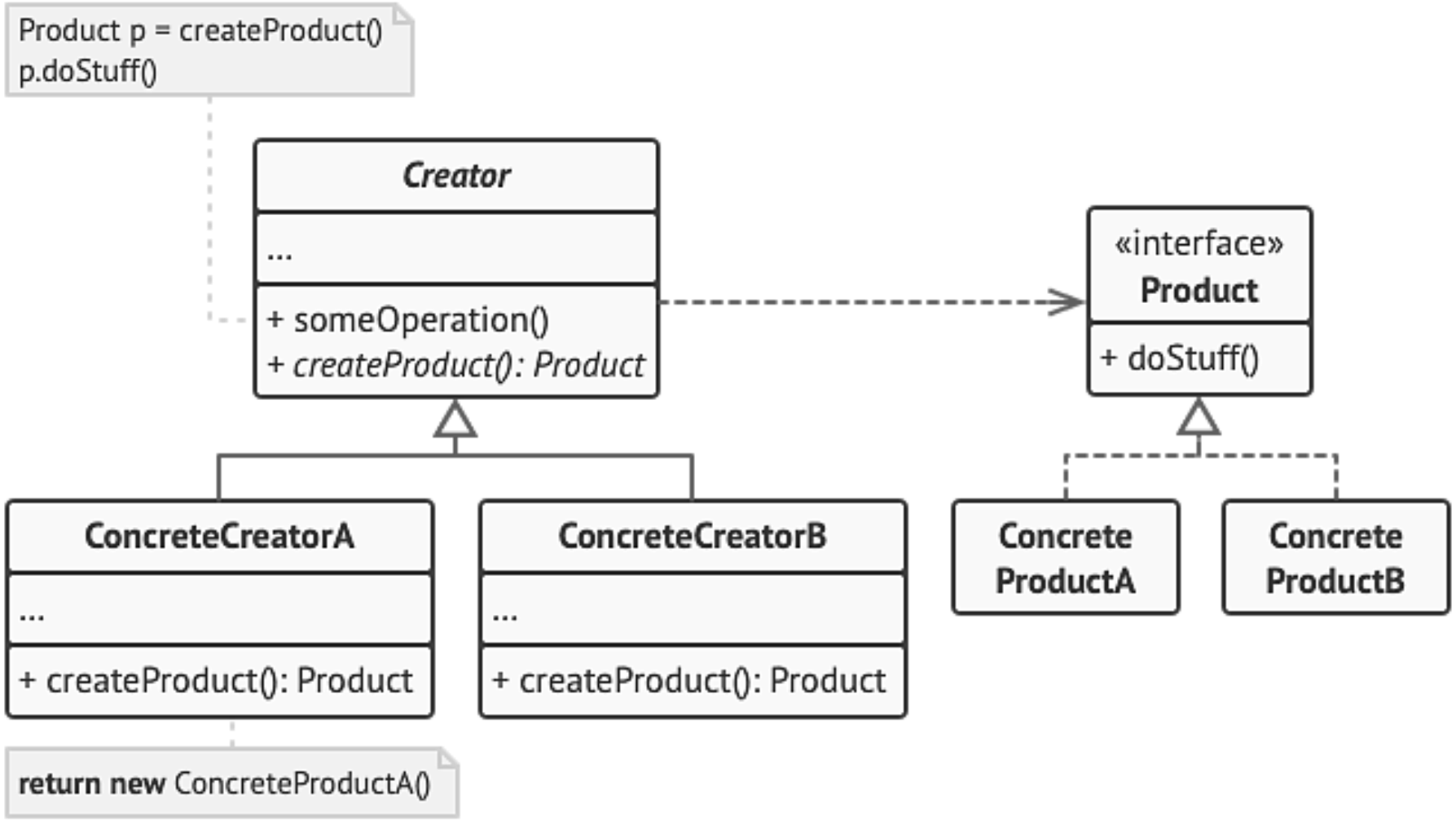


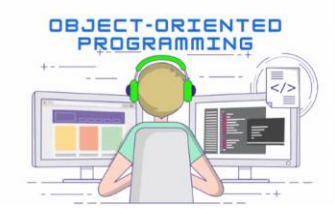
PATRONES DE DISEÑO – FACTORY METHOD

Factory Method – Diagram of sequence



PATRONES DE DISEÑO – FACTORY METHOD





FACTORY METHOD - EJEMPLO

Supongamos que se necesita crear diferentes tipos de objetos Animal (***como Perro, Gato y Pato***), es decir, Perro, Gato y Pato heredan de Animal.

```java
public class Gato {
    public Gato(){

    }

    void hacerSonido() {
        System.out.println(x:"Miaw Miaw");
    }
}
```

```java
public class Pato {
    public Pato(){

    }

    void hacerSonido() {
        System.out.println(x:"Cuac Cuac");
    }
}
```

```java
public class Perro {
    public Perro(){

    }

    void hacerSonido() {
        System.out.println(x:"Gua Gua");
    }
}
```

```java
public class ClienteAnimales {
    Run | Debug
    public static void main(String[] args) {

        Perro perro = new Perro();
        Gato gato = new Gato();
        Pato pato = new Pato();
        perro.hacerSonido();
        gato.hacerSonido();
        pato.hacerSonido();
    }
}
```

FACTORY METHOD - EJEMPLO

Factoria Simple: Una es crear un método estático de fábrica en la clase Animal

```java
public abstract class Animal {
    public Animal(){

    }

    abstract void hacerSonido();

    public static Animal crearAnimal(String tipo) {
        if (tipo.equals(anObject:"Gato")) {
            return new Gato();
        } else if (tipo.equals(anObject:"Perro")) {
            return new Perro();
        } else if (tipo.equals(anObject:"Pato")) {
            return new Pato();
        }
        return null;
    }
}
```

```java
public class ClienteAnimales {
    Run | Debug
    public static void main(String[] args) {



        Animal perro = Animal.crearAnimal(tipo:"Perro");
        Animal gato = Animal.crearAnimal(tipo:"Gato");
        Animal pato = Animal.crearAnimal(tipo:"Pato");

        perro.hacerSonido();
        gato.hacerSonido();
        pato.hacerSonido();
    }
}
```

FACTORY METHOD - EJEMPLO

Factoria Simple: Otra opción es crear una clase que sirva como fábrica que contenga este método para quitarle la responsabilidad a la misma clase Animal de la siguiente

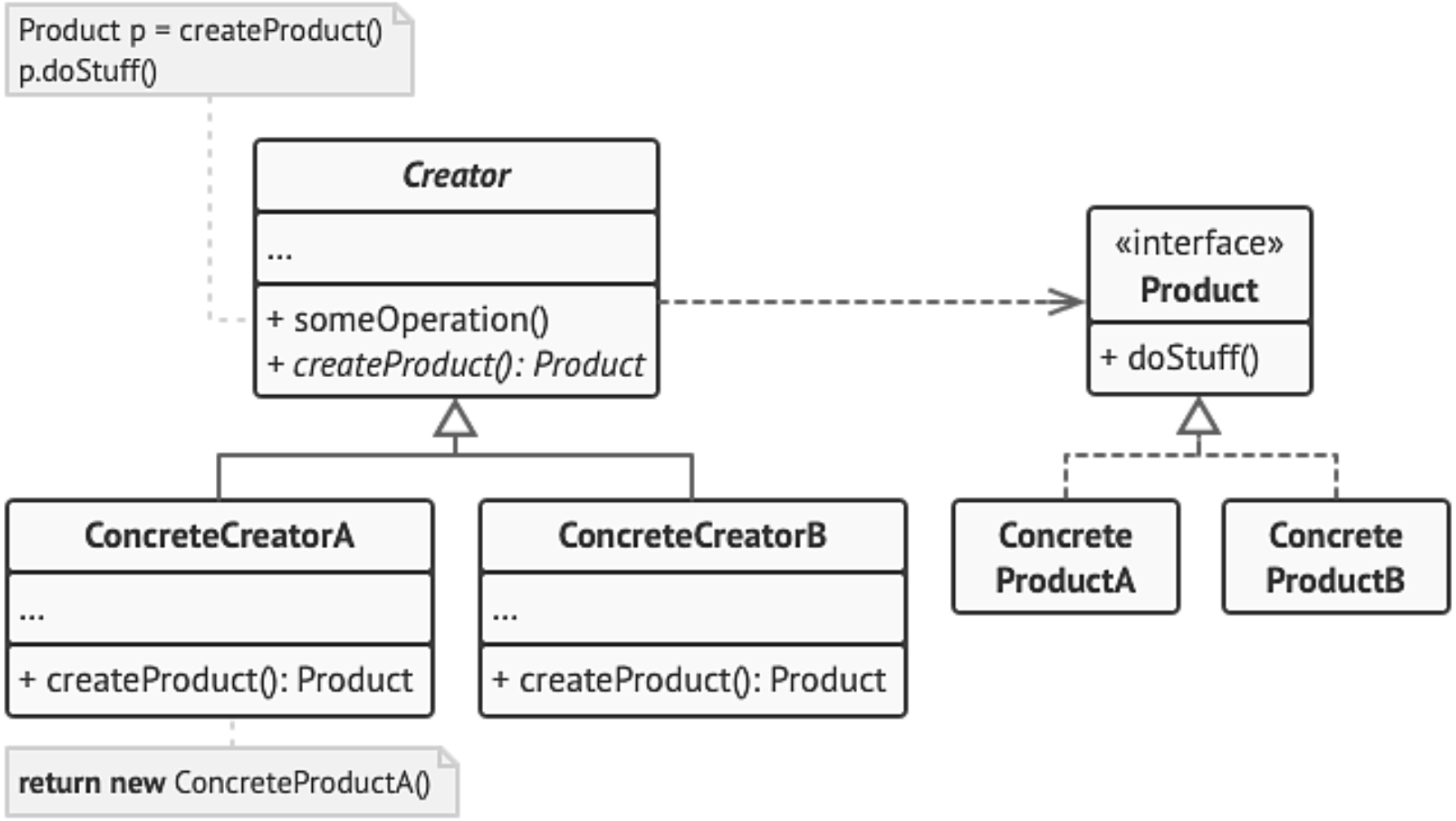
```java
public abstract class Animal {
    public Animal(){

    }

    abstract void hacerSonido();
}
```

```java
public class FactoriaAnimales {
    public static Animal crearAnimal(String tipo) {
        if (tipo.equalsIgnoreCase(anotherString:"Perro")) {
            return new Perro();
        } else if (tipo.equalsIgnoreCase(anotherString:"Gato")) {
            return new Gato();
        } else if (tipo.equalsIgnoreCase(anotherString:"Pato")) {
            return new Pato();
        } else {
            throw new IllegalArgumentException(s:"Tipo de animal no soportado.");
```

```java
public class ClienteAnimales {
    Run | Debug
    public static void main(String[] args) {
        Animal perro = FactoriaAnimales.crearAnimal(tipo:"Perro");
        Animal gato = FactoriaAnimales.crearAnimal(tipo:"Gato");
        Animal pato = FactoriaAnimales.crearAnimal(tipo:"Pato");

        perro.hacerSonido();
        gato.hacerSonido();
        pato.hacerSonido();
    }
}
```

FACTORY METHOD - EJEMPLO

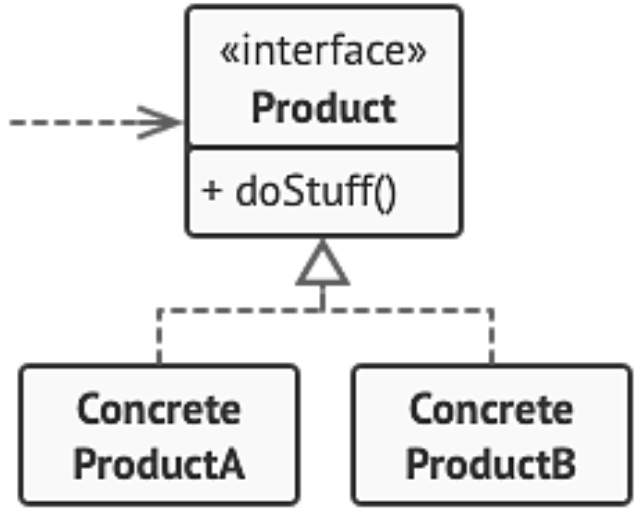
Factoria Method



FACTORY METHOD - EJEMPLO

Factoria Method

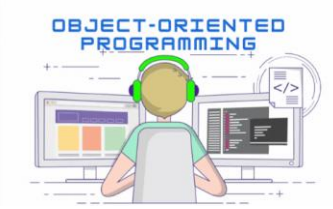
```java
public interface Animal {
    void hacerSonido();
}
```




```java
public class Gato implements Animal {
    public void hacerSonido() {
        System.out.println(x:"Miaw Miaw");
    }
}
```

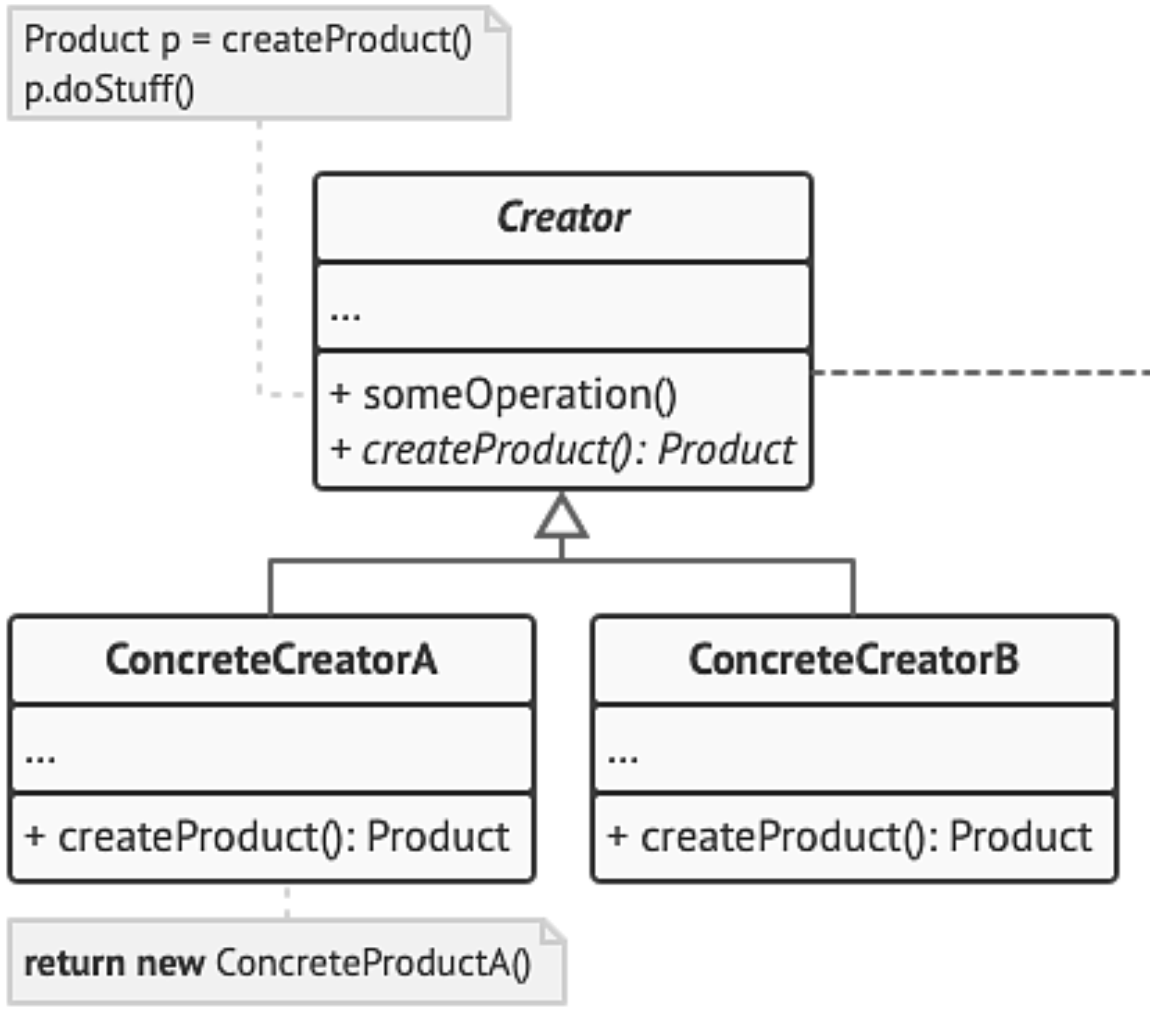
```java
public class Pato implements Animal {
    public void hacerSonido() {
        System.out.println(x:"Cuac Cuac");
    }
}
```

```java
public class Perro implements Animal {
    public void hacerSonido() {
        System.out.println(x:"Gua Gua");
    }
}
```



FACTORY METHOD - EJEMPLO

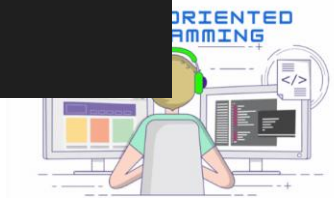
Factoria Method




```java
public abstract class Creator implements Animal {
    public abstract Animal createProduct();

    public void hacerSonido() {
        Animal animal = this.createProduct();
        animal.hacerSonido();
    }
}
```

```java
public class GatoCreator extends Creator {
    public Animal createProduct() {
        return new Gato();
    }
}

public class PatoCreator extends Creator {
    public Animal createProduct() {
        return new Pato();
    }
}
```

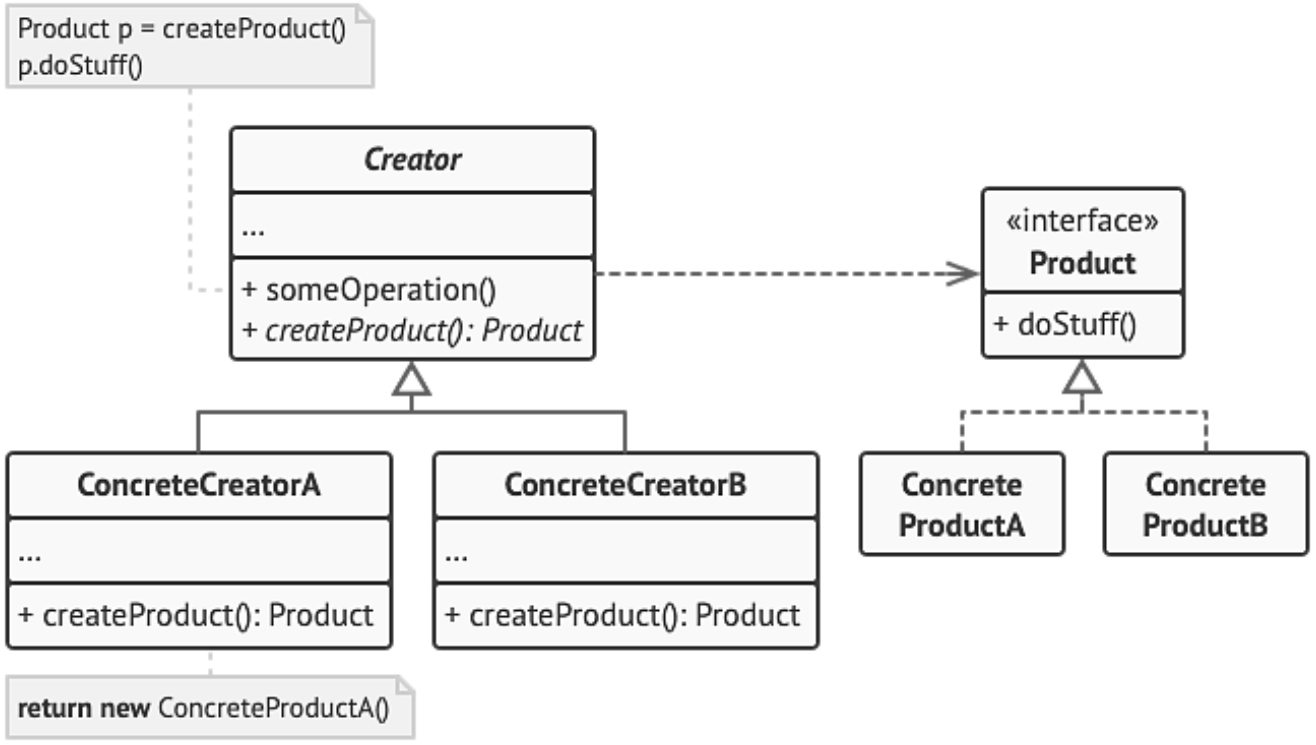
```java
public class PerroCreator extends Creator {
    public Animal createProduct() {
        return new Perro();
    }
}
```



FACTORY METHOD - EJERCICIO

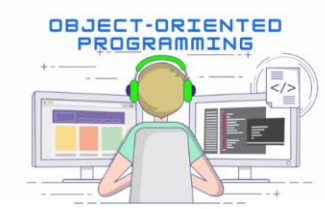
Una empresa de comidas rápidas desea implementar un punto de venta para sus productos. Por el momento ofrecen Pizzas, Hamburguesa y Kebab. En el futuro cercano desean poder agregar más productos en su oferta.

Diseñar y crear las clases implementando el patrón Factory Method para dar solución al punto de venta de la empresa de comidas rápidas.



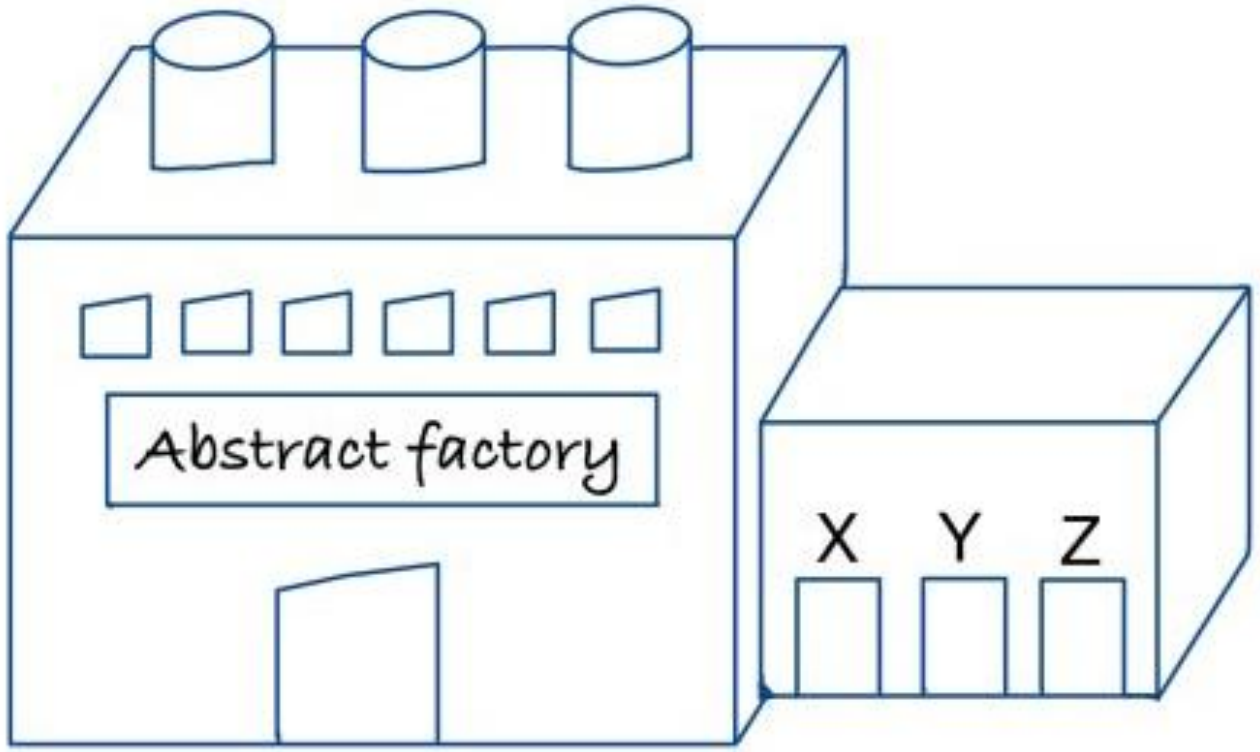
FACTORY METHOD - EJEMPLO

Crear e implementar en Java usando Factory Method una aplicación que permita crear Cuentas de ahorro y Cuentas corrientes para diferentes bancos



ABSTRACT FACTORY METHOD - EJEMPLO

El patrón **Abstract Factory** es un patrón de diseño creacional que proporciona una ***interfaz para crear familias de objetos relacionados*** sin especificar sus clases concretas. Es útil cuando se necesita ***crear objetos que pertenecen a varias categorías relacionadas*** y se requiere asegurar que los objetos creados sean coherentes y compatibles entre sí.

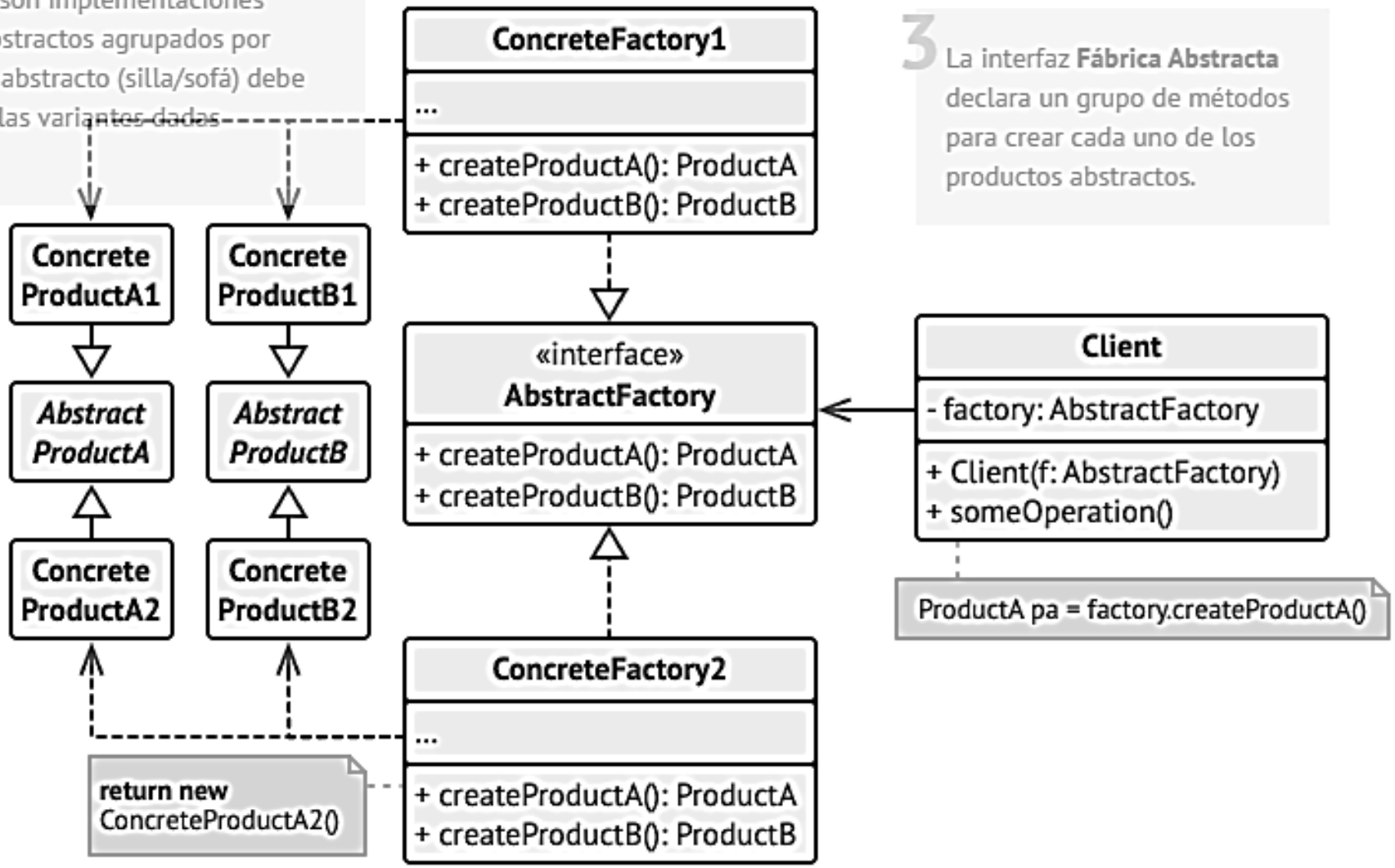


ABSTRACT FACTORY METHOD - EJEMPLO

2 Los **Productos Concretos** son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto (silla/sofá) debe implementarse en todas las variantes dadas (victoriano/moderno).

1 Los **Productos Abstractos** declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.

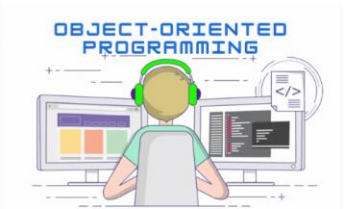
3 La interfaz **Fábrica Abstracta** declara un grupo de métodos para crear cada uno de los productos abstractos.



ABSTRACT FACTORY METHOD - EJEMPLO

Supongamos que existen dos tipos de widgets en una interfaz de usuario: botones y campos de texto, y también existen dos estilos de interfaz: Light y Dark. En este caso, se necesita crear cuatro clases concretas para cada combinación de widgets y estilos:

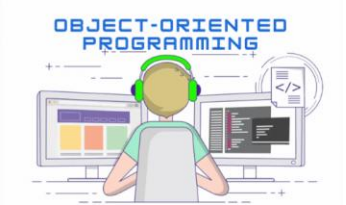
1. Botón Light
2. Texto Light
3. Botón Dark
4. Texto Dark



ABSTRACT FACTORY METHOD - EJEMPLO

```java
// Interfaz para botones
public interface Boton {
    public void render();
}
```

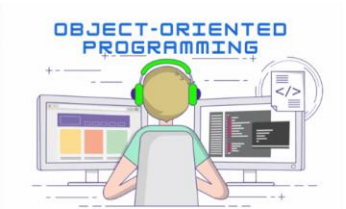
```java
// Clases concretas de botones para cada estilo
class BotonLight implements Boton {
    @Override
    public void render() {
        System.out.println("Boton Light");
    }
}

class BotonDark implements Boton {
    @Override
    public void render() {
        System.out.println("Boton Dark");
    }
}
```



ABSTRACT FACTORY METHOD - EJEMPLO

```java
// Interfaz para campos de texto
public interface CampoTexto {
    void render();
}
```

```java
// Clases concretas de campos de texto para cada estilo
class CampoTextoLight implements CampoTexto {
    @Override
    public void render() {
        System.out.println("Campo de Texto Light");
    }
}

class CampoTextoDark implements CampoTexto {
    @Override
    public void render() {
        System.out.println("Campo de Texto Dark");
    }
}
```

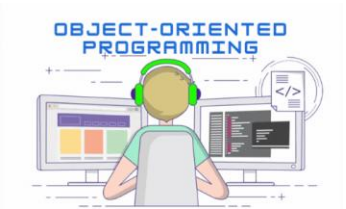


ABSTRACT FACTORY METHOD - EJEMPLO

A continuación, se implementa el patrón Abstract Factory para resolver este problema

```java
// Abstract Factory
public interface InterfazFactory {
    Boton crearBoton();
    CampoTexto crearCampoTexto();
}
```

```java
// Concrete Factory para estilo Light
class InterfazLightFactory implements InterfazFactory {
    @Override
    public Boton crearBoton() {
        return new BotonLight();
    }

    @Override
    public CampoTexto crearCampoTexto() {
        return new CampoTextoLight();
    }
}

// Concrete Factory para estilo Dark
class InterfazDarkFactory implements InterfazFactory {
    @Override
    public Boton crearBoton() {
        return new BotonDark();
    }

    @Override
    public CampoTexto crearCampoTexto() {
        return new CampoTextoDark();
    }
}
```



ABSTRACT FACTORY METHOD - EJEMPLO

Ahora, se puede usar el patrón Abstract Factory para obtener los widgets adecuados según el estilo que desee

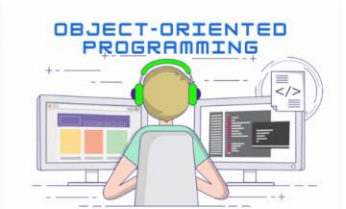
```
// Usando el estilo Light
InterfazFactory factoryLight = new InterfazLightFactory();
Boton botonLight = factoryLight.crearBoton();
CampoTexto campoTextoLight = factoryLight.crearCampoTexto();

botonLight.render(); // Salida: boton Light
campoTextoLight.render(); // Salida: Campo de Texto Light

// Usando el estilo Dark
InterfazFactory factoryDark = new InterfazDarkFactory();
Boton botonDark = factoryDark.crearBoton();
CampoTexto campoTextoDark = factoryDark.crearCampoTexto();

botonDark.render(); // Salida: boton Dark
campoTextoDark.render(); // Salida: Campo de Texto Dark
```

ABSTRACT FACTORY METHOD - EJEMPLO

ABSTRACT FACTORY METHOD - EJEMPLO

Supongamos que queremos un sistema que permita ***crear diferentes tipos*** de cuentas bancarias (por ejemplo, cuentas de ahorro y cuentas corrientes) en diferentes bancos.

El patrón Abstract Factory ***nos permite crear estas familias*** de objetos relacionados ***sin especificar*** sus clases concretas.



ABSTRACT FACTORY METHOD - EJERCICIO

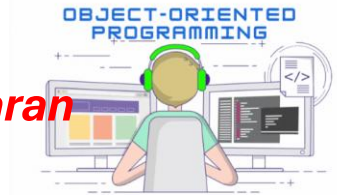
En un sistema bancario existen diferentes familias de productos con características distintas.

- Están asociados a varios tipos de cuenta que dependen del tipo de cliente que las abra:
- Cuenta Joven, para clientes < 25 años
- Cuenta 10, para clientes entre 26 y 65 años y con nómina domiciliada
- Cuenta Oro, para mayores de 65 años con pensión
- Cuenta Estándar, para clientes que no encajan en las anteriores
- Las características de los productos se resumen en la siguiente tabla

Tipo de cuenta	Cuenta	Tarjeta débito	Tarjeta crédito	Regalo
Joven	2% de interés	Gratuita	No	CD música
10	1% de interés 50% descubierto	Gratuita	18 euros 60% nómina	Reproductor CD
Oro	1,5%	Gratuita	Gratuita 60% pensión	Seguro
Estándar	0,5%	5 euros	No	No

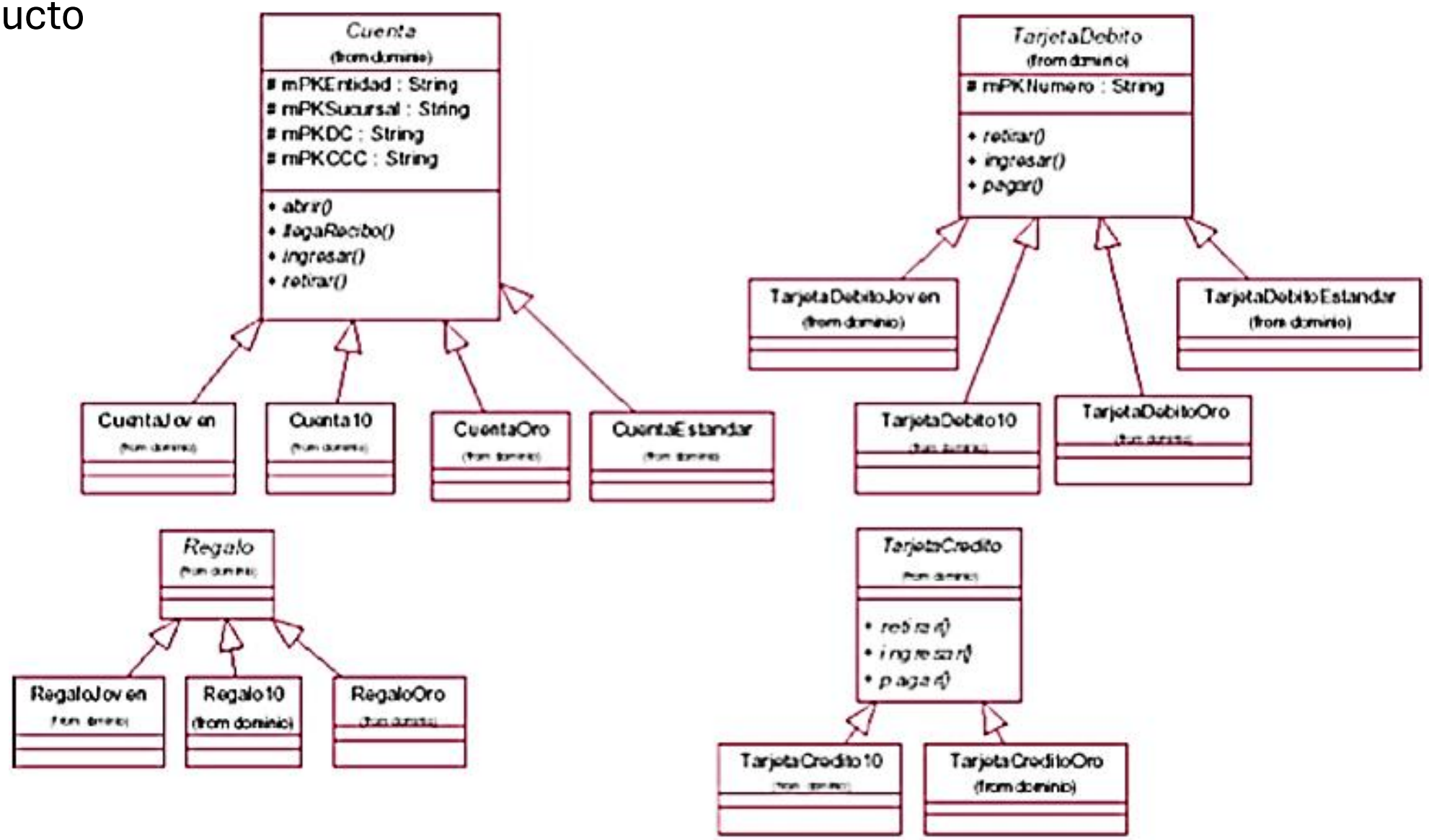
- Tal y como se ve, aunque puede parecer que se ofrecen siempre los mismos productos, existen ciertas particularidades entre ellos que hacen que sean distintos, ciertas características que hacen que se construyan de manera distinta.

Cree un programa en Java que permita tener la opción de que al crear una cierta cuenta se crearan automáticamente los productos asociados al tipo de cuenta



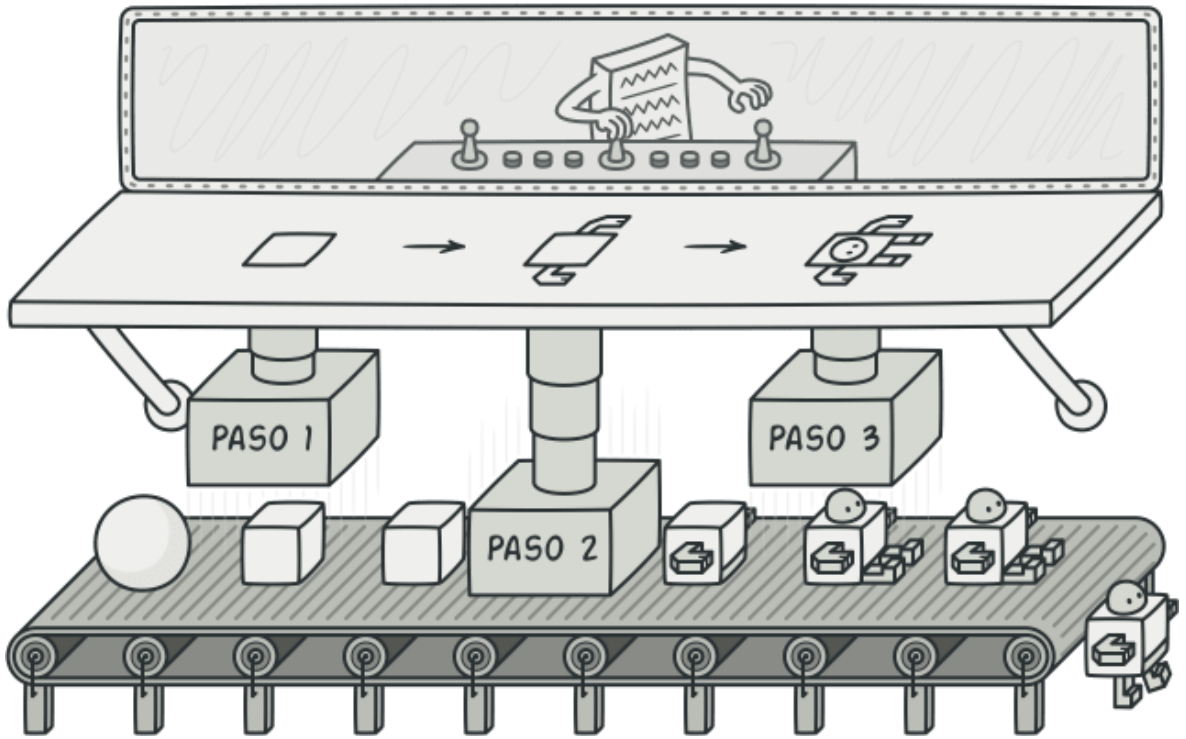
ABSTRACT FACTORY METHOD - EJERCICIO

Al final del documento se encuentra el diagrama de clases que modela el conjunto de producto

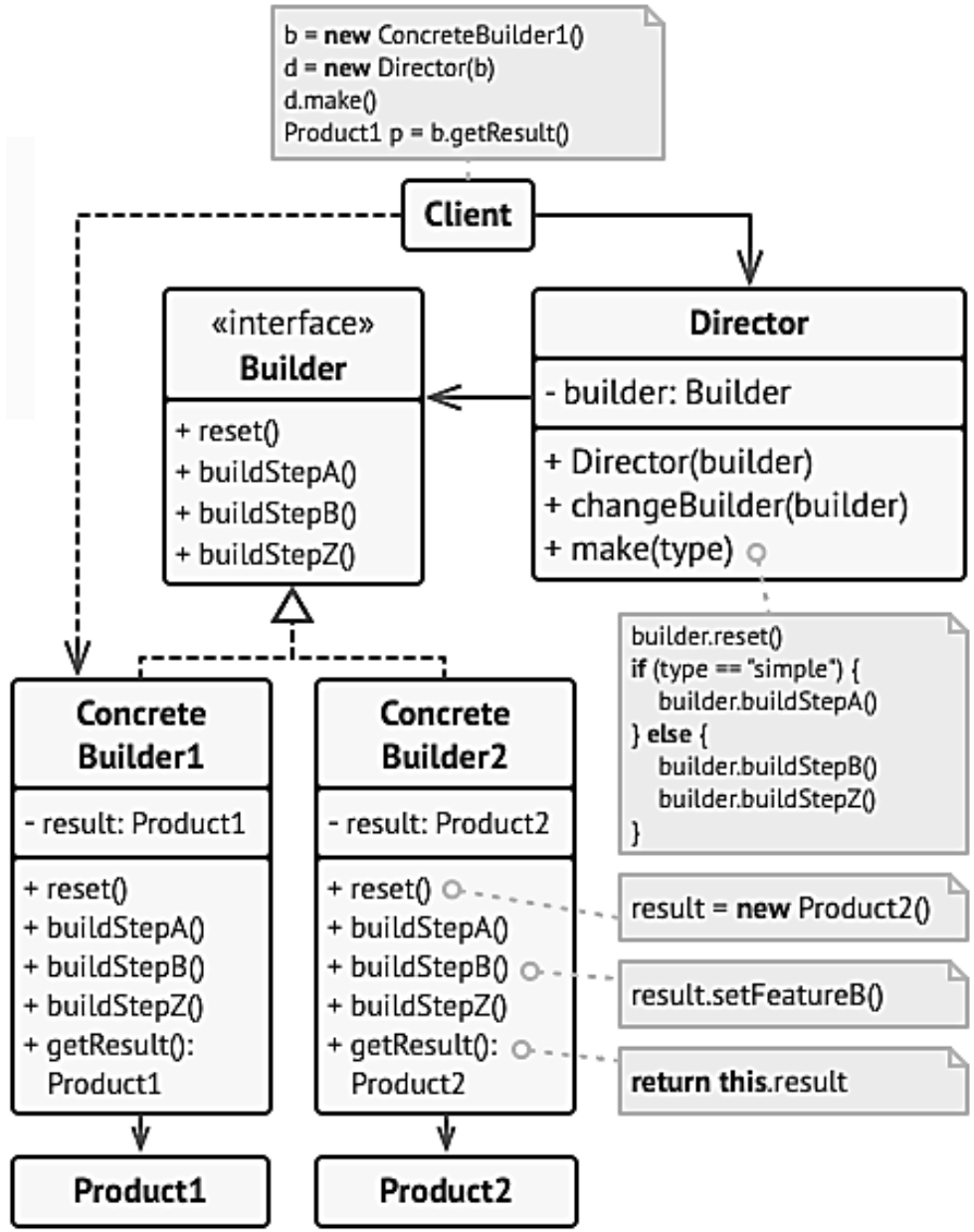


PATRONES DE DISEÑO – BUILDER

El patrón ***Builder*** es un patrón de diseño creacional que se ***utiliza para construir objetos complejos paso a paso***, separando la construcción de un objeto de su representación. Este patrón ***permite crear diferentes variaciones de un objet***o sin modificar su código cliente.

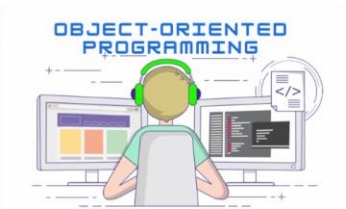


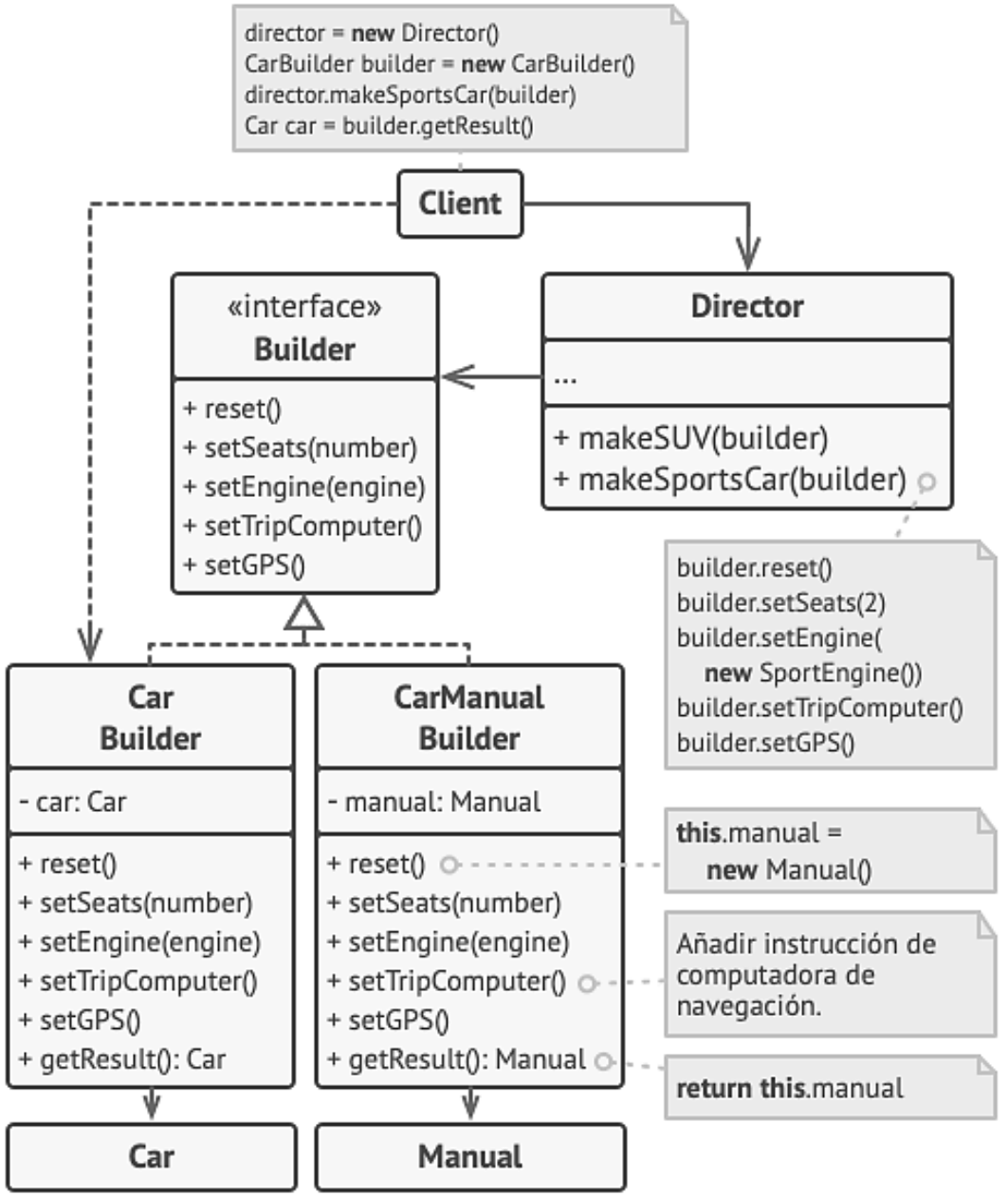
PATRONES DE DISEÑO – BUILDER



PATRONES DE DISEÑO – BUILDER

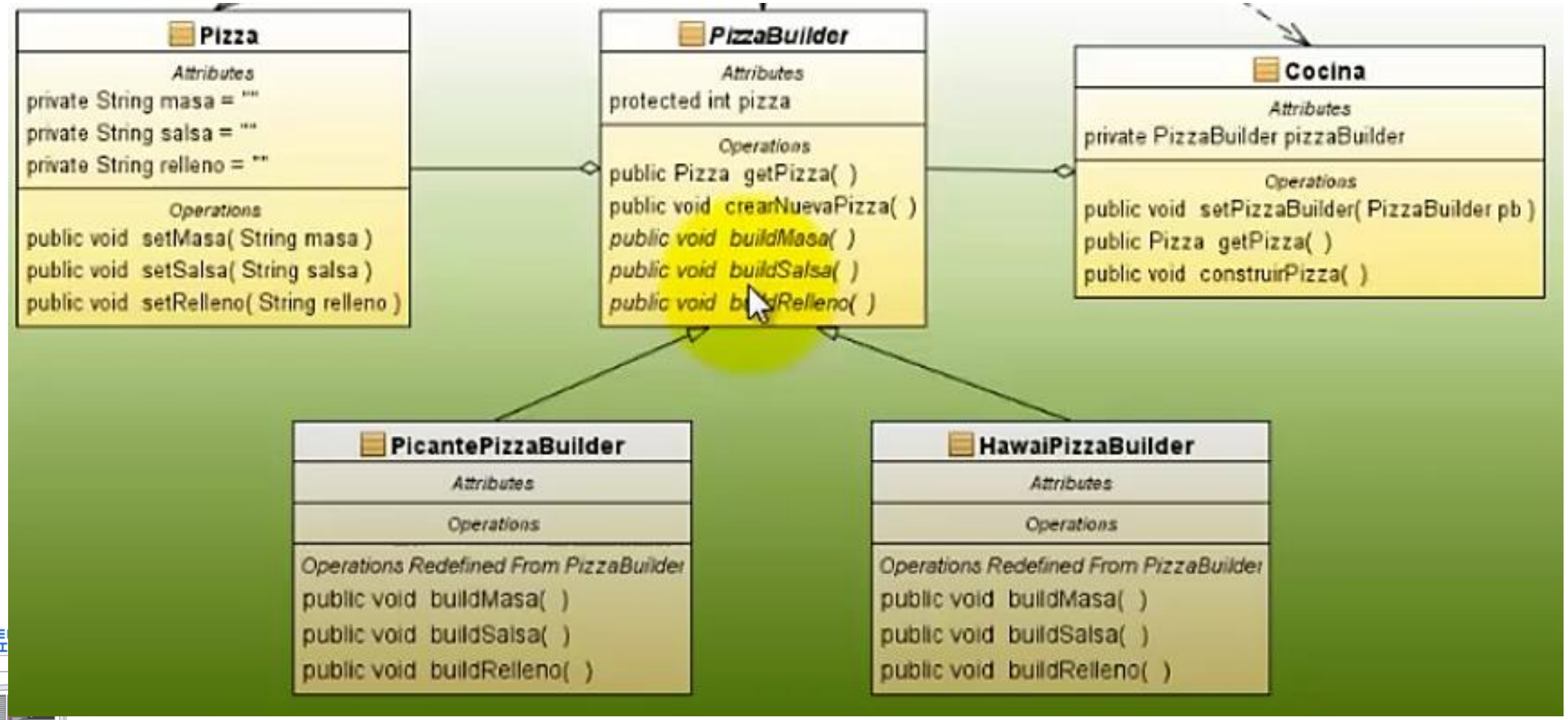
Este ejemplo del patrón **Builder** ilustra cómo se puede reutilizar ***el mismo código de construcción*** de objetos a la hora ***de construir distintos tipos de productos***, como **automóviles**, y crear los correspondientes **manuales** para esos automóviles.





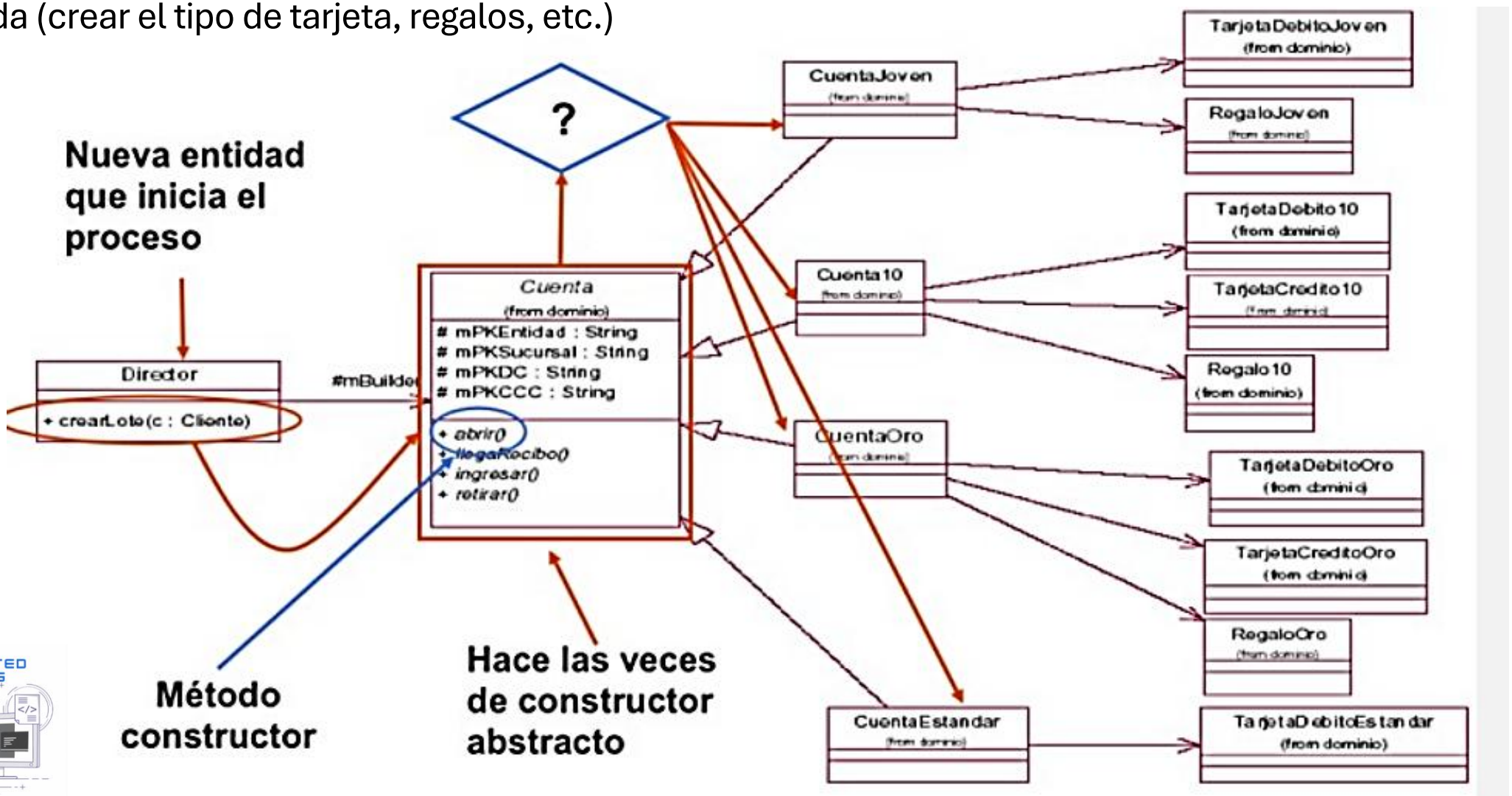
PATRONES DE DISEÑO – BUILDER - EJEMPLO

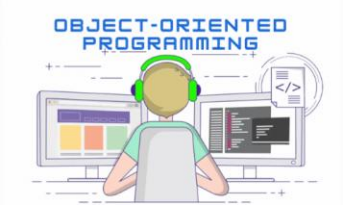
Este ejemplo del patrón **Builder** ilustra cómo se puede construir diferentes tipos de Pizza: Hawaiana y Picante para este caso.



PATRONES DE DISEÑO – BUILDER - EJERCICIO

Siguiendo con el ejemplo anterior de las cuentas bancarias, queremos construir objetos complejos que pueden ser de distinto tipo (los tipos de cuenta) utilizando para ello otros objetos y una secuencia de pasos definida (crear el tipo de tarjeta, regalos, etc.)

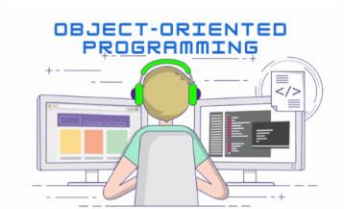




PATRONES DE DISEÑO – PROTOTYPE

El patrón ***Prototype*** es un patrón de diseño creacional que se utiliza para ***crear objetos duplicando un prototipo*** existente en lugar de crear objetos desde cero. Esto permite crear ***copias exactas*** de objetos sin conocer su tipo específico, lo que facilita la creación de nuevos objetos con características similares.

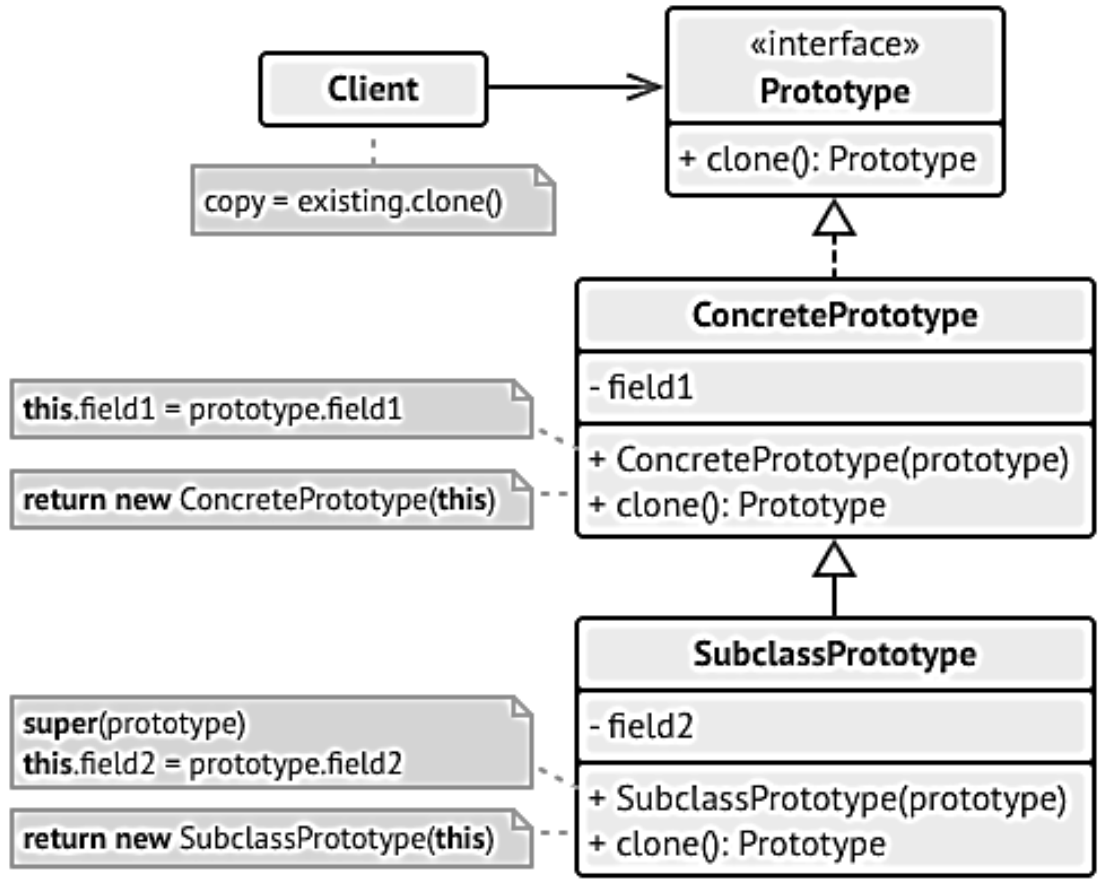
Se utiliza para crear nuevos objetos copiando instancias existentes en lugar de crear nuevas desde cero



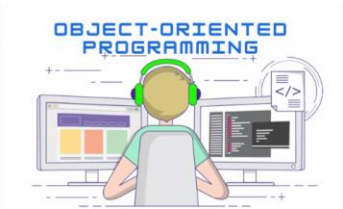
PATRONES DE DISEÑO – PROTOTYPE

3 El **Cliente** puede producir una copia de cualquier objeto que siga la interfaz del prototipo.

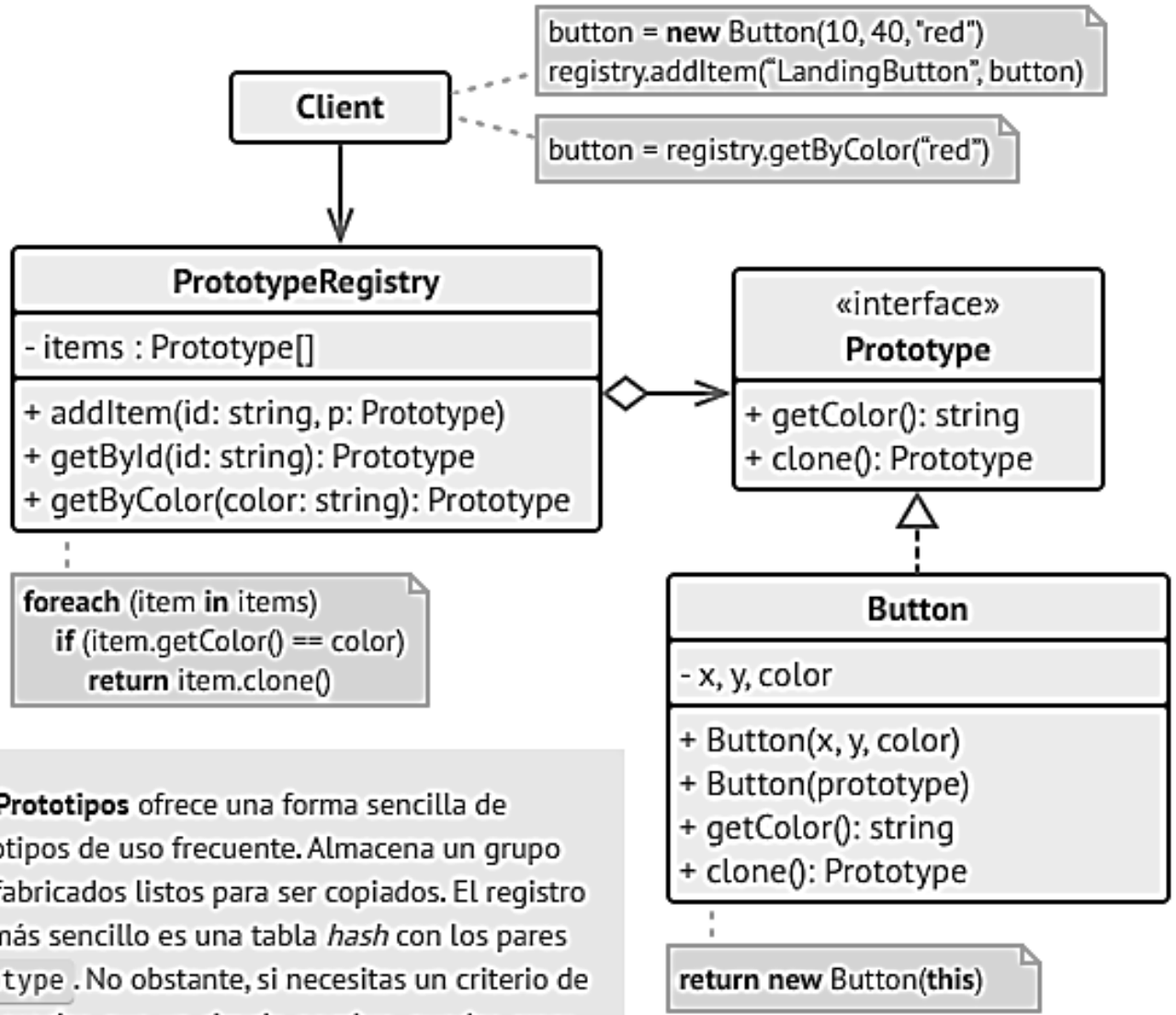
1 La interfaz **Prototipo** declara los métodos de clonación. En la mayoría de los casos, se trata de un único método `clonar`.



2 La clase **Prototipo Concreto** implementa el método de clonación. Además de copiar la información del objeto original al clon, este método también puede gestionar algunos casos extremos del proceso de clonación, como, por ejemplo, clonar objetos vinculados, deshacer dependencias recursivas, etc.



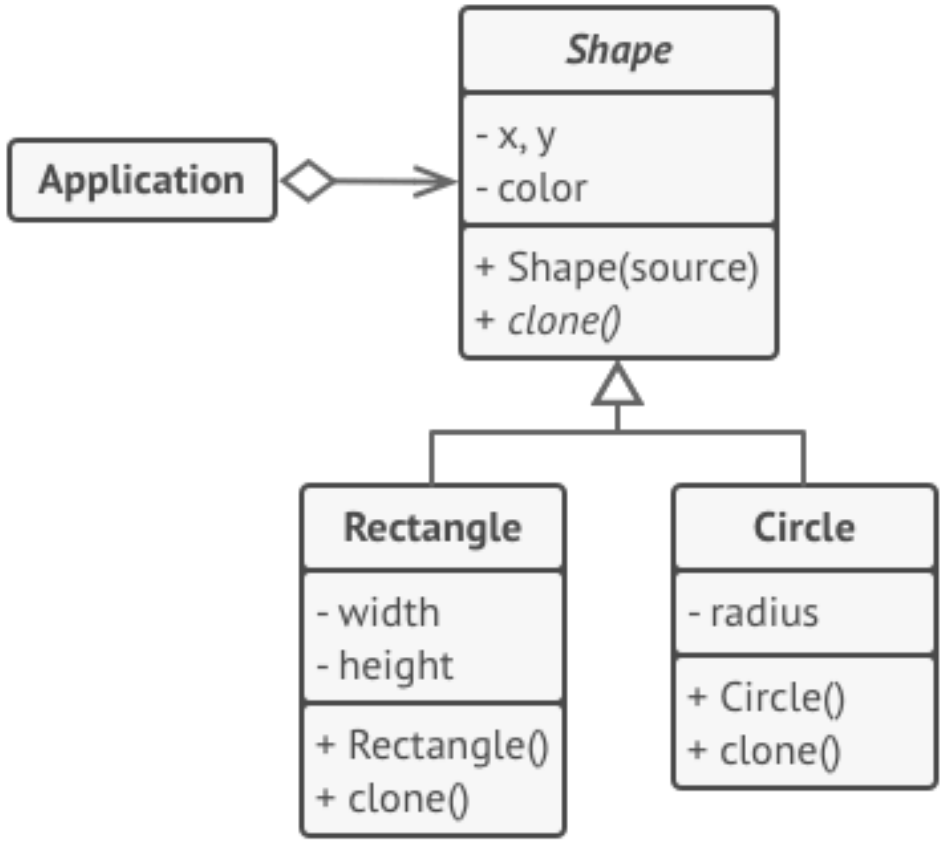
PATRONES DE DISEÑO – PROTOTYPE

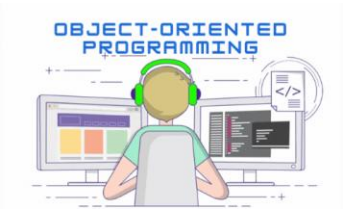


1 El **Registro de Prototipos** ofrece una forma sencilla de acceder a prototipos de uso frecuente. Almacena un grupo de objetos prefabricados listos para ser copiados. El registro de prototipos más sencillo es una tabla *hash* con los pares `name → prototype`. No obstante, si necesitas un criterio de búsqueda más preciso que un simple nombre, puedes crear una versión mucho más robusta del registro.

PATRONES DE DISEÑO – PROTOTYPE - EJEMPLO

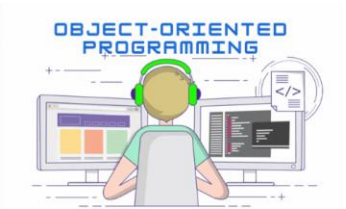
En este ejemplo, el patrón **Prototype** nos permite producir copias exactas de objetos geométricos sin acoplar el código a sus clases.





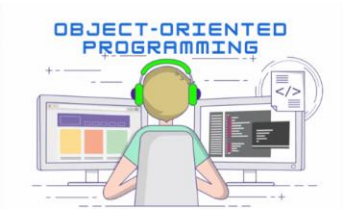
PATRONES DE DISEÑO – PROTOTYPE - EJEMPLO

En este ejemplo, el patrón **Prototype** nos permite producir copias exactas de objetos geométricos sin acoplar el código a sus clases.



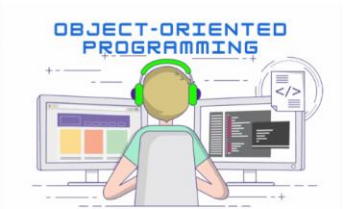
PATRONES DE DISEÑO – PROTOTYPE - EJEMPLO

```
public abstract class Shape {
    protected  int x;
    protected  int y;
    protected  String color;

    public Shape(int x, int y, String color) {
        this.x = x;
        this.y = y;
        this.color = color;
    }

    public Shape(Shape target) {
        if (target != null) {
            this.x = target.x;
            this.y = target.y;
            this.color = target.color;
        }
    }

    @Override
    abstract public Shape clone();
}
```



PATRONES DE DISEÑO – PROTOTYPE - EJEMPLO

```
public class Circulo extends Shape {
    private double radio;

    public Circulo(double radio, int x, int y, String color) {
        super(y, x, color);
        this.radio = radio;
    }

    public Circulo(Circulo target) {
        super(target);
        if (target != null) {
            this.radio = target.radio;
        }
    }

    @Override
    public Shape clone() {
        return (Circulo) new Circulo(this);
    }
}
```

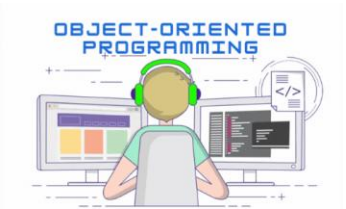


PATRONES DE DISEÑO – PROTOTYPE - EJEMPLO

```
public class Rectangulo extends Shape {
    private int ancho;
    private int alto;

    public Rectangulo(int ancho, int alto, int x, int y, String color) {
        super(x, y, color);
        this.ancho = ancho;
        this.alto = alto;
    }

    public Rectangulo(Rectangulo target) {
        super(target);
        if (target != null) {
            this.ancho = target.ancho;
            this.alto = target.alto;
        }
    }

    @Override
    public Shape clone() {
        return (Rectangulo) new Rectangulo (this);
    }
}
```

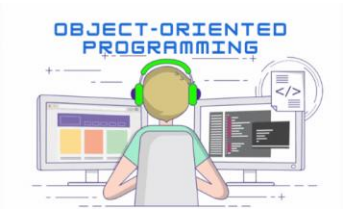
PATRONES DE DISEÑO – PROTOTYPE - EJEMPLO

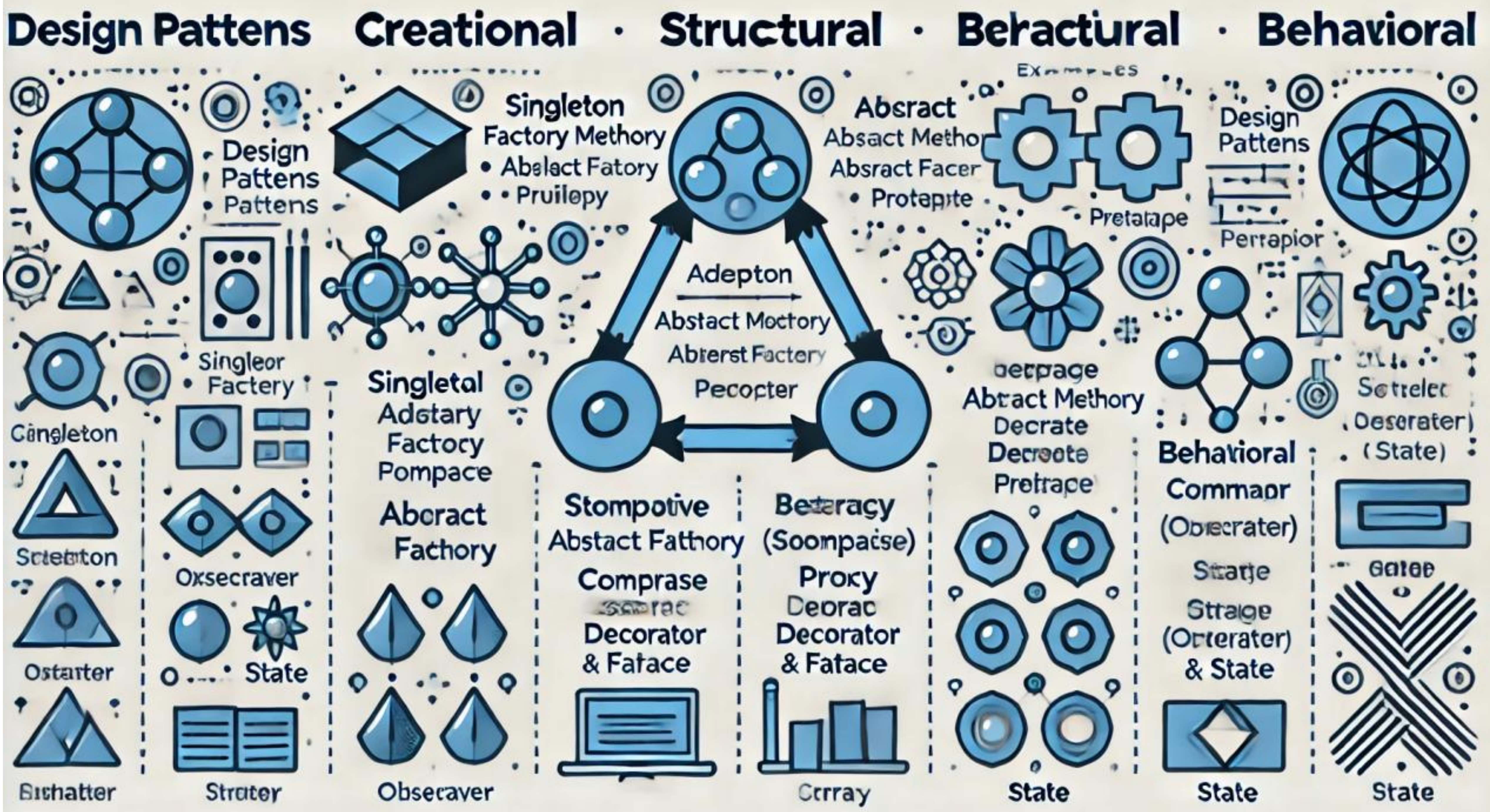
```
public class PruebaShape {
    public static void main(String[] args) {
        Shape circle = new Circulo(10, 10, 10, "Rojo");
        Shape circle2 = circle.clone();

        System.out.println(circle);
        System.out.println(circle2);
    }
}
```

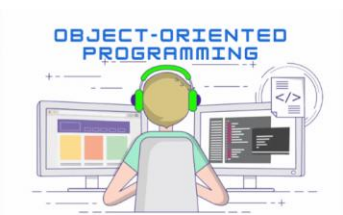


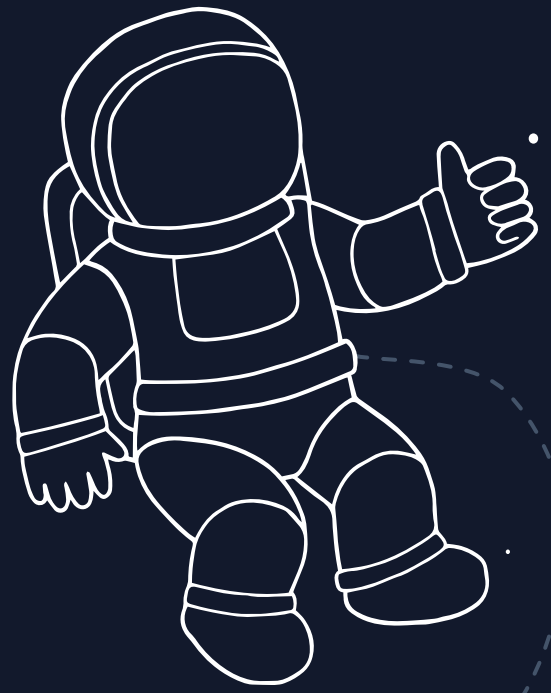
PATRONES DE DISEÑO – PROTOTYPE - EJERCICIO

En una aplicación construida en Java de almacén se instancia objetos de prendas de vestir: camisetas, pantalones, zapatos. Se quiere modificar el código para que permita crear copias de estos objetos.




Design Pattens
Creational
Structural
Beractural
Behavioral
Design
Pattens
Pattens
Singleton
Factory Methory
Ablact Fatory
Pruilopy
Abcract
Absact Metho
Absract Facer
Protepite
Pretatape
Design
Pattens
Perrapior
Adepton
Abstact Moctory
Abterst Factery
Pecocter
Singleor
Factery
Singletal
Adetary
Factory
Pompace
Abcract
Factory
Cingleton
Scteton
Oxsecraver
Ostarter
State
Bichatter
Strater
Obsecaver
Stompotive
Abstact Fathory
Comprase
Decorator
& Fatace
Betaracy
(Soompatse)
Proxy
Deorac
Decorator
& Fatace
Crray
Abtract Methory
Decrate
Decroote
Pretrape
State
Behavioral
Commaor
(Obecrater)
Strarje
Strtage
(Orterater)
& State
State
Sotrelet
Deserater)
(State)
State


OBJECT-ORIENTED
PROGRAMMING



Programa académico CAMPUS

Módulo JAVA