Resumen de patrones de diseño

Patron	Intencion	Aplicabilidad	Tips para saber cuando aplicarla
Adapter	"Convertir" la interfaz de una clase en otra que el cliente espera. Adapter permite que ciertas clases con interfaces incompatibles puedan trabajar en conjunto	Se usa cuando tenemos una interfaz incompatible, es decir cuando tenemos una clase distinta a la que el codigo espera.	No se puede modificar el codigo de ninguna otra clase Vamos a necesitar un Adapter de por medio.
Template Method	Define el esqueleto de un algoritmo en un metodo de una clase base. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar la estructura general.	Cuando hay varias clases que comparten la logica en general, pero cada una necesitapersonalizar algunos pasos en especifico. Se usa para evitar duplicacion de codigo en las subclases. Se aplican varios metodos comunes dentro de un mismo metodo en la superclase.	Template Method reune varios metodos dentro de uno mismo que todas las subclases lo implementa, pero los metodos internos pueden redefinirse Si encontramos codigo duplicado en las subclases, posiblemente se pueda aplicar template method. Se define el algoritmo en la superclase. Las subclases implementan pasos especificos.
Strategy	Permite encapsular cada algoritmo en un objeto y usarlos en forma intercambiable en tiempo de ejecucion segun se necesiten	No es deseable codificarlos todos en una clase y seleccionar cual utilizar por medio de sentencias condicionales.	Cuando tenemos varias estrategias/formas para realizar una misma tarea, podemos usar este patron.

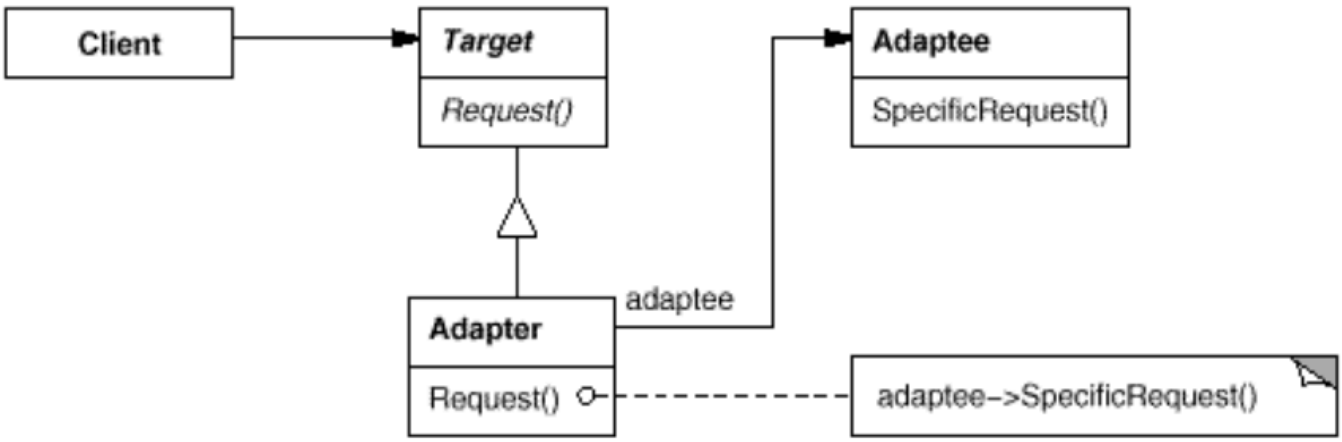
Patron	Intencion	Aplicabilidad	Tips para saber cuando aplicarla
	Permite cambiar en forma dinamica.	Es necesario cambiar el algoritmo en forma dinamica, en tiempo de ejecucion.	Ej : Tarea: Comprimir archivo Estrategias para comprimir: ZIP,RAR,TAR...etc Son distintas estrategias/formas de realizar una tarea (comprimir)
State	Permite que un objeto cambie su comportamiento cuando cambia su estado interno. Permitiendo cambiar de estado durante tiempo de ejecucion.	El comportamiento de un objeto, depende de su estado y puede cambiar durante su ciclo de vida. Se quiere cambiar el comportamiento de manera dinamica.	Si el objeto cambia de estado, o tiene un atributo estado, es una señal de que podemos aplicar el patron State, evitamos tener varios if o switch.
Composite	Componer objetos en estructuras de arbol para representar jerarquias parte-todo. Permite que los clientes traten a los objetos atomicos y a sus composiciones uniformemente.	Usar el patron cuando:: Se quiere presentar jerarquias parte-todo de objetos Se quiere que los objetos "clientes" puedan ignorar las diferencias entre composiciones y objetos individuales.	Podemos aplicar este patron cuando tenemos elementos que pueden contener otros elementos del mismo tipo.
Builder	Separa la construccion de un objeto complejo de su representacion de tal manera que el mismo proceso puede construir diferentes representaciones (implementaciones.)	Cuando tenemos muchos atributis opcionales. No queremos llenar el constructor con distintos parametros. Se quiere construir	Si hacés varios objetos similares con combinaciones distintas de campos. Si estás repitiendo código al crear instancias parecidas con `new`.

Patron	Intencion	Aplicabilidad	Tips para saber cuando aplicarla
		objetos de forma legible y flexible.	Si estás usando muchos setters y el código queda muy disperso o ilegible.

Adapter:

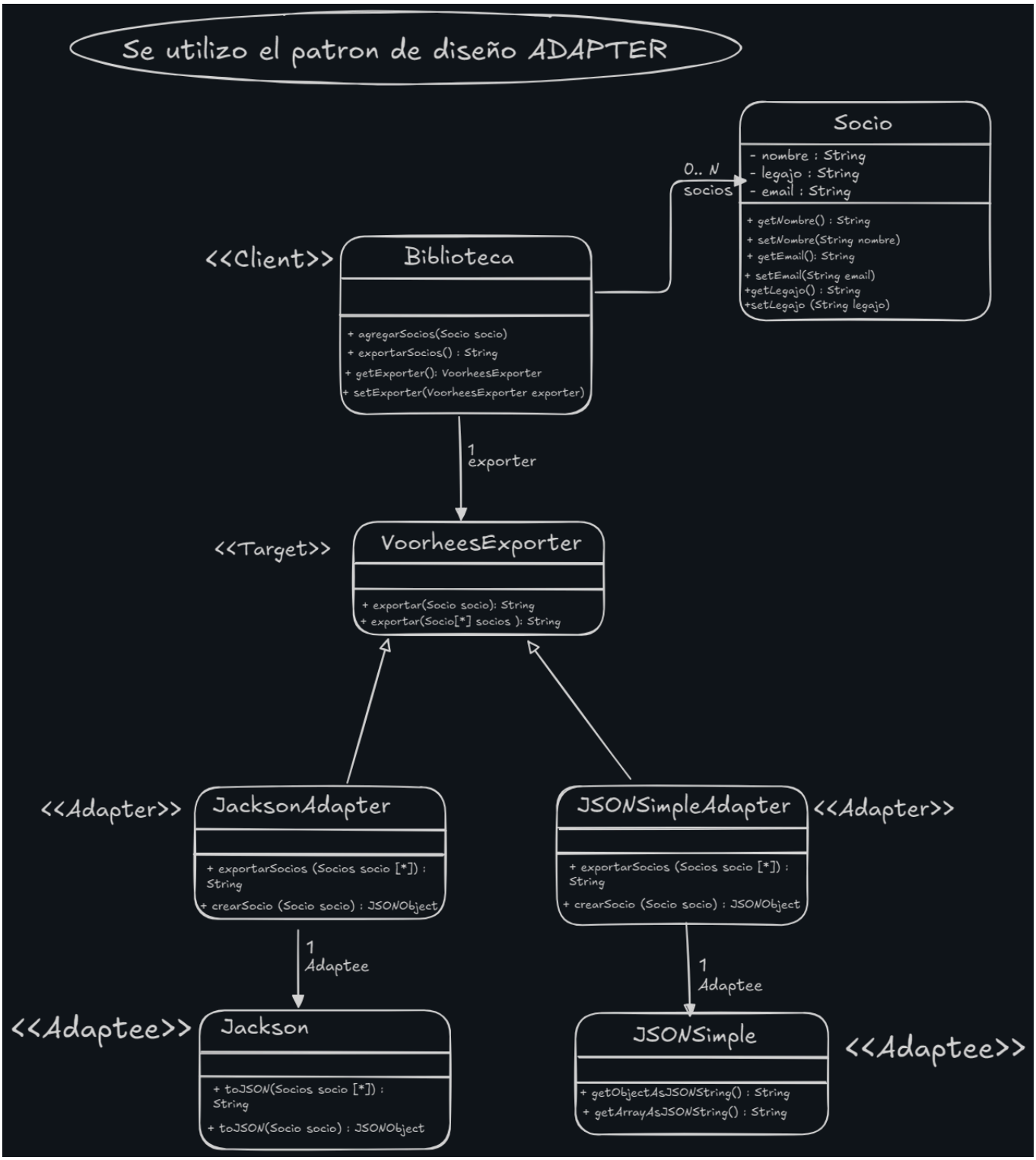
- Vamos a usar adapter cuando tengamos clases con interfaces incompatibles
- Se crea una clase Adapter, que es la encargada de ejecutar el metodo de la clase adaptada.
- Se usa cuando debemos aplicar un patron sin generar cambio en ninguna otra clase.

Estructura:



El cliente espera trabajar con un metodo llamado `Request()` (definido en la clase Targer) pero la clase que se quiere utilizar (Adaptee) no tiene ese metodo, tiene un metodo con otro nombre (`SpecificRequest()`).
De esta manera, para que el cliente pueda seguir usando Request() sin enterarse de como se llama a SpecificRequest() , se crea una clase Adapter.
El cliente usa `target.Request()` y **nunca se entera** de que por dentro se está llamando a `adaptee.SpecificRequest()`. Esta es la esencia del Adapter: **encapsular la incompatibilidad**.

UML de Adapter de un ejercicio de la practica



Participantes:

- Target: Define la interfaz especifica que usa el cliente
- Client: Colabora con objetos que satisfacen la interfaz de target.
- Adaptee: Define una interfaz que precisa ser adaptada
- Adapter: Adapta la interfaz del Adaptee a la interfaz del Target.
-

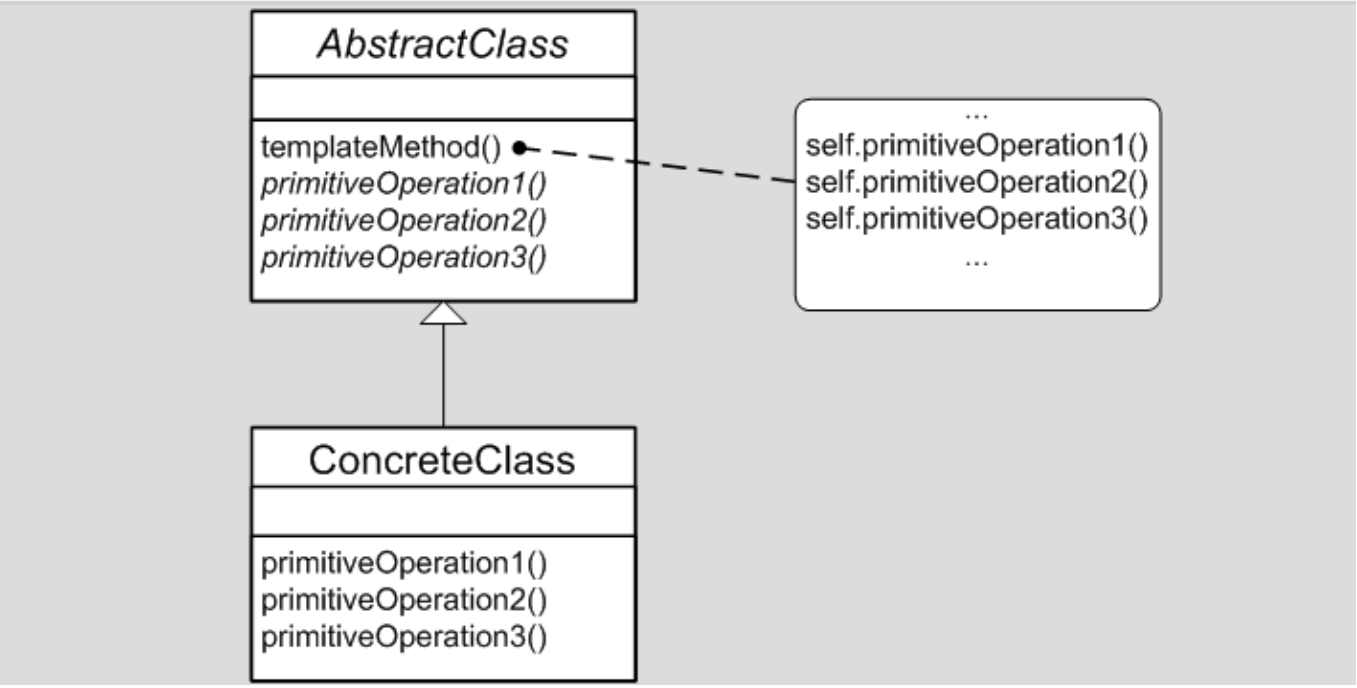
==========

Template Method:

- Vamos a usar template method cuando tenemos varias clases con comportamiento similar
- Queremos evitar duplicar codigo
- Se debe crear un metodo con la logica en comun en una superclase
- Las subclases pueden redefinir la logica de la superclase.

Template Method es un patrón que define un **método principal** (el *template*) que **reúne varios pasos** (otros métodos) dentro de el.
⚙️ **Ese método principal no se cambia**, y todas las subclases lo usan igual.
🛠️ **Pero los métodos internos (los pasos)** que componen el algoritmo **sí pueden ser redefinidos** por las subclases si lo necesitan.

Estructura:



Participantes:

- AbstractClass: Implementa un metodo que contiene el esqueleto del algoritmo.
- ConcreteClass: Implementa operaciones primitivas que llevan a cabo los pasos especificos del algoritmo.

Ejemplo de Template Method:

Nos dan un ejercicio en el que cada empleado puede calcular el sueldo (internamente) de una forma distinta. Pero el sueldo de cualquier empleado se compone de 3 elementos: `sueldo básico, adicional y descuento.`

Entonces, el `template method` seria `calcularSueldo()`, que dentro de el, tiene metodos internos como `calcularSueldoBasico`, `calcularAdicional`, `calcularDescuento`.
Permitiendo que cada subclase puede redefinir esos metodos internos.

	Temporario	Pasante	Planta
básico	$ 20.000 + cantidad de horas que trabajó * $ 300.	$20.000	$ 50.000
adicional	$5.000 si está casado $2.000 por cada hijo	$2.000 por examen que rindió	$5.000 si está casado $2.000 por cada hijo $2.000 por cada año de antigüedad
descuento	13% del sueldo básico 5% del sueldo adicional	13% del sueldo básico 5% del sueldo adicional	13% del sueldo básico 5% del sueldo adicional

> **Contexto:** Cada empleado debe calcular su sueldo, pero **según el tipo de empleado**, lo puede hacer de forma distinta.
> Lo que sí **sabemos con certeza** es que el método `calcularSueldo()` va a usar 3 pasos específicos para hacer ese cálculo.

> **El método `calcularSueldo()` es el Template Method**, ya que define la estructura fija del algoritmo:
>
> - `calcularBasico()`
> - `calcularAdicional()`
> - `calcularDescuento()`

> Las subclases pueden **redefinir** estos pasos para personalizar el comportamiento.

```
public abstract class Empleado {
    private static double montoBasico = 20000;

    public Double calcularSueldo ()
    {
        return this.calcularBasico() + this.calcularAdicional() - this.calcularDescuento();
    }


    protected double calcularDescuento()
    {
        return
                this.calcularPorcentaje(this.calcularBasico(), 0.13)
                + this.calcularPorcentaje(this.calcularAdicional(), 0.05) ;

    }

    protected double calcularBasico ()
    {
        return montoBasico;
    }

    protected abstract double calcularAdicional();


    private double calcularPorcentaje(Double total, Double porcentaje)
    {
        return total * porcentaje;
    }

}
```

==========

Strategy

Permite definir una familia de algoritmos, enca;psular cada uno y hacerlos intercambiables en tiempo de ejecucion. Evita el uso de condicionales como el if o switch.
Lo vamos a usar cuando:

- Hay muchas formas de realizar una misma tarea (por ejemplo distintas formas de comprimir un archivo)
- Puede cambiar dinamicamente en tiempo de ejecucion.
- Se quiere evitar el uso de if o switch

Como darnos cuenta cuando usarlo?

Cuando tenemos varios algoritmos que hacen lo mismo pero de forma distinta.

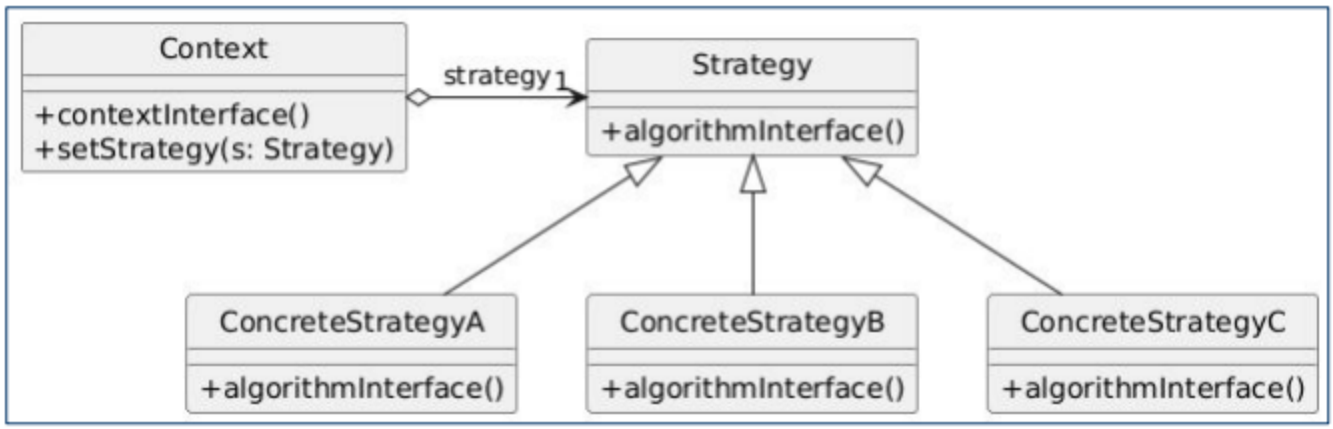
Ejemplos comunes de uso:

- Compresión de archivos (ZIP, RAR, 7z…)
- Métodos de pago (Tarjeta, PayPal, Cripto…)

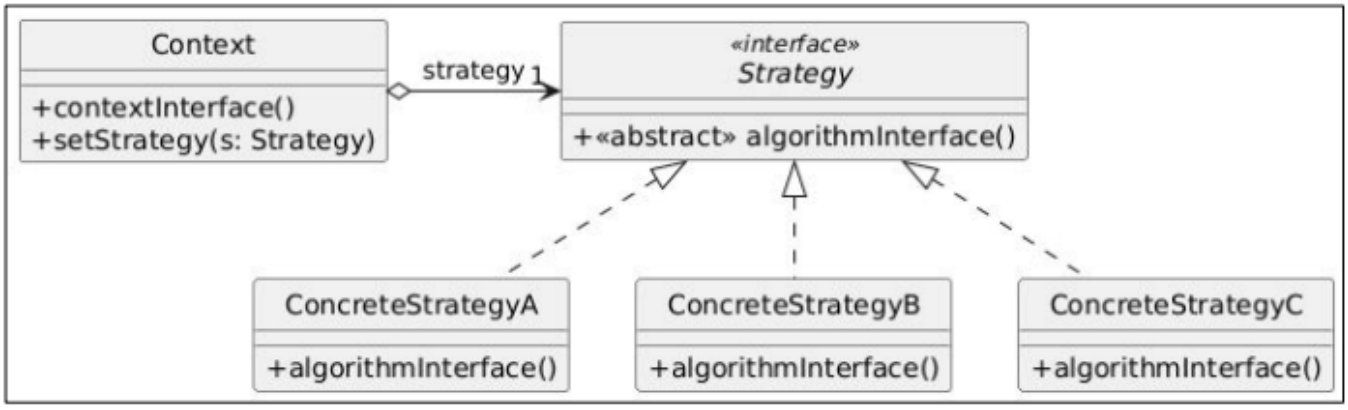
- Algoritmos de ordenamiento (BubbleSort, QuickSort…)
- Cálculo de descuentos (estudiantes, jubilados, promociones…)

Estructura:

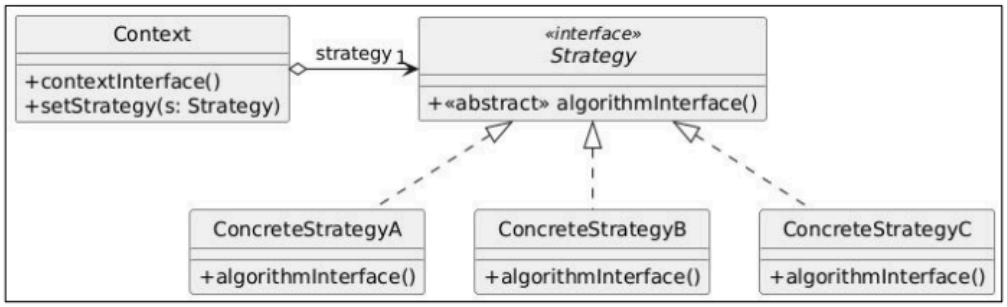
Primera forma de aplicarlo: (Strategy como clase abstracta)

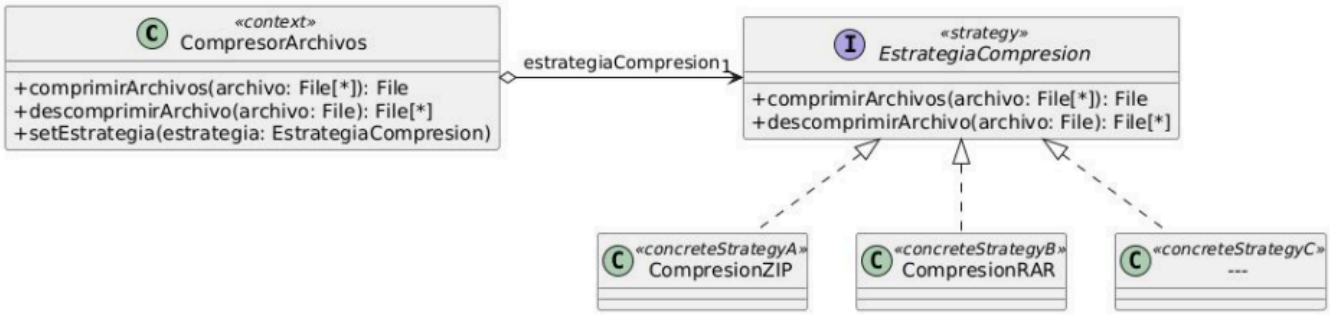


Segunda forma de aplicarlo: (Strategy como interfaz)



Ejemplo aplicandolo en un ejercicio con distintas formas de comprimir un archivo:



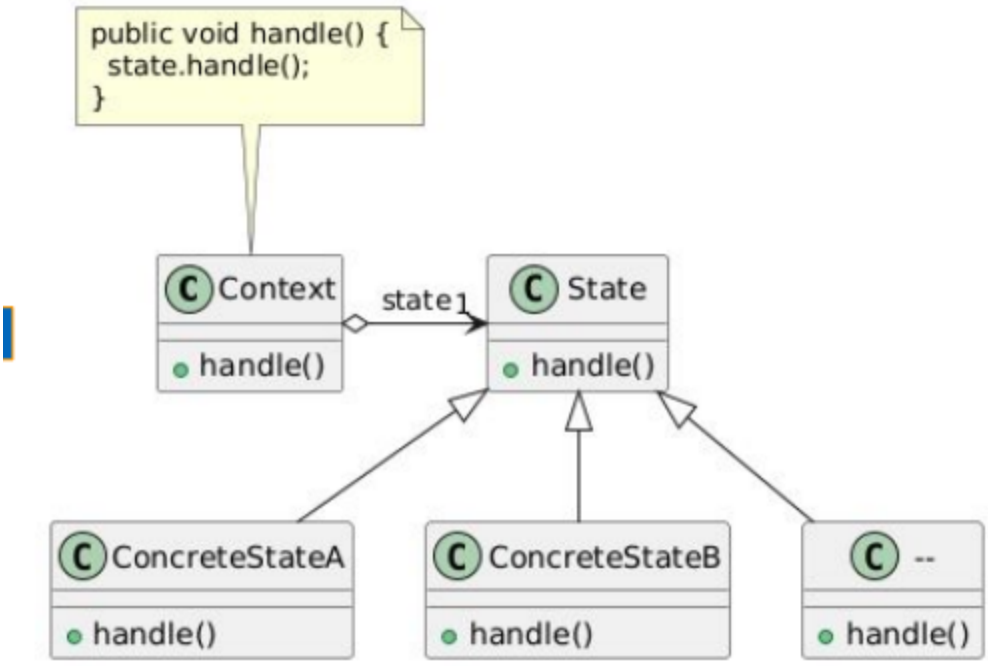


==========

State:

Permite modificar el comportamiento de un objeto cuando su estado interno se modifica.
El comportamiento de un objeto depende en el estado en el que se encuentre.
Generalmente podemos identificar el uso de este patron cuando tenemos distintos estados por los que pasa una clase. Y nuestra clase debe actuar en base al estado en el que se encuentre.

Estructura



Participantes:

- Context: Define la interfaz que conocen los clientes
  Mantiene la instancia de alguna clase de ConcreteState que define el estado corriente.
- State: Define la interfaz para encapsular el comportamiento de los estados del context
- ConcreteState (Subclases): Cada sublcase implementa el comportamiento respecto al estado especifico.

State o Strategy? .....

State

El comportamiento de un objeto depende del estado en el que se encuentre

El patrón State es útil para una clase que debe realizar transiciones entre estados fácilmente.

Strategy

Necesito uno de diferentes **algoritmos opcionales para realizar una misma tarea**

El patrón Strategy es útil para permitir que una clase delegue la ejecución de un algoritmo a una instancia de una familia de estrategias

State

Strategy

- En State, los diferentes estados:
  - son **internos al contexto**,
  - no los eligen las clases clientes
  - la transición se realiza entre los estados mismos

- En Strategy, l**as diferentes estrategias:**
  - son conocidas desde afuera del contexto, por las clases clientes del contexto.

- el Contexto del Strategy **debe contener un mensaje público para cambiar el ConcreteStrategy**.

State vs. Strategy

Resumen

- El **estado es privado del objeto**, ningún otro objeto sabe de él. vs.
≠ El Strategy suele setearse por el cliente, que debe conocer las posibles estrategias concretas.

- Cada State **puede definir muchos mensajes.** vs.
≠ Un Strategy suele tener un único mensaje público.

- Los states concretos **se conocen** entre sí. Saben a cual estado se debe pasar en respuesta a algún mensaje.
≠ Los strategies concretos no.

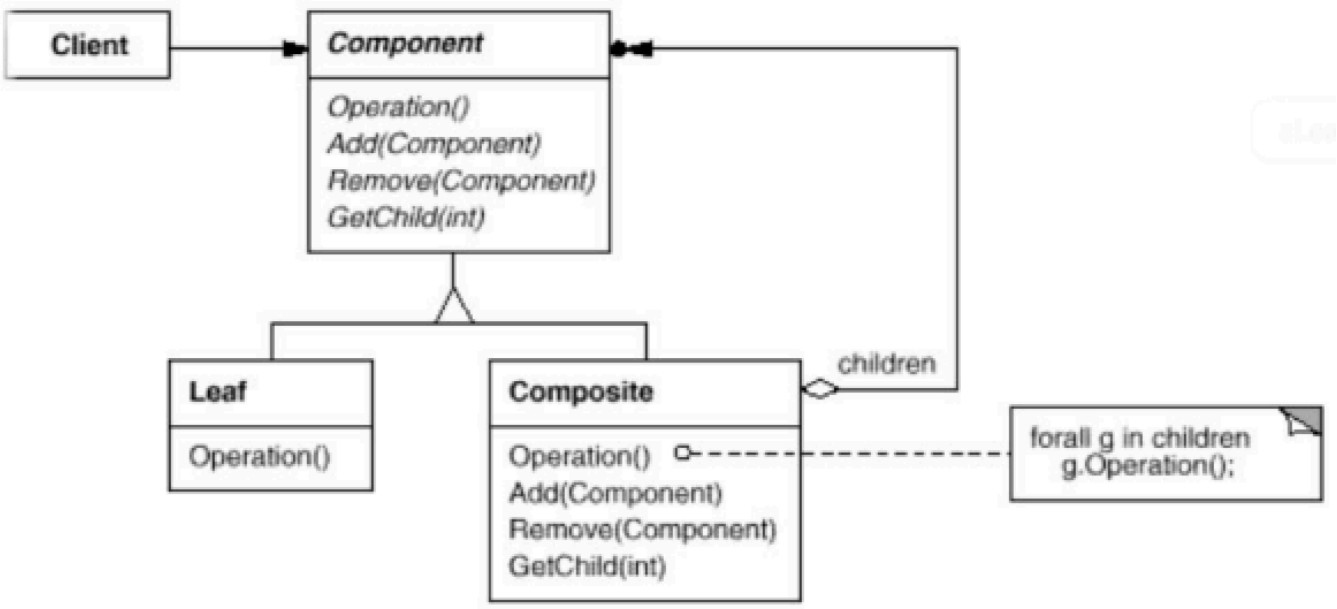
==========

Composite:

El patron composite es util cuando se trabaja con objetos individuales como los grupos de objetos deben tratarse de forma uniforme.
Podemos aplicar este patron cuando:

- Tengamos elementos que pueden contener otros elementos del mismo tipo.
- **Cuando querés tratar objetos individuales y grupos de forma uniforme**
  Por ejemplo, `Graphic` puede ser tanto un `Circle` como un `Group` de gráficos. Ambos implementan una misma interfaz y se usan igual.

Estructura:



Participantes:

- Component: Declara la interfaz para los objetos de la composicion.
- Leaf: Las hojas no tienen subarboles. Define el comportamiento de objetos primitivos en la composicion.
- Composite: Define el comportamiento para componentes complejos.
  Implementa operaciones para manejar subarboles.

Algunos usos y cosas a tener en cuenta

- Ejemplos comunes:
  - group/ungroup de elementos gráficos
  - Carpetas, archivos y link simbolicos (sistemas de archivos)
- Garantías en operaciones
  - Alquileres de inmuebles (garantía: Sueldo, Inmueble)
  - Prod. Financieros (collaterals: acciones, bonos, plazos fijos)
  - Prestamos prendarios (lo visto en el ejemplo)
- Agrupamiento
  - Combos (cajita feliz, talcos, útiles escolares)
  - Productos financieros (prestamos hipotecarios)
  - Pixels/Celdas en sensado remoto (imagenes satelitales)
  - Group/Ungroup de elementos gráficos
  - Paquetes de Alojamiento…
- **Es fundamental entender**
  - **el comportamiento de las hojas y los sub-árboles**
  - **Cuáles son las reglas de composición (configuración)**
    - Solapamiento, continuidad, circularidad, mutua exclusión, etc
  - **¿Qué objetos/métodos se encargan de crear las configuraciones?**

==========

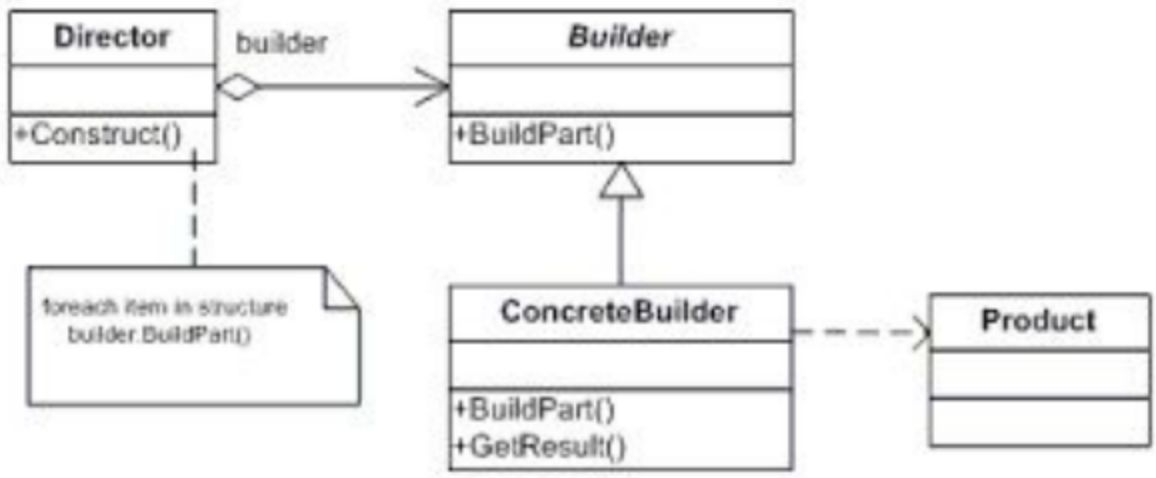
Builder:

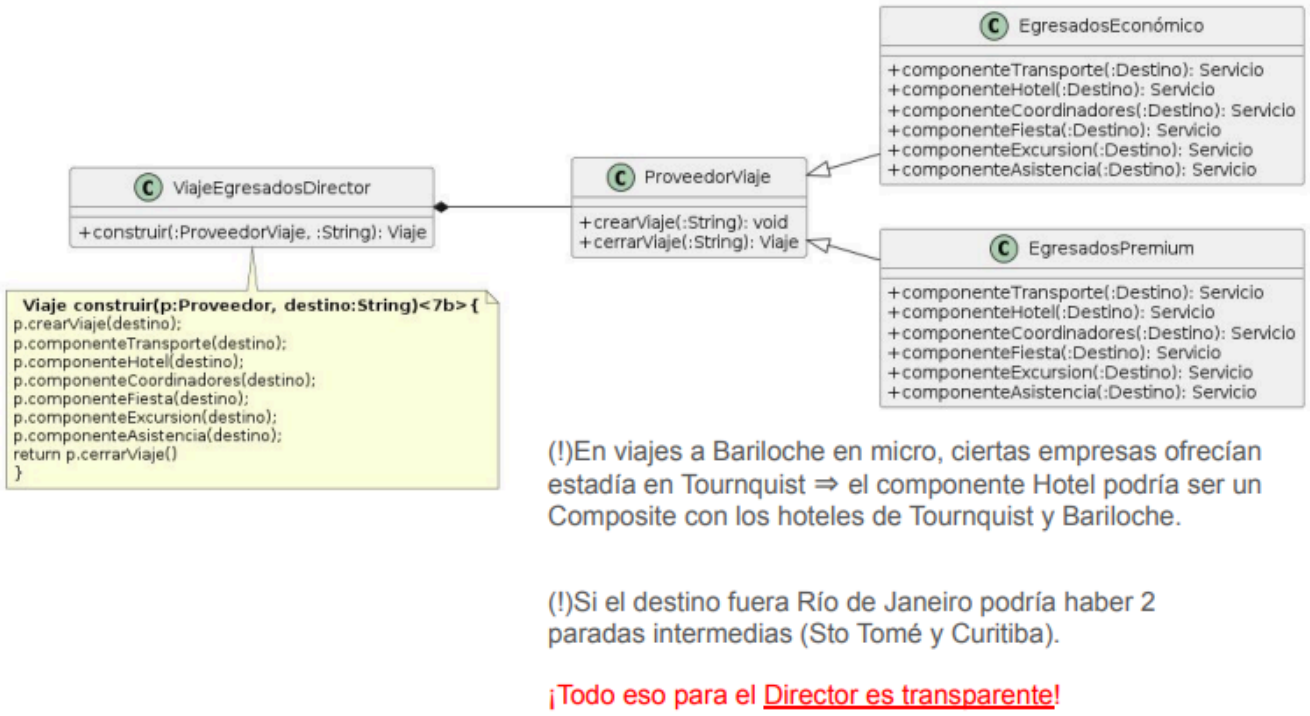
🎯 Intención (intention)

> Separar la construcción de un objeto complejo de su representación, de modo que el mismo proceso de construcción pueda crear diferentes representaciones.

O en palabras más simples:
👉 *Permite construir objetos paso a paso sin necesidad de pasar todos los datos en un solo constructor, y podés tener distintas versiones del objeto final sin cambiar cómo se construye.*

Estructura:





(!)En viajes a Bariloche en micro, ciertas empresas ofrecían estadía en Tournquist ⇒ el componente Hotel podría ser un Composite con los hoteles de Tournquist y Bariloche.

(!)Si el destino fuera Río de Janeiro podría haber 2 paradas intermedias (Sto Tomé y Curitiba).

¡Todo eso para el Director es transparente!

/

Participantes:

- Builder: Especifica una interface abstracta para crear partes de un Producto.
- Concrete Builder: Construye y ensambla partes del producto.
- Director: Conoce los pasos para construir el objeto.
- Product: Es el objeto complejo a ser construido.

Cuando vale la pena el builder?

Vale la pena Builder?

Si!

- Abstrae la construcción compleja de un objeto complejo
- Permite variar lo que se construye Director <-> Builder
- Da control sobre los pasos de contrucción

No!

- Requiere diseñar y implementar varios roles
- Cada tipo de producto requiere un ConcreteBuilder
- Builder suelen cambiar o son parsers de specs (> complejidad)

Este ejemplo de builder, no tiene en cuenta que hay un director. Si no que sirve para crear una clase con demasiados atributos.

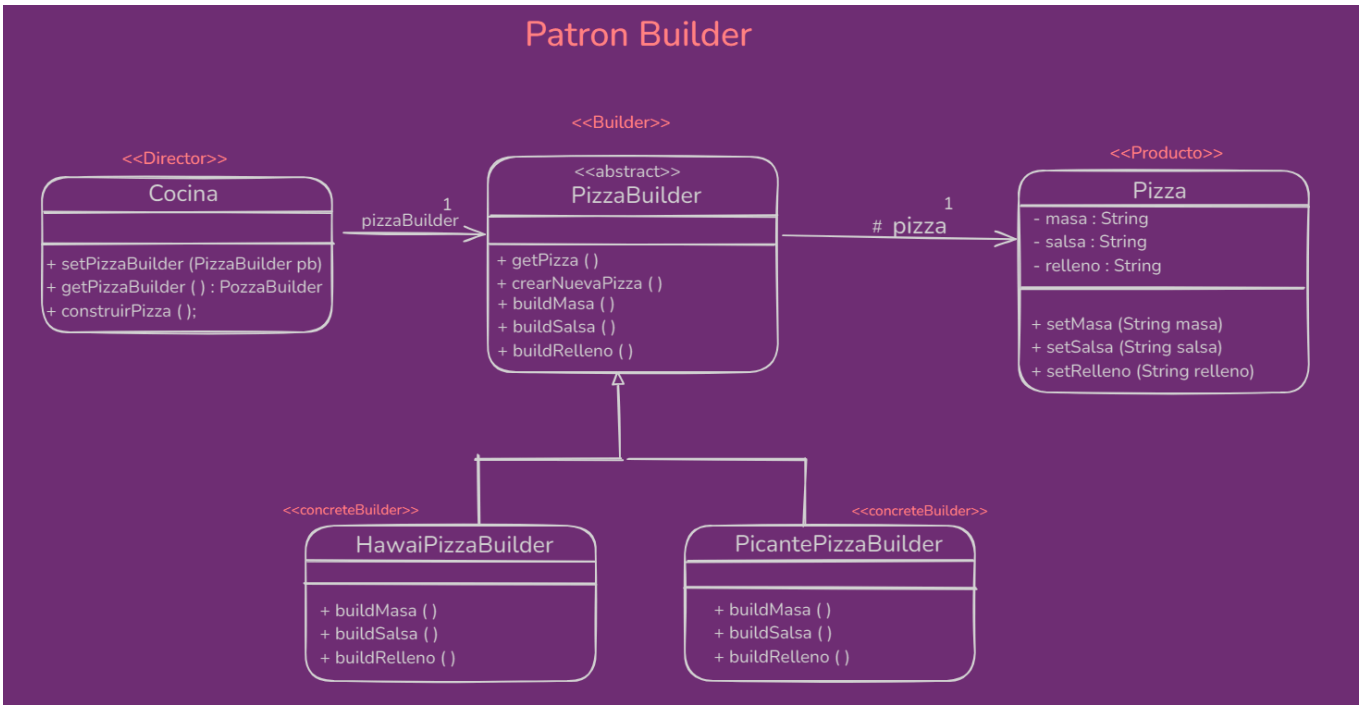
Se tienen dos clases, TsuarioDTO y Builder. Entonces por constructor de UsuarioDTO podemos mandarle directamente un Builder que lo va a entender

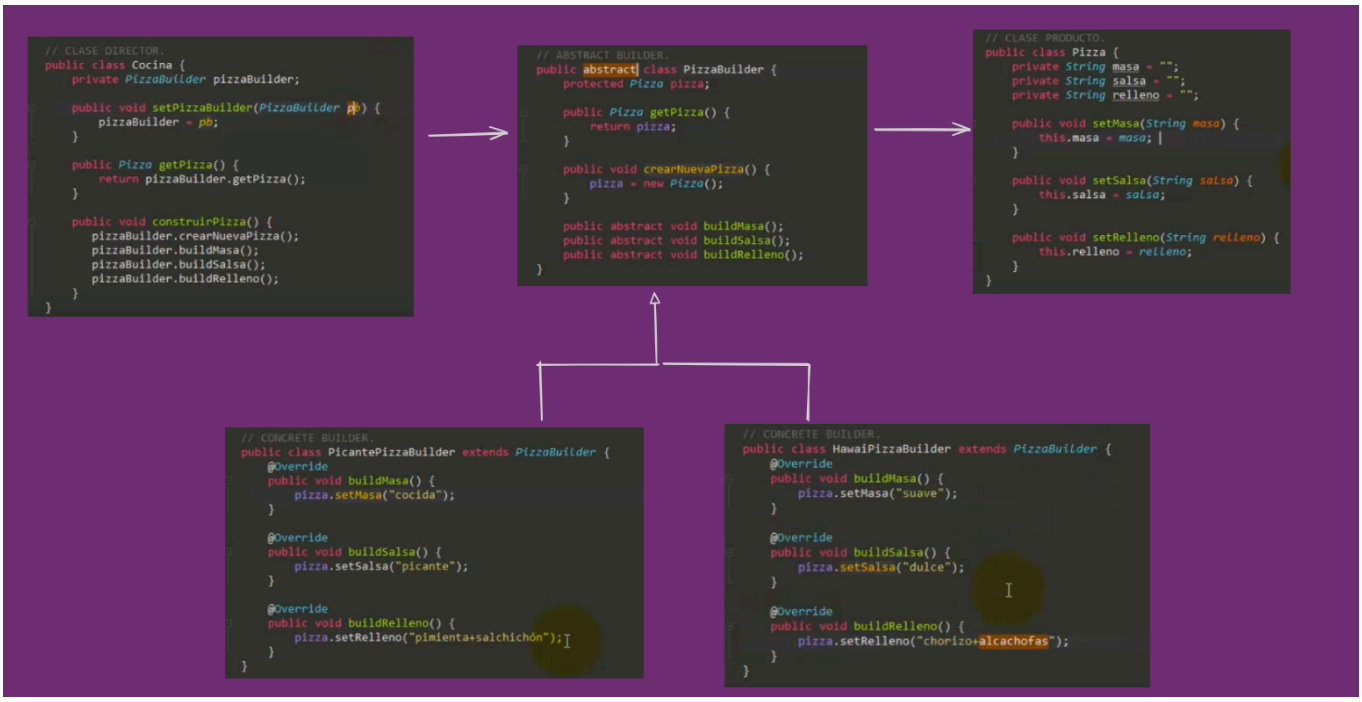
```
public class UsuarioDTO
{
    public UsuarioDTO (Builder builder)
    {
        this.nombre = builder.nombre;
        this.apellido = builder.apellido;
        this.dni = builder.dni;
        this.fechaNacimiento = builder.fechaNacimiento;
        this.email = builder.email;
        this.genero = builder.genero;
        this.estadoCivil = builder.estadoCivil;
    }

    public static class Builder
    {
        private String nombre;
        private String apellido;
        private String dni;
        private LocalDate fechaNacimiento;
        private String domicilio;
        private String email;
```

```
        private String genero;
        private String estadoCivil;
        public UsuarioDTO.Builder nombre (String nombre)
        {
            this.nombre = nombre;
            return this;
        }
        public UsuarioDTO.Builder  apellido (String apellido)
        {
            this.apellido = apellido;
            return this;
        }
        public UsuarioDTO.Builder  dni (String dni)
        {
            this.dni = dni;
            return this;
        }
        public UsuarioDTO.Builder  fechaNacimiento (LocalDate fecha)
        {
            this.fechaNacimiento = fecha;
            return this;
        }
        public UsuarioDTO.Builder  domicilio (String domicilio)
        {
            this.domicilio = domicilio;
            return this;
        }
        public UsuarioDTO.Builder  email (String email)
        {
            this.email = email;
            return this;
        }
        public UsuarioDTO.Builder  genero (String genero)
        {
            this.genero = genero;
            return this;
        }
        public UsuarioDTO.Builder  estadoCivil (String estadoCivil)
        {
            this.estadoCivil = estadoCivil;
            return this;
        }
        // creamos el metodo build.
        public UsuarioDTO build ()
        {
            return new UsuarioDTO(this);
```

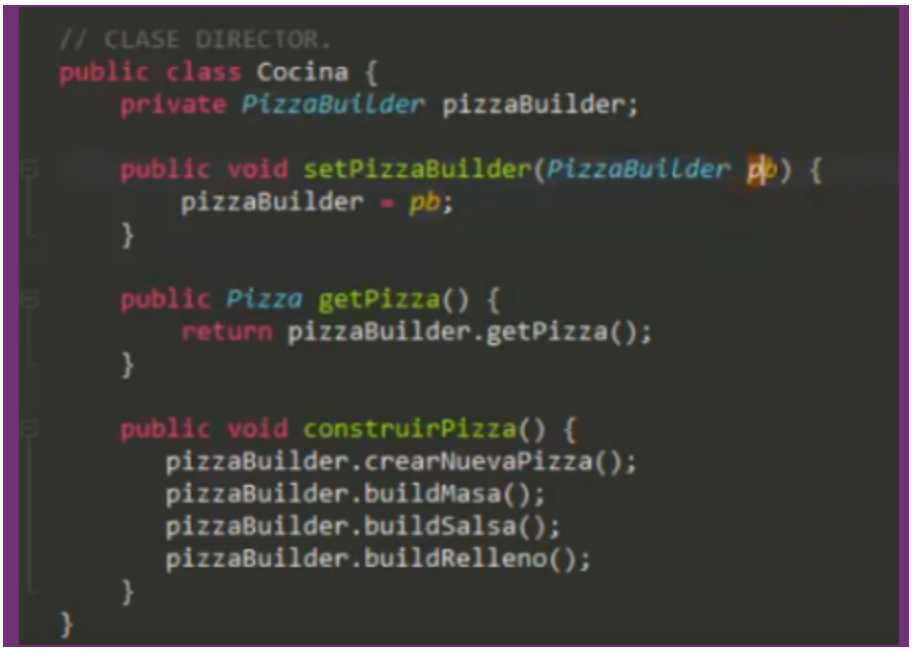
```
        }
    }// fin de clase Builder
}// fin de clase UsuarioDto

// Desde el main, instanciamos el UsuarioDTO a traves de un new Builder.
public class Main
{
    public static void main(String[] args) {
        UsuarioDTO usuario1 = new UsuarioDTO.Builder()
        .nombre("gian")
        .apellido("cardone")
        .dni("333222000")
        .fechaNacimiento(LocalDate.now())
        .email("EsteEsMiEmail@gmail.com")
        .estadoCivil("Soltero paaa")
        .genero("No binario")
        .build();
    }
}
```

Ejemplo de Builder con Director, Producto, Builder, ConcretClass





Por si no se llegan a ver bien las capturas de todas las clases:



```java
// ABSTRACT BUILDER.
public abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() {
        return pizza;
    }

    public void crearNuevaPizza() {
        pizza = new Pizza();
    }

    public abstract void buildMasa();
    public abstract void buildSalsa();
    public abstract void buildRelleno();
}
```

```java
// CLASE PRODUCTO.
public class Pizza {
    private String masa = "";
    private String salsa = "";
    private String relleno = "";

    public void setMasa(String masa) {
        this.masa = masa;
    }

    public void setSalsa(String salsa) {
        this.salsa = salsa;
    }

    public void setRelleno(String relleno) {
        this.relleno = relleno;
    }
}
```

```java
// CONCRETE BUILDER.
public class PicantePizzaBuilder extends PizzaBuilder {
    @Override
    public void buildMasa() {
        pizza.setMasa("cocida");
    }

    @Override
    public void buildSalsa() {
        pizza.setSalsa("picante");
    }

    @Override
    public void buildRelleno() {
        pizza.setRelleno("pimienta+salchichón");
    }
}
```

```java
// CONCRETE BUILDER.
public class HawaiPizzaBuilder extends PizzaBuilder {
    @Override
    public void buildMasa() {
        pizza.setMasa("suave");
    }

    @Override
    public void buildSalsa() {
        pizza.setSalsa("dulce");
    }

    @Override
    public void buildRelleno() {
        pizza.setRelleno("chorizo+alcachofas");
    }
}
```