

# Llista inicial : Patrones según tipo

		Propósito	
	Creacional	Estructural	Conductual
Class			
Scope	Abstract Factory ✓ Builder ✓ Prototype ✓ Singleton ✓ Factory Method ✓	Adapter (Object) ✓ Bridge X Composite ✓ Decorator ✓ Facade ✓ Flyweight X Proxy X	Chain of responsibility X Command X Iterator X Mediator X Memento X Observer ✓ State X Strategy ✓ Visitor X Template Method ✓ Filters de intercepción ✓
Object			

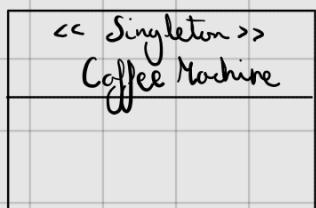
✓ : entra en el examen

✗ : No entra

## CREACIONALES

\* ) **SINGELTON** : Se usa para que una clase sólo pueda tener una instancia

UML

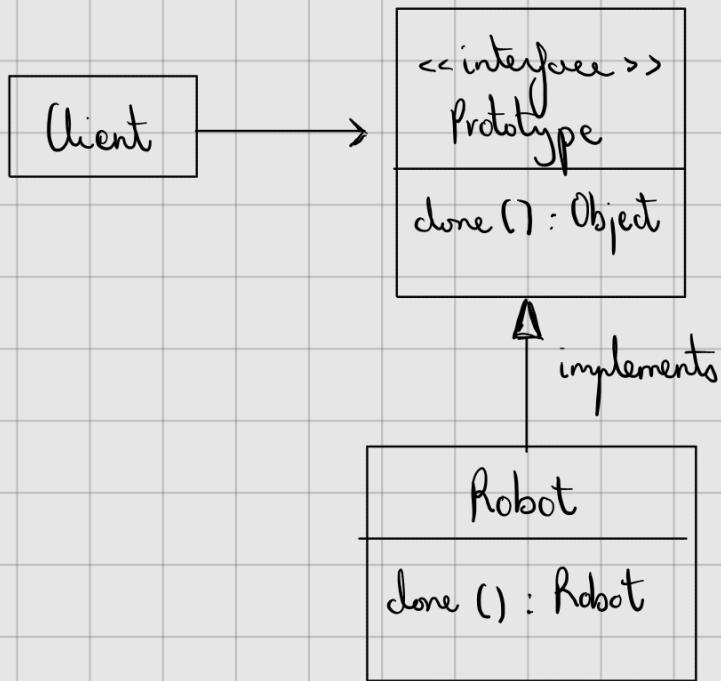


CÓDIGO

```
public class CoffeeMachine {  
    private static CoffeeMachine instance;  
  
    public static CoffeeMachine getInstance {  
        if (instance == null) {  
            instance = new CoffeeMachine();  
        }  
        return instance;  
    }  
}
```

\* ) **PROTOTYPE** : Se usa para hacer una copia profunda (devolviendo una copia del objeto) sin tener que copiar uno a uno los valores de sus atributos.

### UML



### CÓDIGO

```

public interface Prototype {
    public Prototype clone();
}
  
```

```

public class Robot implements Prototype {
    ...
    public Robot (Robot robot) {
        this.componentes = robot.componentes;
    }
  
```

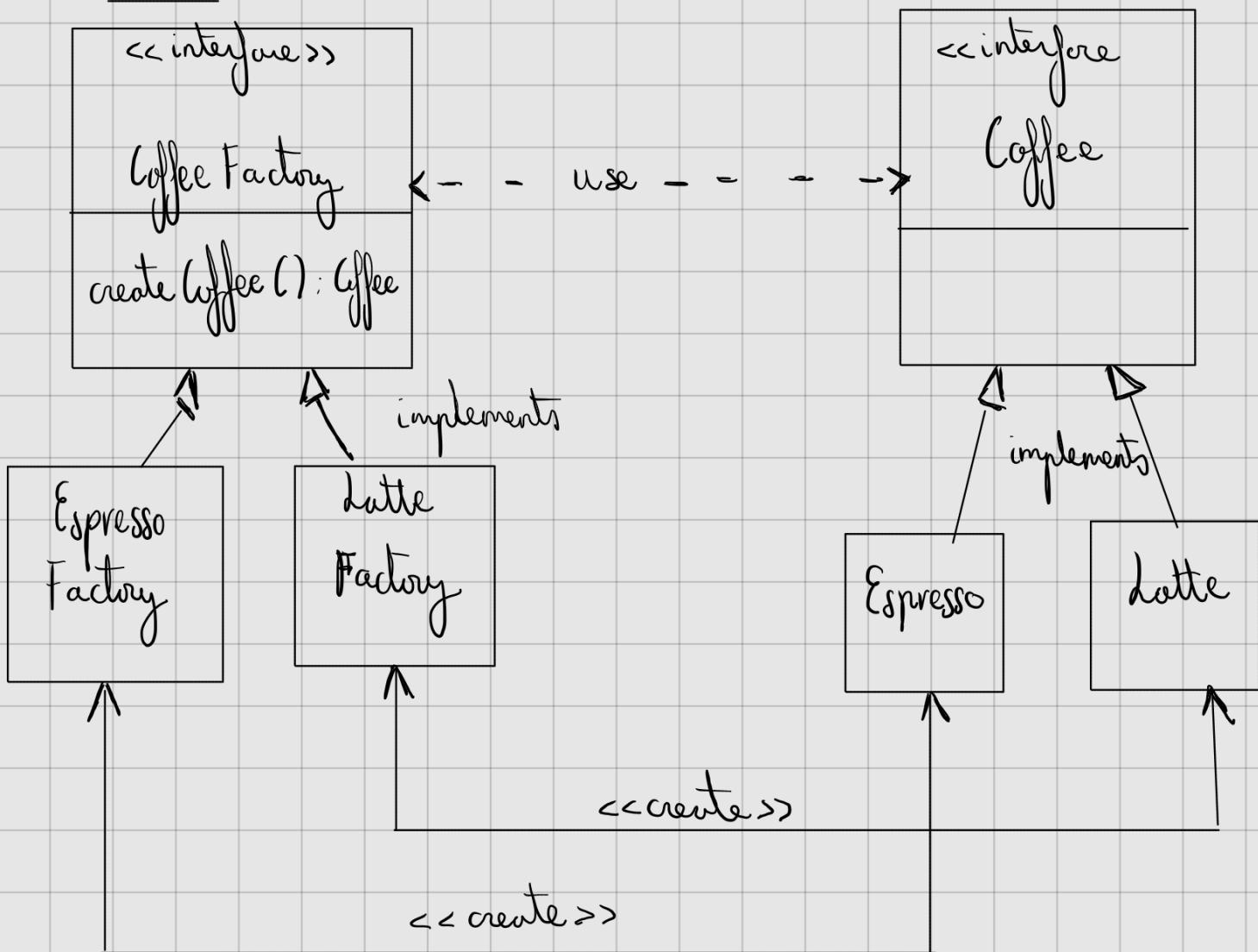
@Override

```
public Prototype clone () {  
    return new Robot (this);  
}
```

}

\* **ABSTRACT FACTORY**: Este patrón es útil cuando un sistema debe ser independiente de cómo se crean, componen y representan sus productos.

UML



## CÓDIGO

```
public interface CoffeeFactory {  
    ...  
    public Coffee createCoffee();  
}
```

```
public class EspressoFactory implements CoffeeFactory {  
    ...
```

```
    public Coffee createCoffee() {  
        return new Espresso();  
    }  
}
```

→ crea coffee dentro  
de fábrica

```
public interface Coffee {  
    ...  
}
```

```
public class Espresso implements Coffee {  
    ...  
}
```

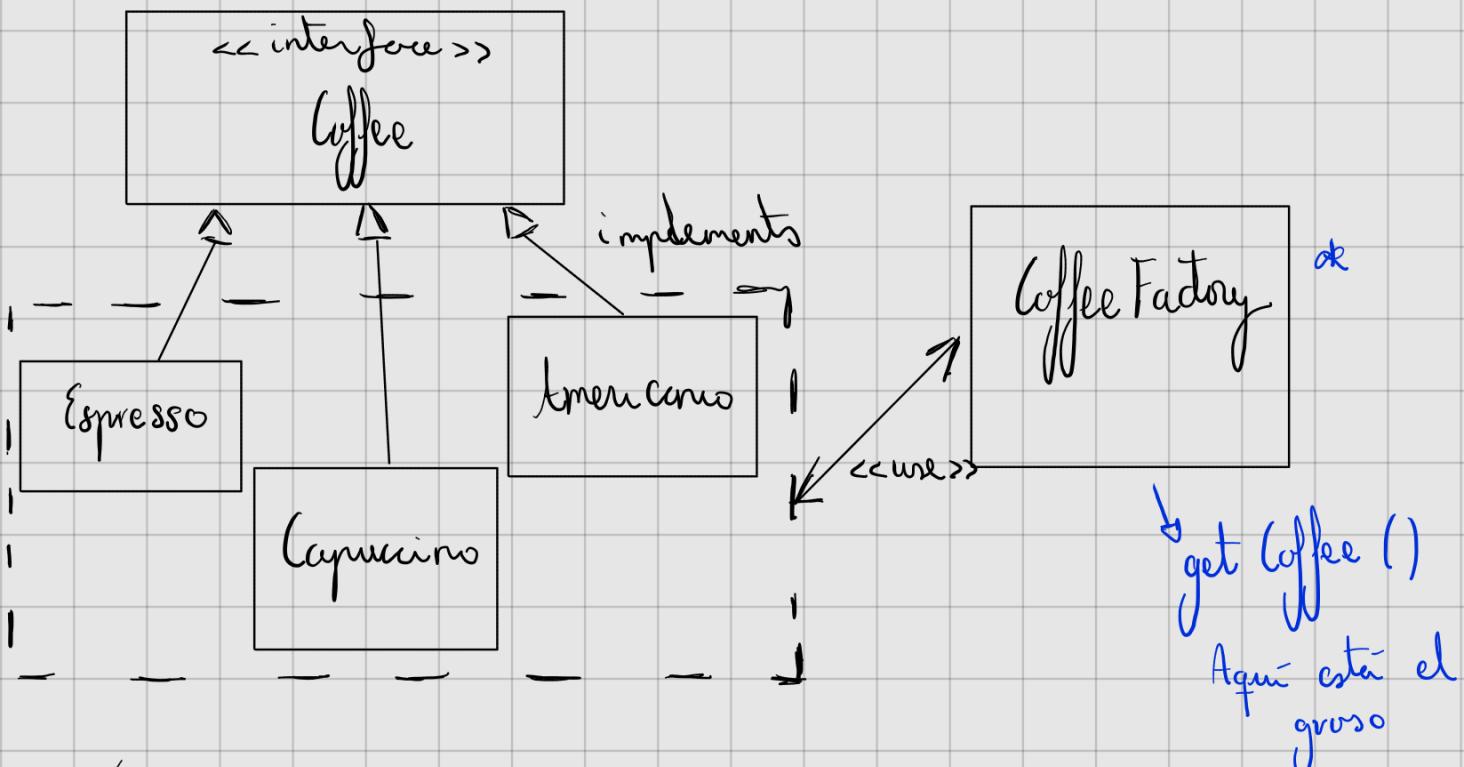
```
{
```

\* do mismo para el latte \*

\* **FACTORY METHOD**: Este patrón delega la responsabilidad de la instanciación de un objeto a los subclases.

Proporciona una interfaz para crear objetos en una superclase.

UML



CÓDIGO

```
public interface Coffee {  
    ...  
    public void steps(Coffee());  
}
```

```
public class Espresso implements Coffee {  
    ...  
    public void steps(Coffee()) {
```

```
System.out.println ("Pasó para Espresso\n");  
}  
}
```

\* De harán los demás tipos

```
class CoffeeFactory {
```

```
    . . .  
    public Coffee getCoffee ( String type ) {
```

```
        switch ( type ) {
```

```
            case "Espresso":
```

```
                return new Espresso();
```

```
            break;
```

```
            case "Capuccino":
```

```
                return new Capuccino();
```

```
            break;
```

```
            case "Americano":
```

```
                return new Americano();
```

```
            break;
```

```
            default:
```

throw new UnsupportedOperationException  
("No existe este tipo");

```
        break;
```

```
public class Main {  
    public static void main (String [] args) {  
        ...  
        // Para crear
```

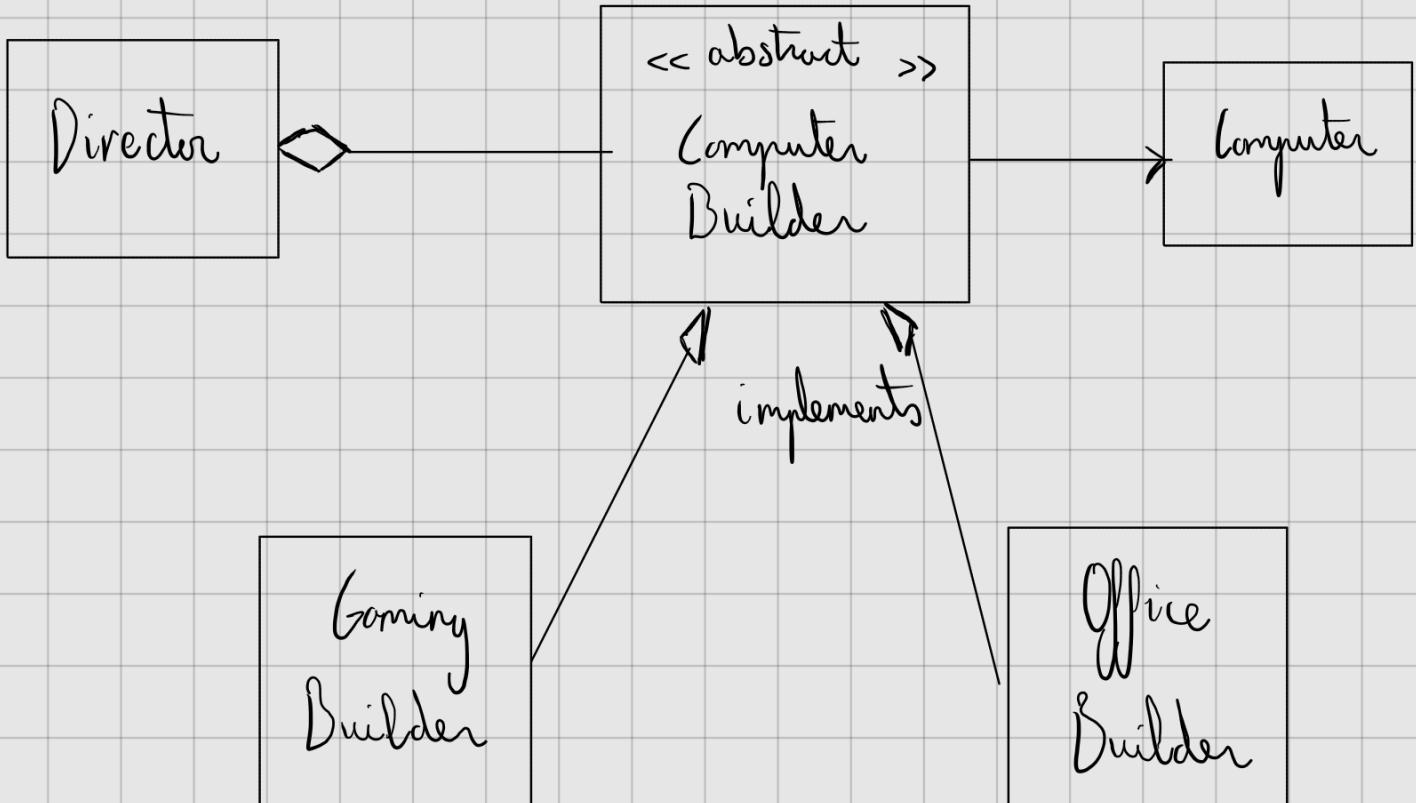
CoffeeFactory cf = new CoffeeFactory();

```
Coffee esp = cf.getcoffee ("Espresso");
```

1 }

\* **BUILDER**: Para manejar la construcción de objetos complejos. Separa la construcción de un objeto complejo de su representación. Oculta detalles de la construcción del objeto.

UML



CÓDIGO

```

public class Director {
    public Builder builder;
    public Director (Builder builder) {
        this.builder = builder;
    }
    public void buildComputer () {
        this.builder.createNewComputer ();
    }
}
  
```

```
this.builder.addComponents();
if (this.builder instanceof GamingBuilder) {
    this.builder.addGraphicCard();
}
}

}

public abstract class ComputerBuilder {
    public Computer compu;
    public ComputerBuilder () {
        this.compu = null;
    }
    public void createNewComputer () {
        this.compu = new Computer();
    }
    public abstract void addComponents();
    public abstract void addGraphicCard();
}
```

```
public class GamingBuilder extends ComputerBuilder {  
    public GamingBuilder () {  
        super ();  
    }  
  
    public void addComponents () {  
        .. - ..  
    }  
  
    public void addGraphicCard () {  
        .. .. ~ ..  
    }  
}  
  
// Lo mismo con el de Oficina
```

```
public class Computer {  
    public String components = null;  
    public String graphicCards = null;  
    public Computer () {}  
}
```

public class Main {

    public static void main (String [] args) {

        GamingBuilder gb = new GamingBuilder ();

        Director d = new Director (gb);

        director.createNewComputer ();

}

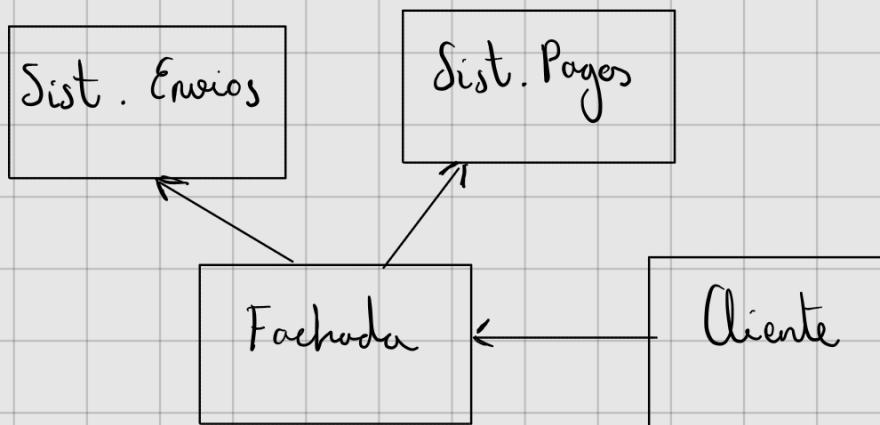
}

# ESTRUCTURALES

o FACADE: Proporciona una interfaz simplificada a un conjunto de otras interfaces en un subsistema.

DESACOPLAMOS la implementación de un sistema de sus clientes y REDUCIMOS COMPLEJIDAD.

## VML



## CÓDIGO

```
public class SistemaPagos {  
    public void procesarPago ( int tarjeta, double monto ) {  
        . . .  
    }  
}  
  
public class SistemaEnvíos {  
    public void procesarEnvío ( String dirección ) {  
        . . .  
    }  
}
```

```
public class Fachada {
    public SistemaPagos sp = null;
    public SistemaEnvios se = null;

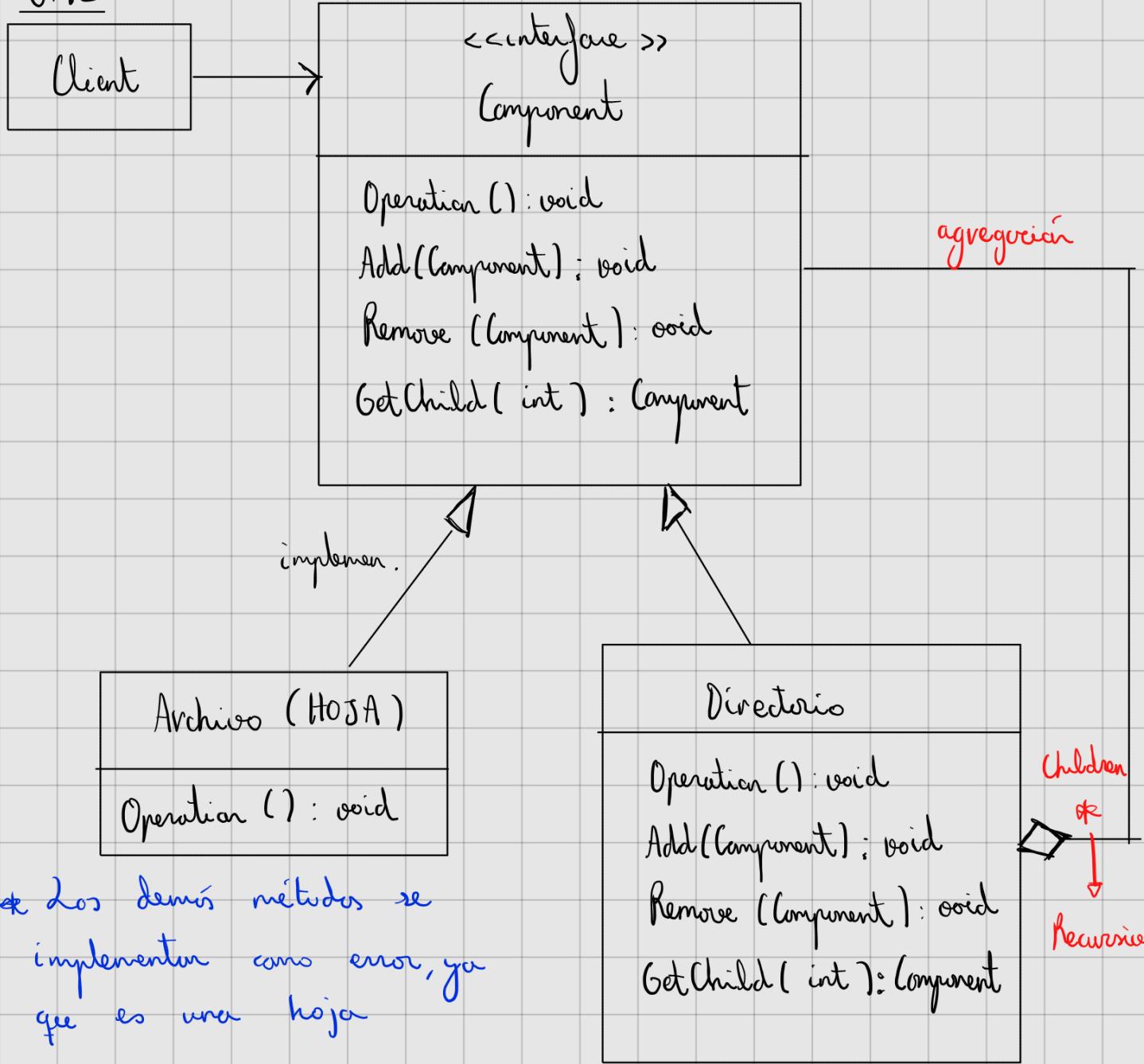
    public Fachada () {
        sp = new SistemaPagos ();
        se = new SistemaEnvios ();
    }

    public void procesarOrden () {
        ...
        this.sp.procesarPago ();
        ...
        this.se.procesarEnvio ();
        ...
    }
}
```

// Toda la gestión la haces desde Fachada

\* **COMPOSITE (COMPOSICIÓN)** : Define un objeto compuesto de forma recursiva

UML



\* Los demás métodos se implementan como error, ya que es una hoja

CÓDIGO

```

public interface Component {
    public void addChild (Component campo);
    public void removeChild (Component campo);
    public Component getChild (int num);
    public void operation ();
}
    
```

```
public class Archivo implements Component {
    public String nombre;
    public Archivo ( String n ) {
        this.nombre = n;
    }
    public void operation () {
        // Implementación
    }
    public void addChild ( Component campo ) {
        throw new UnsupportedOperationException ("No implementado");
    }
    // Lo mismo para getChild () o removeChild
}
```

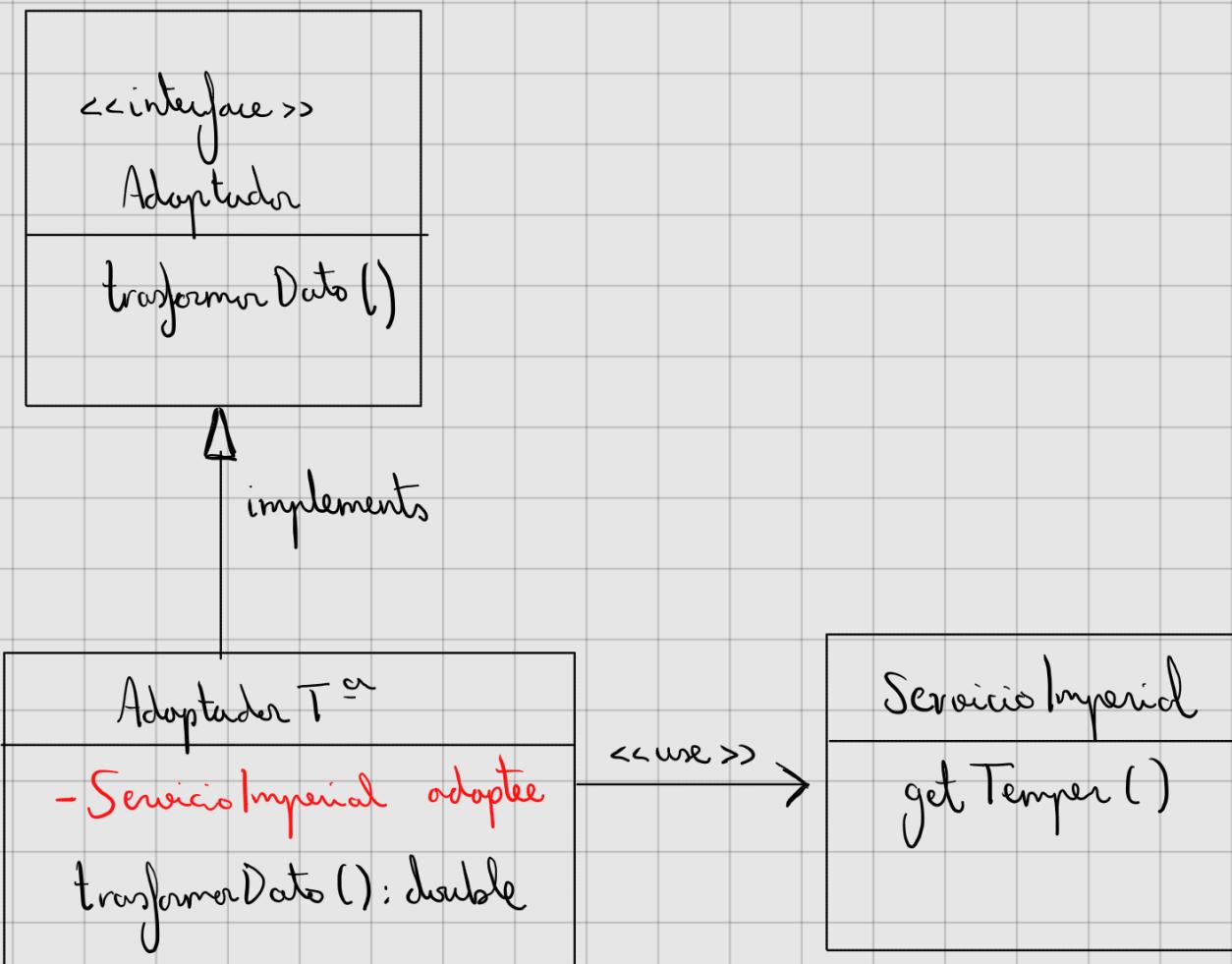
```
public class Directorio implements Component {
    public ArrayList < Component > = new ArrayList < > (); // HJS
    public String nombre;
    public Directorio ( String n ) {
        this.nombre = n;
    }
    public void operation () {
        // Implementación
    }
}
```

// Aquí sí se implementan los métodos add, remove y  
get ...

}

```
public class Main {  
    public static void main (String [] args) {  
        Directorio root = new Directorio ("root");  
        Directorio home = new Directorio ("home");  
        Archivo juan = new Archivo ("juan.txt");  
  
        root.addChild (home);  
        home.addChild (juan);  
  
        root.operation ();  
    }  
}
```

\* ADAPTER (ADAPTADOR): Permite que clases con interfaces incompatibles trabajen juntas, convirtiendo una en otra que el cliente espera.



### CÓDIGO

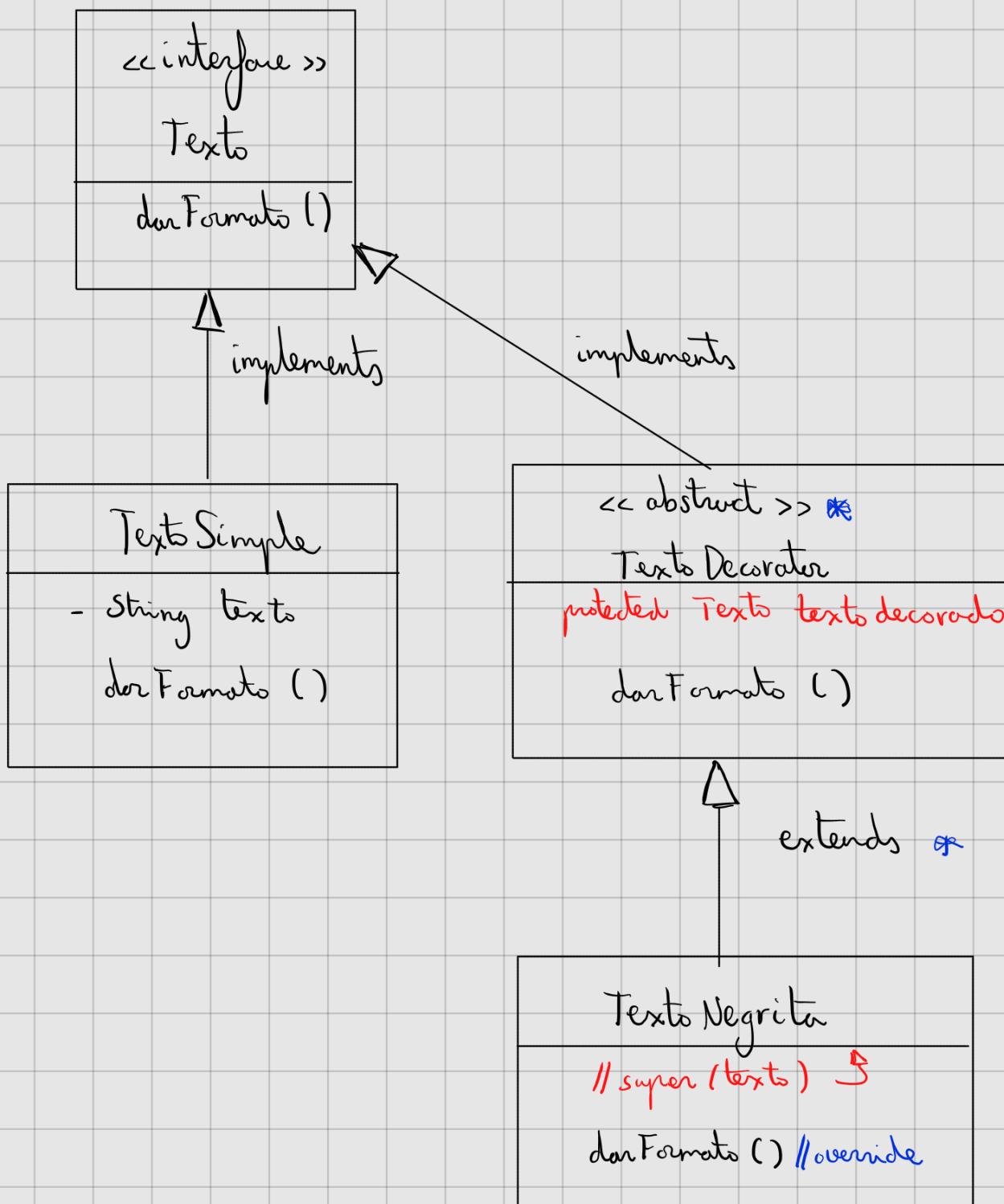
```

public interface Adaptador {
    public double transformarDatos();
}

public class AdaptadorT implements Adaptador {
    private ServicioImperial adaptee;
    public AdaptadorT (ServicioImperial a) {
        this.adaptee = a;
    }
}
  
```

```
    }  
    public double transformarData () {  
        return ( this.adapted.getT° - 32 ) * ( 5 / 9 );  
    }  
}  
  
public class ServicioImperial {  
    private double temp;  
    // constructor, getters, setters y get T°  
}  
  
public class Main {  
    public static void main ( String [] args ) {  
        double temp = 32.00;  
        ServicioImperial adaptedee = new ServicioImperial ( temp );  
        AdaptadorT° adaptador = new AdaptadorT° ( adaptedee );  
        t° Celsius = adaptador.transformarData ();  
    }  
}
```

☞ **DECORATOR (Decorador)** : Permite añadir nuevas funcionalidades a objetos de manera dinámica encediéndoles en clases decoradoras.



## CÓDIGO

```
public interface Texto {  
    String darFormato();  
}  
  
class TextoSimple implements Texto {  
    private String texto;  
    public TextoSimple (String t) {  
        this.texto = t;  
    }  
    public String darFormato () {  
        return texto;  
    }  
}
```

```
abstract class TextoDecorador implements Texto {  
    protected Texto textoDecorado;  
    public TextoDecorador (Texto texto) {  
        this.textoDecorado = texto;  
    }  
    public String darFormato () {  
        return textoDecorado.darFormato();  
    }  
}
```

```
class TextoNegrita extends TextoDecorador {
```

```
    public TextoNegrita (Texto texto) {  
        super (texto);  
    }
```

@Override

```
    public String darFormato () {  
        return "<b>" + super.darFormato () + "</b>";
```

// Esta clase juega mucho con el "super", ya que

// este patrón hereda funcionalidad o características a  
// una clase previamente definida.

```
public class Main {
```

```
    public void static main (String [] args) {
```

```
        TextoSencillo t = new TextoSencillo ("hola");
```

```
        TextoNegrita t2 = new TextoNegrita (t);
```

```
        System.out.println (t.darFormato()); // "hola"
```

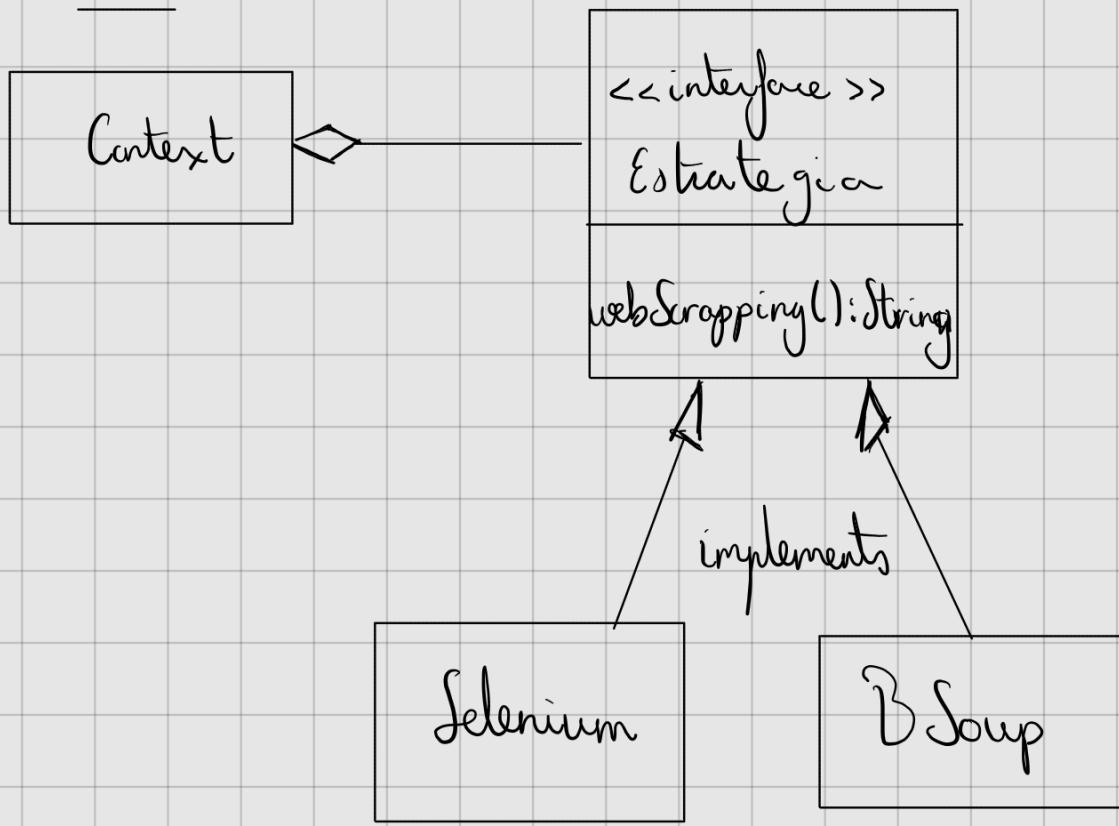
```
        System.out.println (t2.darFormato()); // "hola"
```

```
}
```

## CONDUCTUALES

\* STRATEGY : Define una familia de algoritmos, los encapsula y los hace intercambiables. Permite que el algoritmo varíe independientemente de los clientes que lo utilizan.

### VML



### CÓDIGO

```
public class Context {
    private Estrategia est;
    public Context (Estrategia est) {
        this.est = est;
    }
}
```

```
public String webScraping () {  
    return this.est. webScraping ();  
}  
}
```

```
public interface Estrategia {  
    public String webScraping ();  
}
```

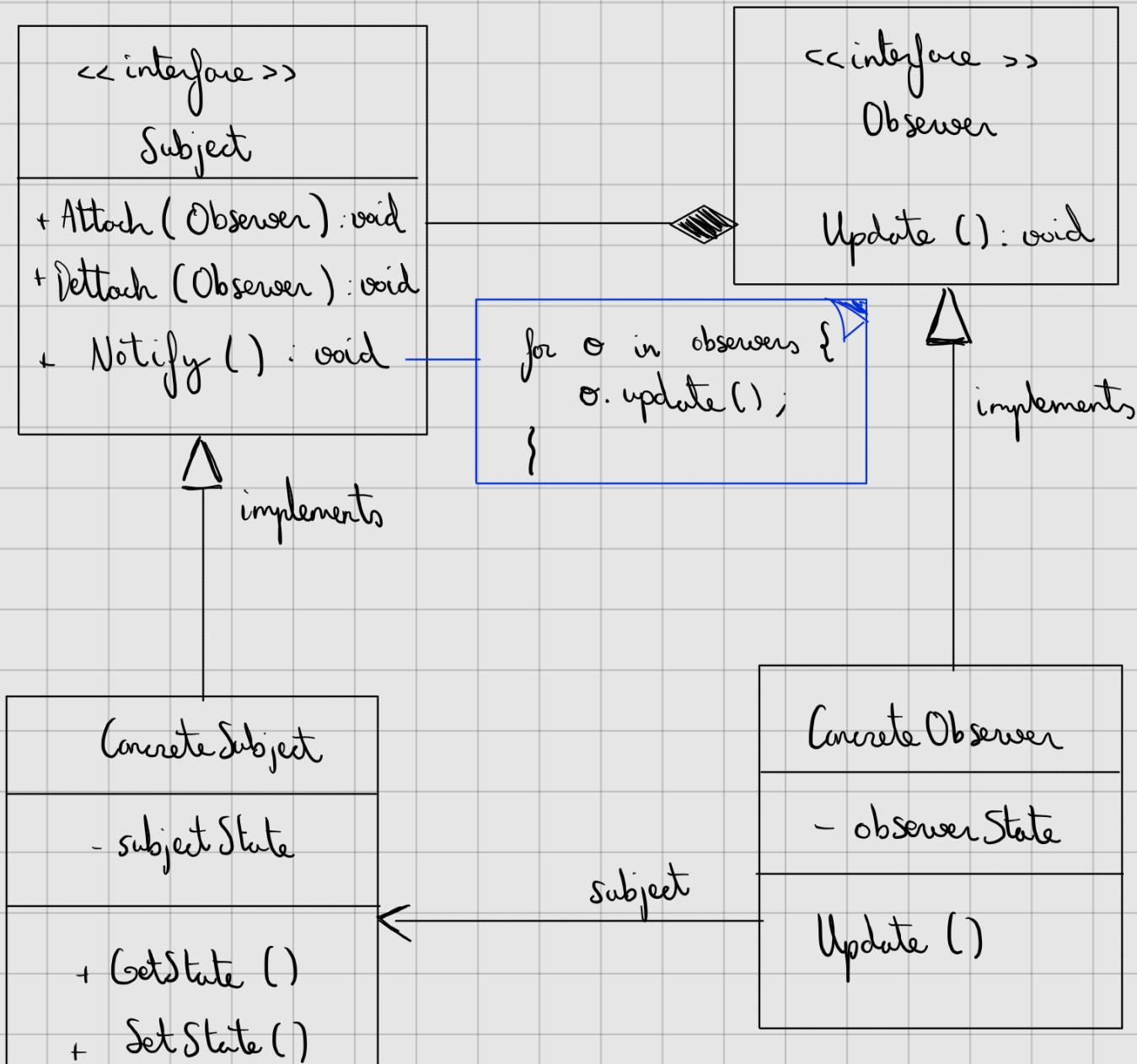
```
public class Selenium implements Estrategia {  
    . . .  
    public String webScraping () {  
        . . .  
    }  
}
```

// Lo mismo para BeautifulSoup

```
public class Main {  
    public static void main (String [] args) {  
        Context con = new Context (new Selenium ());  
        String va = con.webScraping (); // Usar Selenium  
    }  
}
```

\* OBSERVER: Para sistemas flexibles y de bajo acoplamiento  
 Permite que múltiples "observer" respondan a los cambios en un "subject". (Comunicación indirecta).

## UML



## CÓDIGO

// implementaremos los interfaces Subject y Observer

```

public class ConcreteSubject implements Subject {
    private ArrayList<Observers> observers;
    private int estado; // Estado 0 - Nada
    // Estado 1 - Se alertan los observadores
  
```

```
public ConcreteSubject () {
    observers = new ArrayList<>();
    estado = 0;
}

public int getState () {
    return this.estado;
}

public void setState ( int newEstado ) {
    this.estado = newEstado;
}

public void Attach ( Observer o ) {
    this.observers.add ( o );
}

public void Remove ( Observer o ) {
    this.observers.remove ( o );
}

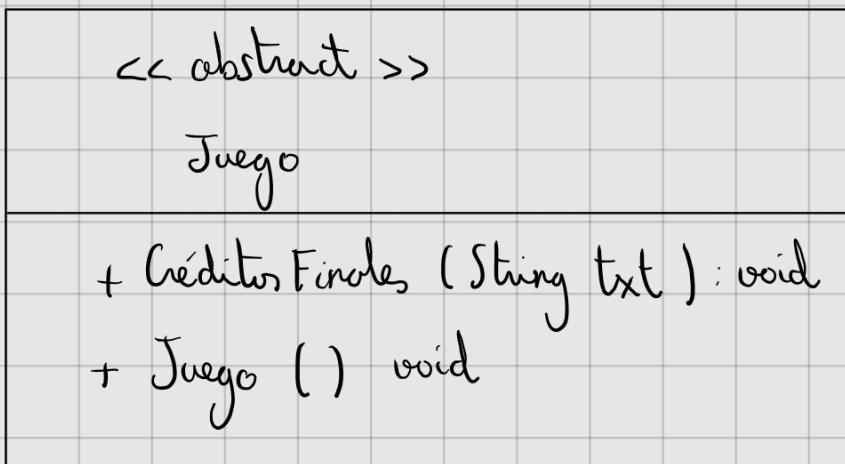
public void Notificar () {
    for ( Observer o : this.observers ) {
        o.update ( this, estado );
    }
}
```

```
public class ConcreteObserver implements Observer {  
    private int estado Observable;  
    public ConcreteObserver () {  
        this.estado Observable = 0;  
    }  
    public void update (int estado) {  
        if (estado == 0) {  
            // Acción normal  
        } else if (estado == 1) {  
            // Otra acción correspondiente a este  
            // estado  
        }  
    }  
}  
  
public class Main {  
    public void static main (String [] args) {  
        ConcreteSubject sub = new ConcreteSubject ();  
        ConcreteObserver ob = new ConcreteObserver ();  
        sub.Attach (ob);  
        sub.Notificar (); // el observador hará una acción  
        sub.set State (1);  
        sub.Notificar (); // el observador hará otra
```

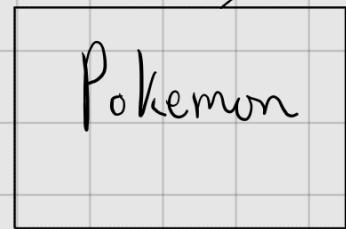
## \* TEMPLATE METHOD

// Es literalmente una clase abstracta  
// con algún método implementado

UML



→ Este método  
está implementado  
y será común para  
los juegos



CÓDIGO

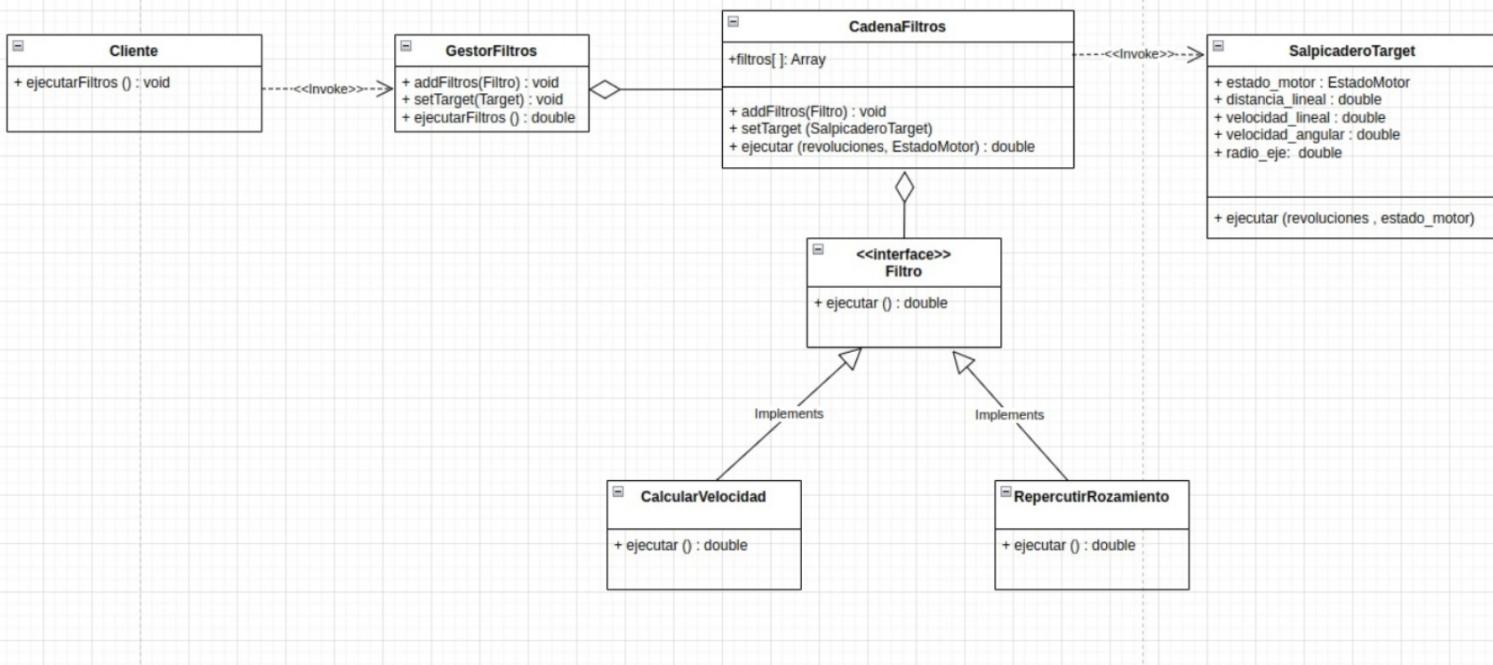
Una clase abstracta bro.

\* INTERCEPTING FILTER : Utiliza una estructura Tubería

y Filtro para procesar datos secuencialmente.

Preprocesamiento y postprocesamiento de peticiones y respuestas.

UML (Perezón dibujarlo)



CÓDIGO

```
class Cliente {  
    private GestorFiltros gestor ;  
    public Cliente ( GestorFiltros g )  
        this . gestor = g ;  
    }  
    public ejecutarFiltros () {  
        this . gestor . ejecutarFiltros () ;  
    }  
}
```

// En este caso haremos que GestorFiltros tenga una CadenaFilters, normalmente  
// se hace que tengan varias.

```
class GestorFiltros {  
    private CadenaFilters cadena;  
    public GestorFiltros () {}  
    public void setCadena ( Cadenafiltros c ) {  
        this.cadena = c;  
    }  
    public void setTarget ( SalpicaderoTarget s ) {  
        this.cadena.setTarget ( s );  
    }  
    public void ejecutarFiltros ( )  
    {  
        . . .  
        revoluciones = cadena.ejecutar ( . . . );  
        return revoluciones;  
    }  
}
```

```
class Cadenafiltros {  
    private ArrayList<Filtro> filtros = new ArrayList<> ();  
    private target = null;  
    public Cadenafiltros () {}  
    public addFiltro ( Filtro filtro ) {  
        this.filtros.add ( filtro );  
    }  
    public setTarget ( SalpicaderoTarget s ) {  
        this.target = s;  
    }
```

```
public ejecutor ( ... ) {  
    double rev;  
    for ( Filtros f : this.filters )  
        rev = f.ejecutor ( ... );  
    }  
    if ( this.target != null ) {  
        this.target.ejecutor ( rev );  
    }  
    return rev;  
}
```

```
public interface Filtros {  
    public double ejecutor ( ... );  
}
```

// Un ejemplo de un filtro en concreto

```
public class RepartirPorcentajes implements Filtros {  
    @Override  
    public double ejecutor ( ... )  
        return rev = ( rev * 0.05 );  
}
```

```
public class SalpicaderoTarget {  
    // getters, setters y método ejecutor que herá  
    // el cliente  
}
```

```
public class Main  
public static void main (String [] args ) {  
    . . .
```

```
    SalpicaderoTarget t = new SalpicaderoTarget (...);
```

```
    CadenaFiltros cadena = new CadenaFiltros ();
```

```
    GestorFiltros gestor = new GestorFiltros (),
```

```
    gestor.setTarget (t);
```

```
// - - - - - - - - -
```

```
    RepentinRozamiento fr = new RepentinRozamiento (),
```

```
    cadena.addFiltro (fr);
```

```
    gestor.setCadenaFiltros (cadena);
```

```
// - - - - - - - - -
```

```
    Cliente di = new Cliente (gestor),
```

```
    . . .  
    revolucion = di.ejecutorFiltros ();
```

```
{ } . . .
```