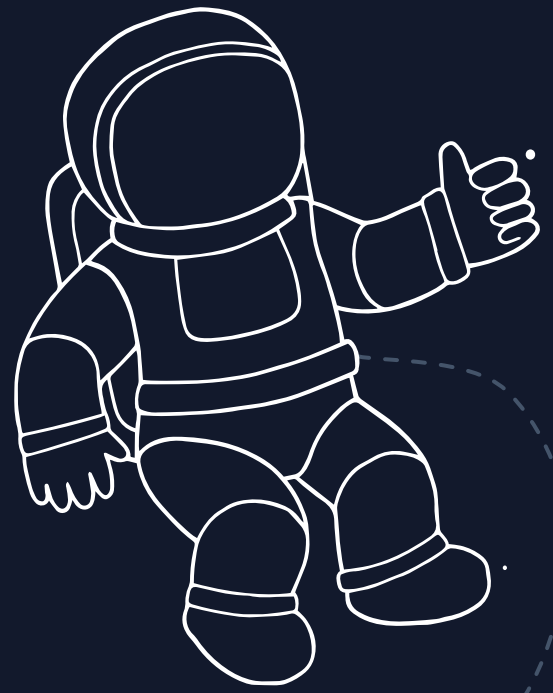
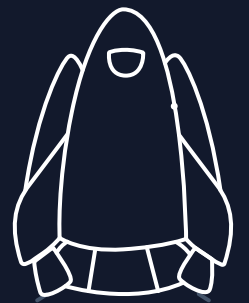


Programa académico CAMPUS

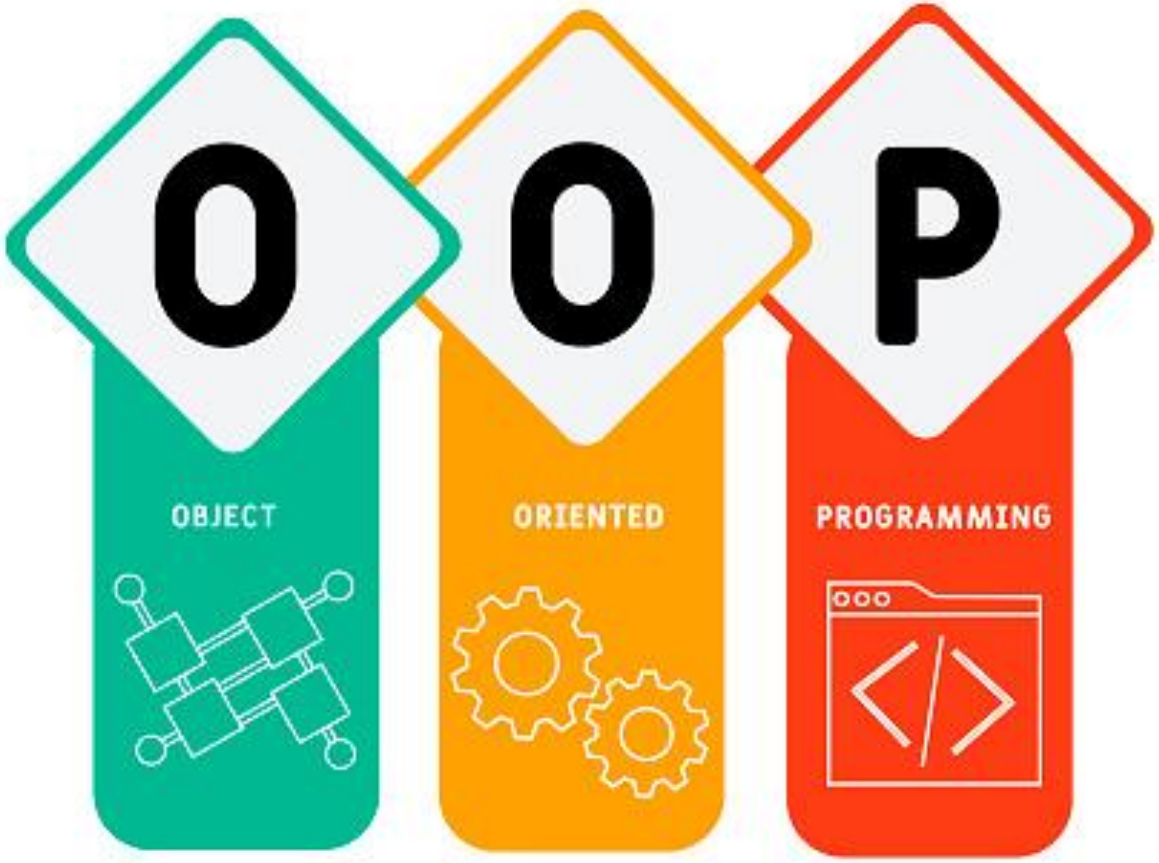
MODULO JAVA
Sesión 13

Patrones de Diseño (Estructurales)



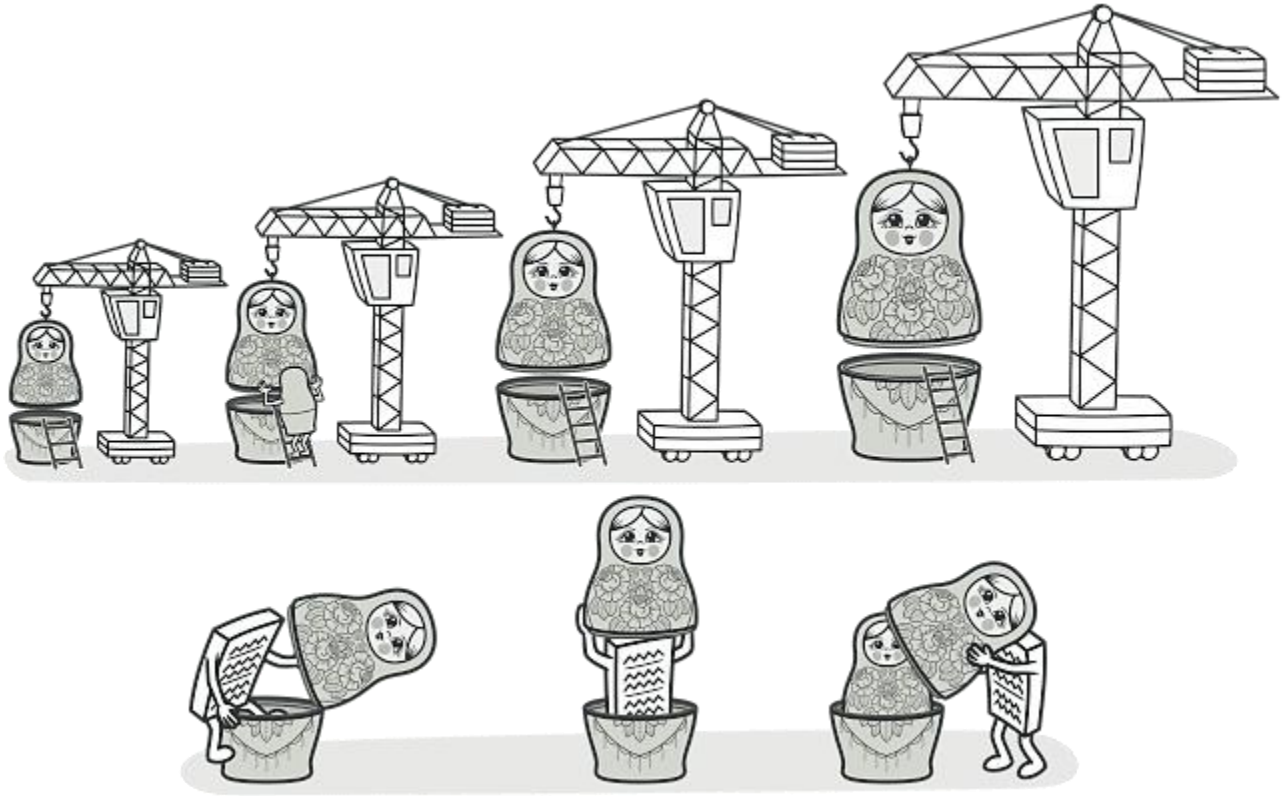
Trainer Carlos H. Rueda C.

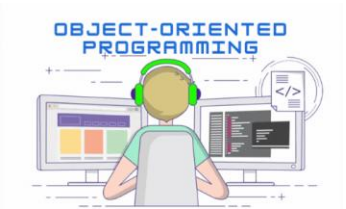

Java™


O
O
P
OBJECT
ORIENTED
PROGRAMMING

PATRONES DE DISEÑO ESTRUCTURALES

Los ***patrones de diseño estructurales*** son un tipo de patrón de diseño en programación que se enfoca en cómo ***organizar*** y ***ensamblar*** las ***clases*** y ***objetos*** para formar estructuras más grandes y funcionales, sin comprometer la flexibilidad o la facilidad de mantenimiento del código.

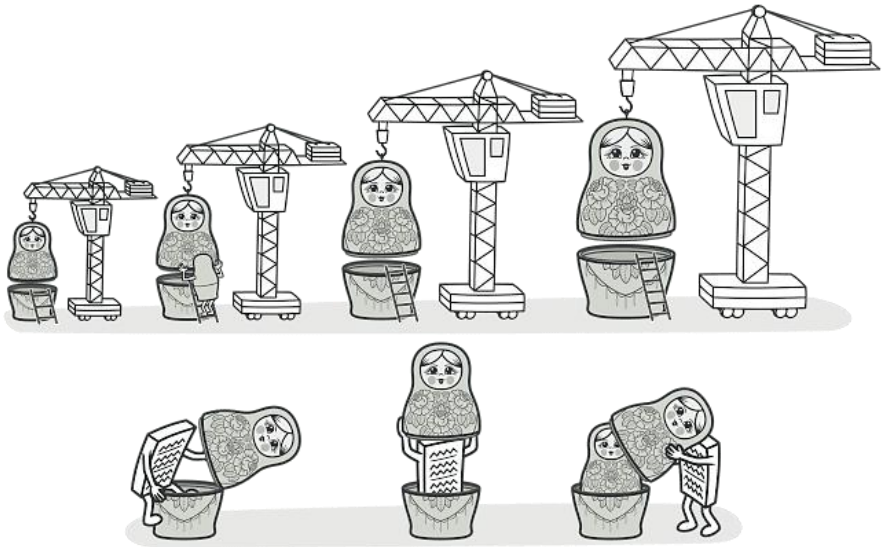


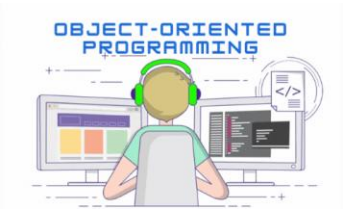


PATRONES DE DISEÑO ESTRUCTURALES

Características principales:

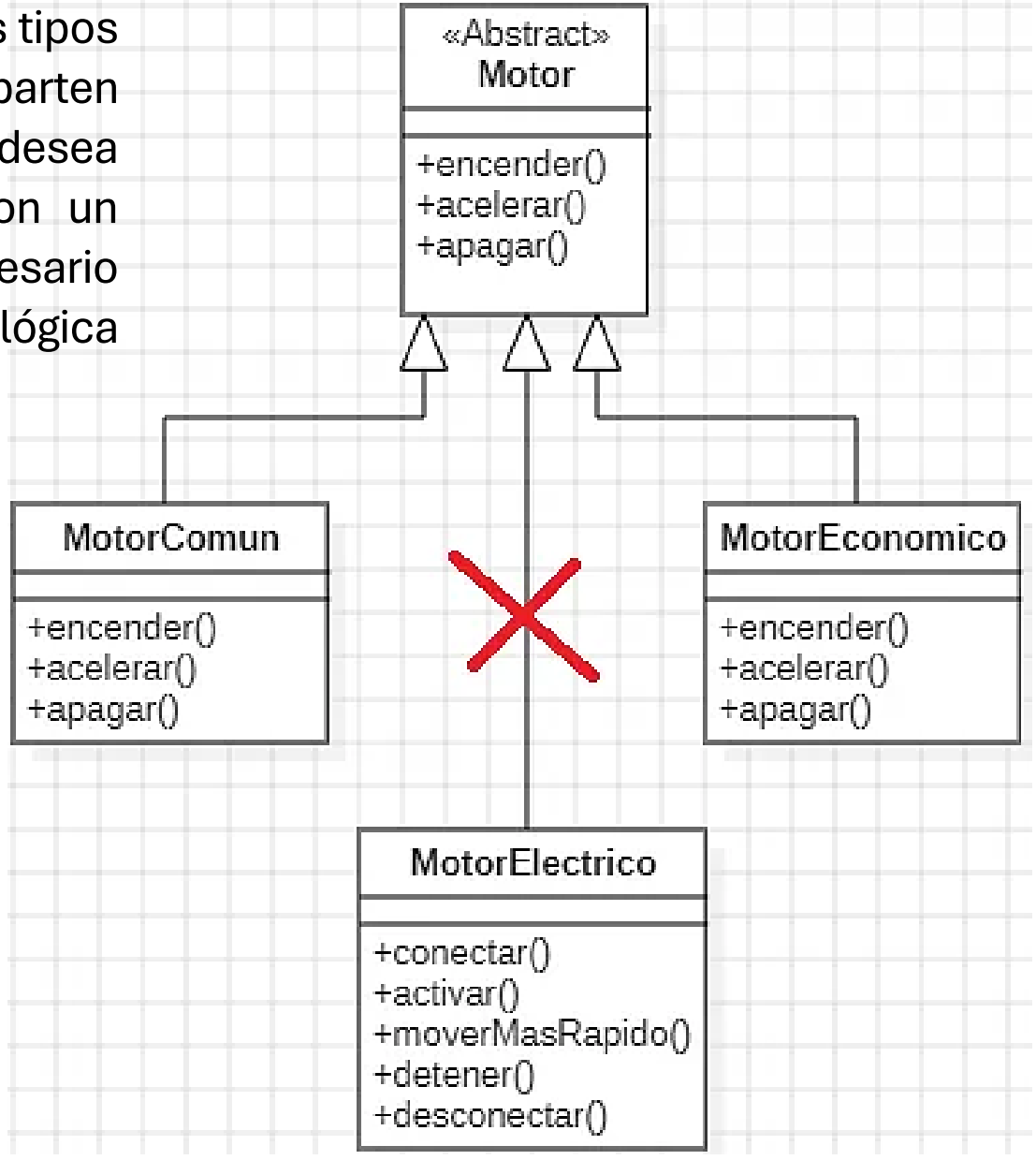
- ✓ **Organización de clases y objetos**:
  Usan la composición o la herencia para definir relaciones entre componentes.
- ✓ **Facilitan la reutilización del código**:
  Promueven el uso de componentes existentes de formas nuevas y flexibles.
- ✓ **Modularidad**:
  Ayudan a dividir un sistema grande en piezas más pequeñas y manejables.
- ✓ **Mantenimiento**:
  Simplifican las dependencias, haciendo que el código sea más fácil de mantener y extender.





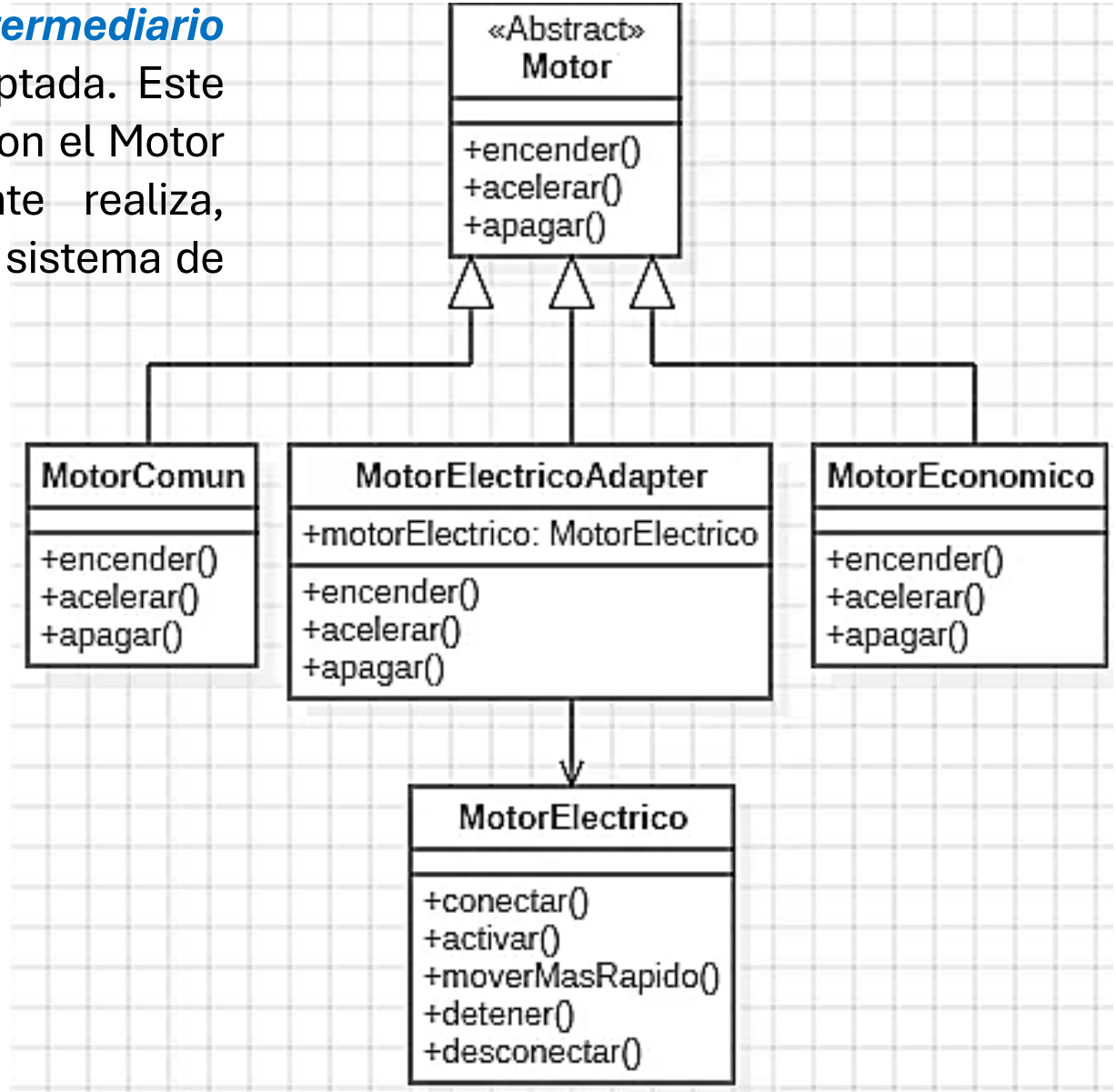
PATRONES DE DISEÑO – ADAPTER – EJEMPLO

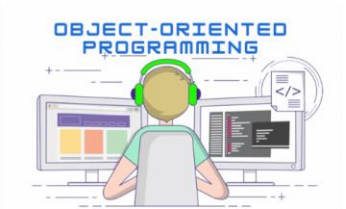
Supongamos que existe un sistema que trabaja con diferentes tipos de motores (Común y Económico), dichos motores comparten características entre sí como su funcionamiento. Pero, se desea vincular al sistema una clase de tipo motor Eléctrico con un funcionamiento diferentes al de los demás, entonces, es necesario adaptar la nueva clase sin que se modifique o se afecte la lógica inicial del sistema.



PATRONES DE DISEÑO – ADAPTER – EJEMPLO

Entra en juego una clase ***Adapter*** que actúa como un ***intermediario*** entre la clase padre y la clase que necesita ser adaptada. Este adaptador se encarga de establecer la comunicación con el Motor Eléctrico y ejecutar las solicitudes que el cliente realiza, permitiendo así la integración del Motor Eléctrico en el sistema de manera eficiente y coherente.

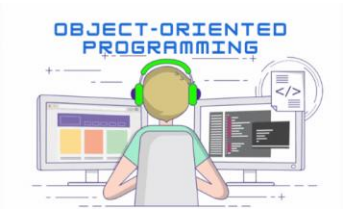




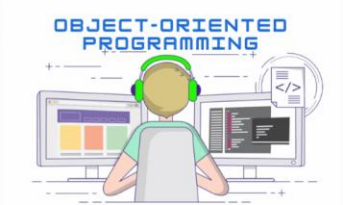
PATRONES DE DISEÑO – ADAPTER – EJEMPLO

```java
public abstract class Motor {

    abstract public void encender();

    abstract public void acelerar();

    abstract public void apagar();
}
```

```java
public class MotorComun extends Motor {

    public MotorComun(){
        super();
        System.out.println("Creando el motor
comun");
    }

    @Override
    public void encender() {
        System.out.println("encendiendo motor
comun");
    }

    @Override
    public void acelerar() {
        System.out.println("acelerando el motor
comun");
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor comun");
    }
}
```

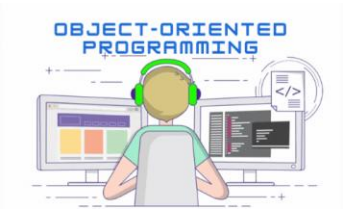


```java
public class MotorElectricoAdapter extends Motor{
    private MotorElectrico motorElectrico;

    public MotorElectricoAdapter(){
        super();
        this.motorElectrico = new MotorElectrico();
        System.out.println("Creando motor Electrico adapter");
    }
    @Override
    public void encender() {
        System.out.println("Encendiendo motorElectricoAdapter");
        this.motorElectrico.conectar();
        this.motorElectrico.activar();
    }
    @Override
    public void acelerar() {
        System.out.println("Acelerando motor electrico...");
        this.motorElectrico.moverMasRapido();
    }
    @Override
    public void apagar() {
        System.out.println("Apagando motor electrico");
        this.motorElectrico.detener();
        this.motorElectrico.desconectar();
    }
}
```



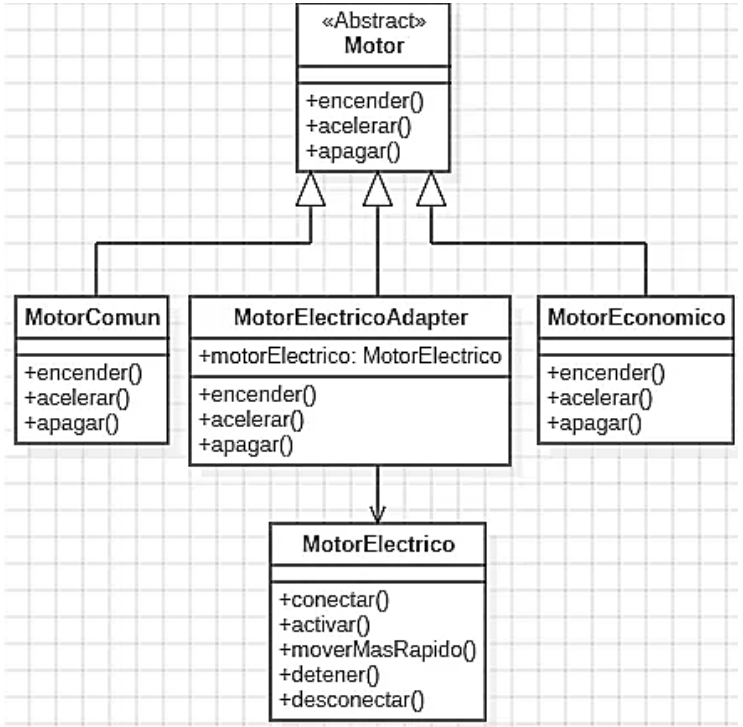
PATRONES DE DISEÑO – ADAPTER – EJEMPLO

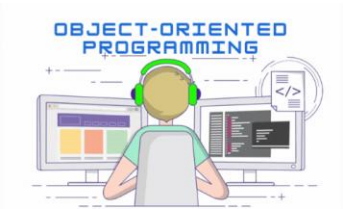
```
public class PruebaAdapter {
    public static void main(String[] args) {
        MotorElectricoAdapter adaptador = new
MotorElectricoAdapter();

        adaptador.encender();
    }
}
```



PATRONES DE DISEÑO – ADAPTER – EJERCICIO

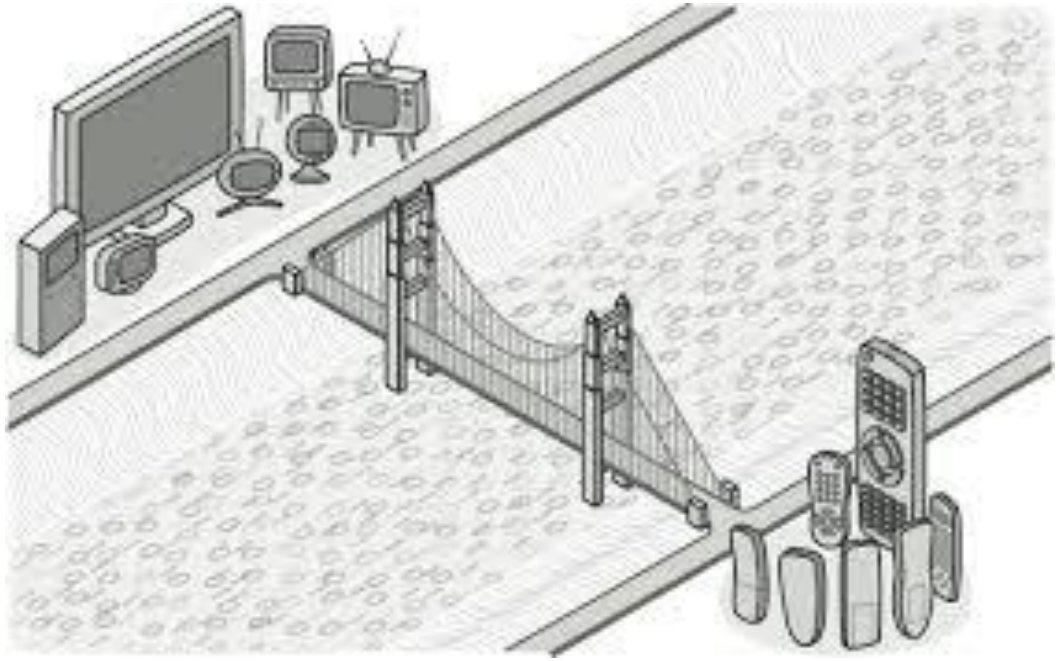
Imagina que tienes un sistema que carga vehículos eléctricos mediante un cargador especial. Sin embargo, en tu flota, también tienes vehículos de combustión interna que no pueden cargarse con este cargador. Tu tarea es implementar el patrón **Adapter** para permitir que los vehículos de combustión sean compatibles con el cargador eléctrico.

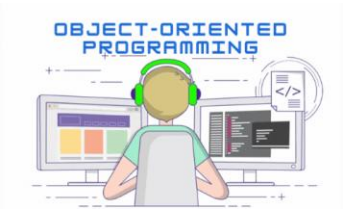




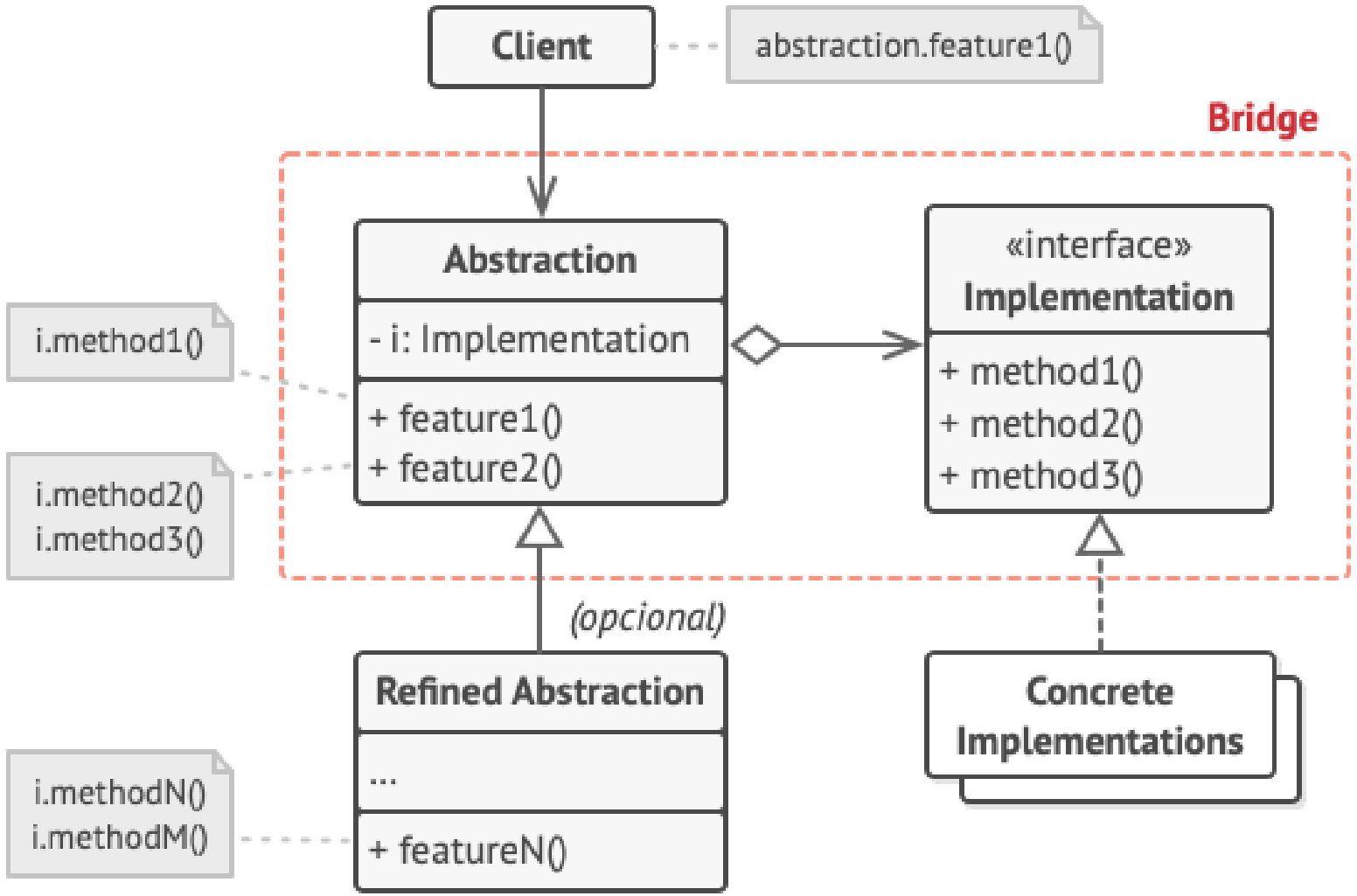
PATRONES DE DISEÑO – BRIDGE

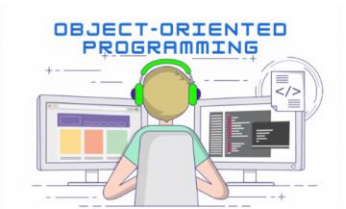
El patrón de diseño ***Bridge*** es una forma de organizar el código para que una ***idea principal*** (**abstracción**) y la ***forma en que funciona*** (**implementación**) estén separadas, de modo que puedas cambiar cualquiera de las dos sin afectar a la otra.
Es útil cuando tienes ***dos tipos de cosas relacionadas***, como "formas geométricas" y "formas de dibujarlas", y no quieres que estén tan conectadas que sea difícil modificarlas o expandirlas.



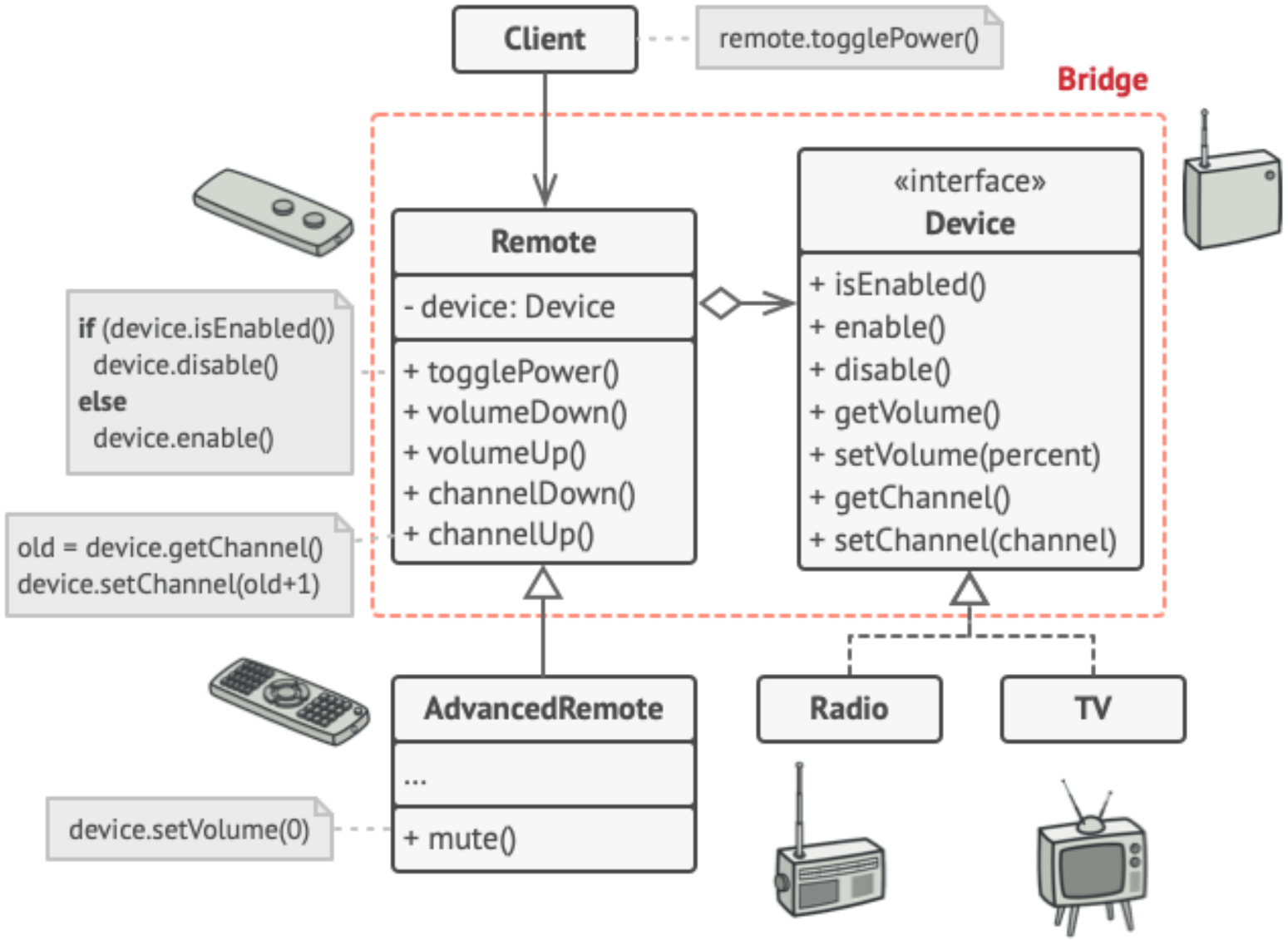


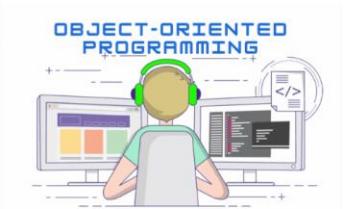
PATRONES DE DISEÑO – BRIDGE





PATRONES DE DISEÑO – BRIDGE

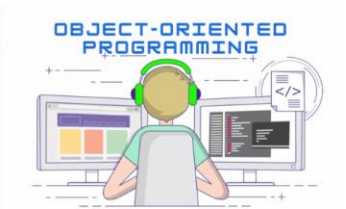




PATRONES DE DISEÑO – BRIDGE

```java
public interface Device {
    boolean isEnabled();
    void enable();
    void disable();
    int getVolume();
    void setVolume(int percent);
    int getChannel();
    void setChannel(int channel);
}
```

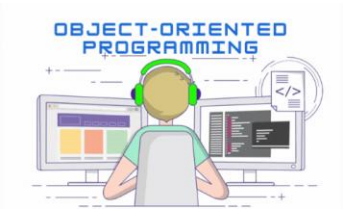
```java
public class Tv implements Device {
    private boolean enabled = false;
    private int volume = 50;
    private int channel = 1;
    @Override
    public boolean isEnabled() {
        return enabled;
    }
    @Override
    public void enable() {
        enabled = true;
        System.out.println("El televisor está encendido.");
    }
    @Override
    public void disable() {
        enabled = false;
        System.out.println("El televisor está apagado.");
    }
    @Override
    public int getVolume() {
        return volume;
    }
    @Override
    public void setVolume(int percent) {
        volume = percent;
        System.out.println("Volumen del televisor ajustado a: " +
volume + "%");
```



PATRONES DE DISEÑO – BRIDGE

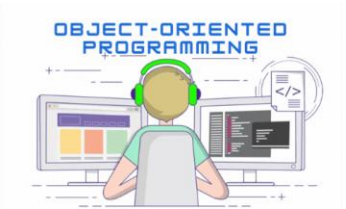
```java
public interface Device {
    boolean isEnabled();
    void enable();
    void disable();
    int getVolume();
    void setVolume(int percent);
    int getChannel();
    void setChannel(int channel);
}
```

```java
public class Radio implements Device {
    private boolean enabled = false;
    private int volume = 30;
    private int channel = 1;
    @Override
    public boolean isEnabled() {
        return enabled;
    }
    @Override
    public void enable() {
        enabled = true;
        System.out.println("La radio está encendida.");
    }
    @Override
    public void disable() {
        enabled = false;
        System.out.println("La radio está apagada.");
    }
    @Override
    public int getVolume() {
        return volume;
    }
    @Override
    public void setVolume(int percent) {
        volume = percent;
        System.out.println("Volumen de la radio ajustado a: " + volume
+ "%");
```



PATRONES DE DISEÑO – BRIDGE

```
public class RemoteControl {
    protected Device device;

    public RemoteControl(Device device) {
        this.device = device;
    }
    public void togglePower() {
        if (device.isEnabled()) { device.disable(); }
        else { device.enable(); }
    }
    public void volumeDown() {
        device.setVolume(device.getVolume() - 10);
    }
    public void volumeUp() {
        device.setVolume(device.getVolume() + 10);
    }
    public void channelDown() {
        device.setChannel(device.getChannel() - 1);
    }
    public void channelUp() {
        device.setChannel(device.getChannel() + 1);
    }
}
```

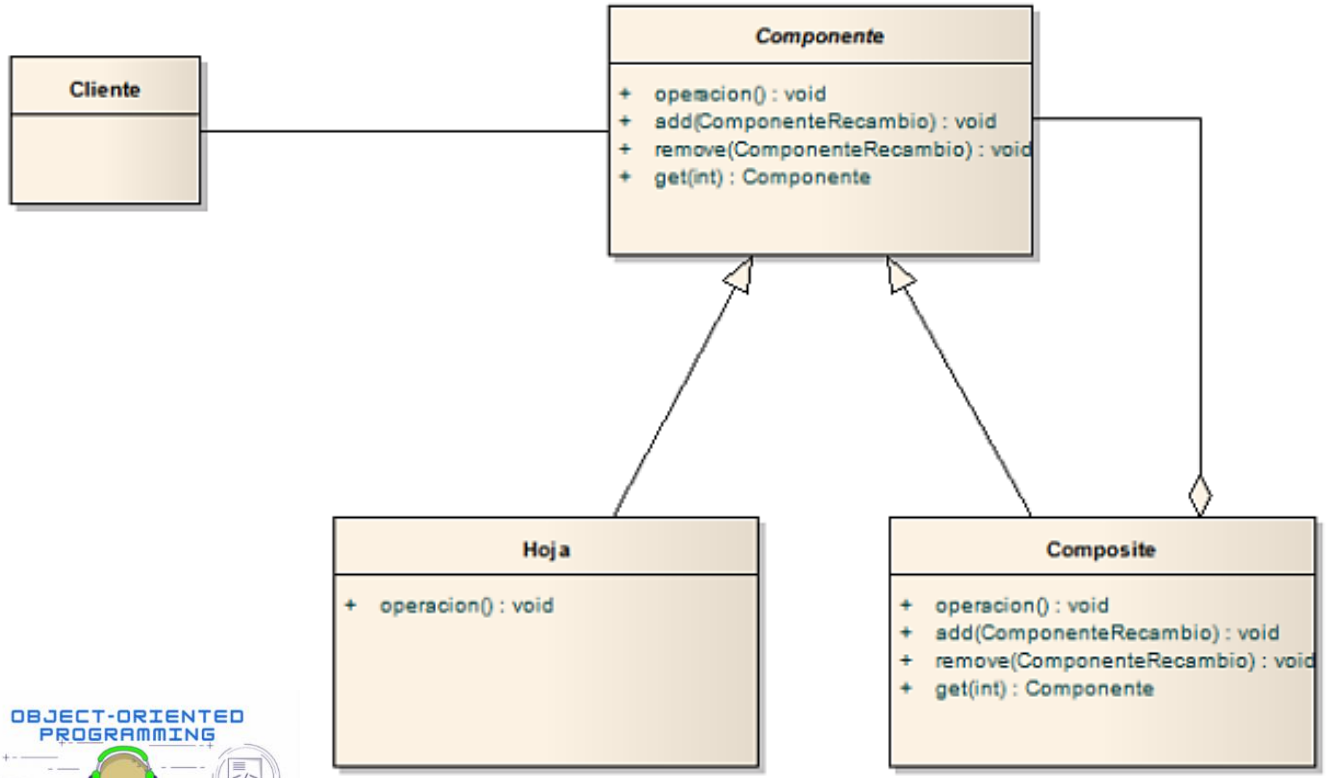


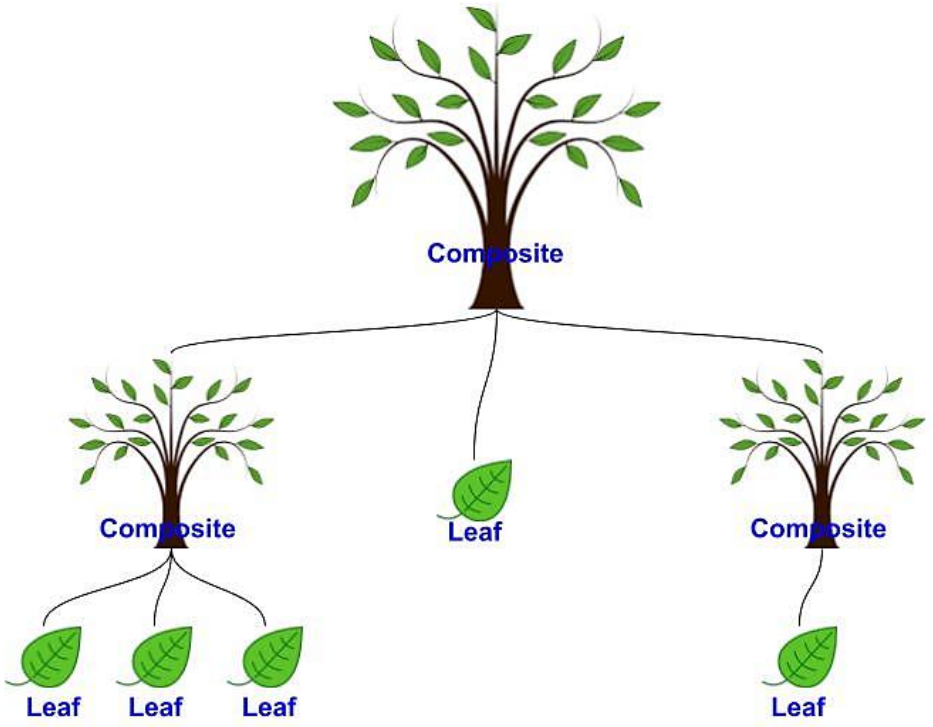
PATRONES DE DISEÑO – BRIDGE

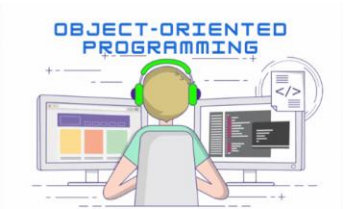
```
public static void main(String[] args) {
        // Crear un televisor y control remoto básico
        Device tv = new Tv();
        RemoteControl remoteControl = new RemoteControl(tv);

        System.out.println("Usando el control remoto básico con el televisor:");
        remoteControl.togglePower();
        remoteControl.volumeUp();
        remoteControl.channelUp();
        remoteControl.togglePower();

        // Crear una radio y un control remoto avanzado
        Device radio = new Radio();
        AdvancedRemoteControl advancedRemoteControl = new AdvancedRemoteControl(radio);

        System.out.println("\nUsando el control remoto avanzado con la radio:");
        advancedRemoteControl.togglePower();
        advancedRemoteControl.volumeUp();
        advancedRemoteControl.mute();
        advancedRemoteControl.togglePower();
    }
```

PATRONES DE DISEÑO – COMPOSITE

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.
Permite tratar objetos individuales y composiciones de objetos de manera uniforme, formando una estructura jerárquica



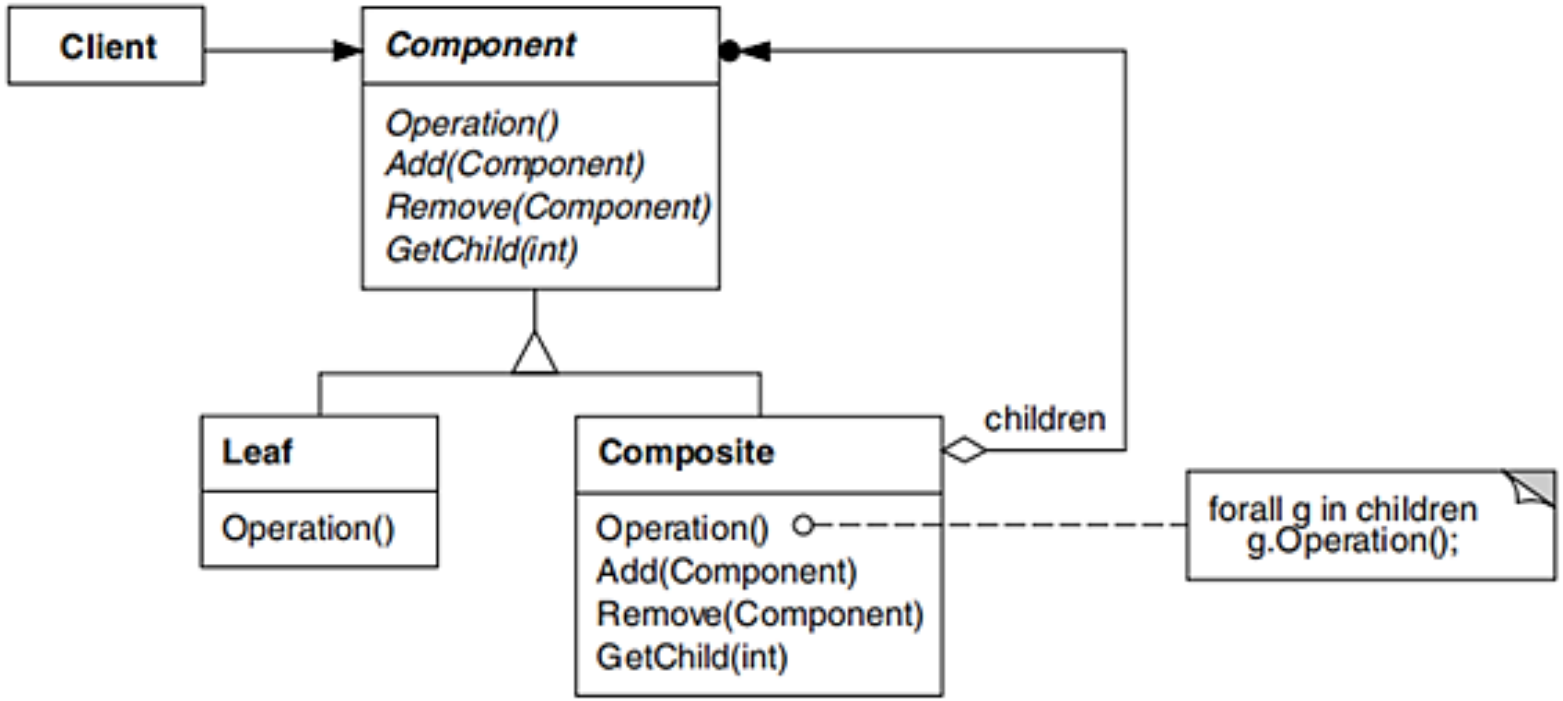


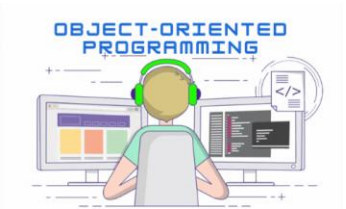


PATRONES DE DISEÑO – COMPOSITE

Los ***objetos individuales*** están representados por ***una clase concreta***. que implementa el ***Componente***, mientras que los ***grupos de objetos*** están representados por ***una clase compuesta*** que también implementa el ***Componente***.

La estructura jerárquica se crea mediante la composición, en la que un grupo de objetos puede incluir otros objetos individuales o grupos de objetos. Esta estructura permite la realización de interacciones recursivas y, por tanto, permite aplicar operaciones a todos los objetos de la jerarquía.





PATRONES DE DISEÑO – COMPOSITE - EJEMPLO

En un supermercado tienes dos formas de comprar un refresco:

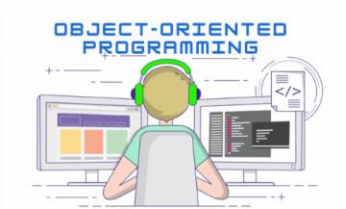
- ¿Es posible comprar la unidad?
- O comprar un paquete con 12 unidades

Aquí el elemento y el compuesto se pueden tratar de la misma manera:

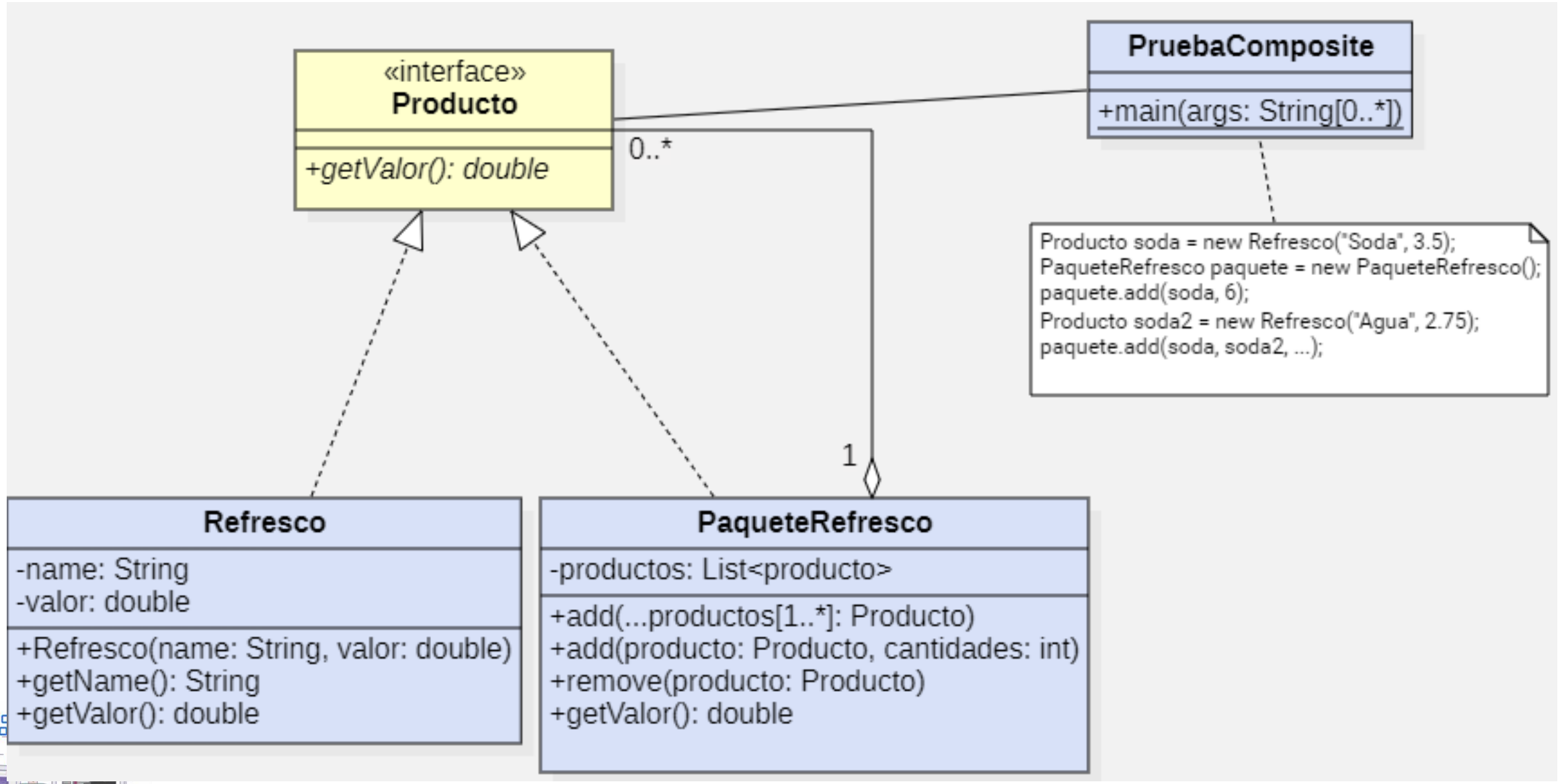
✓ Una ***botella de refresco*** tiene un precio propio, y puedes calcular su costo directamente.

✓ Un ***paquete de refresco*** (*que es un objeto compuesto*) no tiene un precio fijo, sino que su costo se calcula sumando el precio de todas las botellas dentro de él.

El truco es que el ***fardo y la botella pueden usarse igual***, porque el fardo "se comporta como" una botella al implementar la misma interfaz.



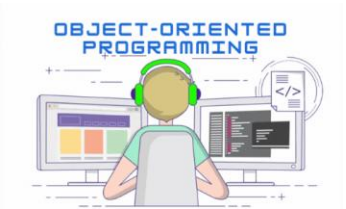
PATRONES DE DISEÑO – COMPOSITE - EJEMPLO



PATRONES DE DISEÑO – COMPOSITE - EJEMPLO

```java
public interface Produto {

    public Double getValor();

}
```

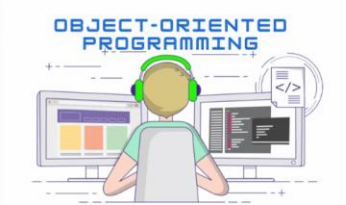
```java
public class Refrigerante implements Produto {

    private String nome;
    private Double valor;

    public Refrigerante(String nome, Double valor) {
        this.nome = nome;
        this.valor = valor;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public Double getValor() {
        return valor;
    }

}
```



PATRONES DE DISEÑO – COMPOSITE - EJEMPLO

```java
public interface Produto {

    public Double getValor();

}
```

```java
public class FardoDeRefrigerante implements Produto {

    private List<Produto> produtos = new ArrayList<>();

    public void add(Produto ...produtos) {
        this.produtos.addAll(Arrays.asList(produtos));
    }

    public void add(Produto produto, int quantidade) {
        for (int i = 0; i < quantidade; i++) {
            this.produtos.add(produto);
        }
    }

    public void remove(Produto produto) {
        produtos.remove(produto);
    }

    @Override
    public Double getValor() {
        Double soma = 0d;

        for (Produto produto : produtos) {
            soma += produto.getValor();
        }

        return soma;
    }
```

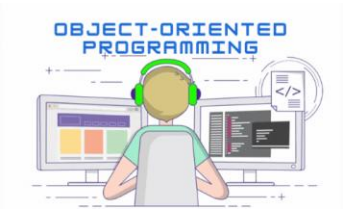


PATRONES DE DISEÑO – COMPOSITE - EJEMPLO

```java
public interface Produto {

    public Double getValor();

}
```

```java
Produto soda = new Refrigerante("Soda", 3.5);
FardoDeRefrigerante fardoDeRefrigerante = new FardoDeRefrigerante();
fardoDeRefrigerante.add(soda, 6);
```

```java
Produto soda1 = new Refrigerante("Soda", 2.75);
Produto soda2 = new Refrigerante("Soda", 3.99);
...

FardoDeRefrigerante fardoDeRefrigerante = new FardoDeRefrigerante();
fardoDeRefrigerante.add(soda1, soda2, ...);
```



PATRONES DE DISEÑO – COMPOSITE - EJEMPLO

A continuación se muestran algunos ejemplos de situaciones en las que se utiliza el patrón compuesto:

- **Estructuras Jerárquicas:**

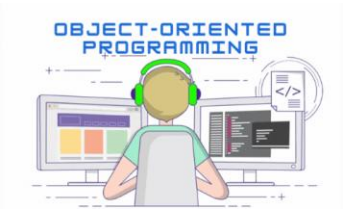
  El patrón Compuesto es ideal para trabajar con estructuras jerárquicas, como árboles o gráficos, donde los nodos pueden ser objetos individuales o grupos de objetos.

- **Modelado de interfaz de usuario:**

  Es común utilizar el patrón Compuesto al crear interfaces de usuario, especialmente cuando hay elementos que pueden contener otros elementos, como menús, botones y paneles.

- **Procesamiento de documentos:**

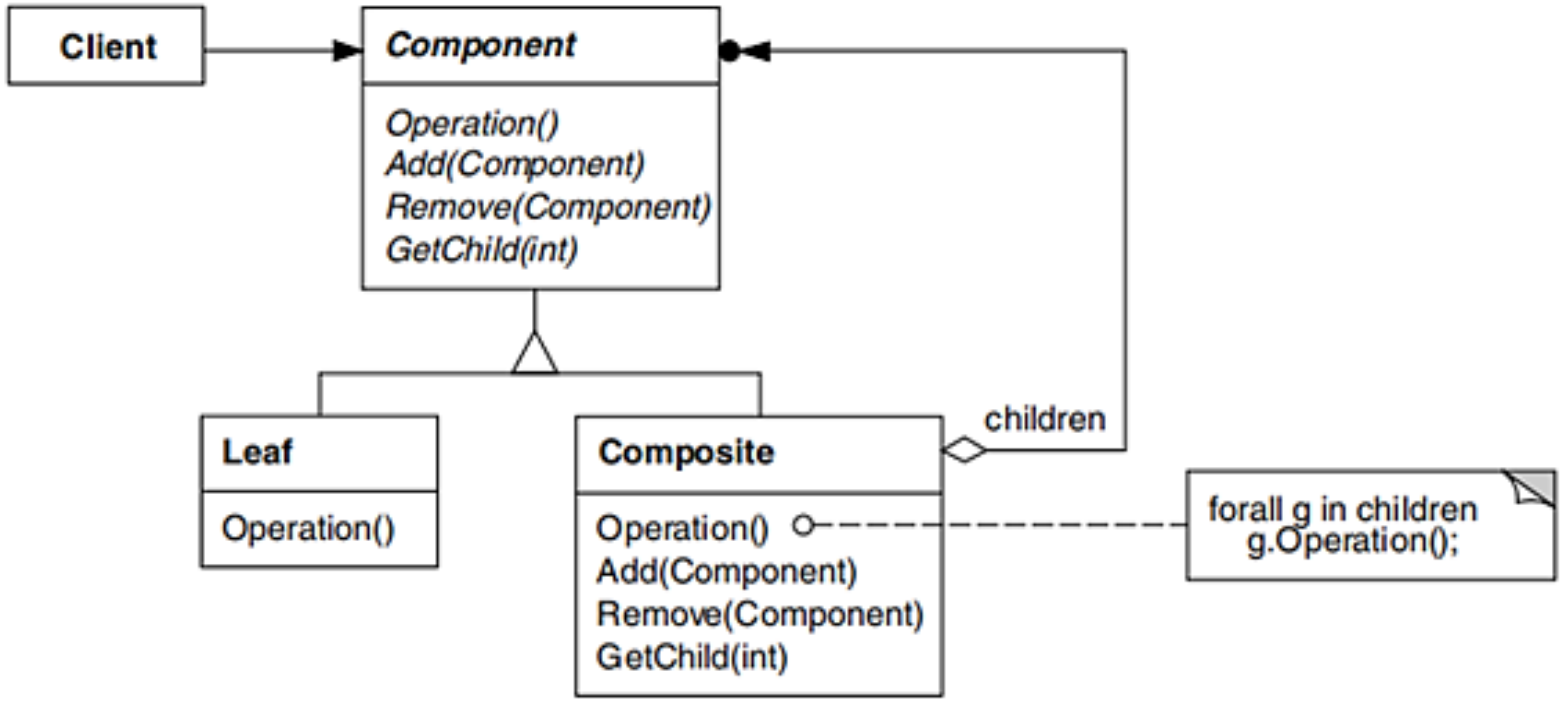
  El compuesto puede resultar útil para tratar elementos como documentos con secciones, párrafos, listas, etc. de manera uniforme.



PATRONES DE DISEÑO – COMPOSITE - EJERCICIO

Supongamos que existe una estructura de clases para representar una empresa con departamentos y empleados. Inicialmente tiene empleados desarrollador, diseñador y administrador. Para el ejemplo los empleados tienen un nombre y la edad y un método que despliega esta información de cualquiera de ellos o de todos los empleados de la empresa.

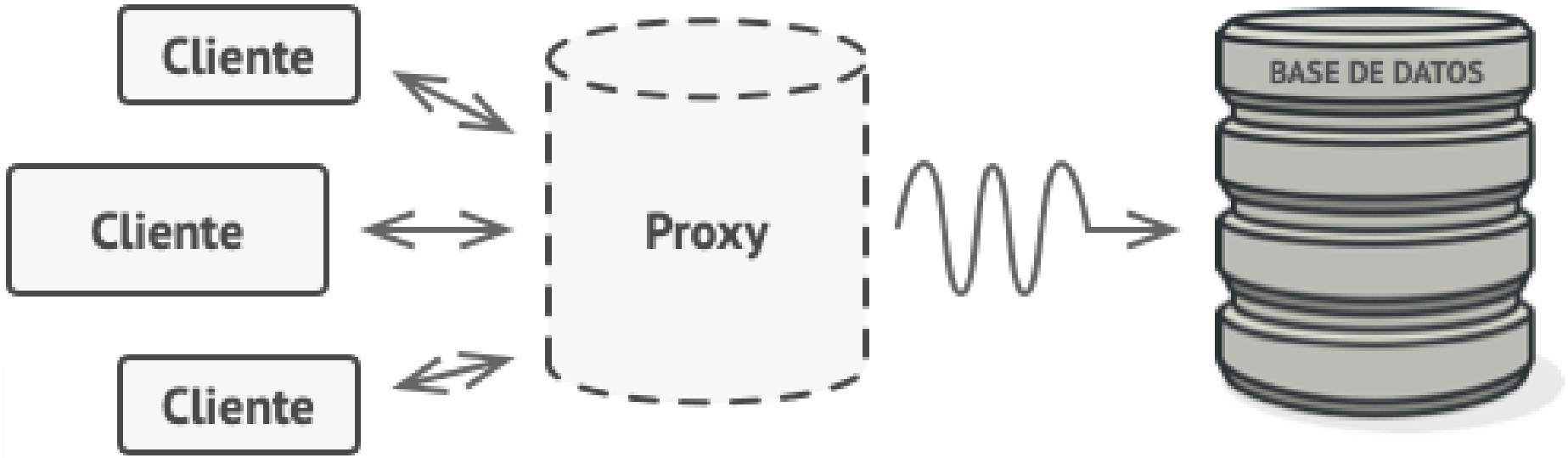
Implemente esta estructura en Java usando patrón de diseño Composite.

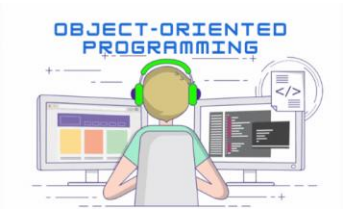


PATRONES DE DISEÑO – PROXY

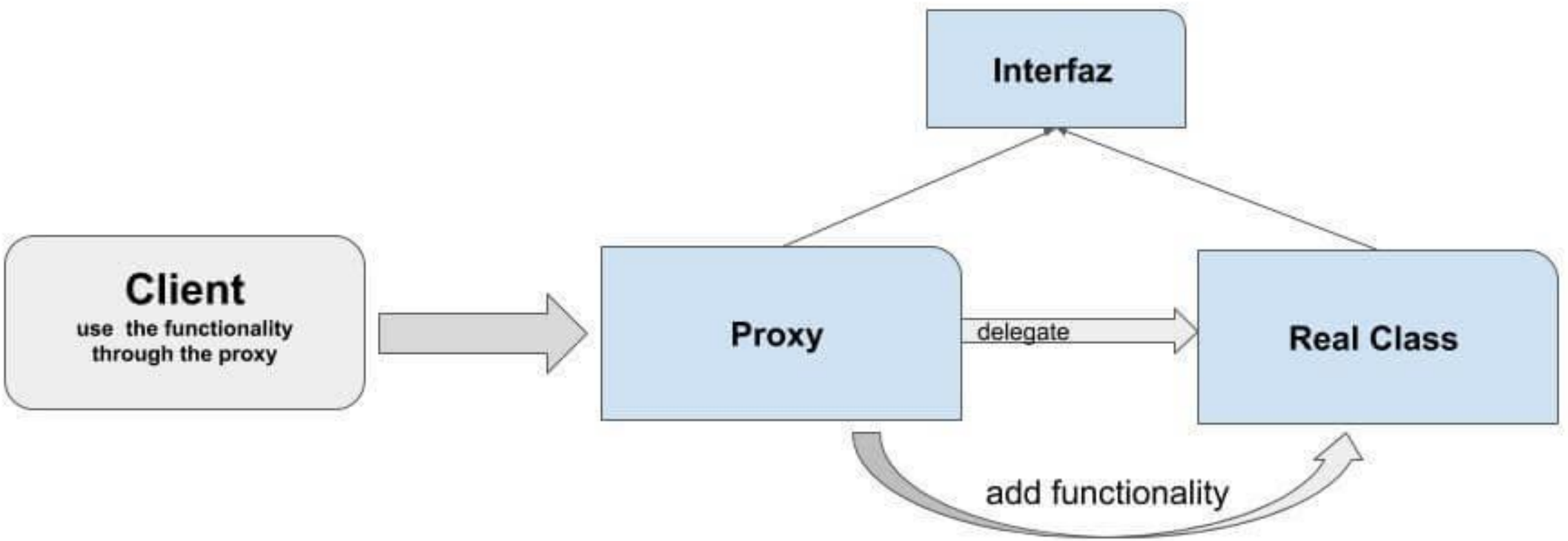
El patrón de diseño Proxy es un patrón estructural que actúa como un intermediario o sustituto de otro objeto para controlar el acceso a éste. Proporciona una representación o proxy de otro objeto y controla el acceso a él. Esto puede ser útil en situaciones en las que se quiere agregar funcionalidad adicional al objeto original sin modificar su código.

Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.



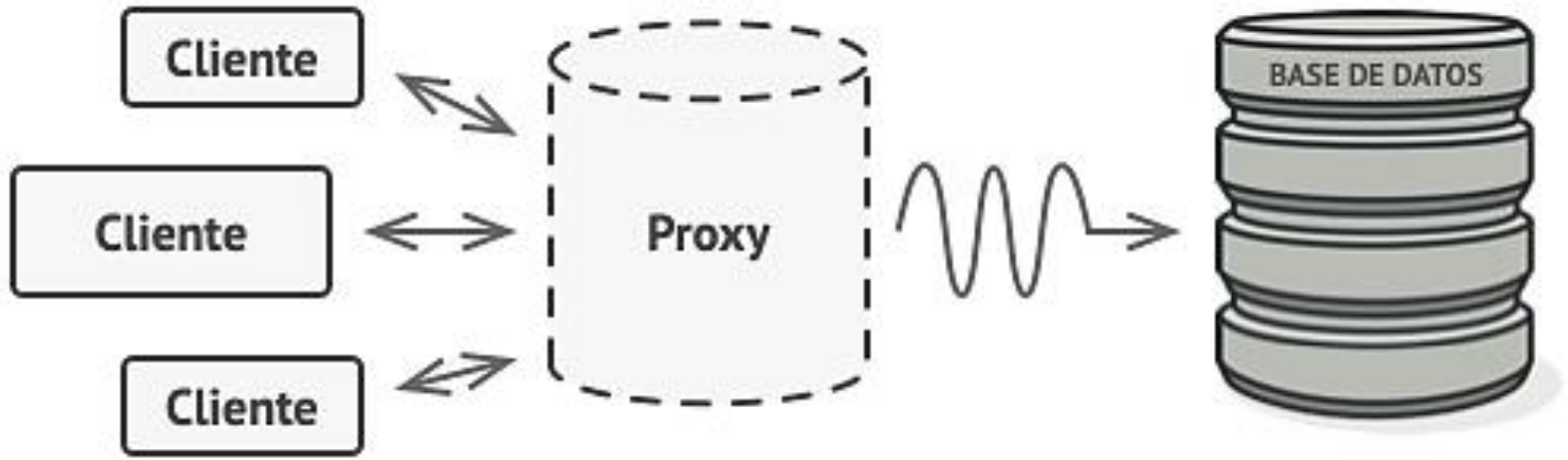


PATRONES DE DISEÑO – PROXY



PATRONES DE DISEÑO – PROXY

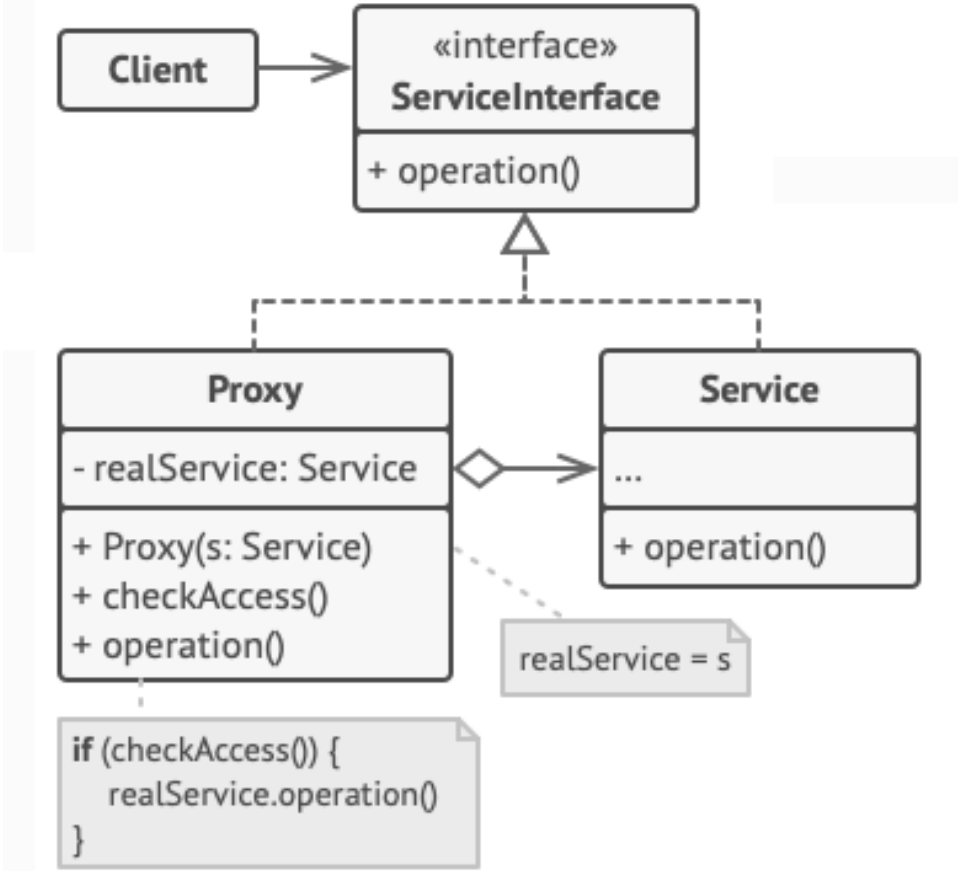
El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

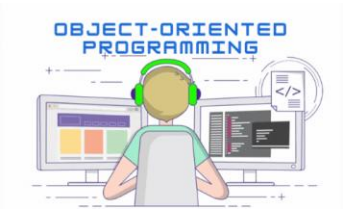


El proxy se camufla como objeto de la base de datos. Puede gestionar la inicialización diferida y el caché de resultados sin que el cliente o el objeto real de la base de datos lo sepan.

PATRONES DE DISEÑO – PROXY

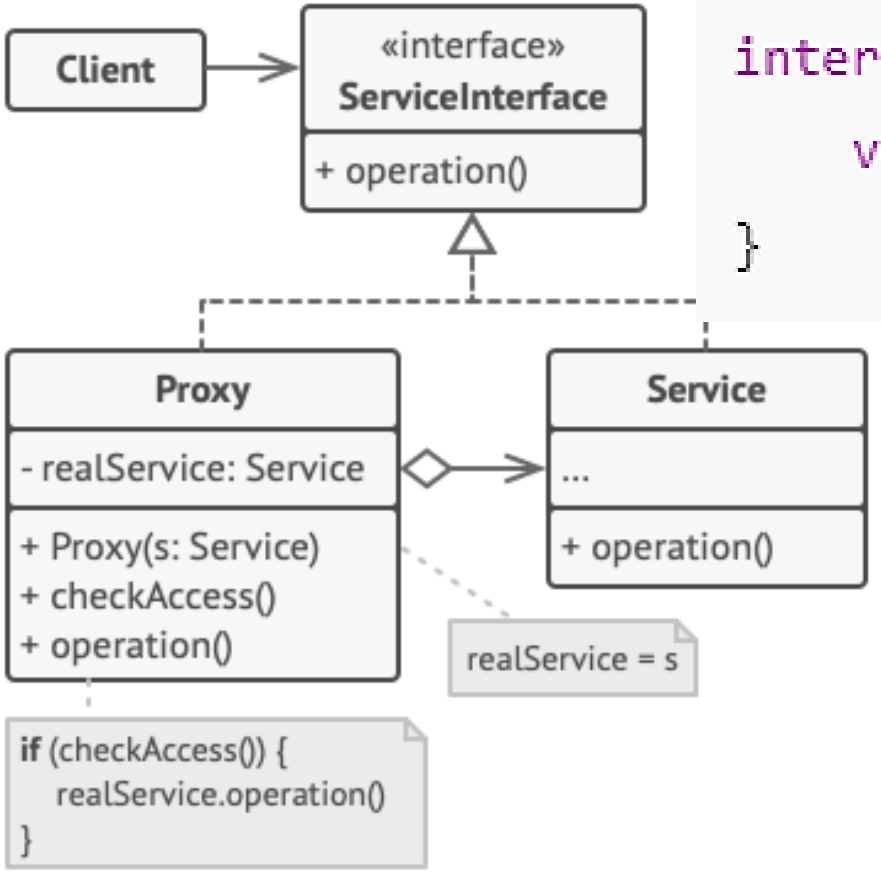
Pero, ¿cuál es la ventaja? Si necesitas ejecutar algo antes o después de la lógica primaria de la clase, el proxy te permite hacerlo sin cambiar esa clase. Ya que el proxy implementa la misma interfaz que la clase original, puede pasarse a cualquier cliente que espere un objeto de servicio real.





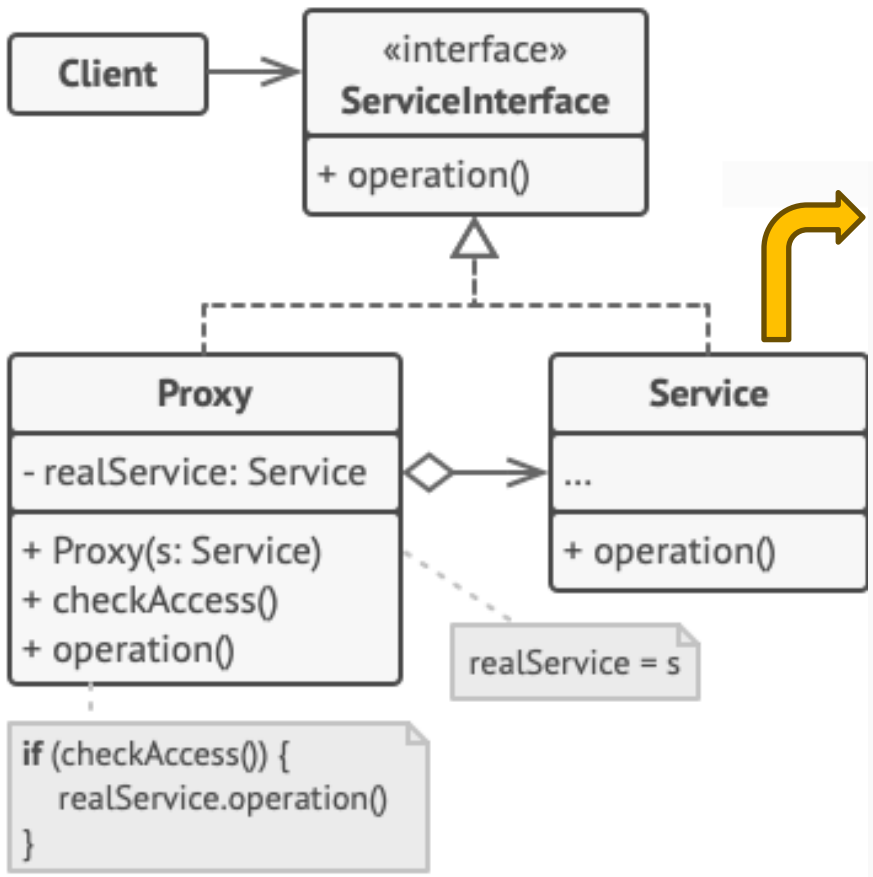
PATRONES DE DISEÑO – PROXY - EJEMPLO

Supongamos que tenemos una interfaz `Image` con un método `display()`, y queremos evitar cargar la imagen real desde el disco hasta que sea necesario.




```java
// Interfaz común
interface Image {
    void display();
}
```

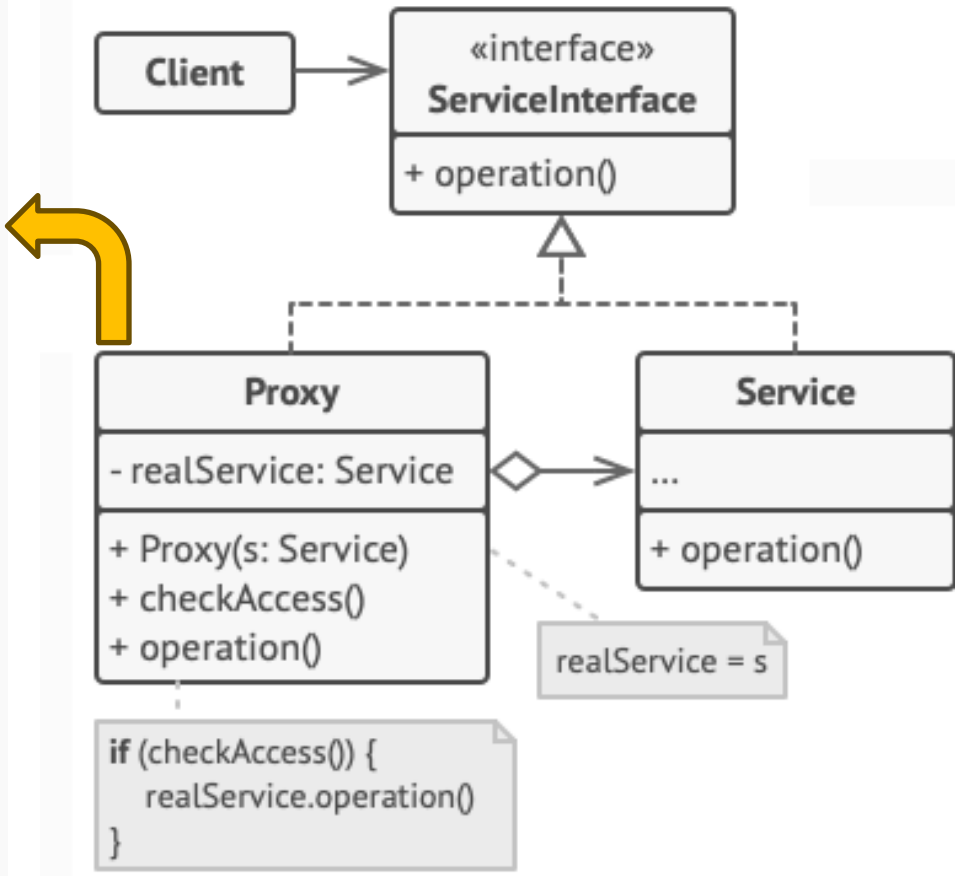
PATRONES DE DISEÑO – PROXY - EJEMPLO




```java
// Clase real que implementa la interfaz
class RealImage implements Image {
    private String fileName;

    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk(); // Operación costosa
    }

    private void loadFromDisk() {
        System.out.println("Cargando imagen desde el disco: " + fileName);
    }

    @Override
    public void display() {
        System.out.println("Mostrando imagen: " + fileName);
    }
}
```

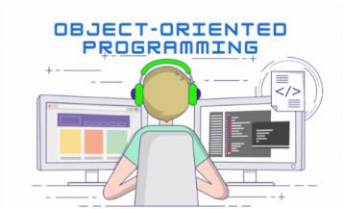
PATRONES DE DISEÑO – PROXY - EJEMPLO

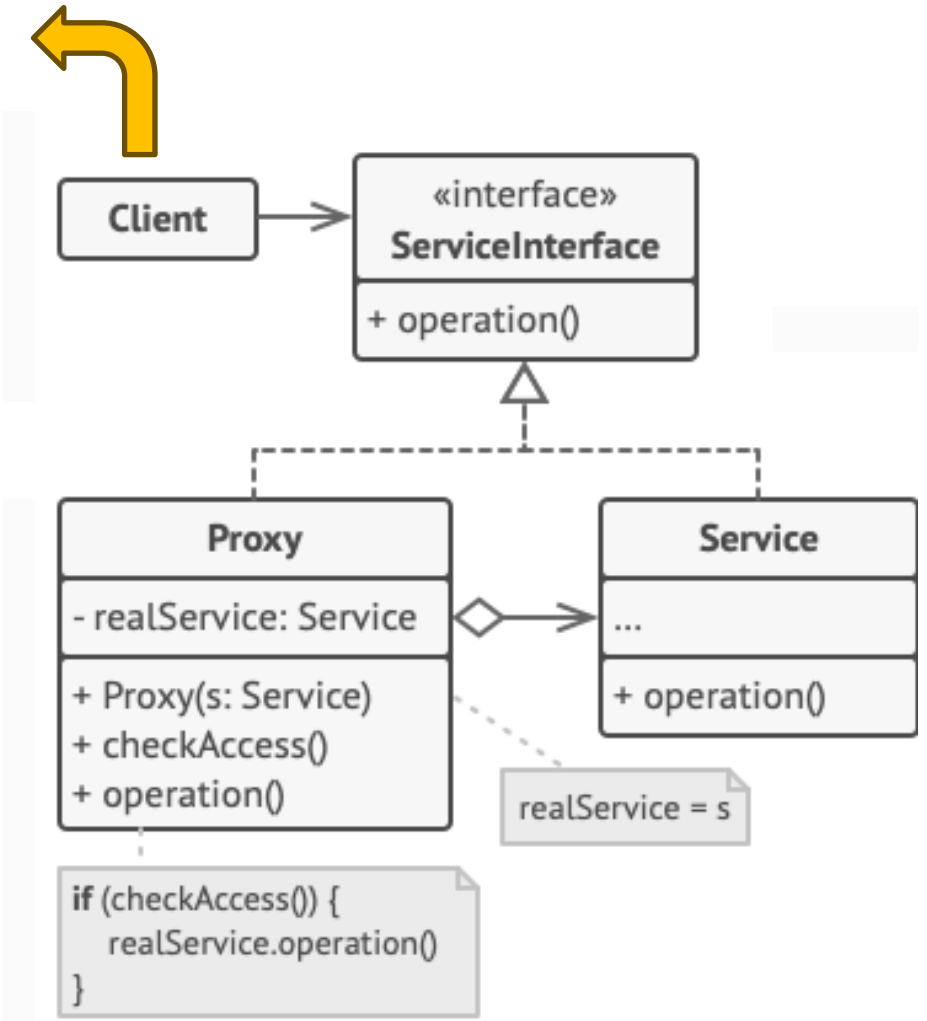
```
class ProxyImage implements Image {
    private RealImage realImage;
    private String fileName;

    public ProxyImage(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}
```



PATRONES DE DISEÑO – PROXY - EJEMPLO

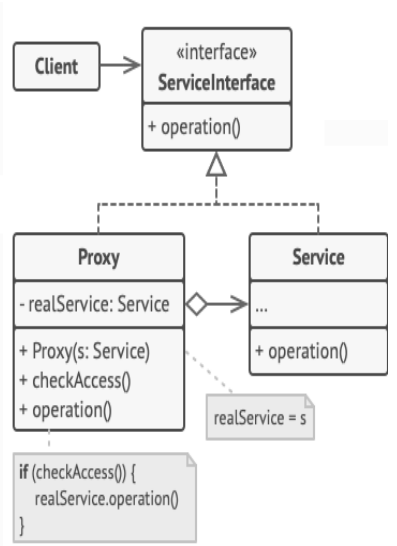
```java
public static void main(String[] args) {
    Image image = new ProxyImage("miFoto.jpg");

    // La primera vez que llamamos display(), se carga desde el disco
    image.display();
    System.out.println();

    // La segunda vez, no se recarga desde el disco
    image.display();
}
```





PATRONES DE DISEÑO – PROXY - EJERCICIO

Se ha desarrollando un sistema de gestión para una importante empresa de investigación científica. Dentro de este sistema, los investigadores necesitan acceder a archivos confidenciales que contienen información crítica, como fórmulas químicas, descubrimientos médicos o datos experimentales. Dado que estos recursos son altamente sensibles, solo los usuarios autenticados pueden tener acceso a ellos.



El equipo de seguridad de la empresa ha identificado un problema: actualmente, el acceso a los archivos confidenciales se realiza directamente, lo que abre la posibilidad de que cualquier usuario, sin importar si está autenticado o no, pueda intentar ***acceder*** y ***desplegar*** la información de estos archivos. Esto pone en riesgo la seguridad de la información y la reputación de la empresa.

Para solucionar este problema, te han encargado implementar una capa de seguridad adicional. Tu objetivo es crear un sistema en el que los usuarios intenten acceder al archivo confidencial, pero ***antes*** de permitirles el acceso, ***el sistema verifique si están autenticados***. Si el usuario no está autenticado, se le debe denegar el acceso y redirigirlo a un método que simule el inicio de sesión. Solo después de autenticarse correctamente podrán acceder al recurso.

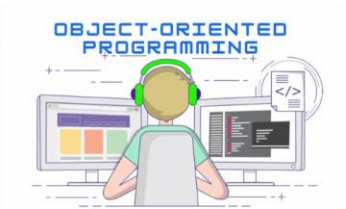
PATRONES DE DISEÑO – PROXY - EJERCICIO

Un almacén gestiona una gran cantidad de productos y, por eficiencia, no quiere cargar toda la información detallada de cada producto en memoria a menos que sea necesario. Solo se quiere cargar la información completa cuando se acceda específicamente al producto.

Tu tarea es implementar un sistema utilizando el **patrón Proxy** para optimizar la gestión de productos en el almacén.

La salida de tu programa es esta:

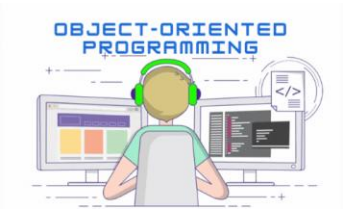
```
Producto: Laptop (ID: 101)
Cargando detalles para el producto con ID: 101...
Detalles del producto: Laptop, Precio: $1200, Descripción: Laptop de alta gama.

Producto: Smartphone (ID: 102)
Cargando detalles para el producto con ID: 102...
Detalles del producto: Smartphone, Precio: $800, Descripción: Smartphone de última generac

Producto: Teclado (ID: 103)
(Detalles no cargados aún)

Producto: Smartphone (ID: 102)
Detalles del producto: Smartphone, Precio: $800, Descripción: Smartphone de última generac
```



PATRONES DE DISEÑO – PROXY - EJERCICIO

Un programa de biblioteca de multimedia permite a los usuarios acceder a elementos como canciones, videos e imágenes descargándolos desde internet. Sin embargo, este proceso es lento y consume muchos recursos cuando el usuario quiere visualizar repetidamente el mismo archivo. Actualmente existe una clase que se encarga de descargar los elementos multimedia por su id. También puede listar los recursos y obtener la información de los videos (nombre e id).

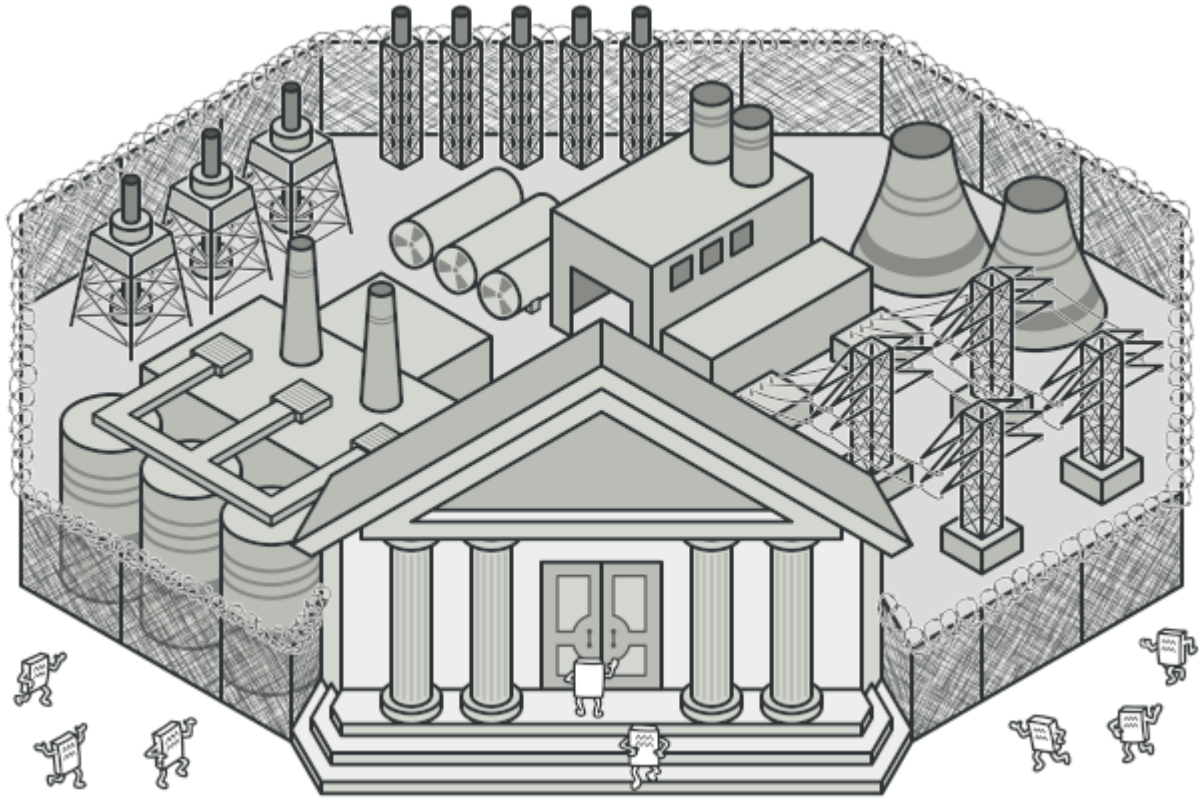
Tu tarea es mejorar este sistema implementando el **patrón Proxy** para **cachear los elementos multimedia** y hacer que su recuperación sea más eficiente.

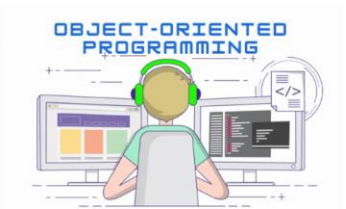


PATRONES ESTRUCTURALES – FACADE

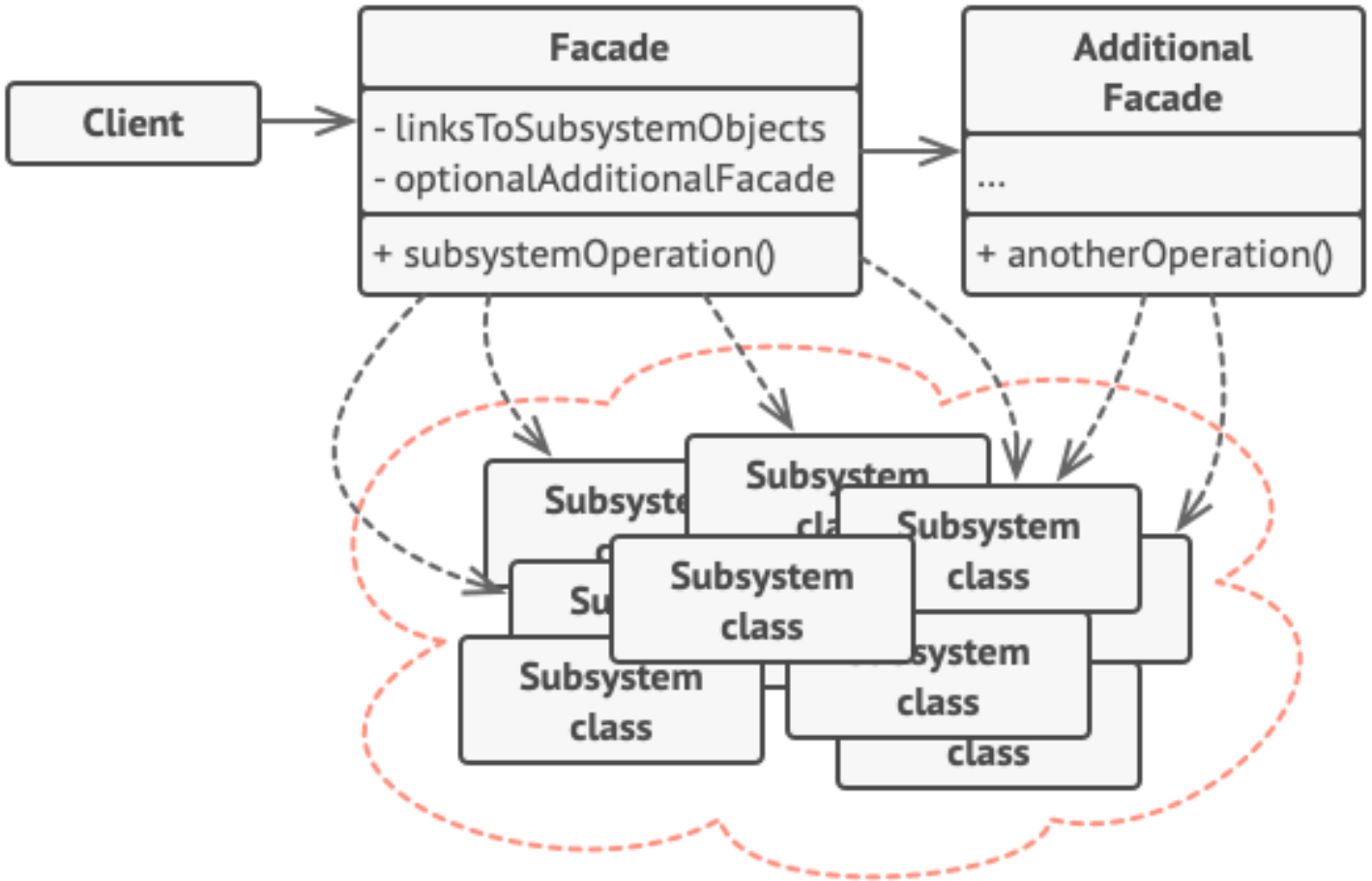
Se utiliza para simplificar la interacción con sistemas complejos al proporcionar una interfaz única y más sencilla que encapsula un conjunto de subsistemas o clases interrelacionadas.

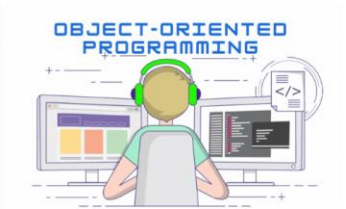
Facade actúa como una fachada o un punto de entrada único a un grupo de clases o servicios, ocultando su complejidad detrás de una interfaz sencilla.





PATRONES ESTRUCTURALES – FACADE DISEÑO

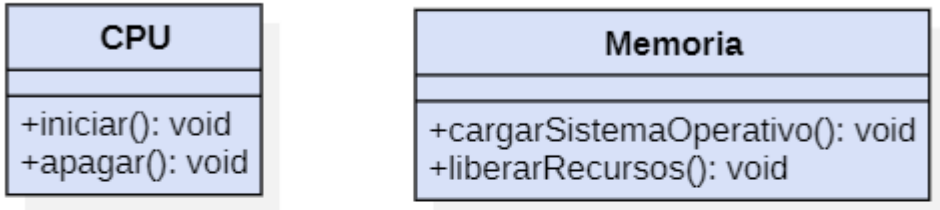


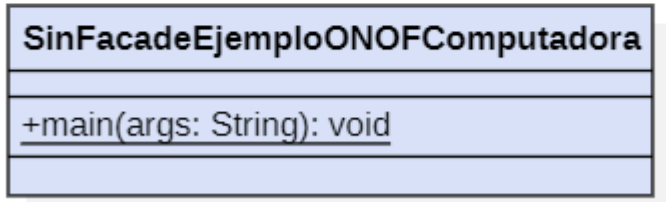


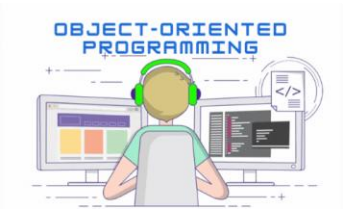
PATRONES ESTRUCTURALES – FACADE EJEMPLO

Supongamos que existe un sistema de computadoras que realizan distintas operaciones para encender y apagar cada computadora. Dichas operaciones, contienen acciones complejas, como cargar el sistema operativo, iniciar recursos y liberar recursos para apagar. Con el patrón Facade, se puede simplificar las operaciones complejas en un conjunto de otras más amigables y sencillas para el usuario final..

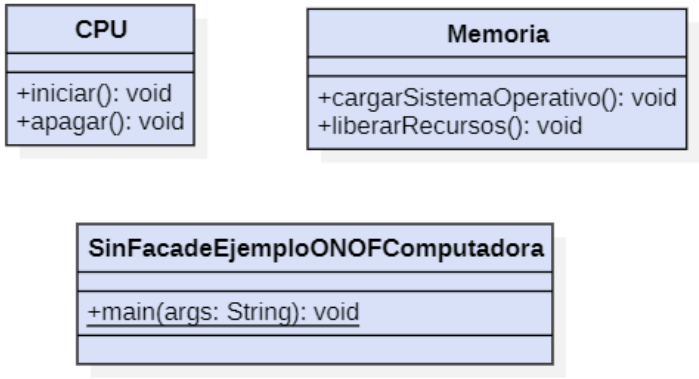
SIN FACADE







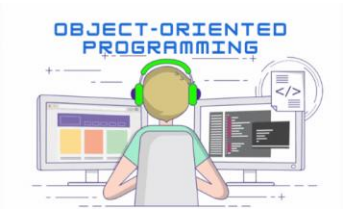
PATRONES ESTRUCTURALES – FACADE EJEMPLO




```java
class Memoria {
    public void cargarSistemaOperativo() {
        System.out.println(x:"Sistema operativo cargado en la memoria");
    }

    public void liberarRecursos() {
        System.out.println(x:"Recursos liberados");
    }
}
```

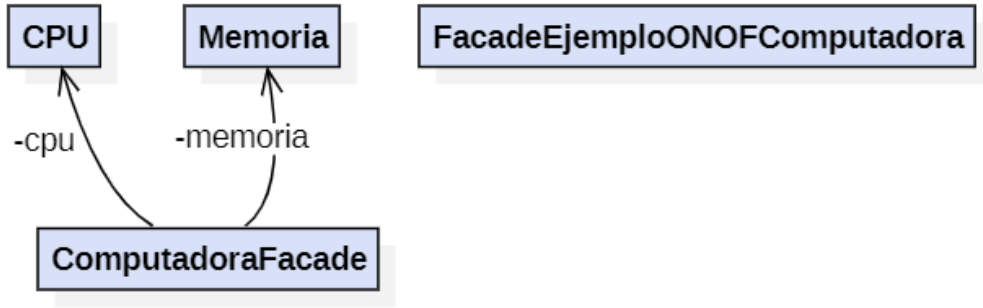
```java
class CPU {
    public void iniciar() {
        System.out.println(x:"CPU iniciada");
    }

    public void apagar() {
        System.out.println(x:"CPU apagada");
    }
}
```

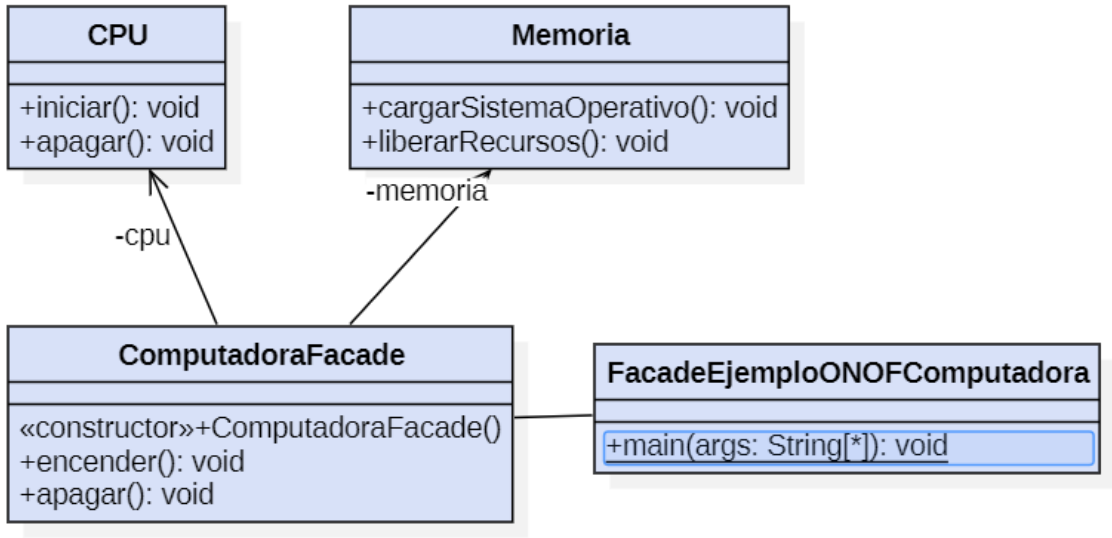
```java
public class SinFacadeEjemploONOFComputadora {
    Run | Debug
    public static void main(String[] args) {
        CPU cpu = new CPU();
        Memoria memoria = new Memoria();

        // Encender la computadora
        cpu.iniciar();
        memoria.cargarSistemaOperativo();
        System.out.println(x:"Computadora encendida");

        // Algun proceso en la computadora
        System.out.println(x:"Realizando algunas operaciones...");

        // Apagar la computadora
        memoria.liberarRecursos();
        cpu.apagar();
        System.out.println(x:"Computadora apagada");
    }
}
```

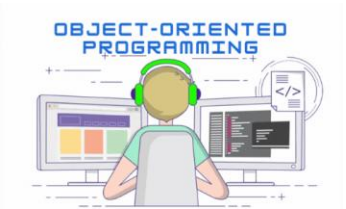


PATRONES ESTRUCTURALES – FACADE EJEMPLO

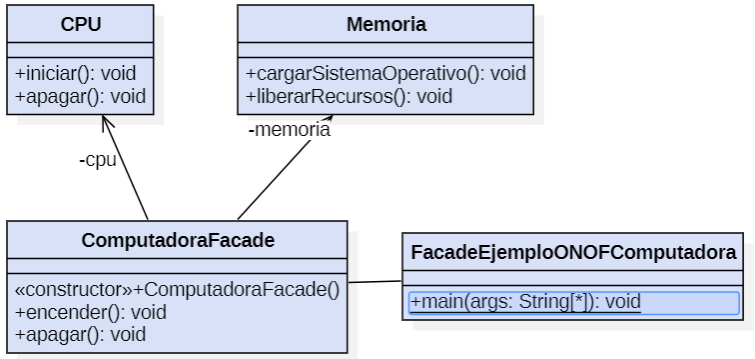
CON FACADE







PATRONES ESTRUCTURALES – FACADE EJEMPLO



CON FACADE

```java
class ComputadoraFacade {
    private CPU cpu;
    private Memoria memoria;

    public ComputadoraFacade() {
        this.cpu = new CPU();
        this.memoria = new Memoria();
    }

    public void encender() {
        cpu.iniciar();
        memoria.cargarSistemaOperativo();
        System.out.println(x:"Computadora encendida");
    }

    public void apagar() {
        memoria.liberarRecursos();
        cpu.apagar();
        System.out.println(x:"Computadora apagada");
    }
}
```

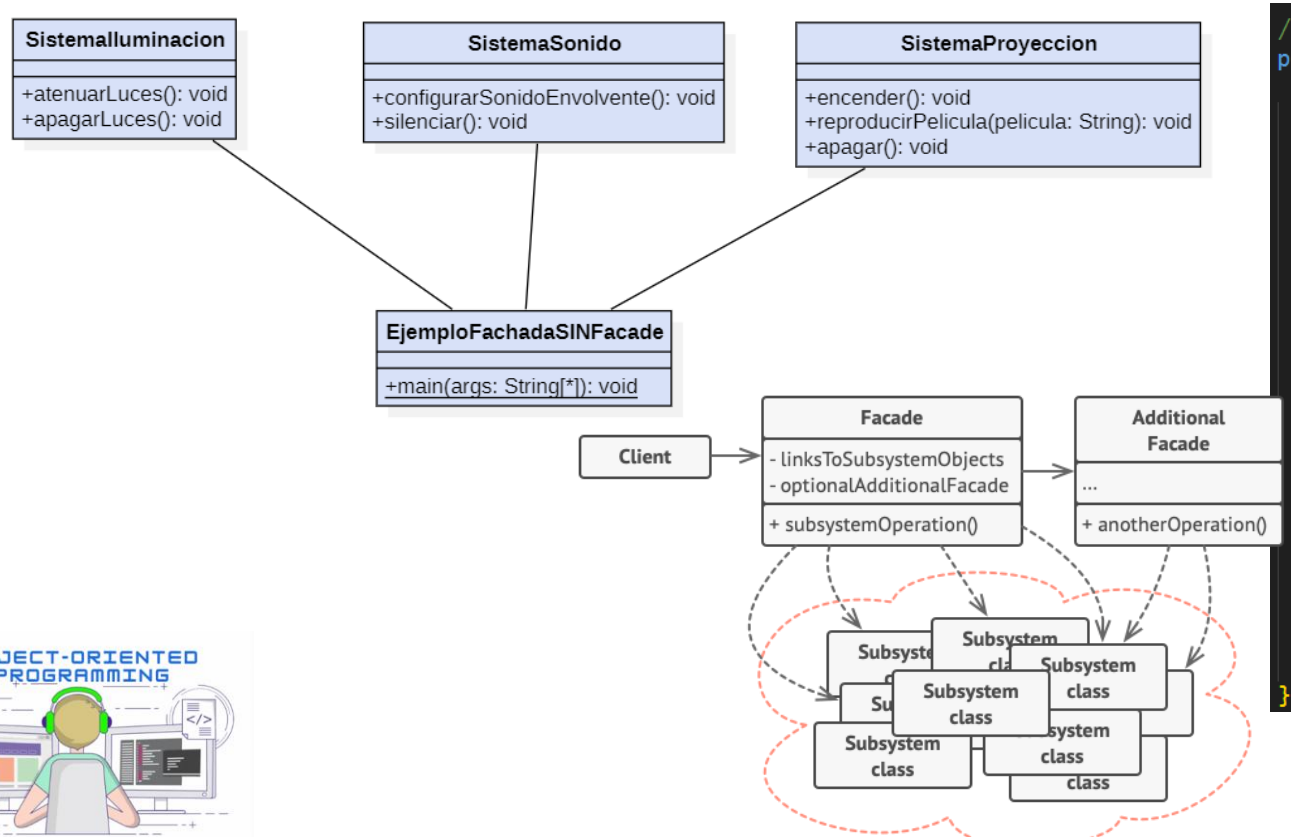
```java
class CPU {
    public void iniciar() {
        System.out.println(x:"CPU iniciada");
    }

    public void apagar() {
        System.out.println(x:"CPU apagada");
    }
}
```

```java
class Memoria {
    public void cargarSistemaOperativo() {
        System.out.println(x:"Sistema operativo cargado en la memoria");
    }

    public void liberarRecursos() {
        System.out.println(x:"Recursos liberados");
    }
}
```

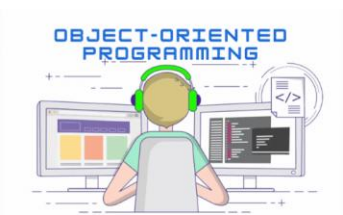
```java
public class FacadeEjemploONOFComputadora {
    Run | Debug
    public static void main(String[] args) {
        ComputadoraFacade computadora = new ComputadoraFacade();
        computadora.encender();
        System.out.println(x:"Realizando algunas operaciones...");
        computadora.apagar();
    }
}
```

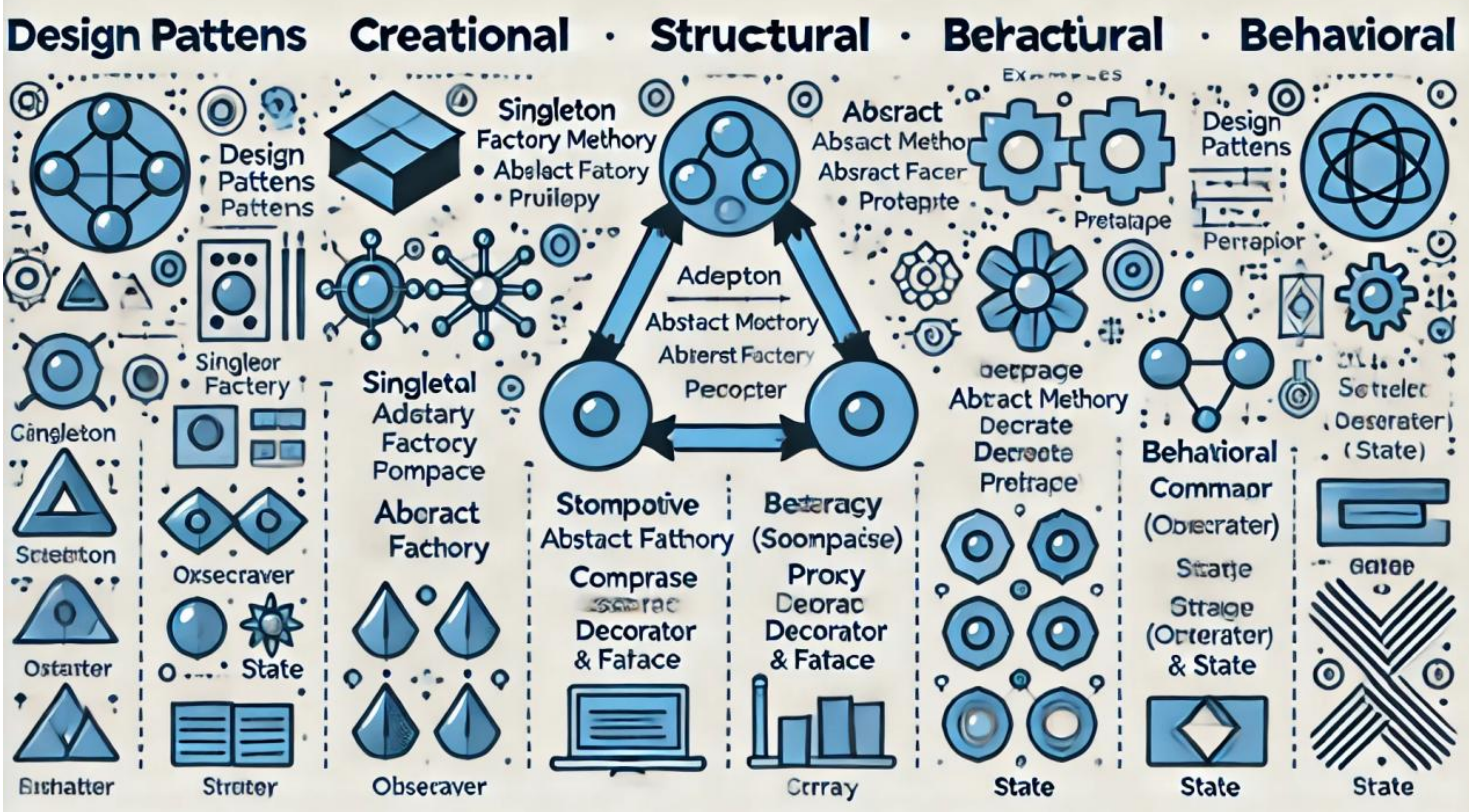
PATRONES ESTRUCTURALES – FACADE EJERCICIO

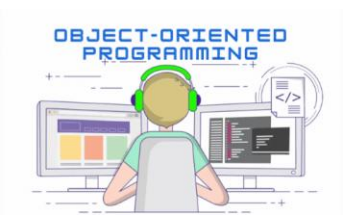
Supongamos un sistema de un cine con múltiples subsistemas para manejar iluminación, sonido y proyección.
A continuación, te entregan el diagrama UML y como está el programa cliente que utiliza los subsistemas.
Tu tarea es aplicar el patrón Facade para integran los subsistemas.

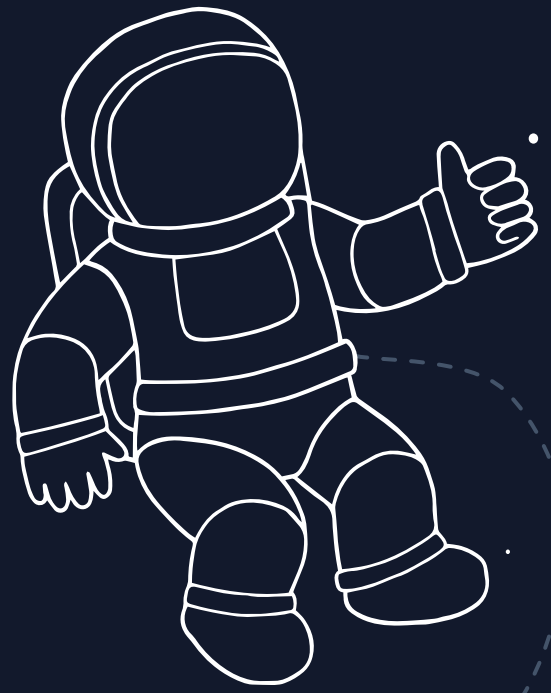



```java
// Cliente: Interacción con el sistema
public class EjemploFachadaSINFacade {
    Run | Debug
    public static void main(String[] args) {
        // Creacion de los subsistemas
        SistemaIluminacion iluminacion = new SistemaIluminacion();
        SistemaSonido sonido = new SistemaSonido();
        SistemaProyeccion proyeccion = new SistemaProyeccion();

        // Subsistemas para ver pelicula "ORIGEN"
        System.out.println(x:"Preparando el sistema para ver una película...");
        iluminacion.atenuarLuces();
        sonido.configurarSonidoEnvolvente();
        proyeccion.encender();
        proyeccion.reproducirPelicula(pelicula:"ORIGEN");

        // Apagando los subsistemas
        System.out.println(x:"Apagando el sistema...");
        proyeccion.apagar();
        iluminacion.apagarLuces();
        sonido.silenciar();
    }
}
```


OBJECT-ORIENTED
PROGRAMMING


Design Pattens Creational · Structural · Beractural · Behavioral
Design
Pattens
• Pattens
Singleton
Factory Methory
• Ablact Fatory
• • Pruilopy
Absract
Absact Metho
Absract Facer
• Protepite
Pretatape
Design
Pattens
Perrapior
Adepton
Abstact Moctory
Abterst Facttery
Pecopter
Singleor
• Factery
Cingleton
Singletal
Adetary
Factory
Pompace
Aberact
Fachory
Stompotive
Abstact Fathory
Comprase
Decorator
& Fatace
Betaracy
(Soompatse)
Procy
Deorac
Decorator
& Fatace
oecpage
Abtract Methory
Decrate
Decroote
Pretrape
Behavioral
Commapr
(Obecrater)
Stratje
Strtage
(Orterater)
& State
Sotreler
Deserater)
(State)
Sctetton
Oxsecraver
Ostarter
State
Bithatter
Stroter
Obsecaver
Crray
State
State
State


OBJECT-ORIENTED
PROGRAMMING



Programa académico CAMPUS

Módulo JAVA