

PATRONES ESTRUCTURALES Y DE COMPORTAMIENTO

UNIDAD 2: DISEÑO E IMPLEMENTACIÓN



Temario

- Patrones Estructurales: Adapter, Facade.
- Patrones de comportamiento: Strategy, Command.

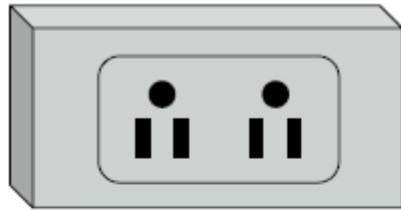
Patrón Estructural: Adapter

- **PROPÓSITO**

- Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes. El adaptador permite que las clases trabajen juntas que de otra manera no podrían debido a interfaces incompatibles.

Patrón Estructural: Adapter

CASO

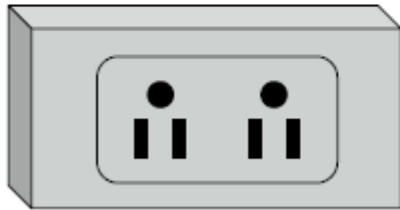


+

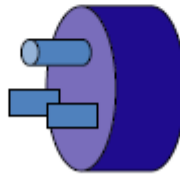


X

SOLUCIÓN



+



+



✓

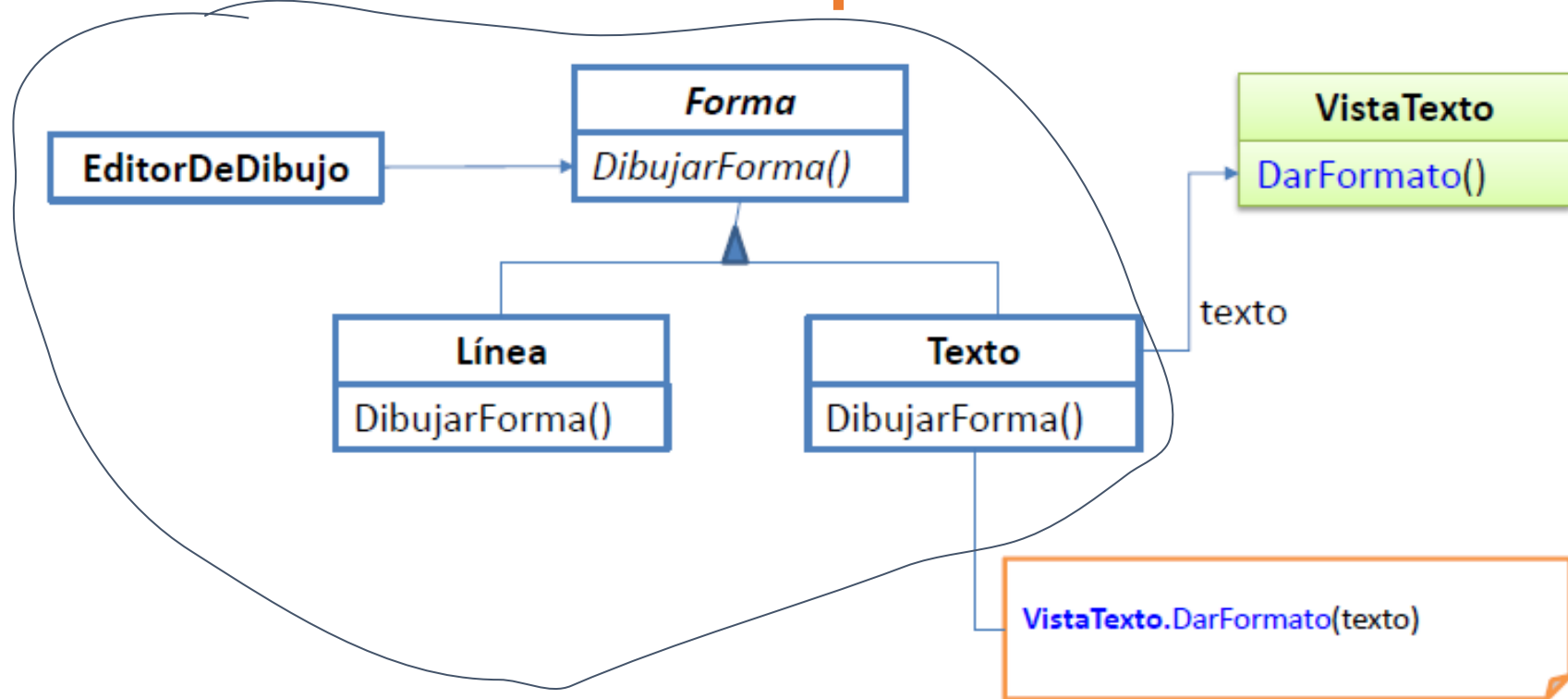
ADAPTADOR

Patrón Estructural: Adapter

- **MOTIVACIÓN**

- Una clase de un toolkit que ha sido rediseñada para reutilizarse, no puede hacerlo porque su interfaz no coincide con la interfaz específica del dominio que requiere la aplicación.

Patrón Estructural: Adapter



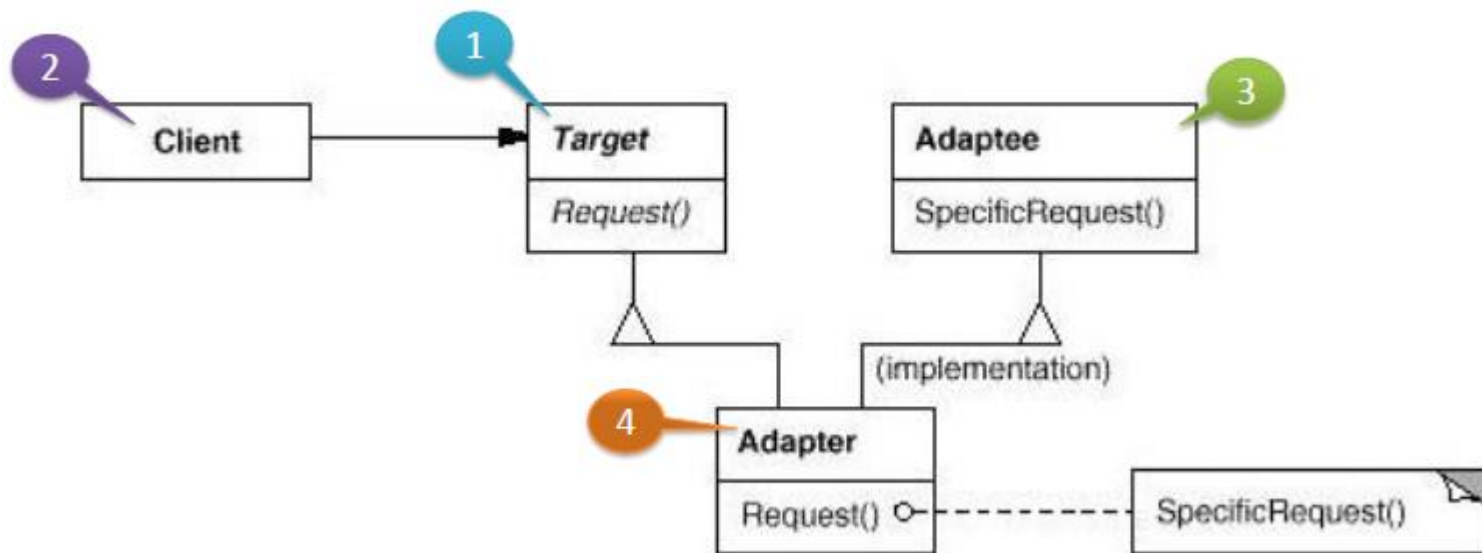
Patrón Estructural: Adapter

- **APLICACIONES**

- Usar el patrón cuando:
 - Se quiere usar una clase existente y su interfaz no concuerda con la que necesita.
 - Se desea crear una clase reutilizable que coopere con clases no relacionadas o que no se ha previsto que tengan interfaces compatibles.
 - Se necesita evitar la necesidad de cambiar el código cuando cambia una interfaz, o para permitir futuras modificaciones o implementaciones cuando diseña clases genéricas.

Patrón Estructural: Adapter

- ESTRUCTURA



Patrón Estructural: Adapter

- **PARTICIPANTES**

1. Target / Objetivo

- Define la interfaz específica del dominio que usa el cliente.

2. Client

- Colabora con los objetos conformando la interfaz del Target

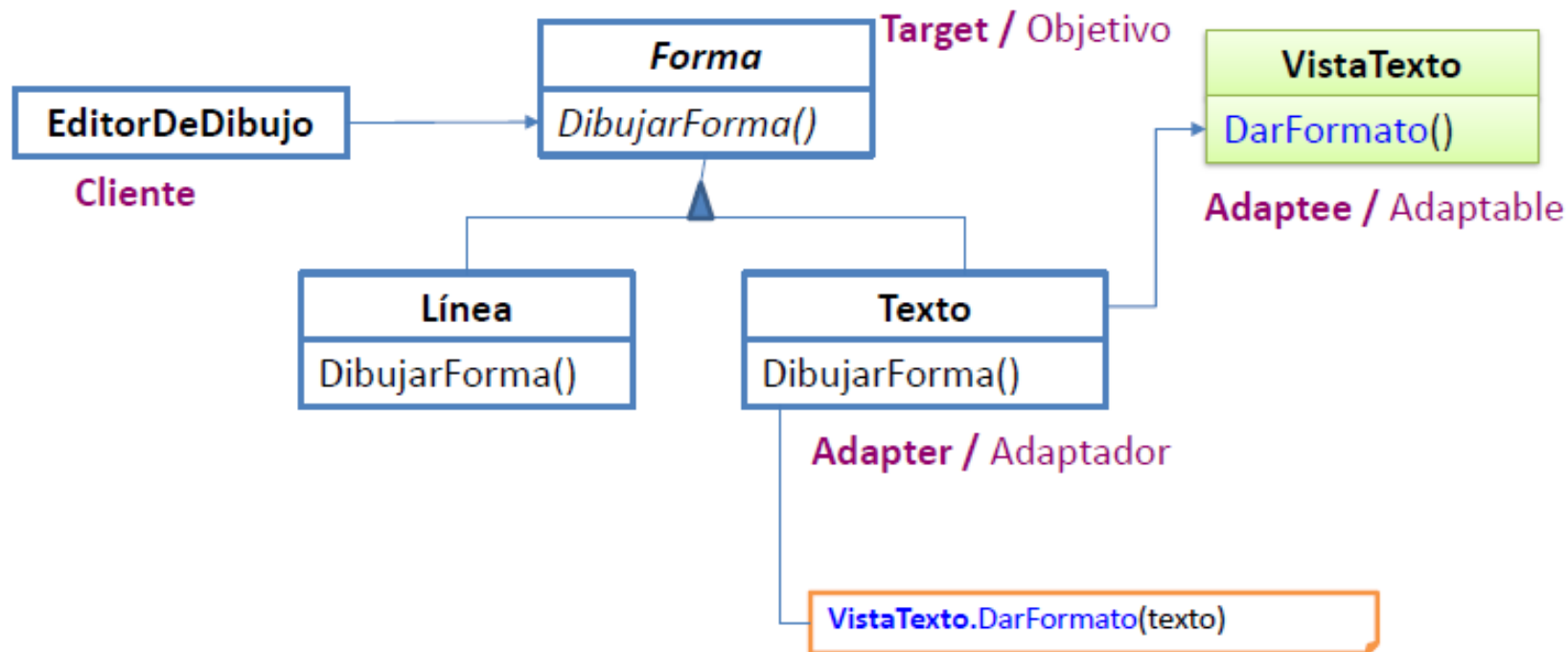
3. Adaptee (Adaptado)

- Define la interfaz existente que necesita adaptarse para aprovechar su funcionalidad específica.

4. Adapter (Adaptador)

- Adapta la interfaz Adaptee para usar en el objeto Target.

Patrón Estructural: Adapter



Patrón Estructural: Adapter

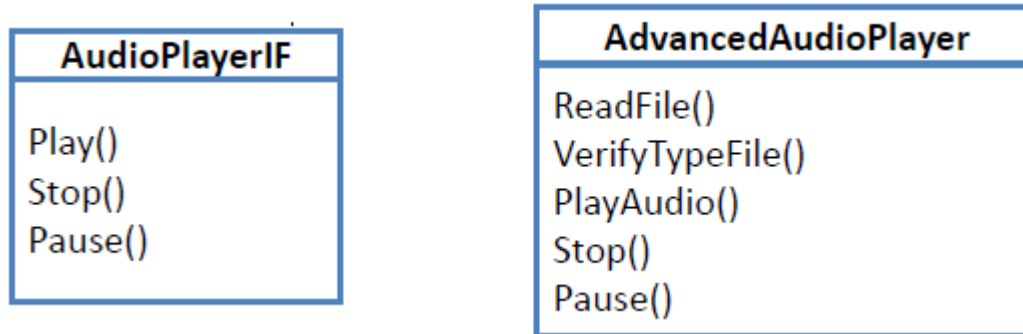
- **COLABORACIONES**

- Los clientes llaman a las operaciones de una instancia del Adapter. A su vez, el Adapter llama a operaciones del Adaptee, qué son las que satisfacen la petición.

Patrón Estructural: Adapter

- **EJERCICIO**

- Se tiene una interfaz para un reproductor de audio `AudioPlayerIF`.
- El cliente utiliza las implementaciones de la clase `AudioPlayerIF` (subclases) para reproducir música.
- Sin embargo, existe un reproductor de Audio Avanzado, `AdvancedAudioPlayer` que cuenta con mejores funciones.
- La aplicación cliente, desearía utilizar las ventajas del reproductor Avanzado sin tener que cambiar internamente, y sólo conoce la interface del `AudioPlayer` simple.



Patrón Estructural: Adapter

AudioPlayer (Simple)

```
package adapter;

public interface AudioPlayerIF {

    void play(String s);
    void stop();
    void pause();
}
```

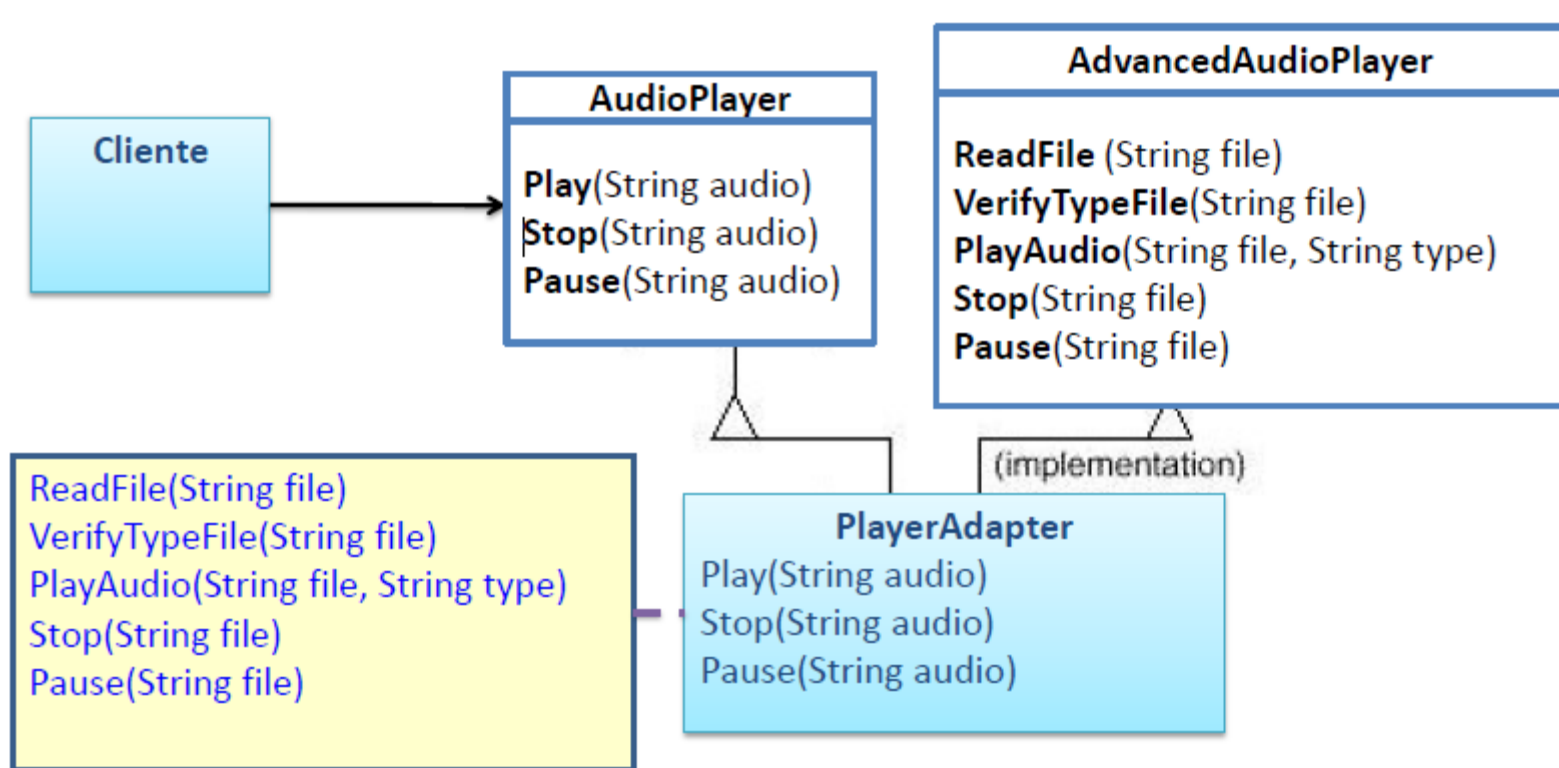
```
public class AudioPlayerMP3 implements AudioPlayerIF{
    @Override
    public void play(String s) {
        System.out.println("Reproduciendo MP3:" + s);
    }
    @Override
    public void stop() {
        System.out.println("Deteniendo reproducción");
    }
    @Override
    public void pause() {
        System.out.println("Pausando reproducción");
    }
}
```

```
public class Main {

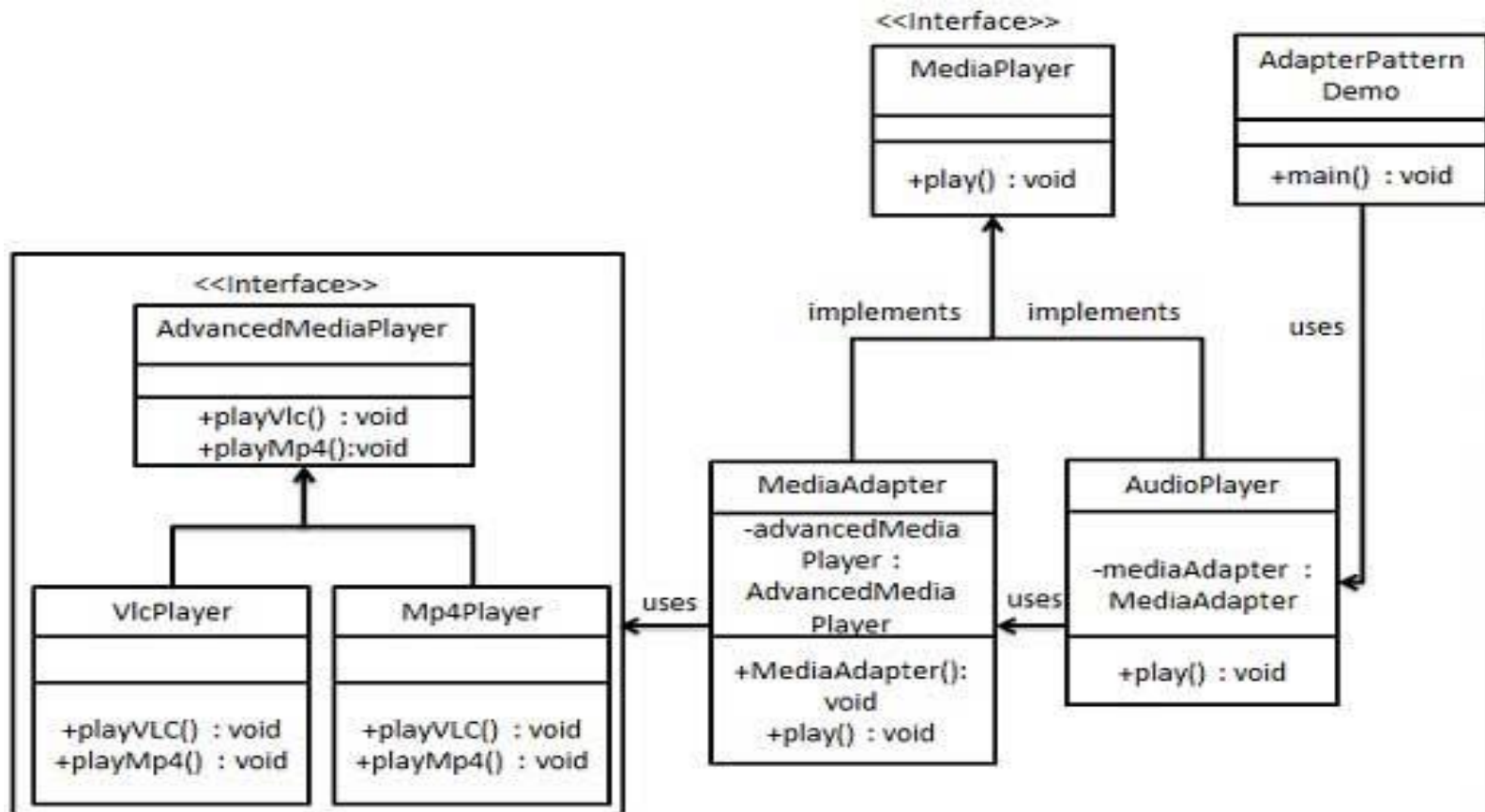
    public static void main(String[] args) {
        AudioPlayerMP3 apMP3 = new AudioPlayerMP3();
        apMP3.play("Rock.mp3");
        apMP3.pause();
        apMP3.stop();

        // TODO code application logic here
    }
}
```

Patrón Estructural: Adapter



Patrón Estructural: Adapter



Patrón Estructural: Adapter

AdvancedAudioPlayer

```
public void ReadFile(String s){
    System.out.println("Leyendo Archivo: " + s);
}

public String VerifyTypeFile(String audio){
    String tipo = audio.substring(audio.indexOf("."));
    System.out.println("Tipo de archivo: " + tipo);
    return tipo;
}

public void PlayAudio(String audio, String tipo){
    switch (tipo) {
        case ".mp3":
            //FUNCION DE REPRODUCCION DE Mp3
            System.out.println("Reproduciendo formato MP3: " + audio);
            break;
        case ".mp4":
            //FUNCION DE REPRODUCCION DE MP4
            System.out.println("Reproduciendo formato MP4: " + audio);
            break;
        default:
            System.out.println("Formato no Soportado");
            break;
    }
}
```

Soporta más extensiones!



Patrón Estructural: Facade

- **PROPÓSITO**
 - Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea fácil de usar.

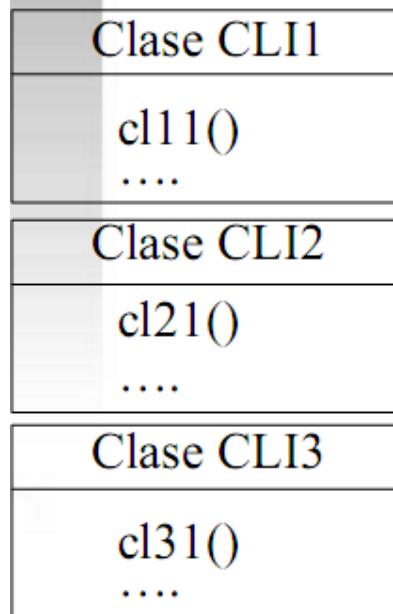
Patrón Estructural: Facade

- **MOTIVACIÓN**

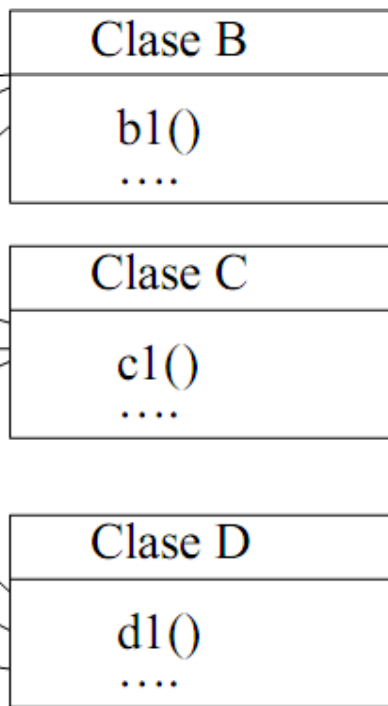
- Estructurar un sistema en subsistemas ayuda a reducir su complejidad.
- Se debe minimizar las comunicaciones y dependencias entre subsistemas.
- Un modo de conseguir esto es introducir un objeto FACADE que proporcione una interfaz única y simplificada a los servicios más generales del subsistema

Patrón Estructural: Facade

Clases CLIENTES



Clases SERVIDORAS



usa

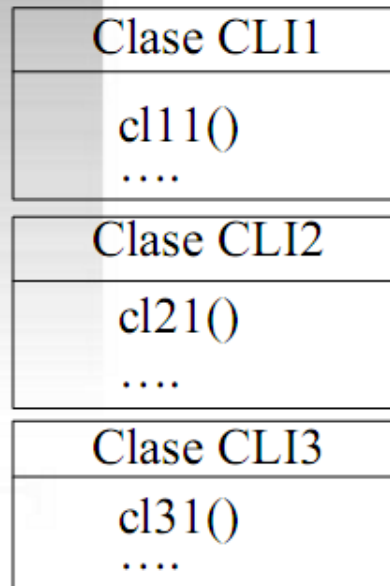
usa...

Problema: Además puede haber muchas clases cliente...



Patrón Estructural: Facade

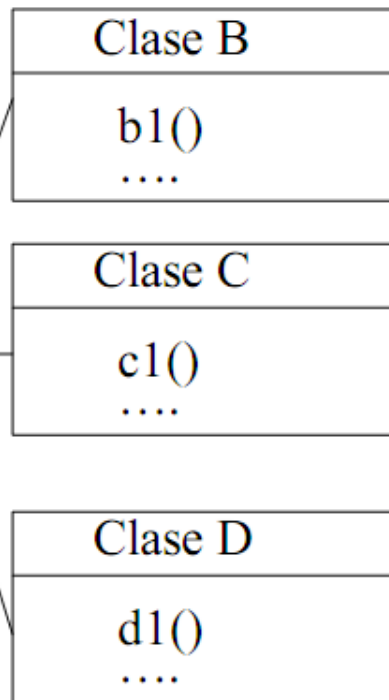
Clases CLIENTES



```
public class Facade {  
    B  objB = new B();  
    C  objC = new C();...  
    void b1() { objB.b1();}..}
```

Clase Facade
b1(), c1(), d1(),..

Clases SERVIDORAS



usa

usa

Solución: Proporcionar una clase que implemente todos los servicios (b1()...). Los clientes sólo usarán dicha clase.



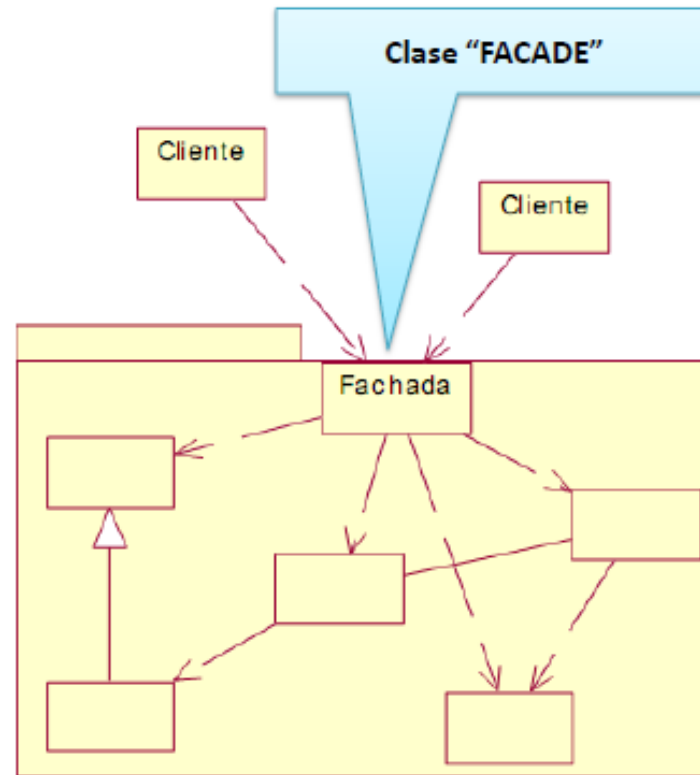
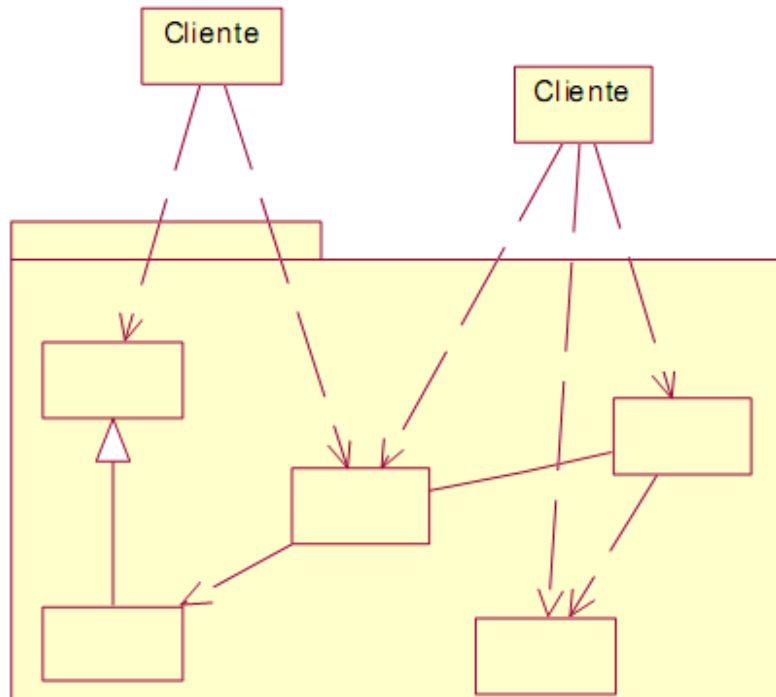
Patrón Estructural: Facade

- **APLICACIONES**

- Usar el patrón cuando:
 - Necesitamos proporcionar una interfaz simple a un subsistema complejo.
 - Sólo los clientes que necesitan más personalización necesitarán ir más allá de la fachada.
 - Existan muchas dependencias entre los clientes y las clases que implementan una abstracción. La fachada desacopla el subsistema de sus clientes y otros subsistemas (mejora acoplamiento y portabilidad).
 - Queramos dividir en capas nuestros subsistemas. La fachada es el punto de entrada a cada subsistema.

Patrón Estructural: Facade

- ESTRUCTURA**



Patrón Estructural: Facade

- **PARTICIPANTES**

1. Facade (Fachada)

- Clase que centraliza las comunicaciones a los subsistemas.
- Sabe qué clases del subsistema son las responsables por una petición.
- Delega las peticiones de los clientes en los objetos apropiados del subsistema.

2. Clases del Subsistema

- Implementan la funcionalidad del subsistema
- Realizan las labores encomendadas por el objeto Facade.

Patrón Estructural: Facade

- **COLABORACIONES**

- Los clientes se comunican con el subsistema enviando peticiones al objeto Facade, el cual las reenvía a los objetos apropiados del subsistema.
- Los clientes que usan el objeto Facade no tienen que acceder directamente a los objetos del subsistema.

Patrón Estructural: Facade

- **CONSECUENCIAS**

- Oculta a los clientes los componentes del subsistema.
 - Reduce el número de objetos con los que tienen que tratar los clientes.
- Disminuye el acoplamiento entre un subsistema y sus clientes.
 - Un menor acoplamiento facilita el cambio de los componentes del subsistema sin afectar a sus clientes.
 - Las fachadas permiten estructurar el sistema en capas

Patrón Estructural: Facade

- **Ejemplo 1: Real-Life Example**

- Suppose you are going to organize a birthday party and you have invited 100 people. Nowadays, you can go to any party organizer and let him/her know the minimum information—(party type, date and time of the party, number of attendees, etc.). The organizer will do the rest for you. You do not even think about how he/she will decorate the party room, whether people will take food from self-help counter or will be served by a caterer, and so on.

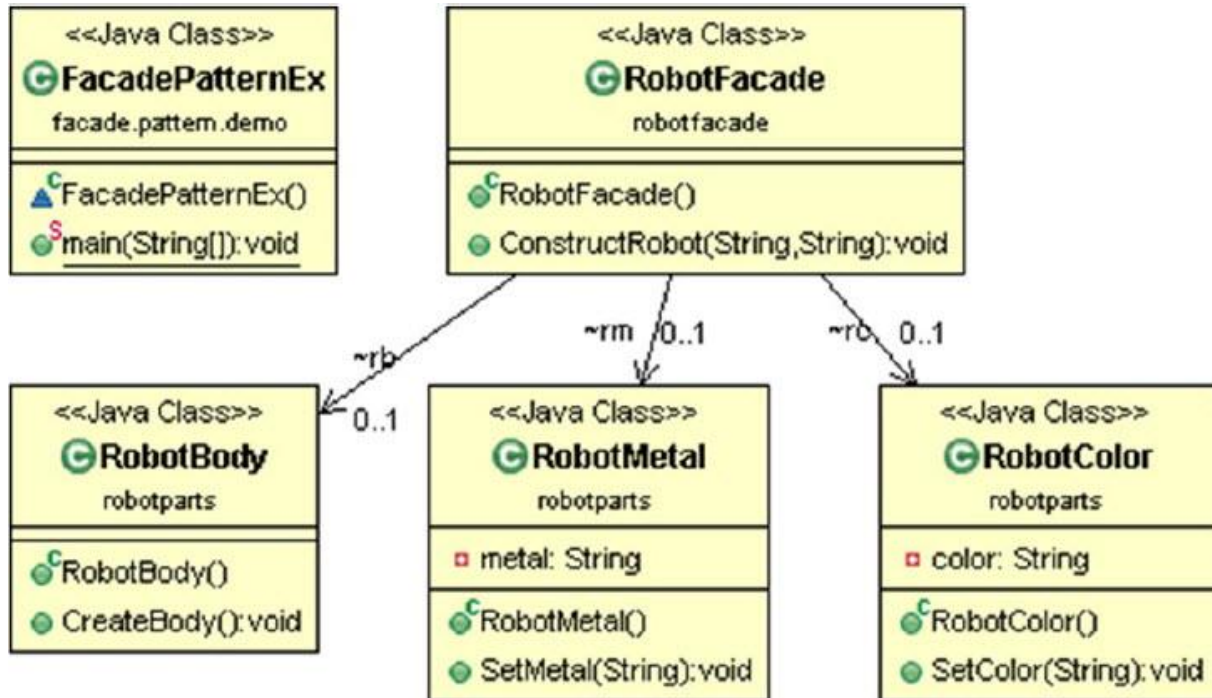
Patrón Estructural: Facade

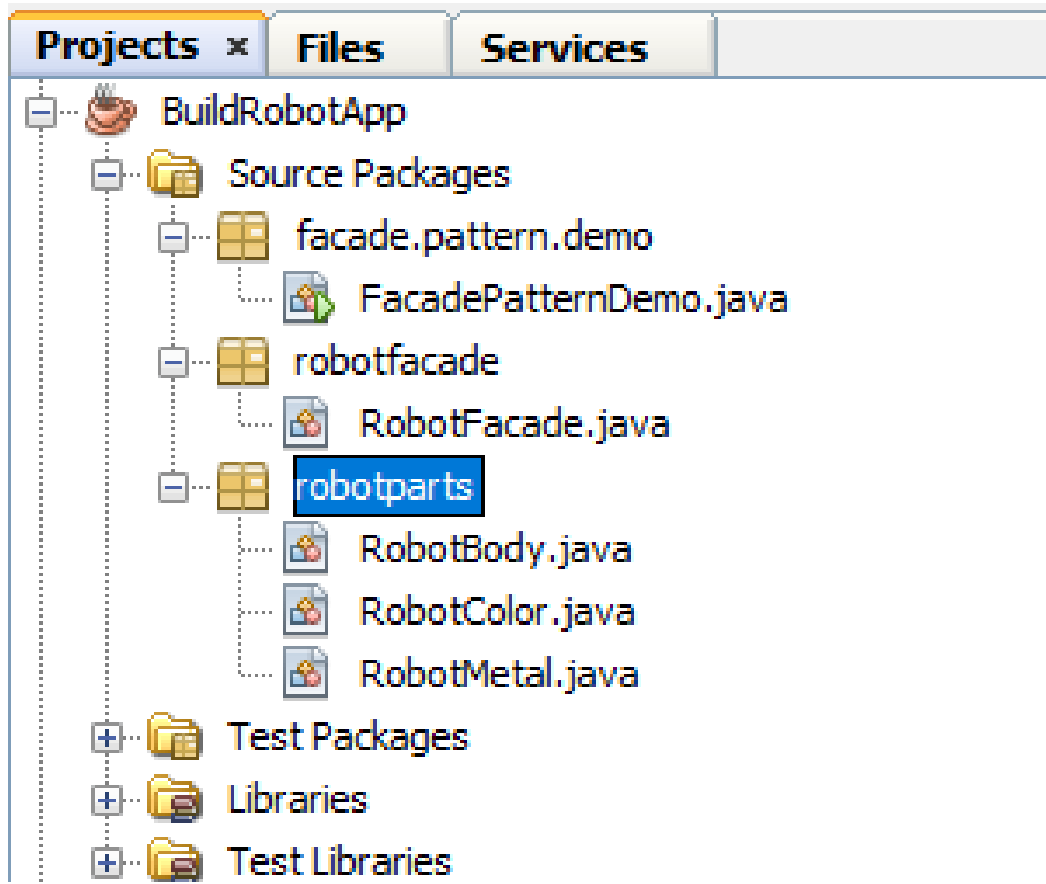
- **Ejemplo 2: Computer World Example**

- We can think about a case where we use a method from a library. The user doesn't care how the method is implemented in the library. He/she just calls the method to serve his/her easy purpose. The pattern can be best described by the example that follows.
 - Here our aim is to build/construct robots. And from a user point of view he/she needs to supply only the color and material for his/her robot through the RobotFacade (See our FacadePatternEx.java file.) Our RobotFacade (RobotFacade.java) will in turn create objects for RobotBody, RobotColor, RobotMetal and will do the rest for the user. We need not worry about the creation of these separate classes and their calling sequence. All of the classes have their corresponding implementation here.

Patrón Estructural: Facade

- Ejemplo 2: Computer World Example





```
1 package robotparts;  
2 public class RobotBody {  
3     public void createBody() {  
4         System.out.println("Body creation done");  
5     }  
6 }  
7
```

RobotBody.java x RobotColor.java x

Source


History



```
1
2 package robotparts;
3
4 public class RobotColor {
5     private String color;
6     public void setColor(String color){
7         this.color = color;
8         System.out.println("Color is set to: "+this.color);
9     }
10 }
11
```

RobotBody.java x RobotColor.java x RobotMetal.java x

Source History



1

package robotparts;

2

public class RobotMetal {

3

private String metal;

4

public void setMetal(String metal){

5

this.metal = metal;

6

System.out.println("Metal is set to: "+this.metal);

7

}

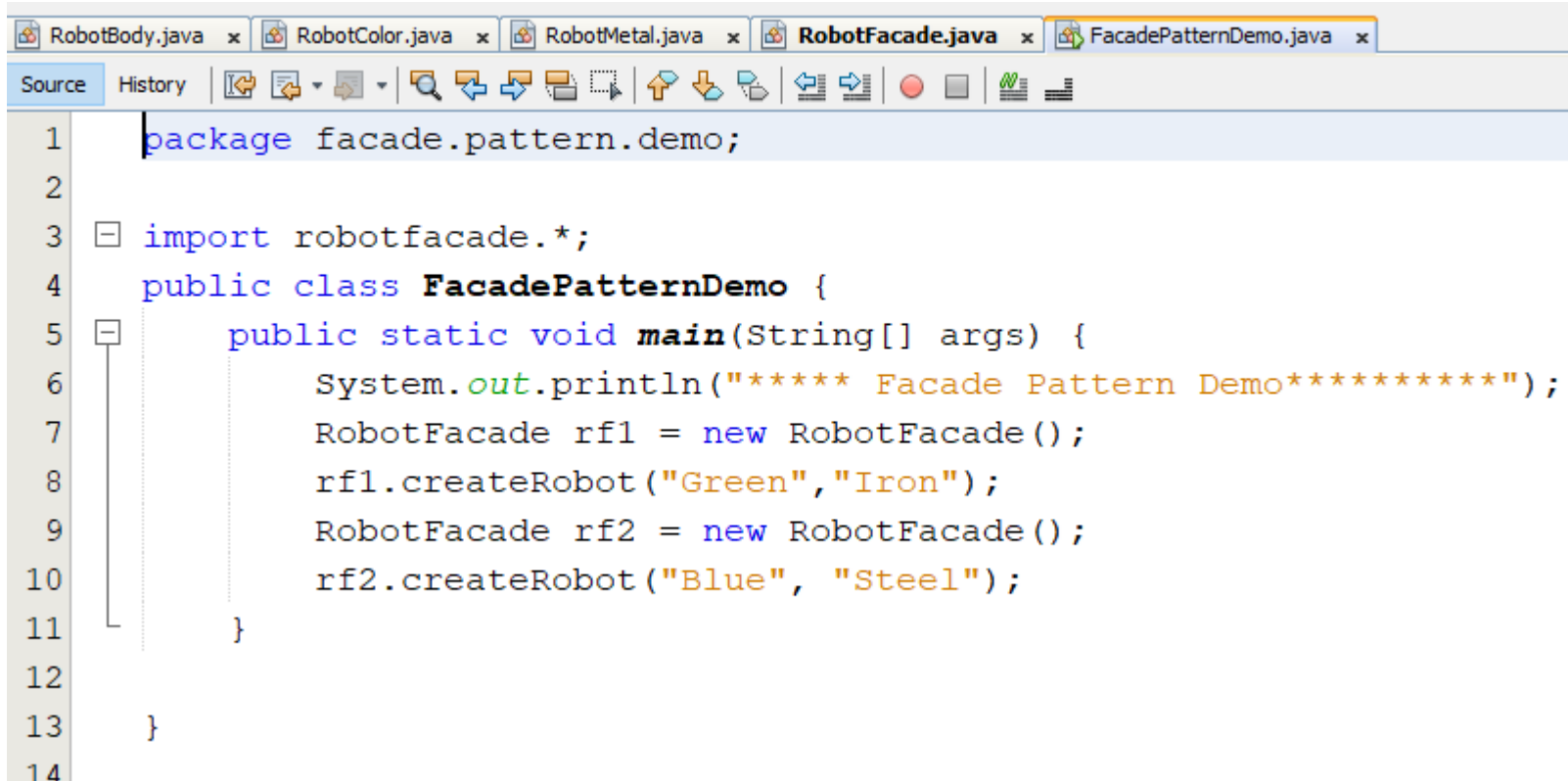
8

}

9



```
1 package robotfacade;
2 import robotparts.*;
3 public class RobotFacade {
4     private RobotBody rb;
5     private RobotColor rc;
6     private RobotMetal rm;
7     public RobotFacade() {
8         rb = new RobotBody();
9         rc = new RobotColor();
10        rm = new RobotMetal();
11    }
12    public void createRobot(String color, String metal) {
13        System.out.println("\nCreation of the robot start");
14        rc.setColor(color);
15        rm.setMetal(metal);
16        rb.createBody();
17        System.out.println(" \nRobot creation end");
18        System.out.println();
19    }
20 }
```



```
1 package facade.pattern.demo;
2
3 import robotfacade.*;
4 public class FacadePatternDemo {
5     public static void main(String[] args) {
6         System.out.println("***** Facade Pattern Demo*****");
7         RobotFacade rf1 = new RobotFacade();
8         rf1.createRobot("Green", "Iron");
9         RobotFacade rf2 = new RobotFacade();
10        rf2.createRobot("Blue", "Steel");
11    }
12
13 }
14
```

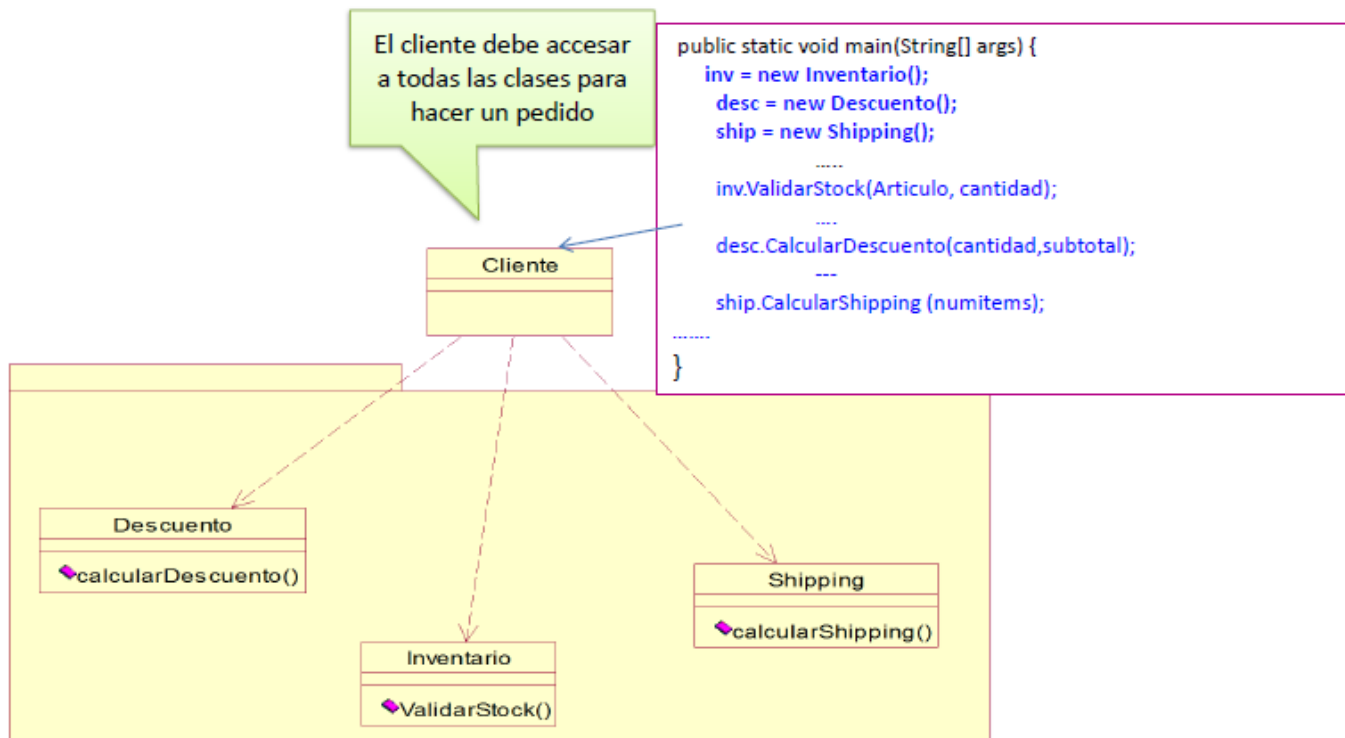
Patrón Estructural: Facade

- **Ejemplo 2**

- Página de compra por internet
- Las clases son:
 - Items: Clase cuyos objetos almacenarán la información de un artículo: nombre, cantidad, precio.
 - Inventario: Donde existirá la lista de productos y su stock disponible y la lógica para verificar su disponibilidad.
 - Shipping: Donde, de acuerdo a la cantidad de ítems, agregará el costo de envío.
 - Descuento: Clase que tiene la funcionalidad de calcularDescuento. Si se compra más de 10 items, se aplica el 10% de descuento del total.
 - Cliente: Formulario.

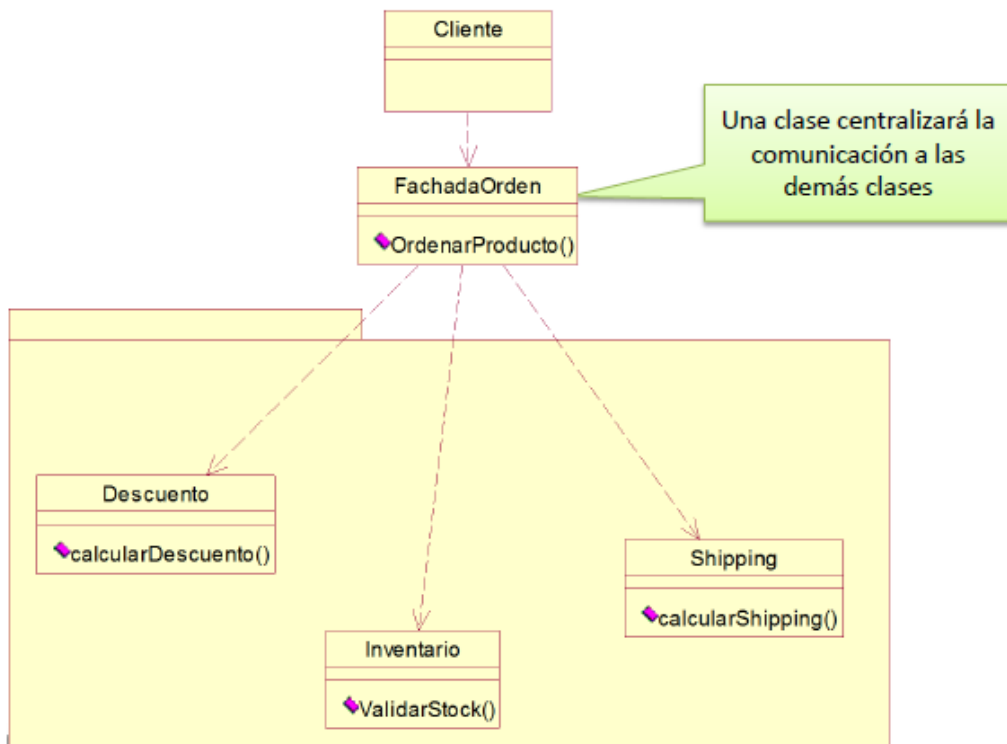
Patrón Estructural: Facade

- Ejemplo



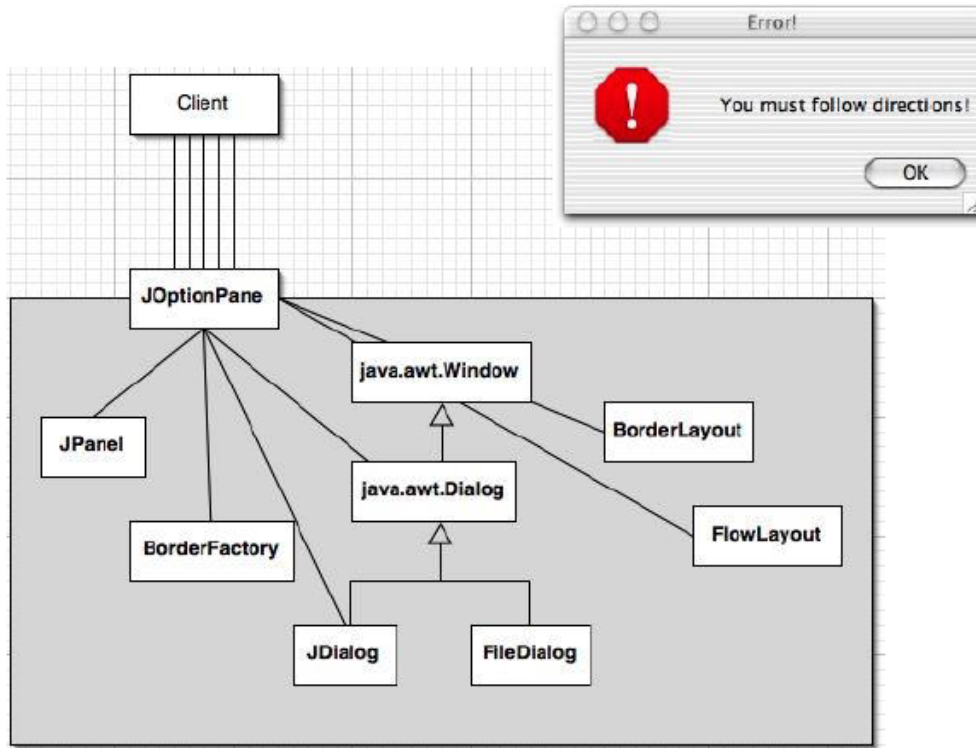
Patrón Estructural: Facade

- Ejemplo



Patrón Estructural: Facade

- Ejemplo



Patrón Estructural: Facade

- **EJERCICIO 1**

- Considere una aplicación que simule el proceso de retirar dinero de un cajero automático.
- Se requiere un gran número de operaciones para garantizar la seguridad y la integridad.
- Clases que intervienen:
 - Autenticación: Leer tarjeta, introducir clave, comprobar clave, ObtenerCuenta,
 - Cajero: Introducir cantidad, Verificar Saldo, ExpedirDinero, Imprimir Ticket.
 - Cuenta: comprobarSaldoDisponible, bloquearCuenta, retirarSaldo, actualizarCuenta, etc.

Patrón Estructural: Decorator

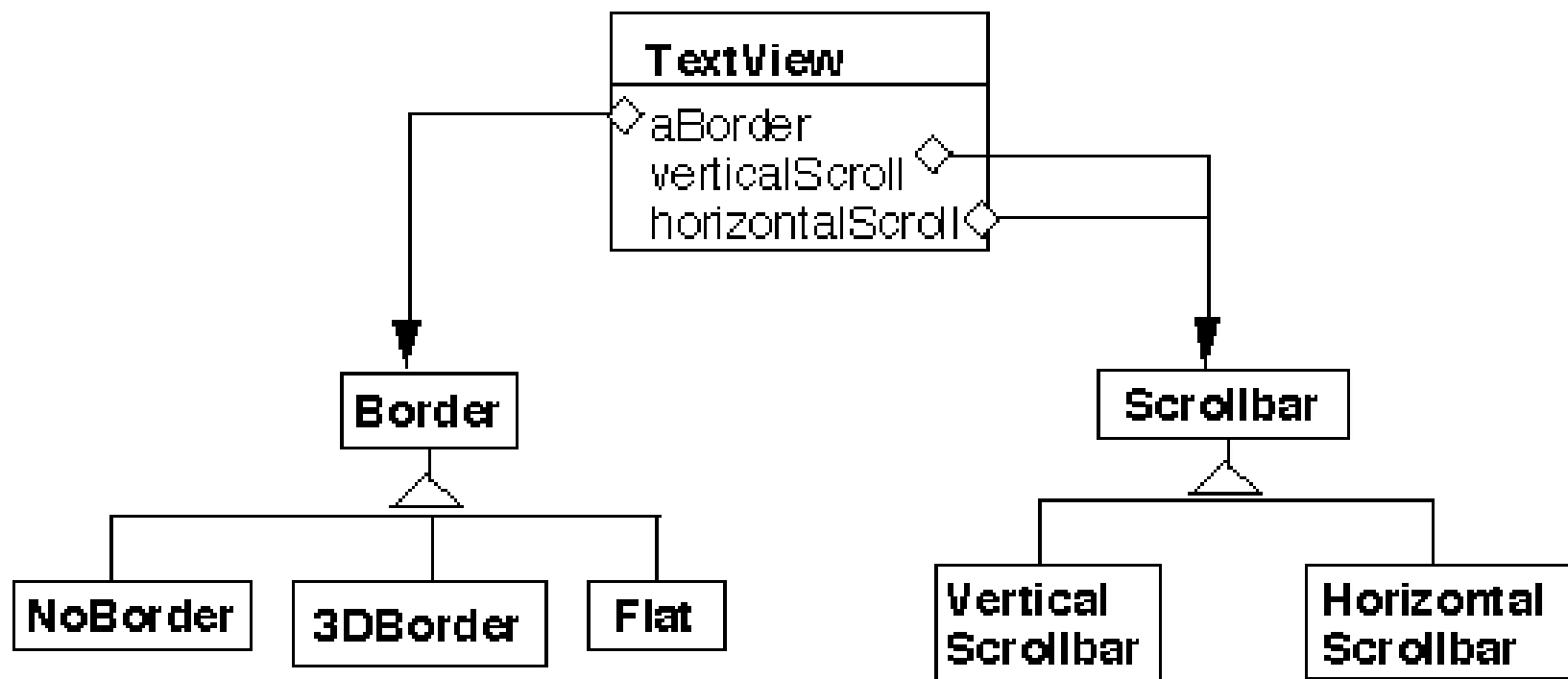
- **PROPÓSITO**
 - Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Patrón Estructural: Decorator

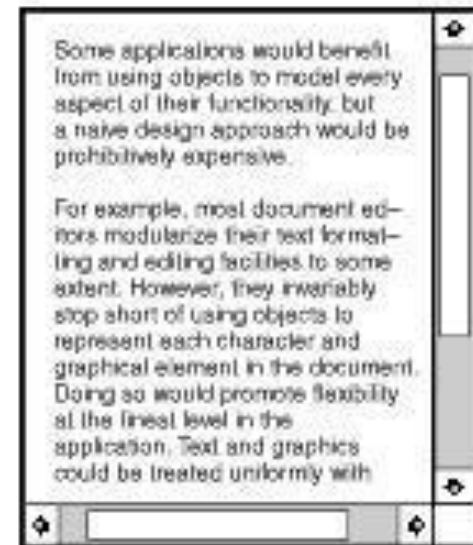
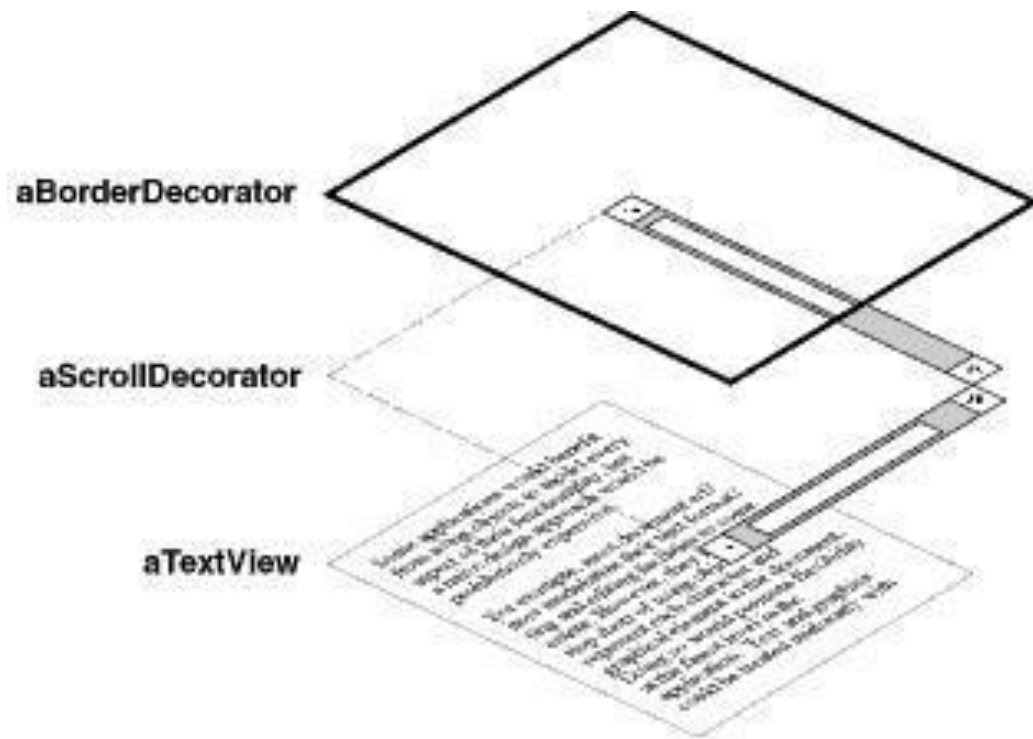
- **MOTIVACIÓN**

- Disponemos de una herramienta para crear interfaces gráficas, que permite añadir funcionalidades como bordes o barras de desplazamiento a cualquier componente de la interfaz.
- Una posible solución sería utilizar la herencia para extender las responsabilidades de la clase. Si optamos por esta solución, estaríamos haciendo un diseño inflexible (estático), ya que el cliente no puede controlar cuando y como decorar el componente con esa propiedad.
- La solución está en encapsular dentro de otro objeto, llamado Decorador, las nuevas responsabilidades. El decorador redirige las peticiones al componente y, además, puede realizar acciones adicionales antes y después de la redirección. De este modo, se pueden añadir decoradores con cualidades añadidas recursivamente.

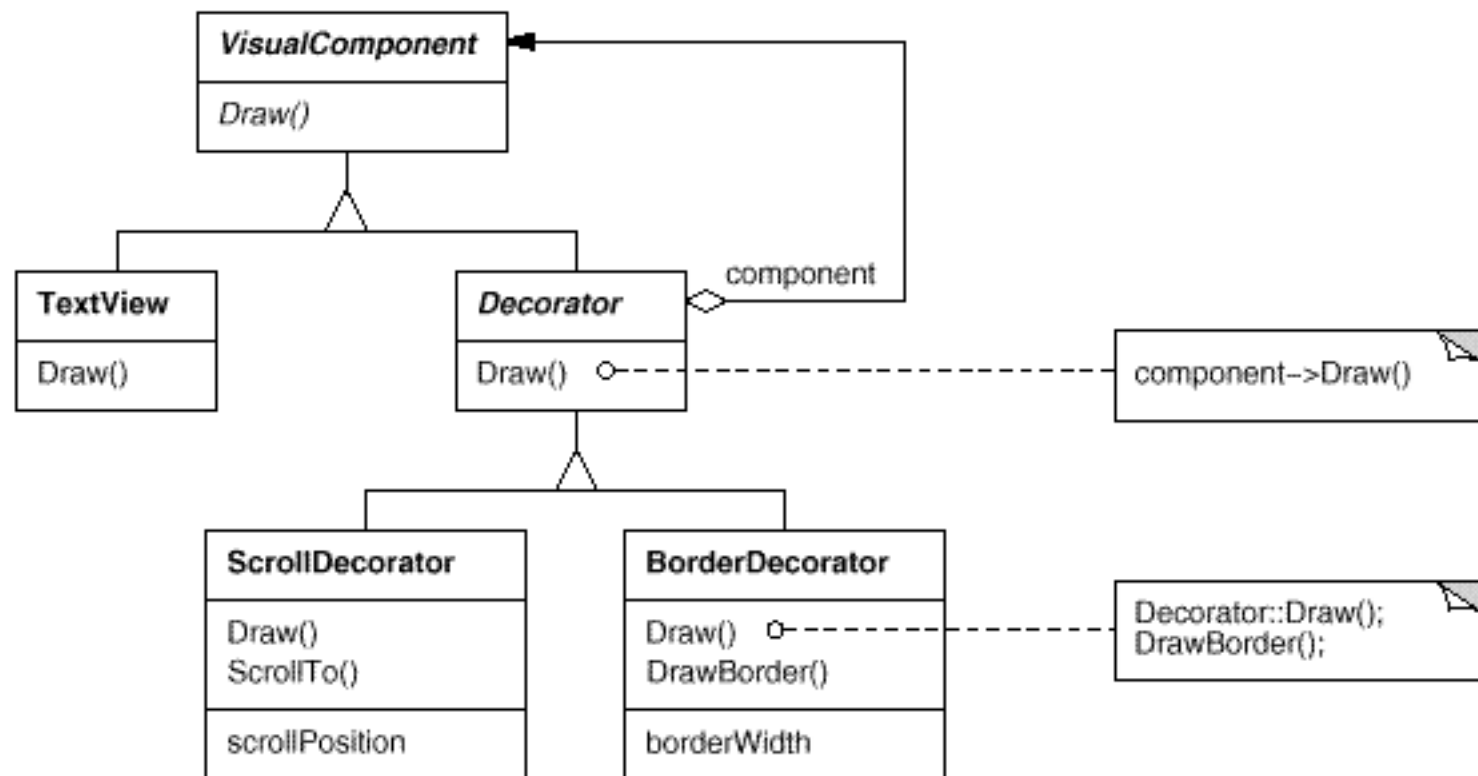
Patrón Estructural: Decorator



Patrón Estructural: Decorator



Patrón Estructural: Decorator



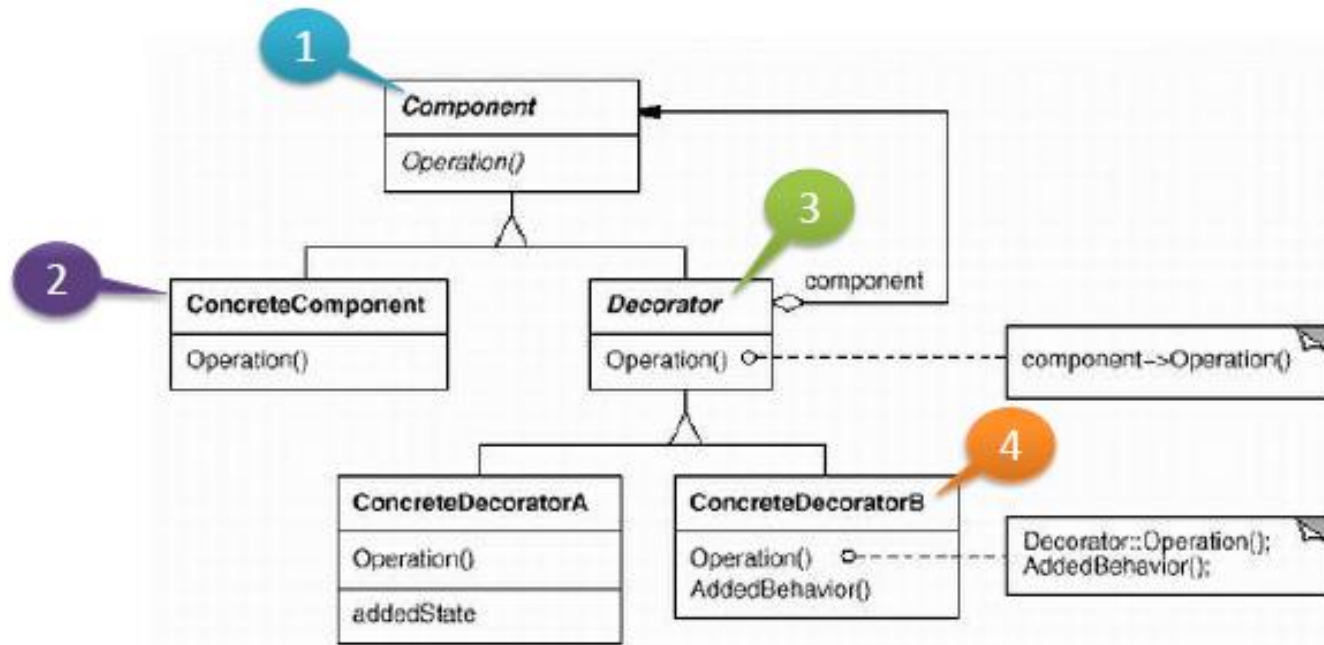
Patrón Estructural: Decorator

- **APLICACIONES**

- Añadir objetos individuales de forma dinámica y transparente
- Responsabilidades de un objeto pueden ser retiradas
- Cuando la extensión mediante la herencia no es viable.

Patrón Estructural: Decorator

- ESTRUCTURA



Patrón Estructural: Decorator

- **PARTICIPANTES**

1. **Componente.** Define la interfaz para los objetos que pueden tener responsabilidades añadidas.
2. **Componente Concreto.** Define un objeto al cual se le pueden agregar responsabilidades adicionales.
3. **Decorator.** Mantiene una referencia al componente asociado. Implementa la interfaz de la superclase Componente delegando en el componente asociado.
4. **Decorator Concreto.** Añade responsabilidades al componente

Patrón Estructural: Decorator

- **COLABORACIONES**

- El Decorator redirige peticiones a su objeto Componente. Opcionalmente puede realizar operaciones adicionales antes y después de reenviar la petición.

Patrón Estructural: Decorator

- **CONSECUENCIAS**

- Más flexible que la herencia. Al utilizar este patrón, se pueden añadir y eliminar responsabilidades en tiempo de ejecución. Además, evita la utilización de la herencia con muchas clases y también, en algunos casos, la herencia múltiple.
- Evita la aparición de clases con muchas responsabilidades en las clases superiores de la jerarquía. Este patrón nos permite ir incorporando de manera incremental responsabilidades.
- Genera gran cantidad de objetos pequeños. El uso de decoradores da como resultado sistemas formados por muchos objetos pequeños y parecidos.

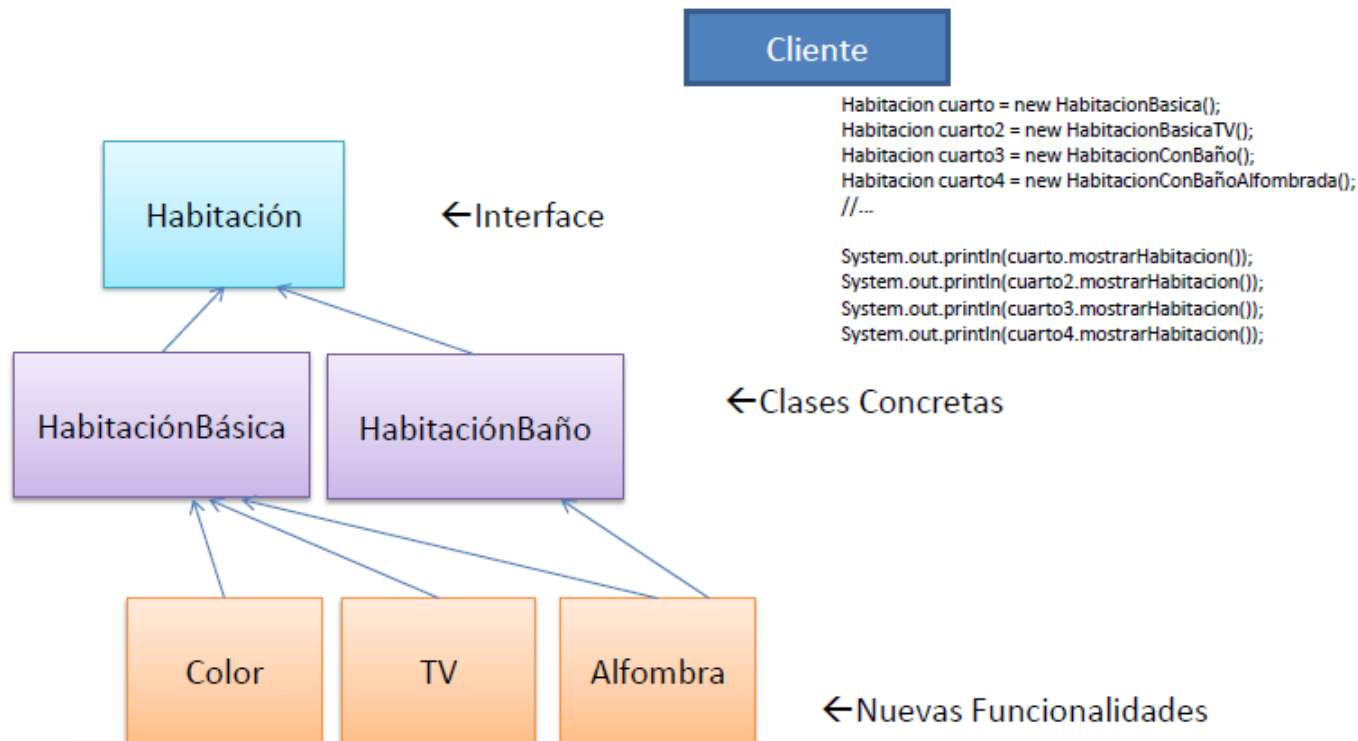
Patrón Estructural: Decorator

- **EJEMPLO**

- Tenemos una aplicación que simula la creación de habitaciones de un hotel.
- Las habitaciones son simples y puede solicitarse que sean alfombradas, de algún color en especial, que tengan cortinas oscuras y/o que tengan TV.
- Tendríamos que crear una subclase específica o trabajar con herencias por cada tipo de combinación posible. Sin embargo, la combinación elegida debería ser en tiempo de ejecución.

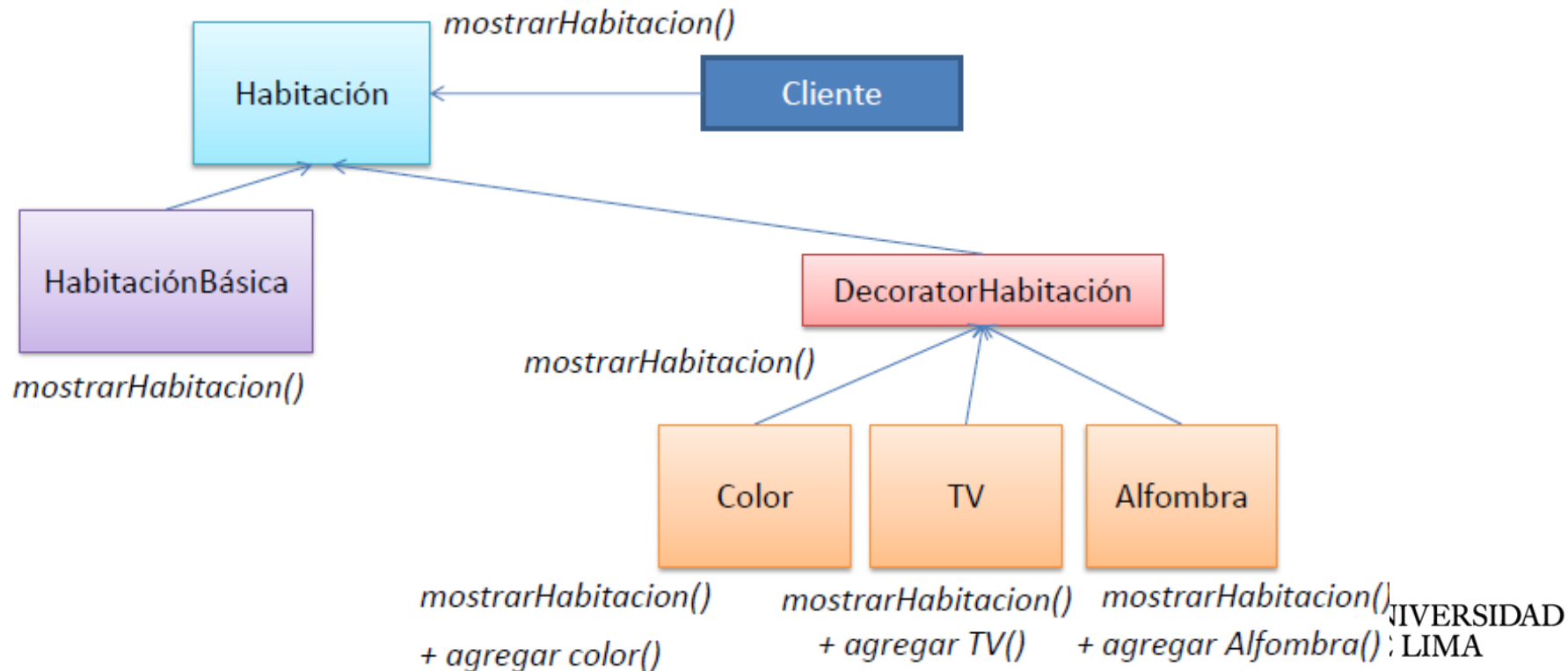
Patrón Estructural: Decorator

- EJEMPLO**

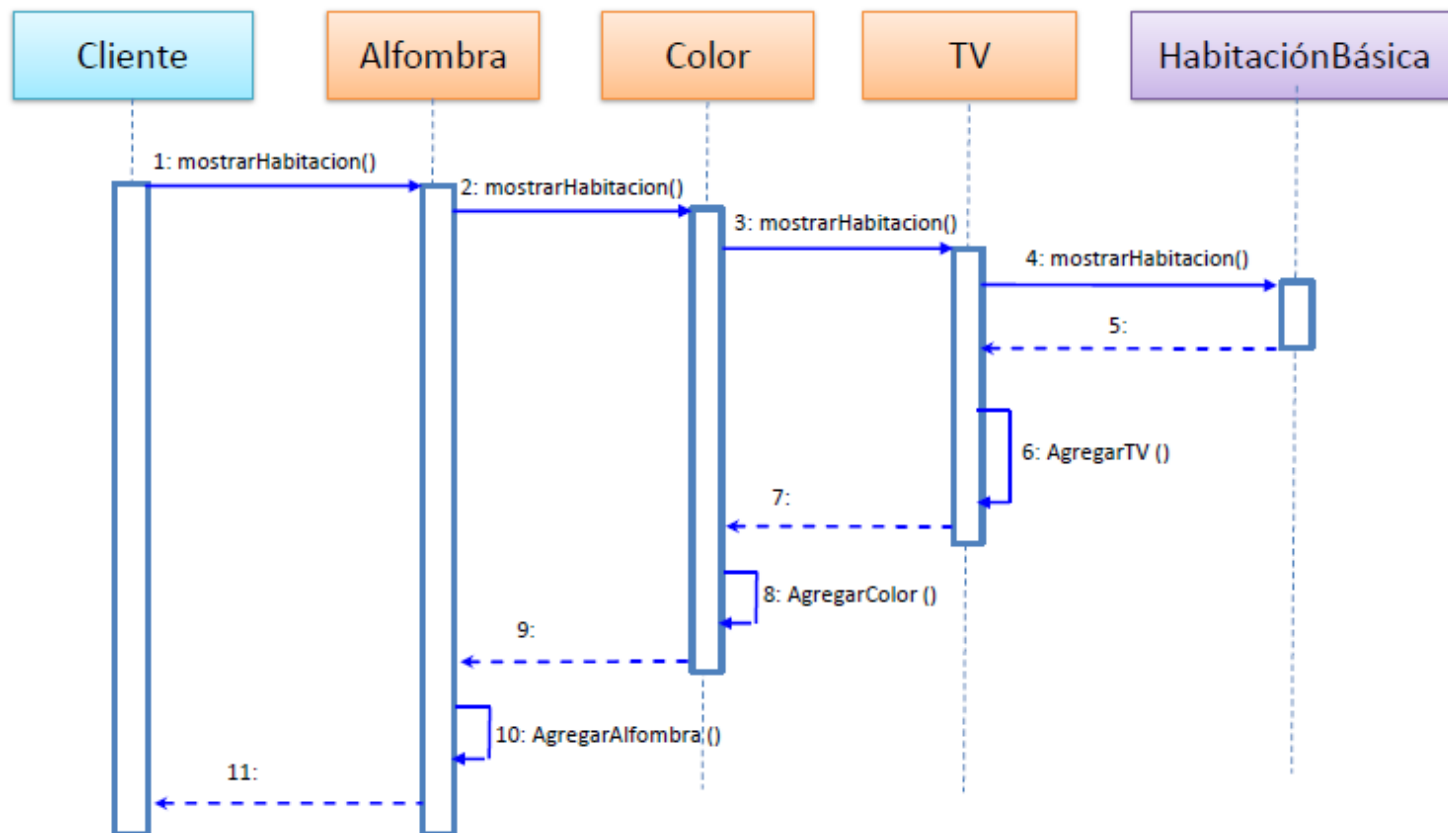


Patrón Estructural: Decorator

- EJEMPLO**

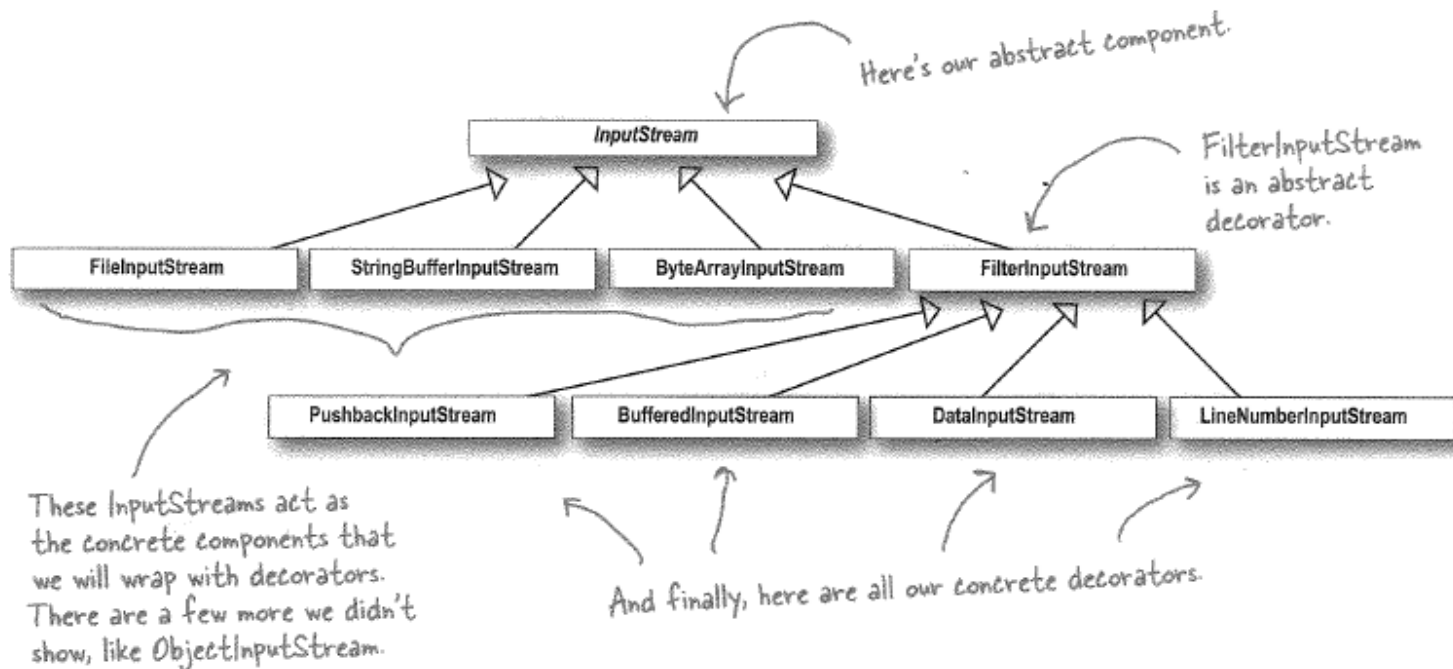


Patrón Estructural: Decorator



Patrón Estructural: Decorator

En Java



Patrón Estructural: Bridge

- **PROPÓSITO**
 - Desacoplar una abstracción de su implementación, de manera que ambas puedan ser modificadas independientemente.

Patrón Estructural: Bridge

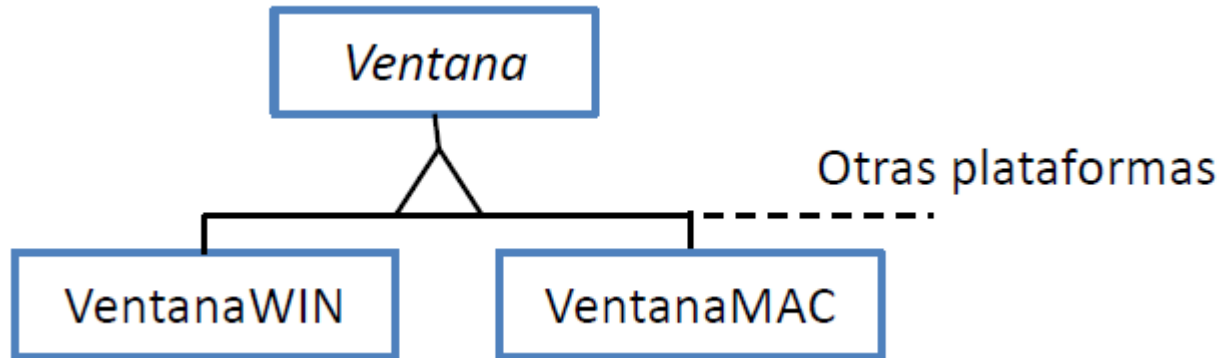
- **MOTIVACIÓN**

- La herencia permite que una abstracción tenga varias implementaciones
- Una clase abstracta define la interfaz a la abstracción y la aplicación de las subclases concretas en diferentes maneras.
- No siempre es lo suficientemente flexible, obligando a la herencia a estar ligada a la implementación, por lo que resulta difícil de modificar, ampliar, y usar la reutilización de abstracciones e implementaciones independiente.

Patrón Estructural: Bridge

- **MOTIVACIÓN**

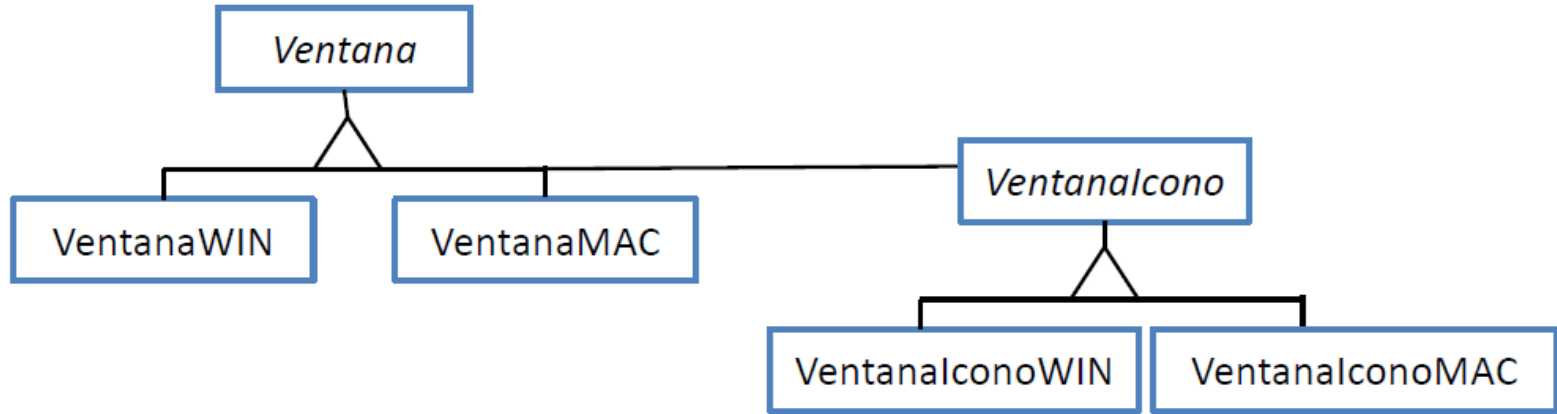
- Tenemos un framework de interfaces de usuario.
- La clase Ventana es una clase abstracta que nos permite crear ventanas en diferentes plataformas (Windows, MAC OS) mediante la herencia.



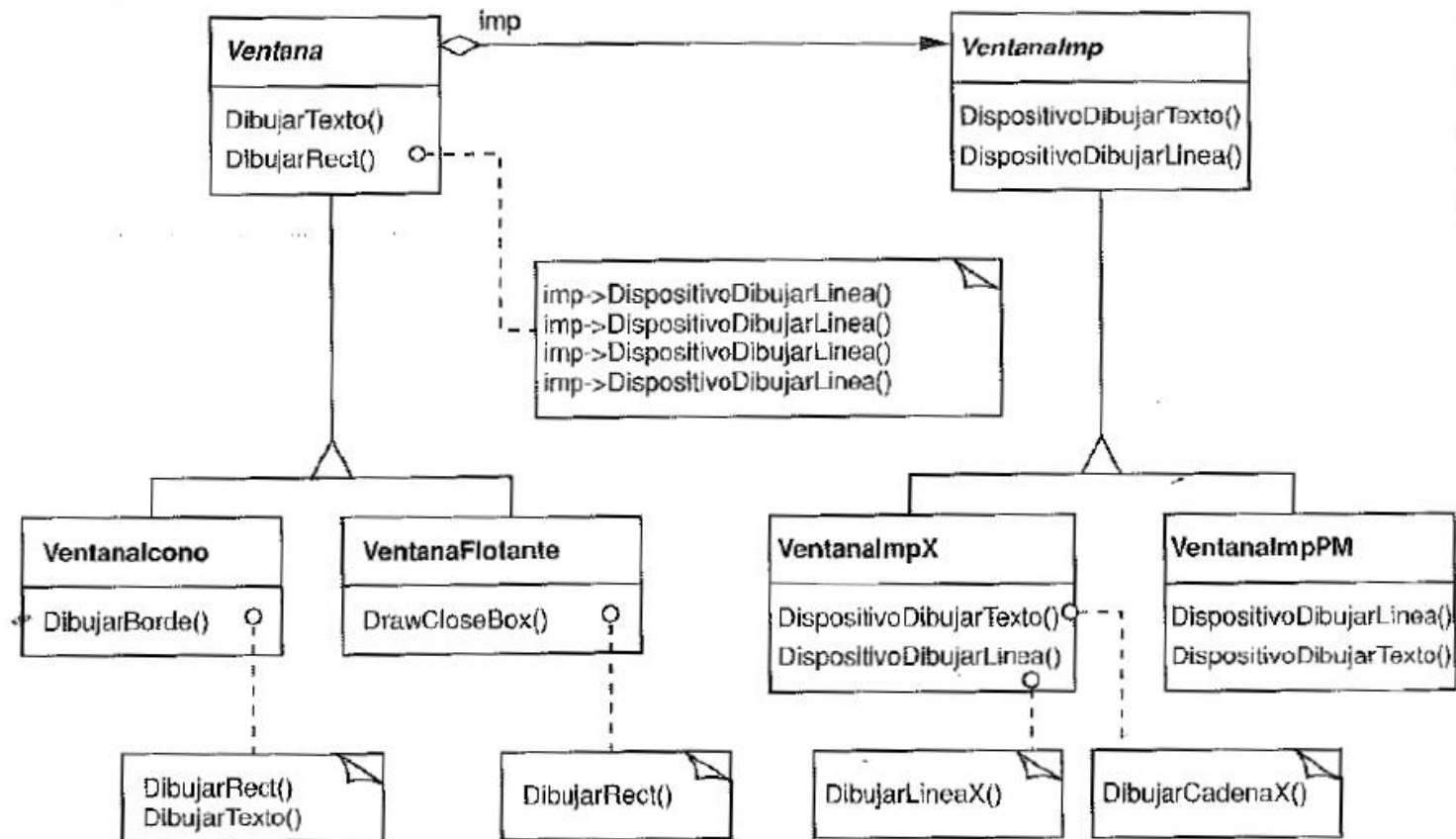
Patrón Estructural: Bridge

- **MOTIVACIÓN**

- Imaginemos que a su vez, existen distintos tipos de ventana: VentanaIcono (para representar íconos), VentanaFlotante, quienes a su vez, deberían poder crearse para distintas plataformas como Windows y Mac.



Patrón Estructural: Bridge



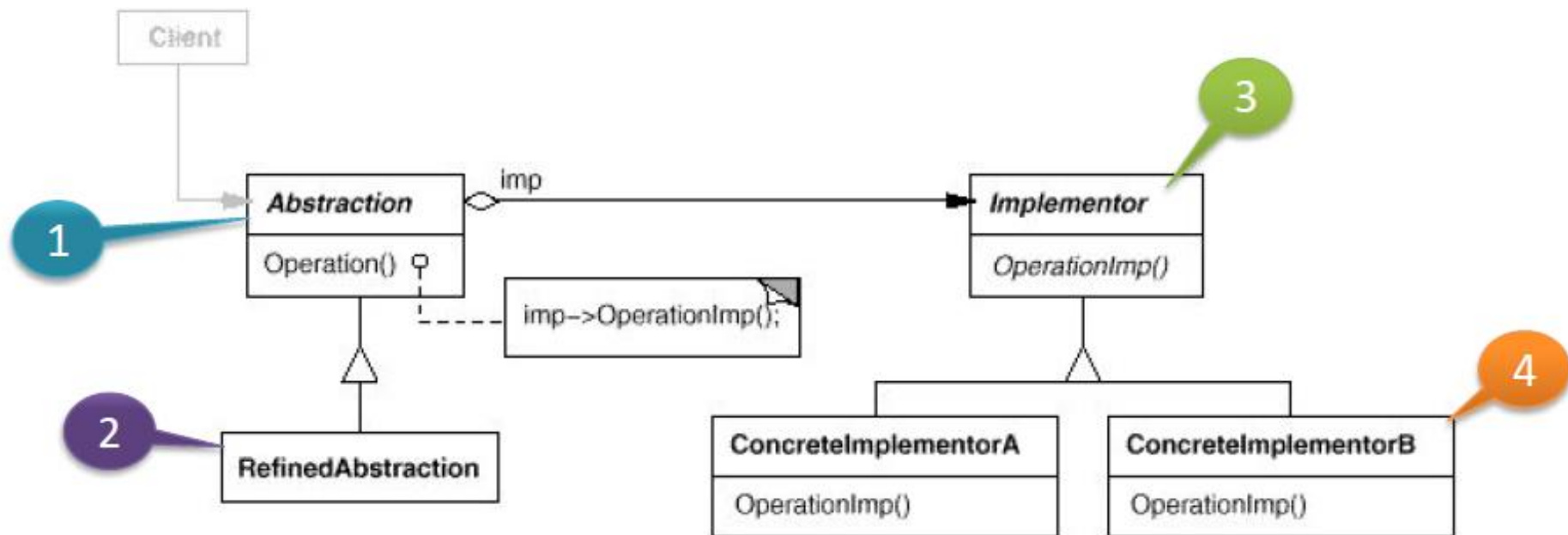
Patrón Estructural: Bridge

- **APLICACIONES**

- Se desea evitar un enlace permanente entre la abstracción y su implementación. Esto debido a que la implementación debe ser seleccionada o cambiada en tiempo de ejecución.
- Tanto las abstracciones como sus implementaciones deben ser extensibles por medio de subclases. En este caso, el patrón Bridge permite combinar abstracciones e implementaciones diferentes y extenderlas independientemente.
- Cambios en la implementación de una abstracción no deben impactar en los clientes.
- Se desea compartir una implementación entre múltiples objetos, y este hecho debe ser escondido a los clientes.

Patrón Estructural: Bridge

- ESTRUCTURA



Patrón Estructural: Bridge

- **PARTICIPANTES**

1. **Abstracción:** define la interface de la abstracción, mantiene referencia a un objeto de tipo implementor.
2. **RefinedAbstraction:** extiende la interface definida por Abstracción.
3. **Implementor:** define la interface para la implementación de clases. Esta interface no tiene que corresponder exactamente con la de la Abstracción.
4. **Concrete Implementor:** implementa la interfaz de Implementor y define su implementación concreta.

```
interface Implementor{  
    public abstract void operacionImpl();  
}  
  
/** primera implementacion de Implementador */  
class ImplementacionA implements Implementor{  
    public void operacionImpl() {  
        System.out.println("Esta es la implementacion A");  
    }  
}  
  
/** segunda implementacion de Implementador */  
class ImplementacionB implements Implementor{  
    public void operacionImpl() {  
        System.out.println("Esta es una implementacion de B");  
    }  
}
```

```
interface Abstraccion {
```

```
    public void operacion();  
}
```

```
/** clase refinada que implementa la abstraccion **/
```

```
class AbstraccionRefinada implements Abstraccion{  
    private Implementador implementador;  
    public AbstraccionRefinada(Implementador implementador){  
        this.implementador = implementador;  
    }  
    public void operacion(){  
        implementador.operacionImpl();  
    }  
}
```


En el cliente

....

```
Abstraccion a = new AbstraccionRefinada(new ImplementacionA());  
a.operacion();|
```

Patrón Estructural: Bridge

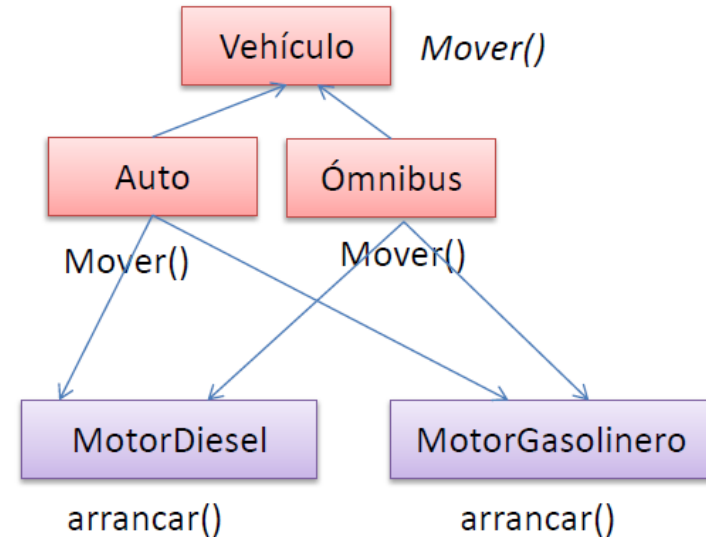
- **CONSECUENCIAS**

- Desacoplando la interfaz de la implementación. La implementación no está limitada permanentemente a una interfaz. La implementación de una abstracción se puede configurar en tiempo de ejecución.
- Extensibilidad mejorada. Puede ampliar las jerarquías de la abstracción y del ejecutor independientemente.

Patrón Estructural: Bridge

- **EJEMPLO**

- Tenemos una app que simula el funcionamiento de vehículos.
- Existe una abstracción Vehículo de la cual se derivarán los diferentes tipos de vehículos terrestres que pueden existir (Auto, Buses)
- Así mismo, los vehículos podrían tener diferentes tipos de motor (MotorDiesel, Gasolinero), y cada uno de ellos implementa su propio mecanismo interno para encender.
- El problema es que, debería implementar una clase por cada combinación posible a ser implementada y la abstracción ya no sería reusable



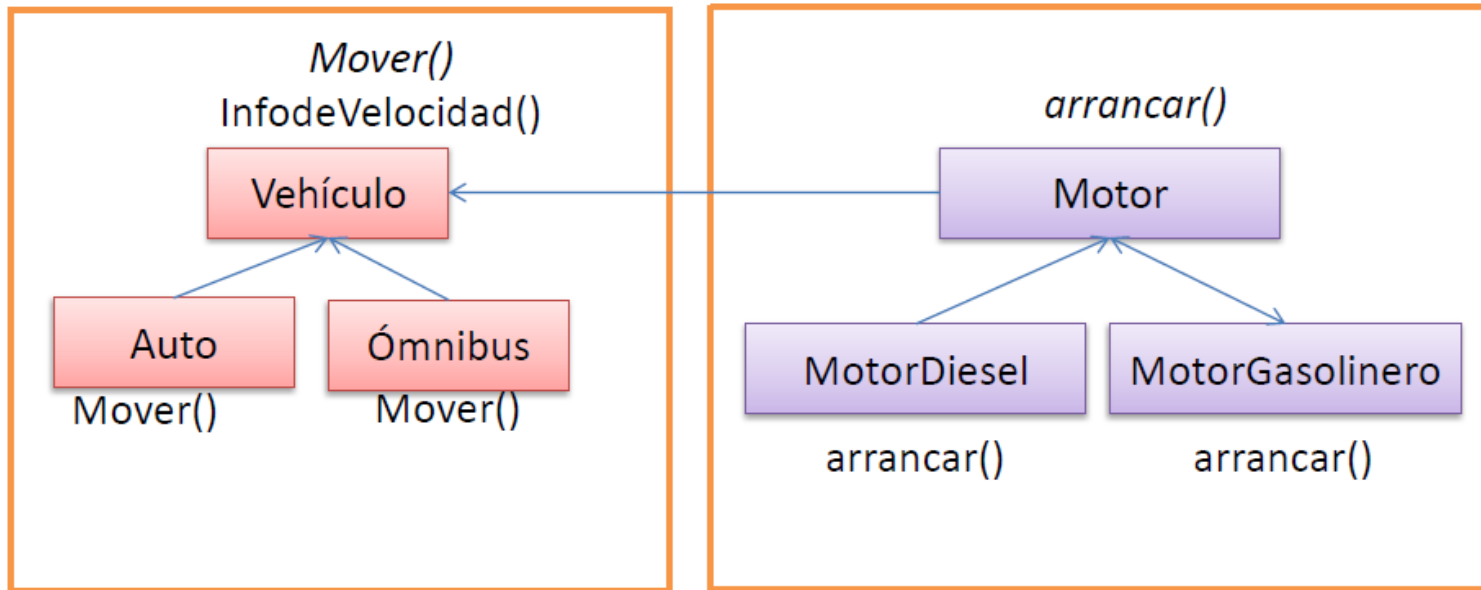
Patrón Estructural: Bridge

- **EJEMPLO**

- Cada motor tiene una cantidad de caballos de fuerza que determinan su potencia.
- Además de ello, todo Vehículo deberá implementar una operación que muestre la información de la velocidad basado en los caballos de fuerza del motor que usa y el peso del mismo.
 - Si el ratio < 3 : El vehículo va a gran velocidad
 - Si el ratio está entre 3 y 8 : El vehículo va a velocidad moderada.
 - De lo contrario : El vehículo va a baja velocidad
 - Donde: $\text{ratio} = \text{pesoVehiculo} / \text{caballos de fuerza}$

Patrón Estructural: Bridge

- EJEMPLO**



Abstracción (Tipos de Vehículo)

Implementación (Diferentes
Plataformas: Motores)

Patrón de Comportamiento: STRATEGY

- **PROPÓSITO**

- Definir una familia de algoritmos y encapsular cada uno de ellos para hacerlos intercambiables.
- Permite que un algoritmo varíe independientemente de los clientes que lo usan.

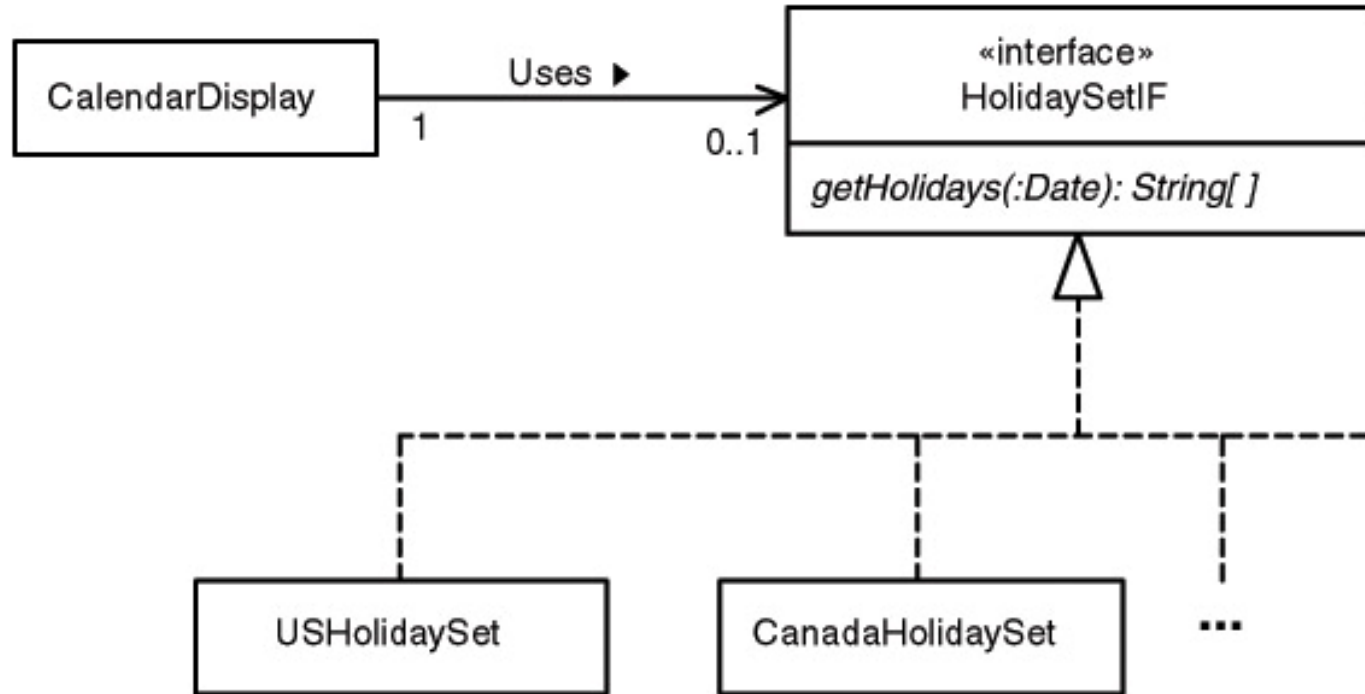
Patrón de Comportamiento: STRATEGY

- **MOTIVACIÓN**

- Imagine una aplicación **CalendarDisplay** que muestra calendarios.
- Uno de los requerimientos del programa, es que debe mostrar los feriados **por cada país**.
- Como debe existir una lógica para obtener los feriados por cada país y grupo religioso, el cliente debería incluir cada lógica en su codificación, lo cuál lo haría muy complejo y dificultaría su modificación.
- Una solución contar con una interface en común llamada **ESTRATEGIA** y subclases que implementen la lógica de cada grupo de feriado. Cada subclase será una **ESTRATEGIA CONCRETA**.
- Se podrán agregar fácilmente nuevas estrategias si fuese necesario.

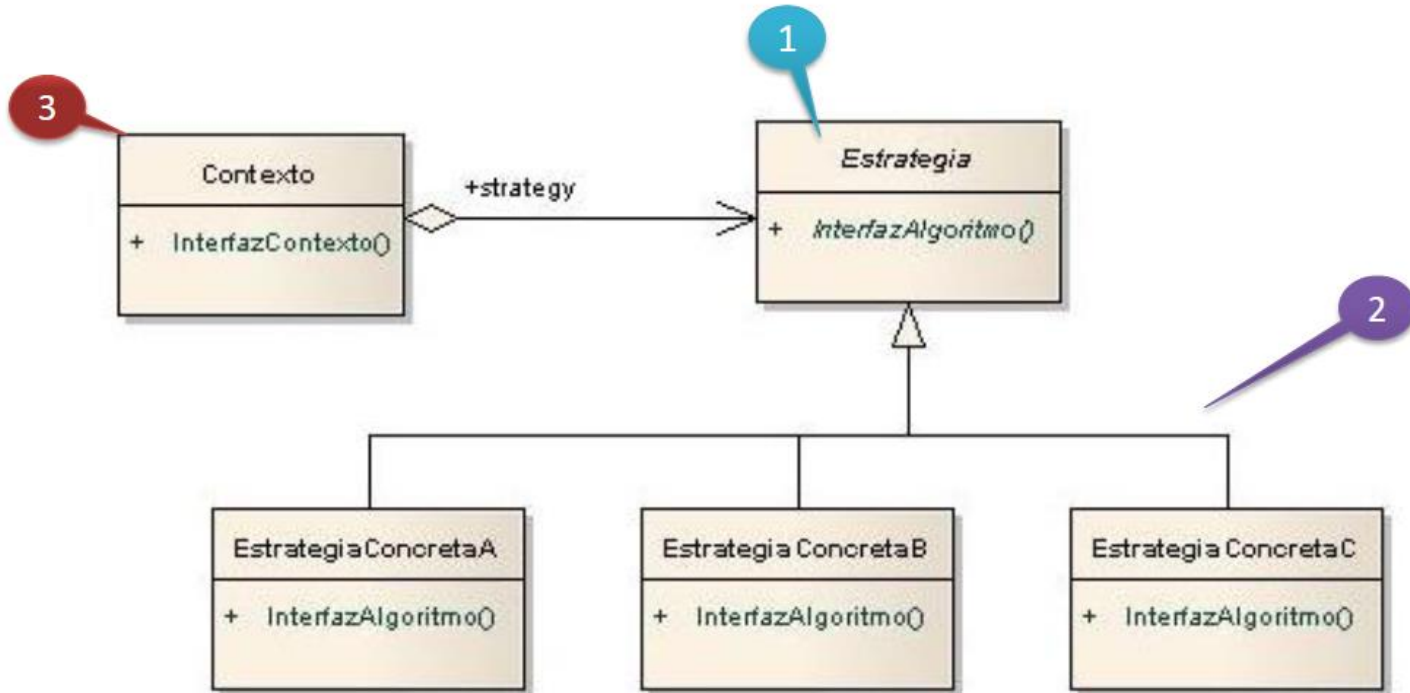
Patrón de Comportamiento: STRATEGY

- MOTIVACIÓN**



Patrón de Comportamiento: STRATEGY

- ESTRUCTURA



Patrón de Comportamiento: STRATEGY

- **PARTICIPANTES:**

1. **Contexto (Context)** : Es el elemento que usa los algoritmos, por tanto, delega en la jerarquía de estrategias. Configura una estrategia concreta mediante una referencia a la estrategia necesaria.
2. **Estrategia (Strategy)**: Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el Contexto para invocar a la estrategia concreta.
3. **EstrategiaConcreta (ConcreteStrategy)**: Implementa el algoritmo utilizando la interfaz definida por la estrategia.

Patrón de Comportamiento: STRATEGY

- **CONSECUENCIAS:**

- El uso del patrón proporciona una alternativa a la extensión de contextos, ya que puede realizarse un cambio dinámico de estrategia.
- Los clientes deben conocer las diferentes estrategias y debe comprender las posibilidades que ofrecen. Por tanto, este patrón debe usarse cuando el cliente conozca y le sea relevante la diferencia entre algoritmos.

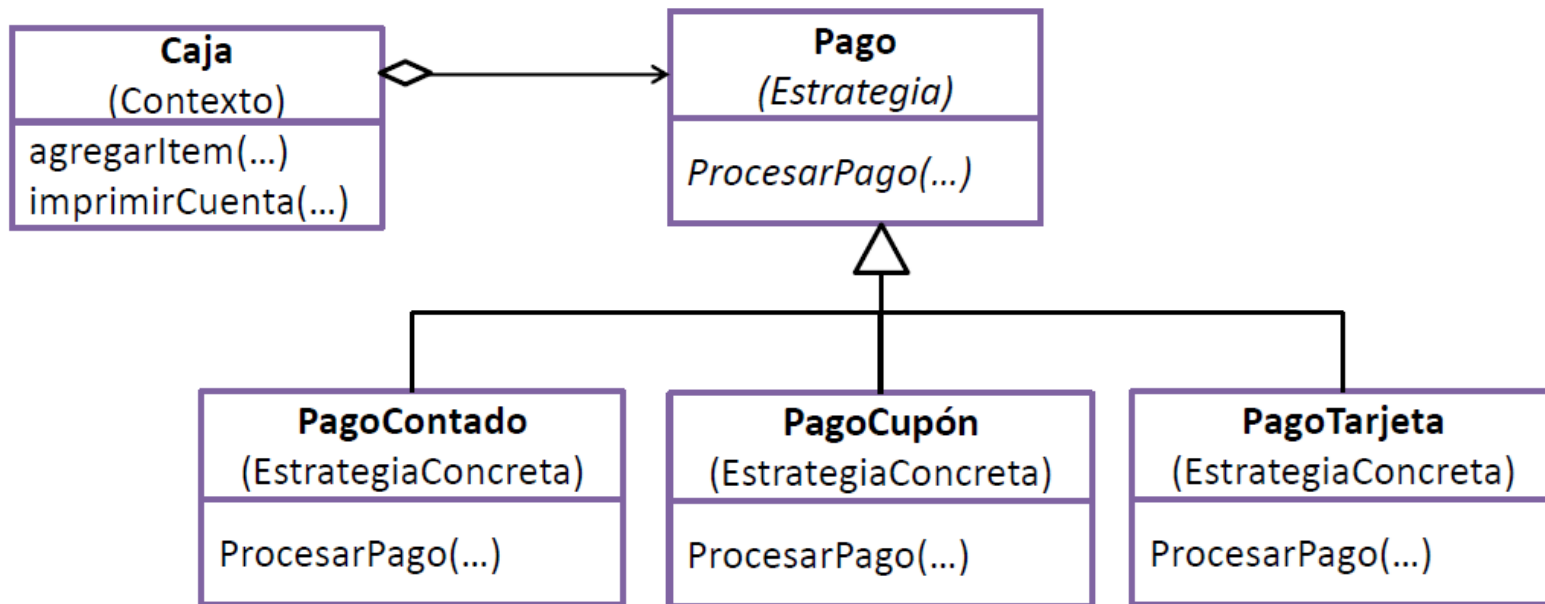
Patrón de Comportamiento: STRATEGY

- **Ejemplo:**

- En un sistema de ventas, tenemos diferentes tipos de pago:
 - Al contado
 - Con cupón – al cuál se le aplica el 10% de descuento al monto final
 - Con Tarjeta – al cuál se le aplica 5% extra por comisión del banco.
- Tenemos la clase abstracta Pago que define la interfaz común de los pagos.
- Así también, tenemos la clase Caja que se encarga de registrar la lista de productos comprados e imprimir la cuenta final.

Patrón de Comportamiento: STRATEGY

- **Ejemplo:**



Patrón de Comportamiento: COMMAND

- **PROPÓSITO**
 - Encapsula la petición en un objeto, y de esta manera podemos gestionar colas o registros de peticiones y deshacer operaciones.

Patrón de Comportamiento: COMMAND

- **MOTIVACIÓN:**

- Muchas veces necesitamos enviar una petición a objetos sin conocer la operación en sí o el receptor de dicha petición.
- Un utilitario para crear interfaces de usuario, incluye objetos como botones o menús que ejecutan una petición en respuesta de una entrada del usuario (Click).
- El utilitario no puede implementar la petición directamente en el botón o en el menú porque son las aplicaciones que hacen uso del utilitario quienes saben qué debería hacerse y en cuál objeto.

Patrón de Comportamiento: COMMAND

- **MOTIVACIÓN:**

- **El patrón command permite que los objetos del utilitario hagan peticiones a objetos de la aplicación** que no han sido especificados, convirtiendo la petición en un objeto que puede ser guardado y enviado.
- En su forma más completa, permite crear una estructura de datos en la cual se pueden almacenar las peticiones para **mantener un historial** que, a su vez, nos permitirá deshacer y rehacer acciones.

Patrón de Comportamiento: COMMAND

- **MOTIVACIÓN:**

- Pensemos en una aplicación para procesar texto donde se pueda deshacer y rehacer acciones.
- Para ello, consideramos una clase abstracta **Orden (Command)** con sus métodos abstractos **Ejecutar()** y **Deshacer()**, declarando así una interfaz para crear operaciones.
- A partir de esta clase, nacen otras subclases por cada orden como Copiar, Cortar, o Pegar, que implementarán la función **Ejecutar()** y **Deshacer()**

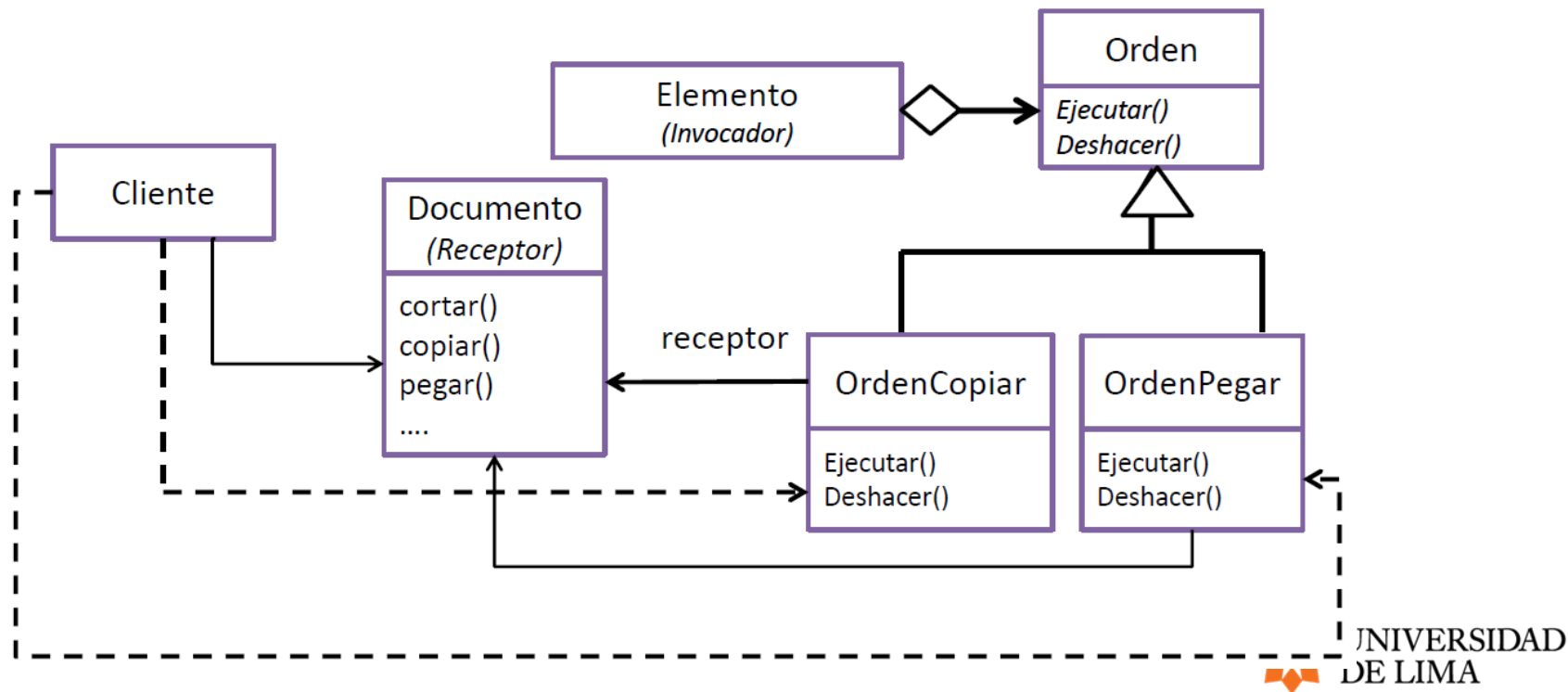
Patrón de Comportamiento: COMMAND

- **MOTIVACIÓN:**

- El cliente dará clic en un Elemento de la Pantalla que a su vez deberá ejecutar la acción que le corresponde
- Para invocar una acción, se debe crear un objeto de las subclases **OrdenPegar**, **OrdenCopiar**, **OrdenCortar**, **etc.**

Patrón de Comportamiento COMMAND

EJEMPLO:



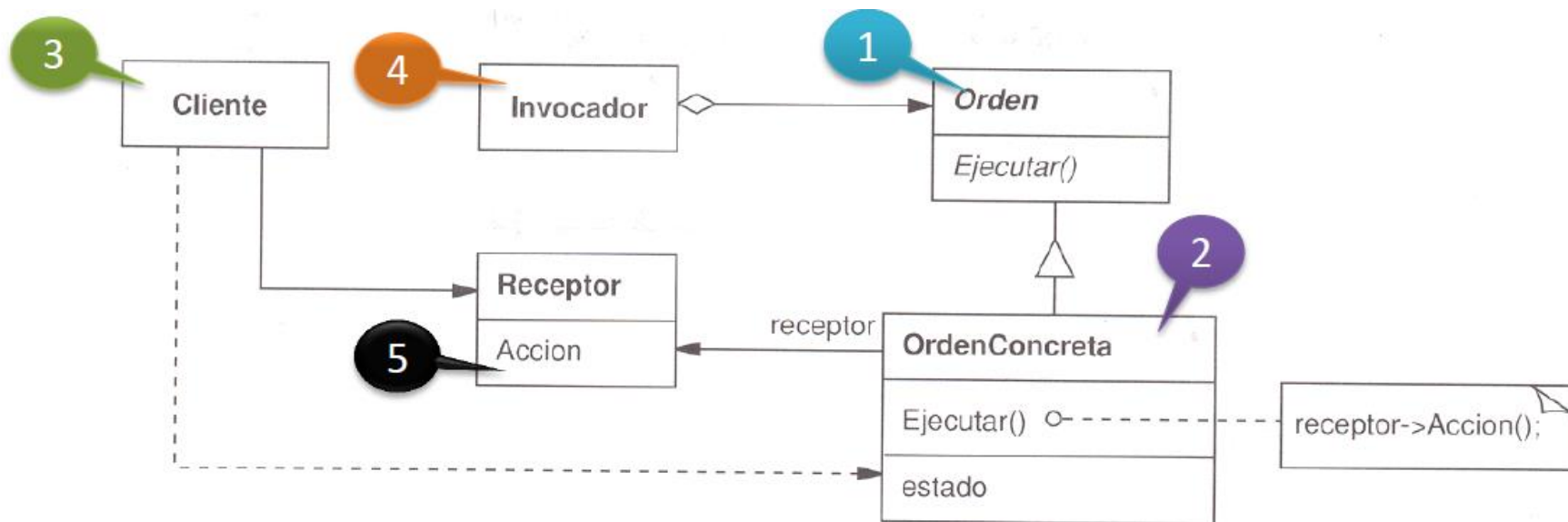
Patrón de Comportamiento: COMMAND

- **APLICABILIDAD:**

- Aplicar command cuando se necesite:
 - Funciones callback: Función que está registrada en algún sitio para que sea llamada más tarde.
 - Hacer cola, ejecutar órdenes en diferentes instantes de tiempo.
 - Permitir deshacer.
 - Registrar los cambios en caso se necesite rehacer la operación al producirse una caída.
 - Estructurar un sistema alrededor de operaciones de alto nivel construidas sobre operaciones básicas (Transacciones)

Patrón de Comportamiento: COMMAND

- **ESTRUCTURA:**



Patrón de Comportamiento: COMMAND

- **PARTICIPANTES:**

1. **Orden(Command):** Declara la interface para la ejecución de la operación.
2. **OrdenConcreta:** Define la relación entre el objeto Receptor y una acción, implementa Ejecutar() al invocar las operaciones correspondientes del Receptor.
3. **Cliente:** Crea un objeto OrdenConcreta y lo relaciona con su Receptor
4. **Invocador:** Le pide a la Orden que ejecute la petición.
5. **Receptor:** Sabe cómo ejecutar las operaciones asociadas a la solicitud. Es el objeto que implementará la funcionalidad real que queremos abstraer.

Patrón de Comportamiento: COMMAND

- **Colaboraciones:**

1. El cliente crea un objeto Orden Concreta y especifica su receptor (Pantalla)

En el cliente...

```
public Orden oCopiar = new OrdenCopiar(Pantalla,...,...)
```

1. El objeto Invocador almacena el objeto OrdenConcreta.

En el cliente...

```
Invocador.almacenaryEjecutar(oCopiar)
```

Patrón de Comportamiento: COMMAND

- **Colaboraciones:**

3. El Invocador envía una petición llamando a Ejecutar sobre la orden

En clase Invocadora:

```
public void AlmacenaryEjecutar(Orden orden){  
    Historial_comandos.add(orden) //almacena la orden en un historial  
    Orden.execute() // llama al "Execute"  
}
```

4. El Objeto OrdenConcreta invoca operaciones de su receptor para llevar a cabo la petición

En clase OrdenConcreta: OrdenCopiar

```
Public void execute(){  
    pantalla.copiar(...);  
}
```


Patrón de Comportamiento: COMMAND

- **CONSECUENCIAS:**

- La Orden (Command) **desacopla el objeto que invoca la operación de aquel que sabe cómo realizarla**
- Es fácil añadir nuevas órdenes, ya que no hay cambiar las clases existentes.

Referencias

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley.
2. <https://github.com/RameshMF/gof-java-design-patterns>
3. <https://refactoring.guru/design-patterns/abstract-factory/java/example>