Patrones de diseño estructurales

Semana 8

Agenda

- Patrones estructurales
- Patrón Adapter
- Patrón Composite
- Patrón Facade
- Patrón Decorator

Patrones estructurales

Patrones estructurales

Patrones estructurales

- Son patrones que se encargan de la composición de las clases.
- Usa conceptos de herencia para componer interfaces.
- Indica maneras de proveerles nuevas funcionalidades a los objetos.
- Se trata de conseguir que cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos.

Breve Descripcion de cada uno

- **Adapter**:

  Adapta interfaces de distintas clases.

- **Bridge**:

  Separa una interfaz de su implementacion.

- **Composite**:

  Indica una estructura en forma de arbol que contiene objetos simples y compuestos.

- **Decorator**:

  Añade dinámicamente responsabilidades a los objetos.

Breve Descripcion de cada uno

- **Facade**:

  Una sola clase que representa un sistema completo.

- **Flyweight**:

  Utilizar una instancia liviana por motivos de eficiencia.

- **Private Class Data**:

  Restringe el acceso a los datos de un objeto.

- **Proxy**:

  Un objeto representa otro objeto.

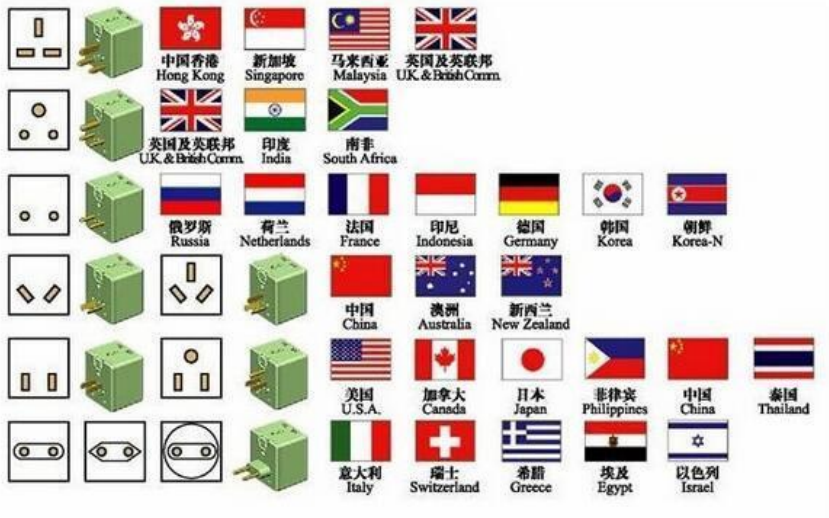
Patrón Adapter - Wrapper

Papá Noél existe

- Un tío viene de visita de España y le regala una MacBook 2017.

Adapter

- **Propósito**
  - Convertir la interfaz de una clase en otra que esperan los clientes.
- **Motivación**
  - Para reutilizar una clase de una librería, aunque su interfaz no correspondiera exactamente con el que requiere una aplicación concreta.
  - Para añadir funcionalidad que la clase reutilizada no proporciona.

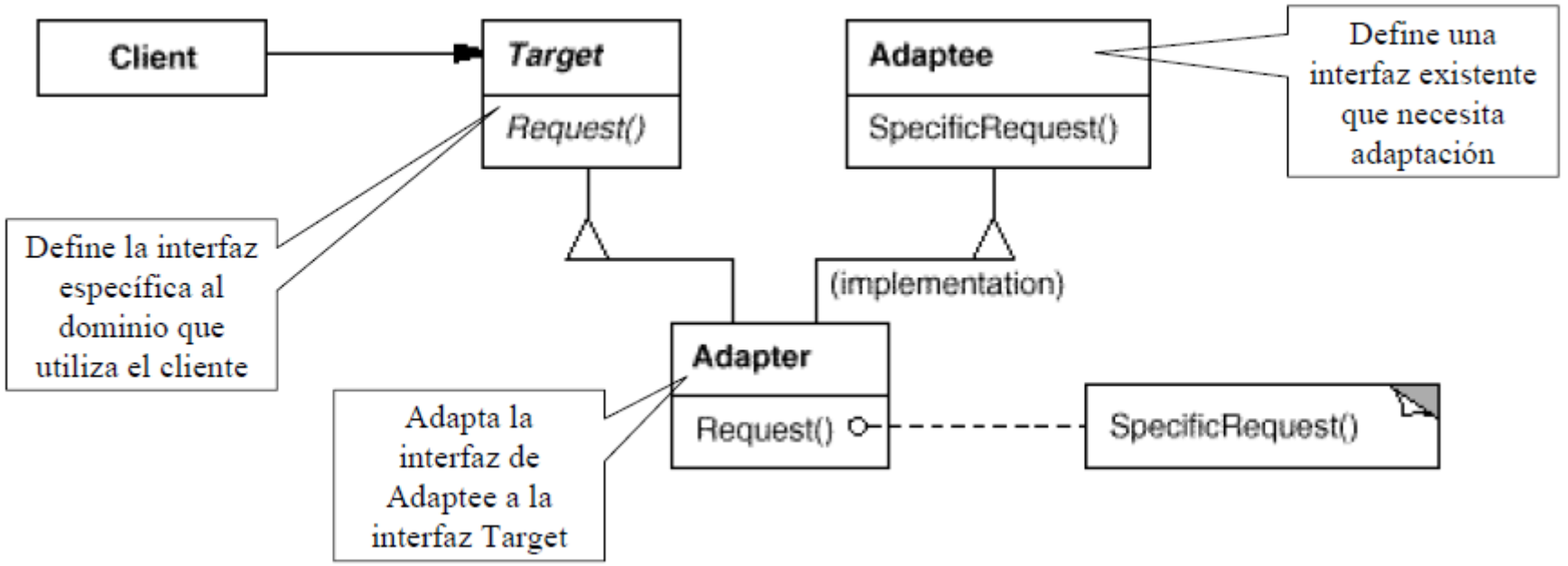


Adapter - Wrapper

- **Aplicación**
  - Para usar una clase existente cuya interfaz no se corresponde con el que se necesita.
  - Para crear una clase reutilizable que coopera con clases imprevistas (esto es, que no tienen necesariamente interfaces compatibles).
  - El adaptador de objeto: para utilizar varias subclases existentes para las que sería poco práctico adaptar su interfaz heredando de cada una. Un adaptador de objeto puede adaptar la interfaz de su clase padre.

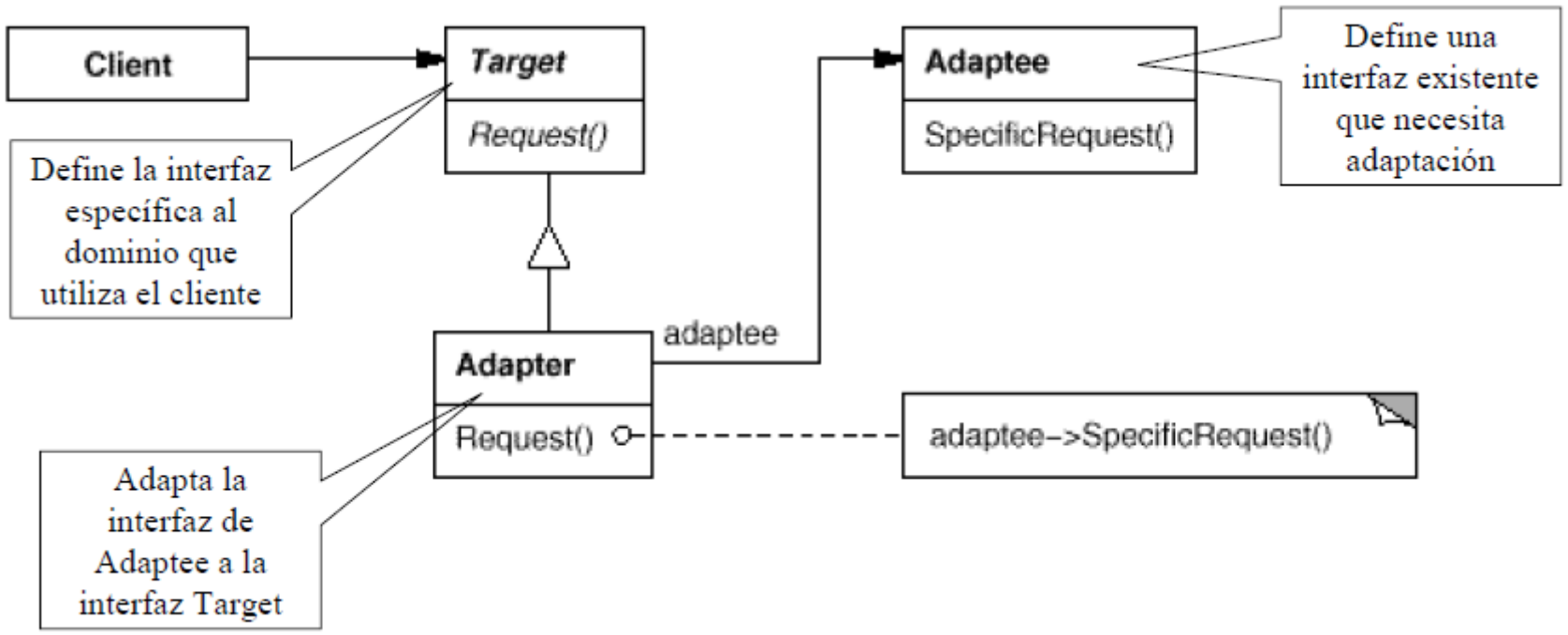
Adapter - Clase

- Los clientes llaman a las operaciones de un objeto Adaptador.
- A su vez, el Adaptador llama a las operaciones heredadas de la clase Adaptada que tratan la petición.



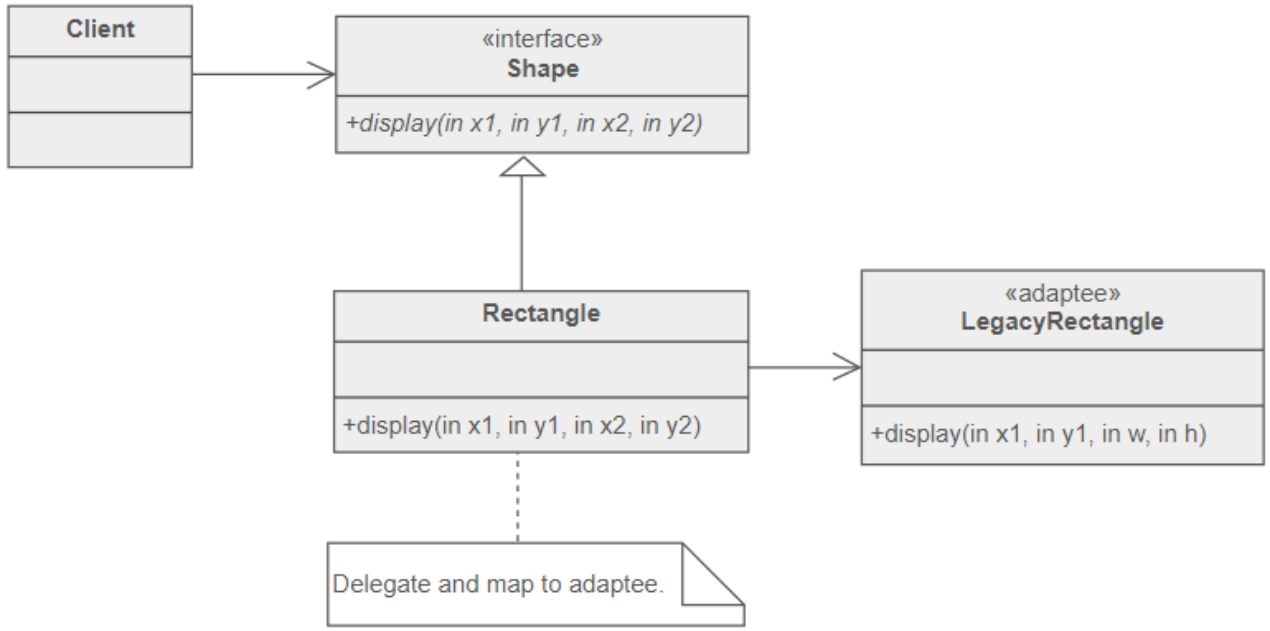
Adapter - Objeto

- Los clientes llaman a las operaciones de un objeto Adaptador.
- A su vez, el Adaptador llama a las operaciones del Adaptado que tratan la petición.



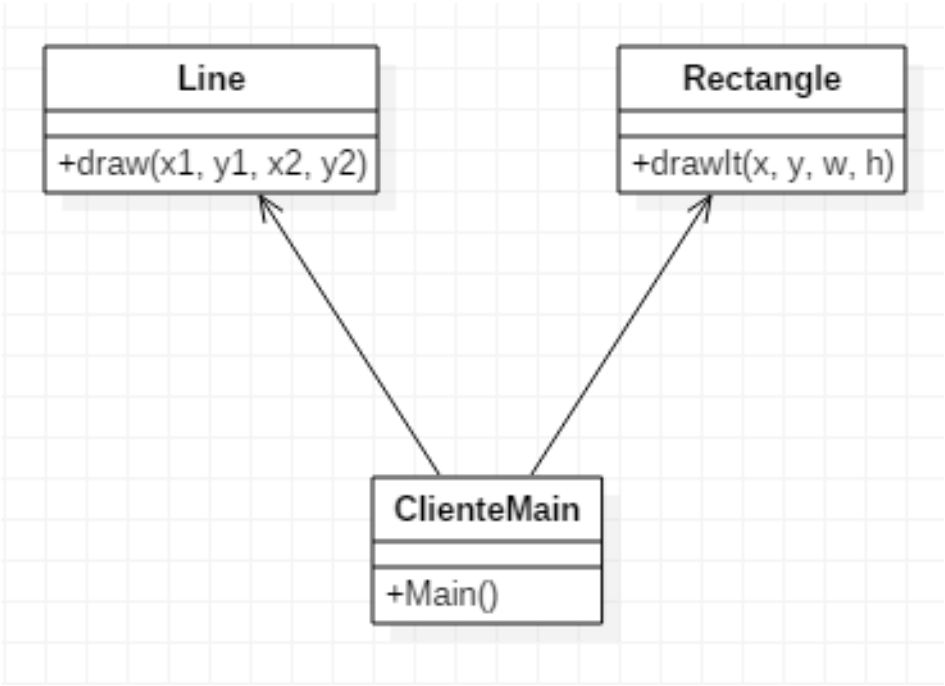
Ejemplo – Adapter Objeto

- El cliente desea mostrar un Shape, y se tiene un LegacyRectangle que no tiene la misma interfaz que cliente necesita.
- Rectangle adapta el LegacyRectangle para que implemente Shape.



Problema

- ClienteMain desea dibujar distintas formas geométricas, pero ya hay algunas implementadas.
- Tienen métodos y/o atributos distintos.



Codificando 1/3

```
class Line {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Line from point A(" + x1 + ";" + y1 +
            "), to point B(" + x2 + ";" + y2 + ")");
    }
}

class Rectangle {
    public void drawIt(int x, int y, int width, int height) {
        System.out.println("Rectangle with coordinate left-down point (" +
            x + ";" + y + "), width: " + width + ", height: " + height);
    }
}

public class ClienteMain {
    public static void main(String[] args) {
        Object[] shapes = {new Line(), new Rectangle()};
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        int width = 40, height = 40;
        for (Object shape : shapes) {
            if (shape.getClass().getSimpleName().equals("Line")) {
                ((Line)shape).draw(x1, y1, x2, y2);
            } else if (shape.getClass().getSimpleName().equals("Rectangle")) {
                ((Rectangle)shape).drawIt(x2, y2, width, height);
            }
        }
    }
}
```

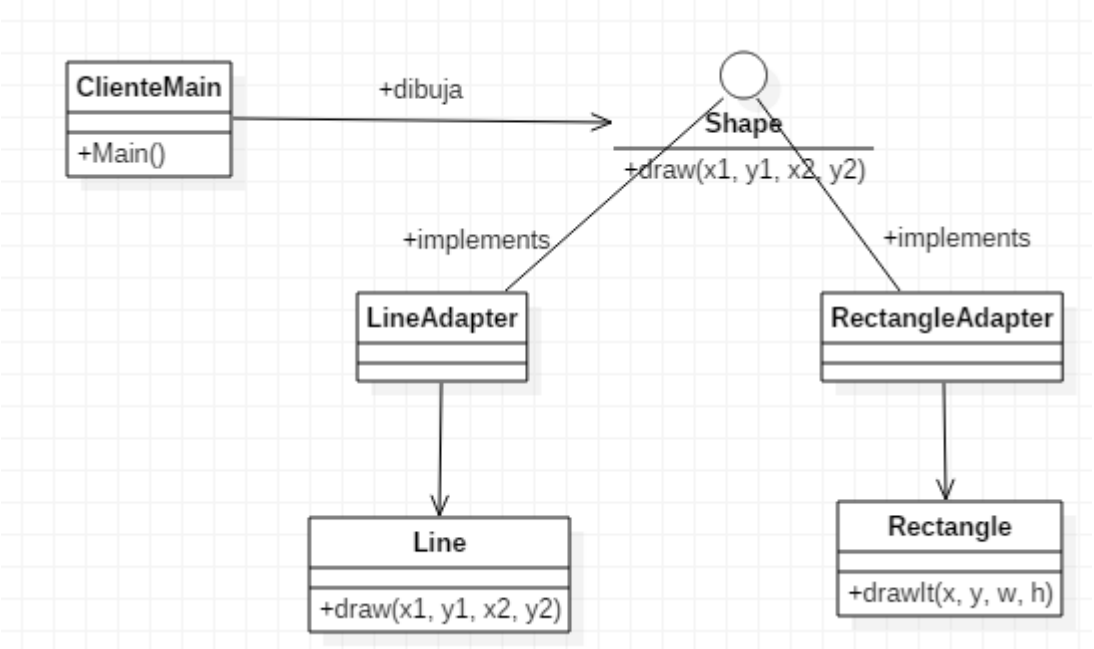
¿Dónde está el problema?

Codificando 2/3

```
interface Shape {
    void draw(int x, int y, int z, int j);
}
class LineAdapter implements Shape {
    private Line adaptee;

    public LineAdapter(Line line) {
        this.adaptee = line;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        adaptee.draw (x1, y1, x2, y2);
    }
}

class RectangleAdapter implements Shape {
    private Rectangle adaptee;

    public RectangleAdapter(Rectangle rectangle) {
        this.adaptee = rectangle;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int x = Math.min(x1, x2);
        int y = Math.min(y1, y2);
        int width = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);
        adaptee.drawIt (x, y, width, height);
    }
}
```

Codificando 3/3

```
public class ClienteMain {
    public static void main(String[] args) {
        Shape[] shapes = {new RectangleAdapter(new Rectangle()),
                          new LineAdapter(new Line())};
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (Shape shape : shapes) {
            shape.draw(x1, y1, x2, y2);
        }
    }
}
```



Consecuencias

- Adapter de clase:
  - Adapta una clase Adaptee a una interfaz Target reutilizando los métodos de la clase Adaptee. Por lo tanto, no funcionará cuando se quieran adaptar la clase adaptada y todas sus subclases.
  - La clase adapter redefine algunos métodos de la clase Adaptee.
  - Sólo se introduce en un objeto y no hace falta delegar en otro adaptado.

Consecuencias

- Adapter de objeto:
  - Permite trabajar con un solo “Adapter” muchos “Adaptee” (La clase adaptada y sus hijas).
  - Se puede agregar funcionalidad a todos los “Adaptee” de una vez.
- Se puede cambiar el nombre de los métodos e incluso agregar nuevas funcionalidades.

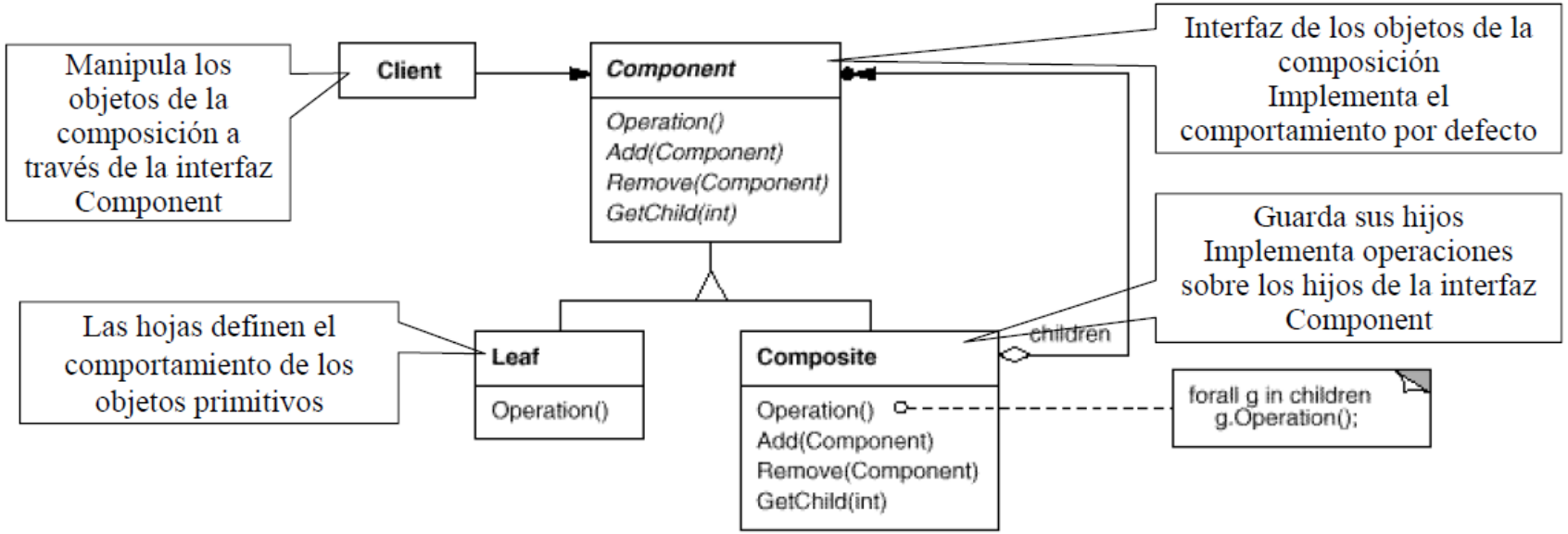
Patrón Composite

Composite

- **Propósito**
  - Construir objetos complejos mediante la composición recursiva de objetos similares.
  - Similar a un árbol.
- Motivación
  - Las aplicaciones gráfica tienen componentes que pueden agruparse para formar componentes mayores.

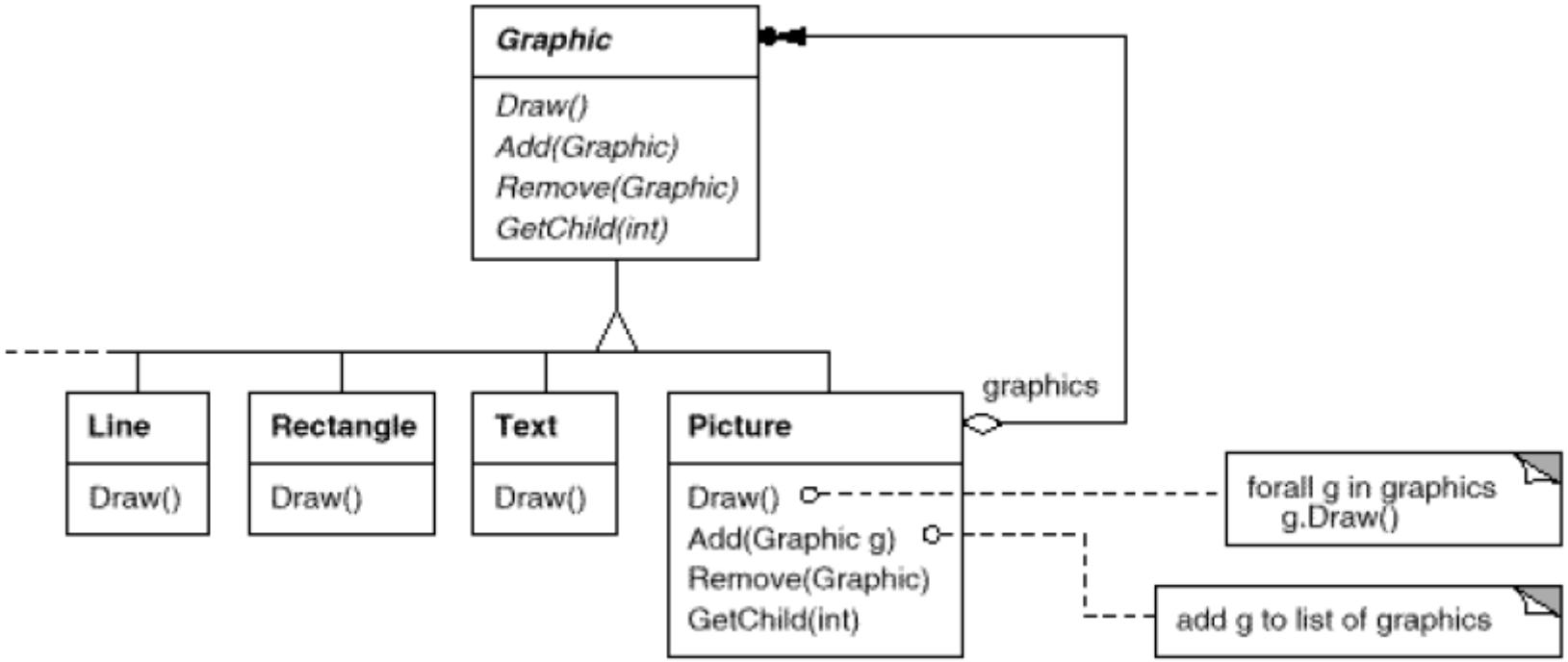
Composite

- **Aplicación**
  - Cuando se desea que el cliente pueda manejar indistintamente objetos simples y compuestos.
    - El cliente trata a todos los objetos de uniformemente.
  - Cuando hay una composición recursiva.



Ejemplo - Composite

- Un Rectangulo, un texto, etc, son subtipos de Graphic.
- Picture también es subtipo, pero puede contener varios Graphics.



Composite

- Consecuencias:
  - Define jerarquías de clases que tienen objetos primitivos y objetos compuestos (composite).
  - Facilita la adición de nuevas clases de componentes.
  - Puede hacer que el diseño sea demasiado general.
    - ❖Hace más difícil restringir los componentes de un composite.
    - ❖Si se quiere hacer que un composite sólo tenga ciertos componentes (no todos), hay que codificar las comprobaciones para que se realicen en tiempo de ejecución.

Patrón Facade

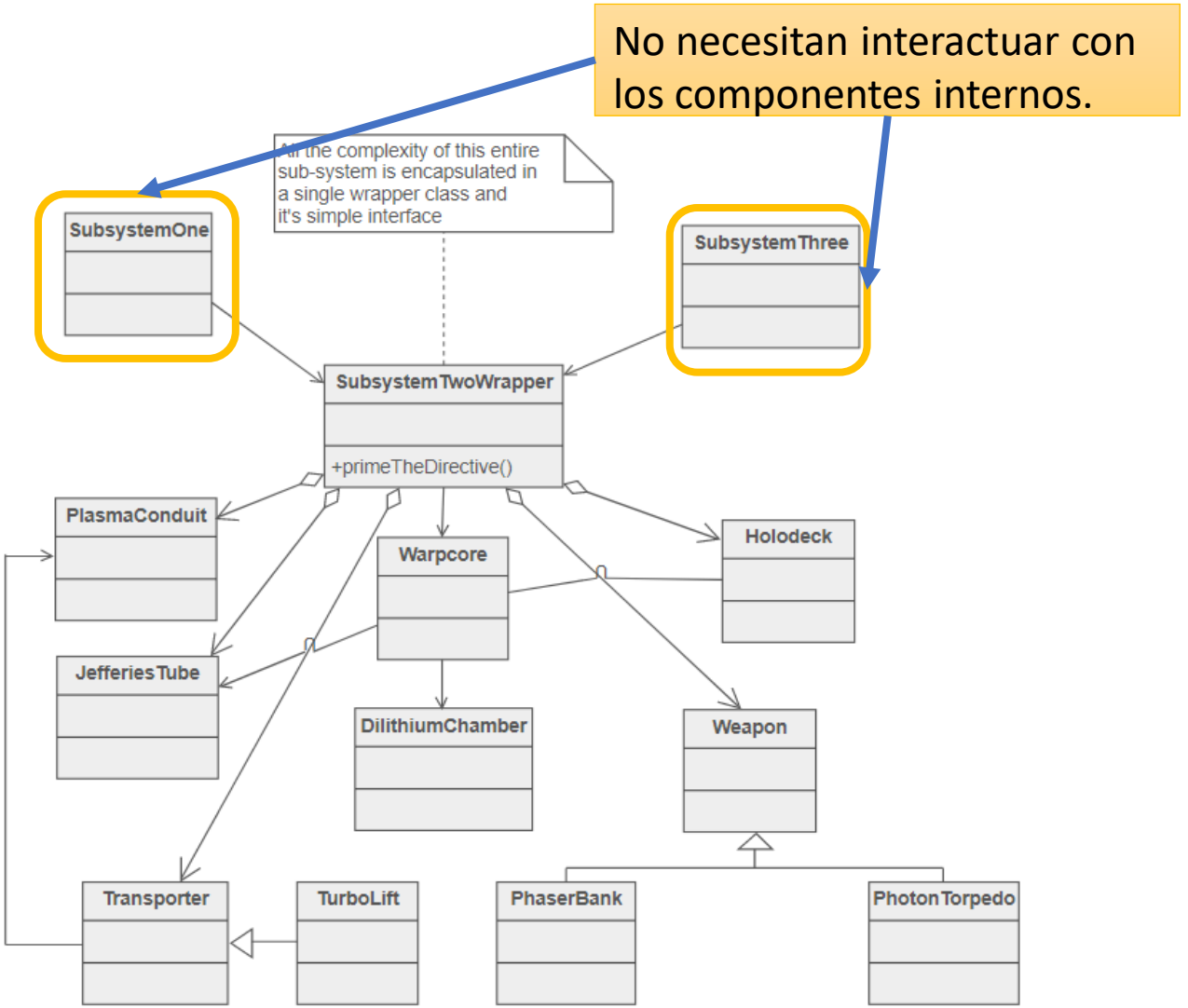
Facade - Fachada

- Propósito
  - Proveer una única interfaz para acceder a un subsistema (grupo de funcionalidades bien delimitadas interrelacionadas) y hacerlo fácil de usar.
  - Envolver un subsistema complejo con una interfaz simplificada.
- Motivación
  - Minimizar las comunicaciones y dependencias entre sistemas.

Facade - Fachada

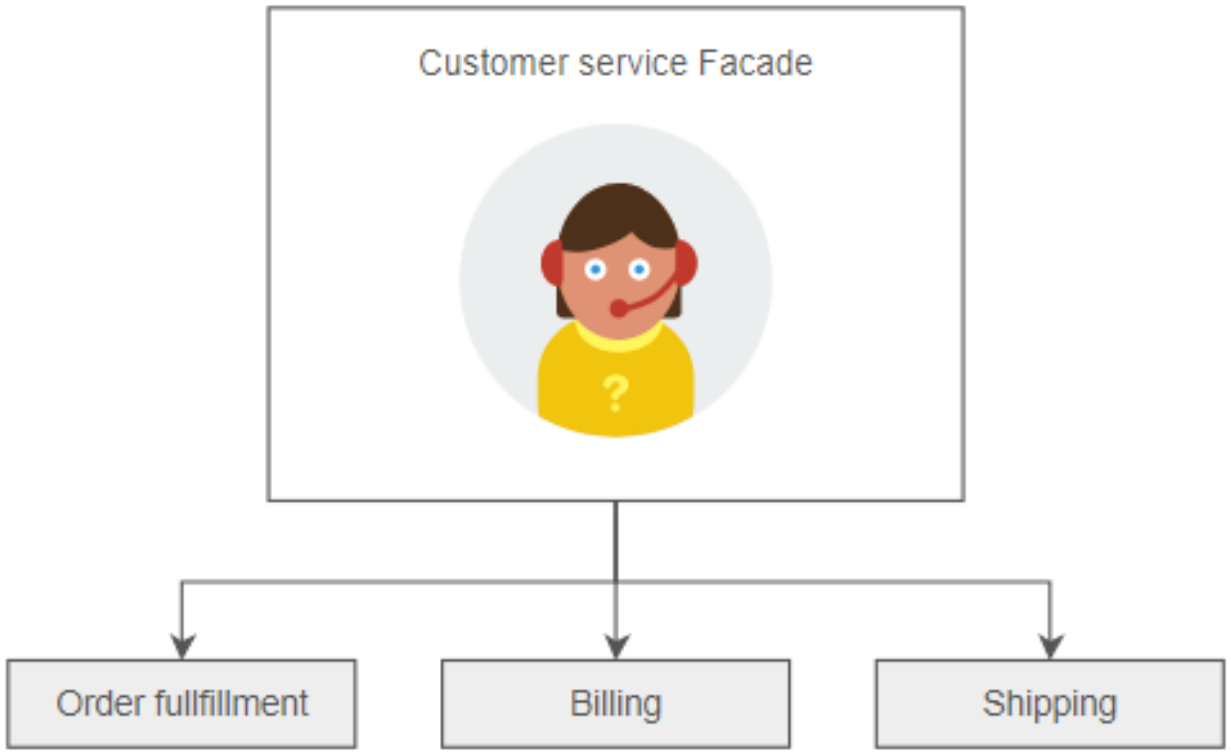
- Aplicación
  - Cuando se desea utilizar sólo algunas de funcionalidades de un grupo de clases.
  - Cuando se desea ocultar un grupo de clases y se prefiere mostrar una interfaz unificada para accederlas sin necesidad de conocerlas directamente.
  - Cuando se desea estructurar un sistema en capas.
    - La fachada define el punto de entrada en cada nivel.
    - Se puede simplificar la dependencia entre sistemas obligando a utilizar únicamente su fachada.

Facade - ejemplo



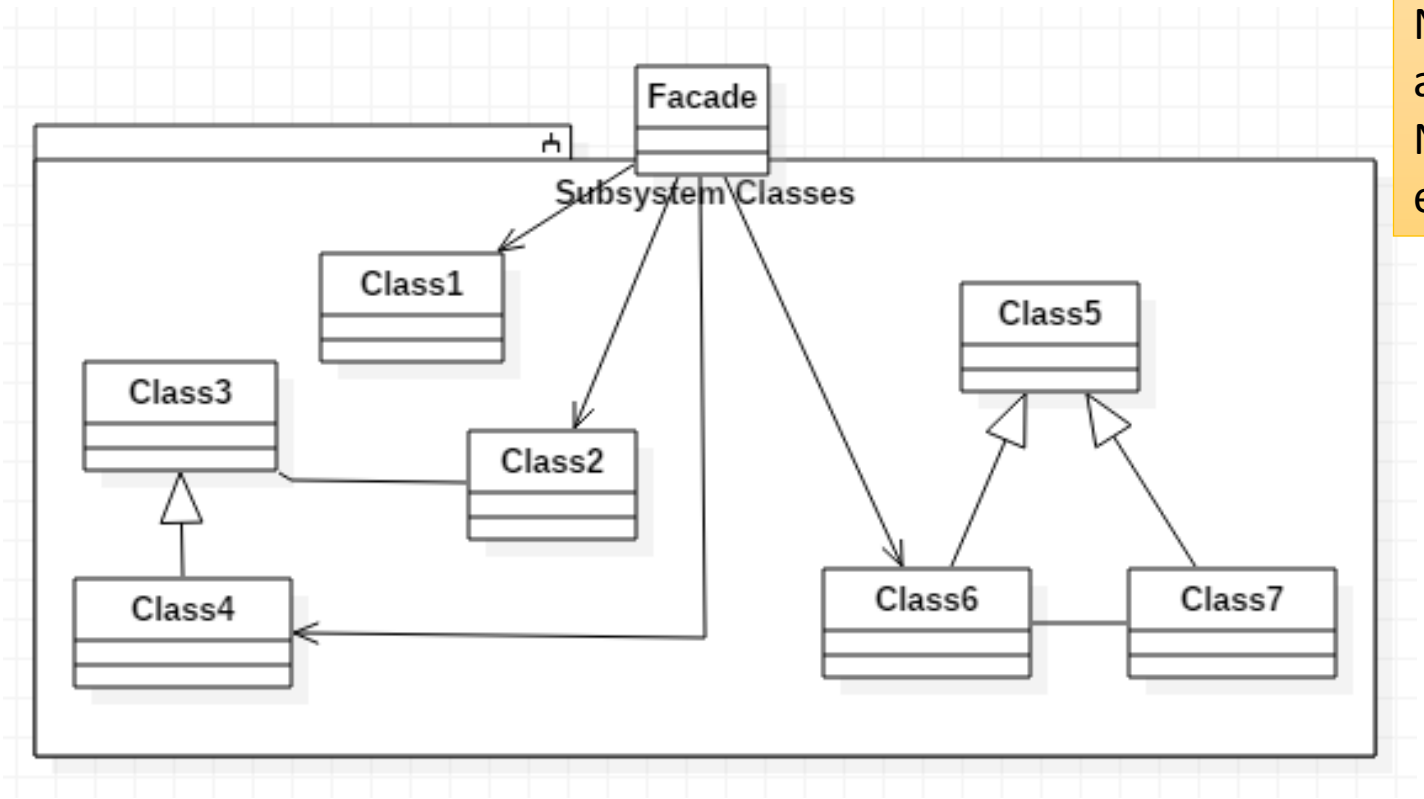
Facade – Ejemplo 2

- El sistema de Servicio al Cliente tiene una fachada (interfaz) que permite resolver distintos problemas a partir de una sola interfaz.



Analice

- ¿Se puede tener un solo diagrama de clases para Facade?
- ¿Hay un DC de Facade, como en Adapter o Builder?



No, porque sólo es una clase que actúa como interfaz de subsistema. No conocemos como está compuesto el subsistema.

Facade

- Consecuencias
  - Reduce la cantidad de objetos con los que interactúa el cliente.
  - Disminuye el acoplamiento entre un subsistema y sus clientes.
  - No evitan que los clientes utilicen las clases del subsistema.

Facade

- Implementación y patrones relacionados.
  - Normalmente sólo hace falta un objeto fachada, por lo tanto, se suele utilizar en conjunto con Singleton.
  - Para reducir el acoplamiento entre Cliente-Subsistema se puede crear una clase fachada abstracta con fachadas concretas para las distintas implementaciones del subsistema.

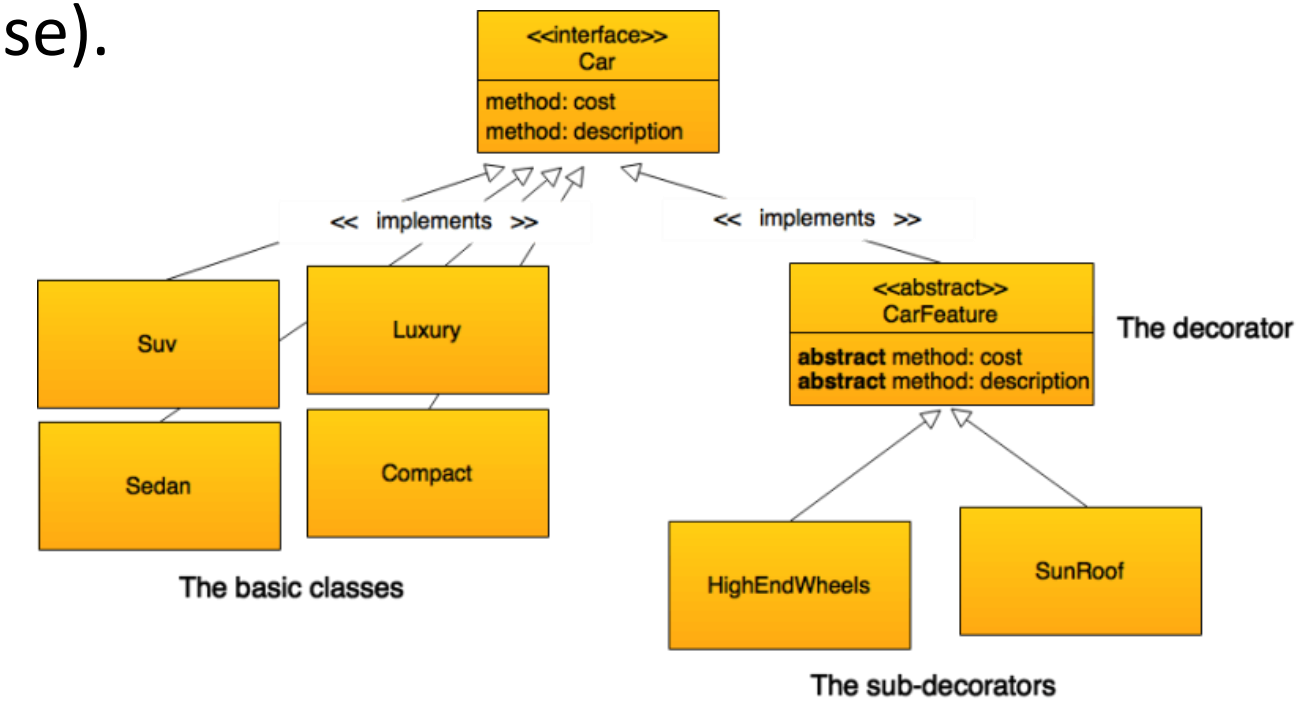
Patrón Decorator

Decorator

- **Propósito**
  - Asignar funcionalidades/características a un objeto dinámicamente. Es una alternativa a la creación de subclases por herencia.
- **Motivación**
  - Para agregar nuevas funcionalidades/características a objetos individuales y no a toda la clase.
    - Una ventana puede tener bordes, barras de desplazamiento, fondo semitransparente.
    - Se los puede agregar dinámicamente.

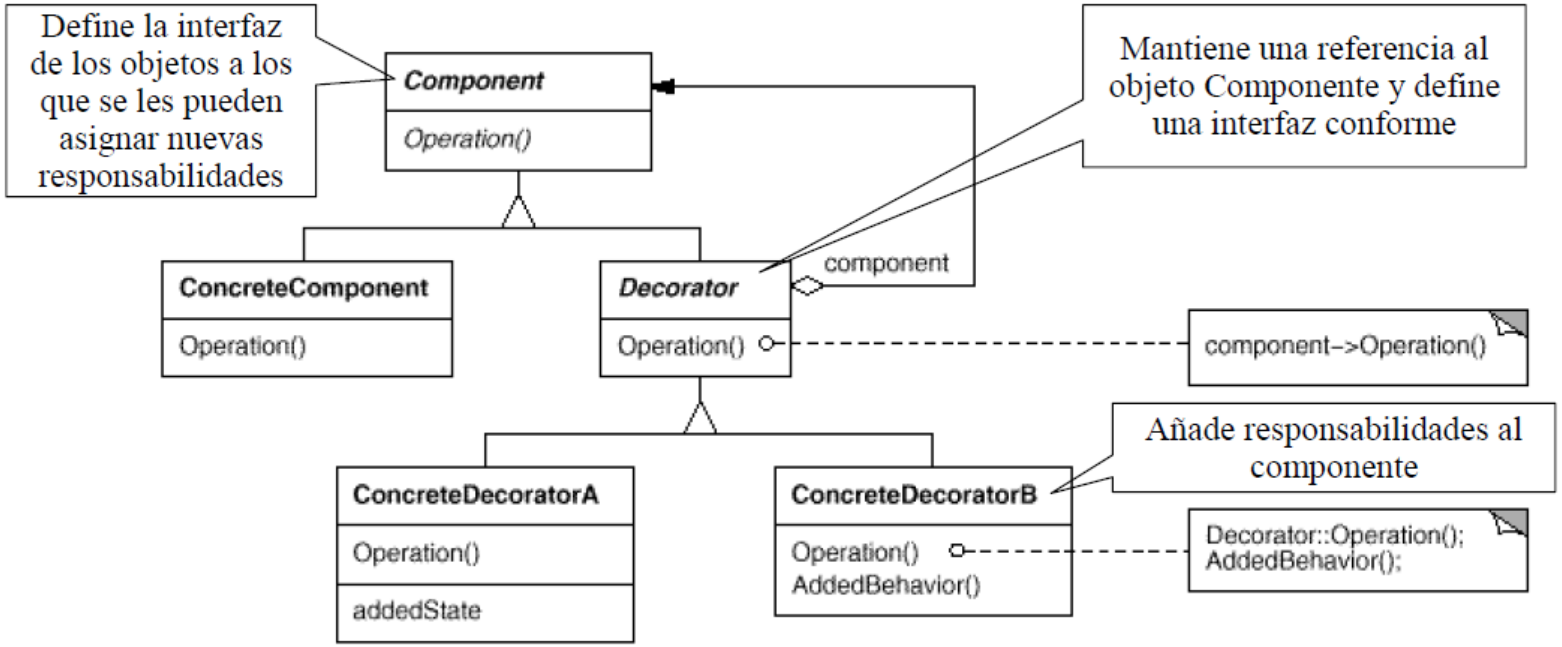
Decorator

- **Aplicación**
  - Cuando se desea agregar funcionalidades a objetos individuales, sin afectar a otros objetos.
  - Cuando no es viable utilizar herencia (muchas clases o no se tiene la definición de la clase).



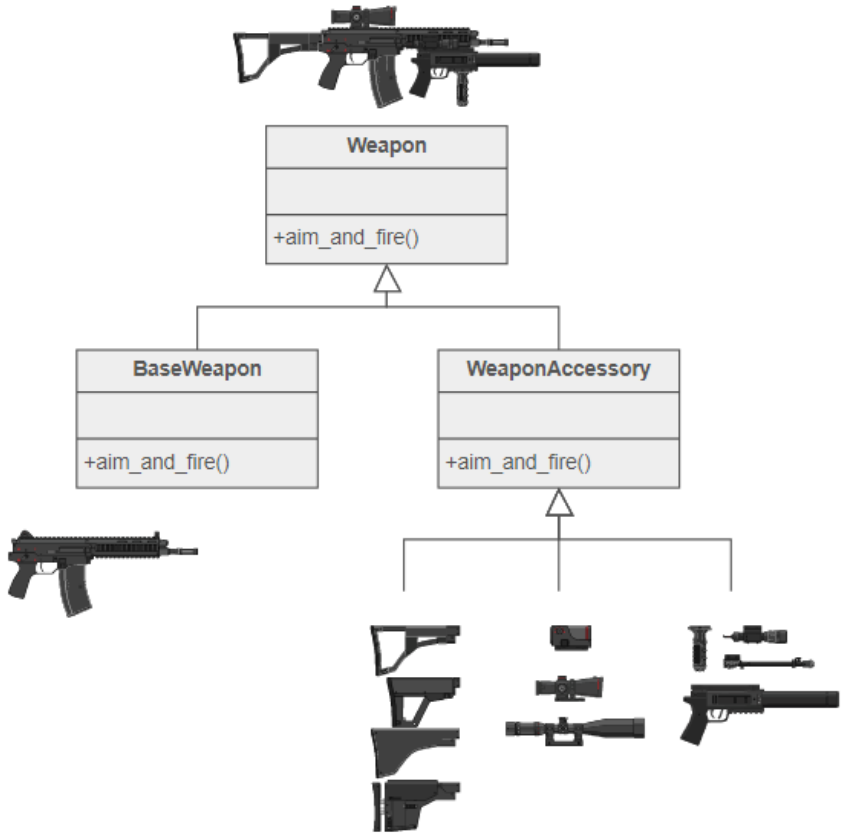
Decorator

- Existe el ConcreteComponent con la funcionalidad básica.
- Se puede extender las funcionalidades del Component con los ConcreteDecorator* sin heredar o modificar el básico (ConcreteComponent).

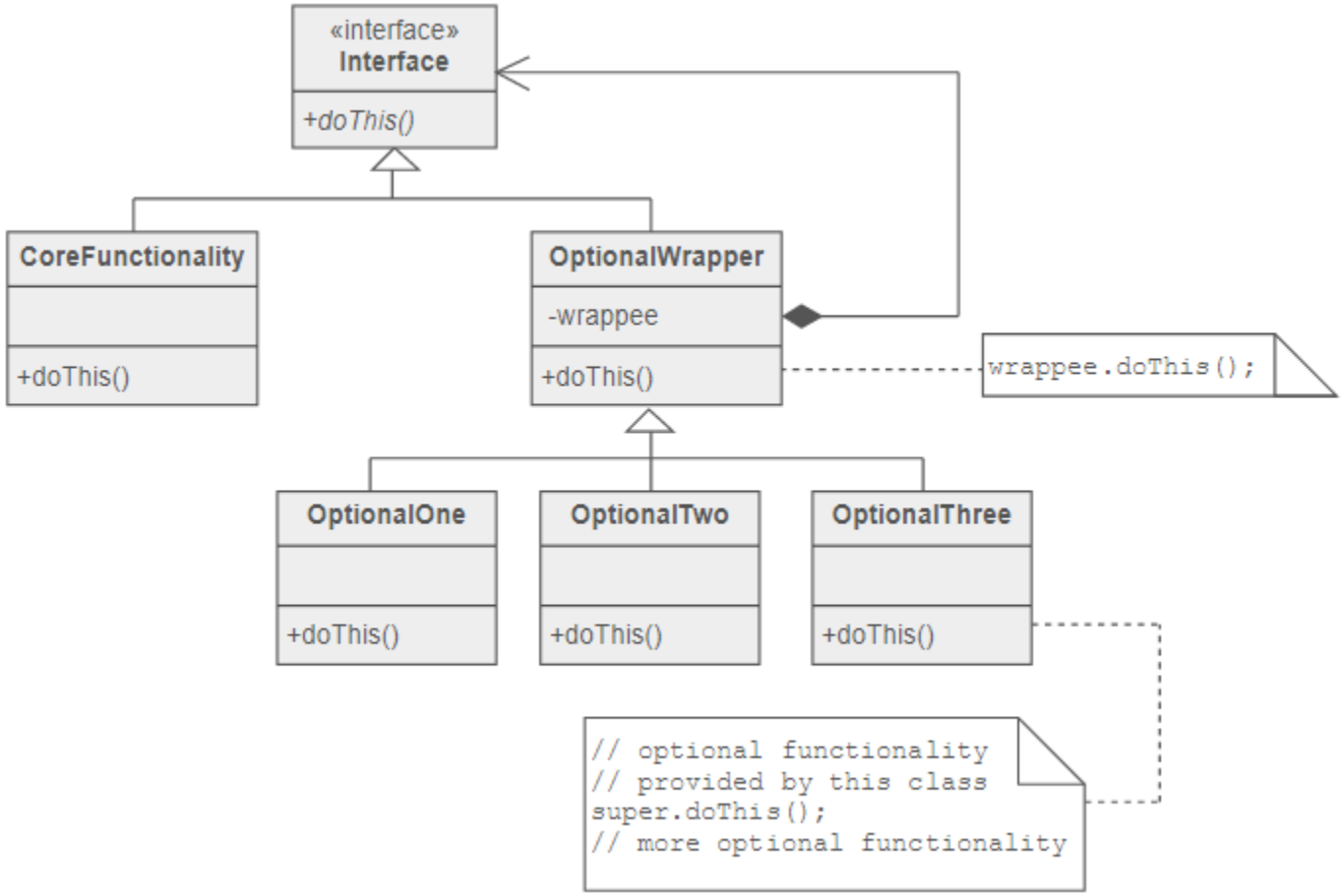


Decorator - Ejemplo 2

- Un arma de asalto ya es peligrosa.
- Pero podría agregarle aditamentos para hacerla más precisa, más silenciosa o más ponderosa.
- Agregando “decoraciones”.



Decorator - Ejemplo 3

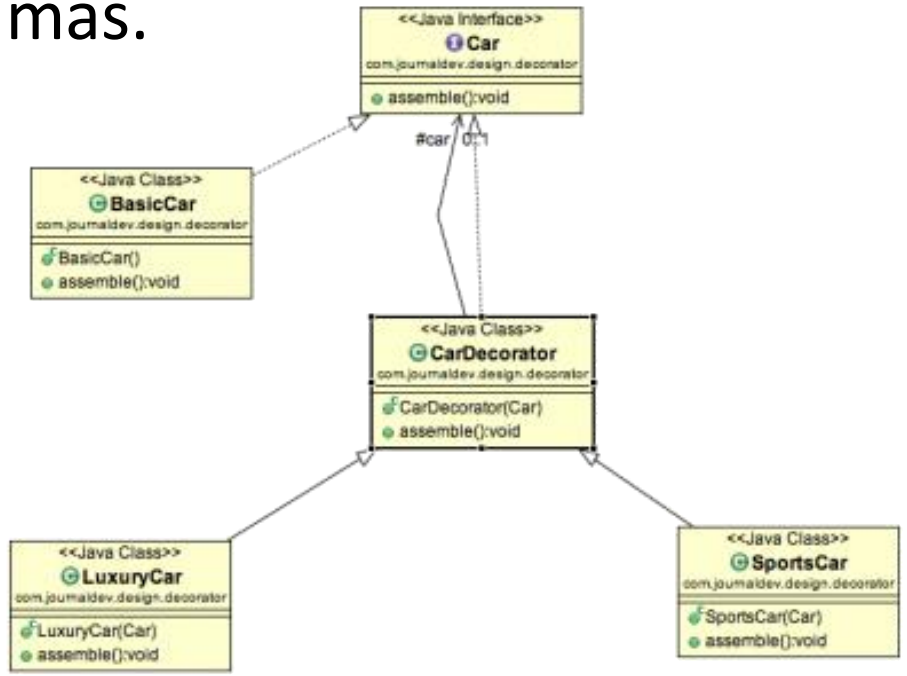


Pasos para utilizar decorator

1. Create a "lowest common denominator" that makes classes interchangeable.
2. Create a second level base class for optional functionality
3. "Core" class and "Decorator" class declare an "is a" relationship
4. Decorator class "has an" instance of the "lowest common denominator"
5. Decorator class delegates to the "has an" object
6. Create a Decorator derived class for each optional embellishment
7. Decorator derived classes delegate to base class AND add extra stuff
8. Client has the responsibility to compose desired configurations

Decorator

- Problema
  - Ya tengo un objeto de “BasicCar”.
  - Deseo agregarle caracteristicas de lujoso “LuxuryCar” y Deportivo “SportCar” **en tiempo de ejecución**. Por lo tanto, **ya tengo creado el BasicCar**, pero quiero extenderlo más.



Codificando 1/3

```java
public interface Car {
    public void assemble();
}

public class BasicCar implements Car {
    @Override
    public void assemble() {
        System.out.print("Basic Car.");
    }
}

public abstract class CarDecorator implements Car {
    protected Car car;
    public CarDecorator(Car c){
        this.car=c;
    }
    @Override
    public void assemble() {
        this.car.assemble();
    }
}
```

1. Crear una interfaz común denominador.
2. La clase concreta debe implementar la interfaz.
3. Crear una clase decorator abstracta, que implemente la interfaz.
4. El decorator debe tener como atributo un objeto de la interfaz.
5. Delegar los métodos a otro que implemente la interfaz.

Codificando 2/3

- Las características adicionales para el BasicCar (SportCar y LuxuryCar) extienden del decorator abstracto.

```
public class SportsCar extends CarDecorator {
    public SportsCar(Car c) {
        super(c);
    }
    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Sports Car.");
    }
}

public class LuxuryCar extends CarDecorator {
    public LuxuryCar(Car c) {
        super(c);
    }
    @Override
    public void assemble(){
        super.assemble();
        System.out.print(" Adding features of Luxury Car.");
    }
}
```

Codificando 3/3

- Finalmente para agregarle características al BasicCar, sólo tengo enviarlo al decorator adecuado, o a más de uno.

```
public class DecoratorPatternTest {

    public static void main(String[] args) {
        Car sportsCar = new SportsCar(new BasicCar());
        sportsCar.assemble();
        System.out.println("\n*****");

        Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));
        sportsLuxuryCar.assemble();
    }

}
```

Antes de finalizar

Puntos para recordar

- ¿Qué es un patrón de diseño estructural?
  - ¿Cuál es la lista de patrones dentro de esta categoría?
  - Defina la utilidad de cada patrón estructural en sus propias palabras
- Asocie un ejemplo que le resulte fácil de recordar para cada patrón de diseño

Lectura adicional

- Gamma et al. , “Design Patterns: Elements of Reusable Object-Oriented Software”
- Shalloway and Trott, "Design Patterns Explained"
- Source Making, “Design Patterns”
  - https://sourcemaking.com/design_patterns

Próxima sesión

- Patrones de diseño de comportamiento