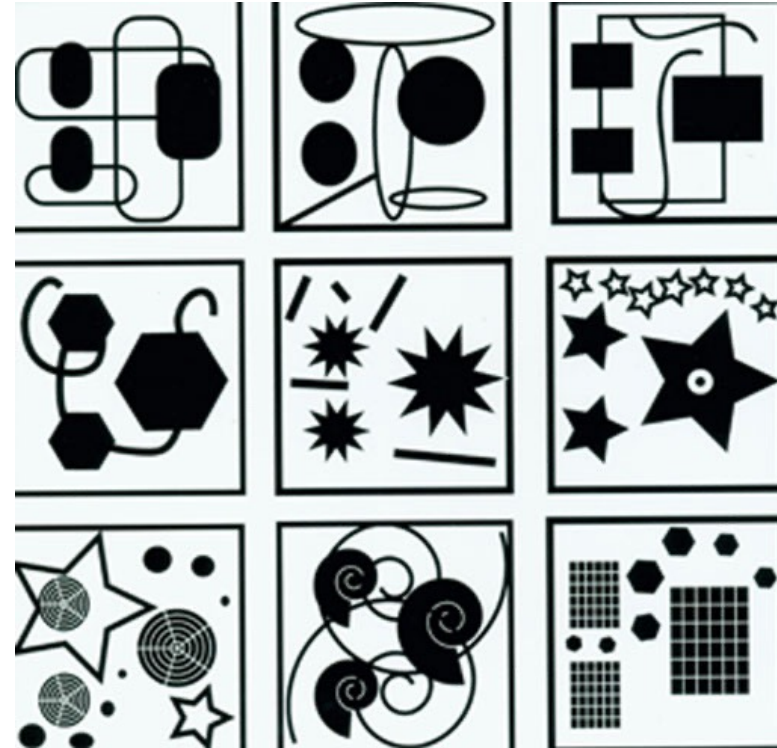


Objetos 2

Nuevos patrones: Wrappers



Alejandra Garrido:
garrido@lifa.info.unlp.edu.ar

- Adapter (estructural)
- Composite (estructural)
- Template Method (comportamiento)
- State (comportamiento)
- Strategy (comportamiento)



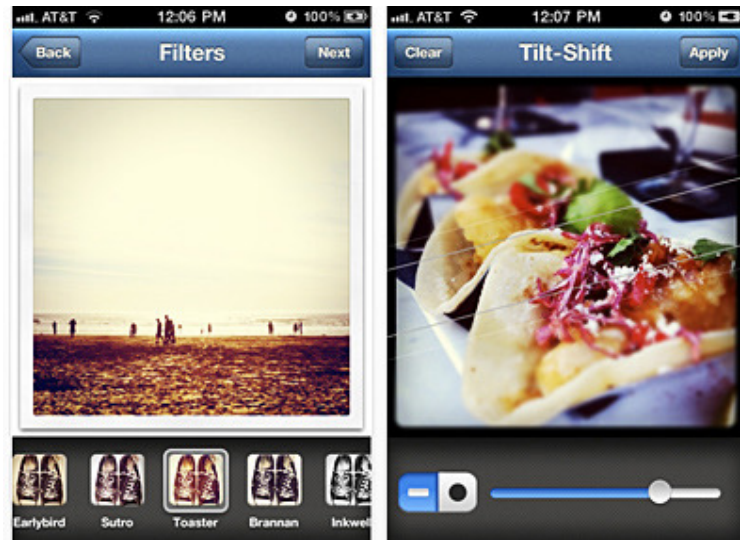
Hoy veremos 2 nuevos
patrones
estructurales

1er nuevo patrón



Ejercicio 1: Edición de fotos en Instagram

- A cada foto podemos aplicarle distintos efectos:
 - Filtros
 - Shadows
 - Vignette
 - Brillo, contraste, temperatura, saturación, etc. Etc.
- Cada una de estas características puede agregarse o quitarse.



Ejercicio 1: Soluciones posibles?

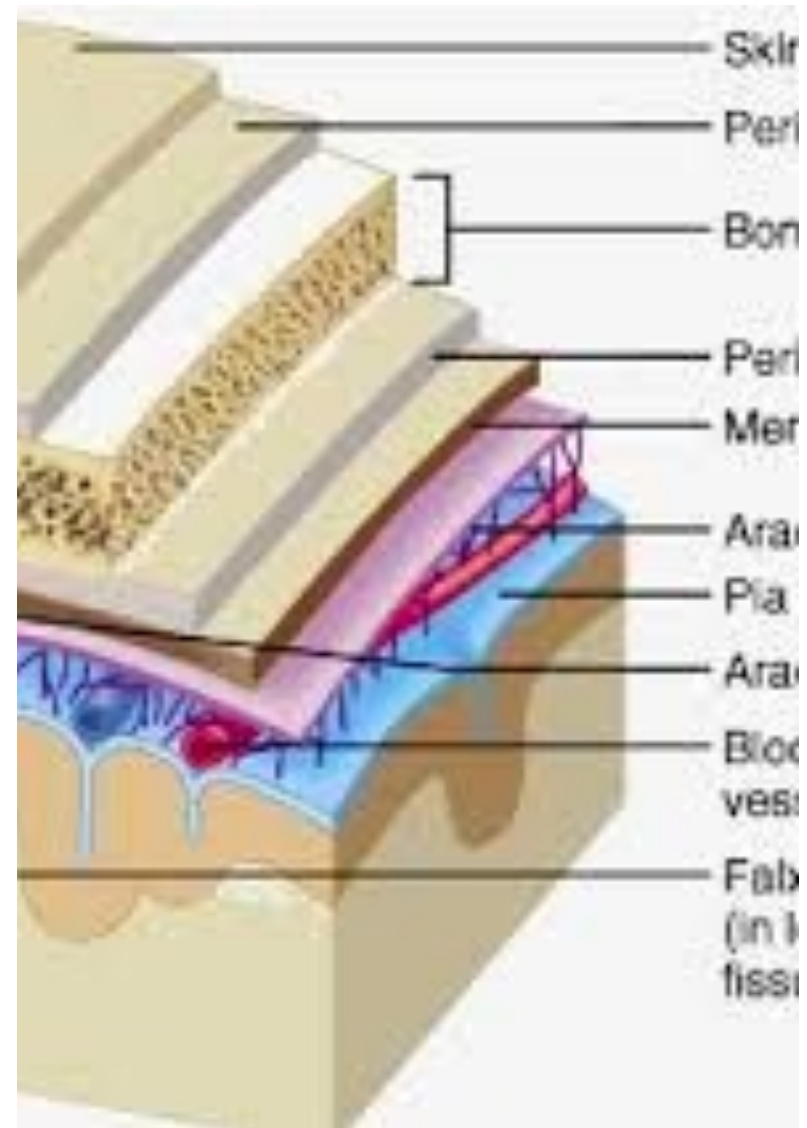
1. Crear subclases de Foto para los distintos efectos
2. Crear una jerarquía separada de efectos, hacer que Foto conozca a todos sus efectos, y agregar métodos en Foto para aplicar los distintos efectos

Ejercicio 1 - Fuerzas del problema

- Queremos agregar responsabilidades a algunos objetos individualmente y no a toda una clase
- Estas responsabilidades pueden agregarse o quitarse dinámicamente
- Si usamos herencia (solución 1) para agregar responsabilidades solo en una subclase de objetos la solución es inflexible, porque se decide estáticamente y no podríamos quitarlas
- Si usamos composición (solución 2) queda un protocolo y una responsabilidad muy grande para la clase original

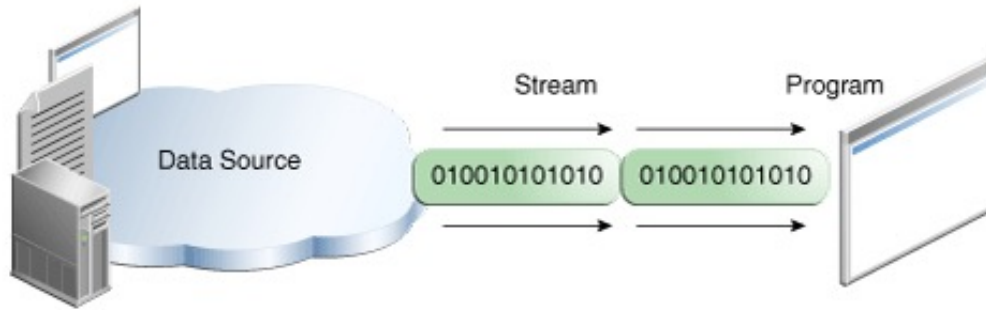
Ejercicio 1: otra solución?

- Agregar los efectos por fuera de la foto

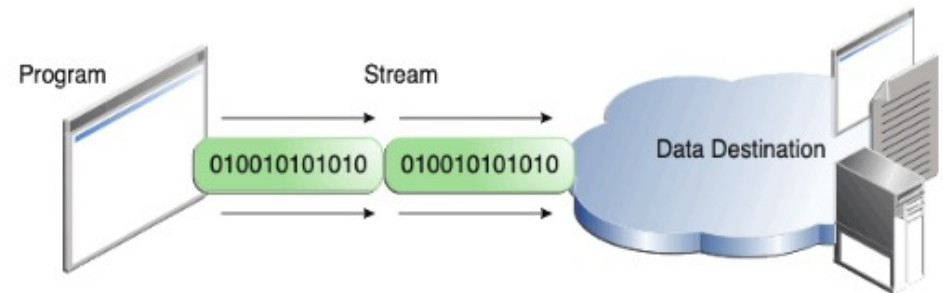


Ejercicio 2: File I/O Streams

- Cuando se necesita procesar una entrada o escribir a una salida en forma secuencial, los streams resultan la manera más eficaz.
- En Smalltalk, en Java, en .Net existe el concepto de streams para el manejo de archivos de i/o



Reading information into a program.



Writing information from a program.

Ejercicio 2: File I/O Streams

- Existen I/O streams para diferentes tipos de datos:
 - byte streams
 - character streams (con distinto encoding)
 - data streams (boolean, int, ...) / Object streams
- Tienen un protocolo que permite:
 - Abrir un stream para lectura o escritura
in = new FileInputStream("myfile.txt"); (*byte stream*)
out = new FileOutputStream("myfile.txt");

in = new FileReader("myfile.txt"); (*character stream*)
out = new FileWriter("charOutput.txt");
 - leer / escribir el elemento de la posición actual
read() / write()

Reading / writing streams

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) { in.close(); }
            if (out != null) { out.close(); }
        }
    }
}
```

Ejercicio 2: File I/O Streams

- Además los streams se pueden acceder de manera más eficiente usando buffering:
 - `BufferedInputStream`
 - `BufferedOutputStream`
- Además los streams deberían poder comprimirse /descomprimirse
 - `ZipInputStream`
 - `ZipOutputStream`
 - `GZipInputStream` - `GZipOutputStream`
 - `JarInputStream` - `JarOutputStream`

Ejercicio 2 - Fuerzas del problema = Ejercicio 1

- Queremos agregar responsabilidades a algunos objetos individualmente
- Estas responsabilidades pueden agregarse o quitarse dinámicamente
- Si usamos herencia (solución 1) para agregar responsabilidades solo en una subclase de objetos la solución es inflexible, porque se decide estáticamente y no podríamos quitarlas
- Si usamos composición (solución 2) queda un protocolo y una responsabilidad muy grande para la clase original

Ejercicio 2: Streams

- Otra manera.
- Supongamos que tenemos 3 posibilidades:
 - FileInputStream
 - BufferedInputStream
 - GZipInputStream

GZipInputStream

FileInputStream

BufferedInputStream

FileInputStream

GZipInputStream

BufferedInputStream

FileInputStream

Ejercicio 2: Streams - en código

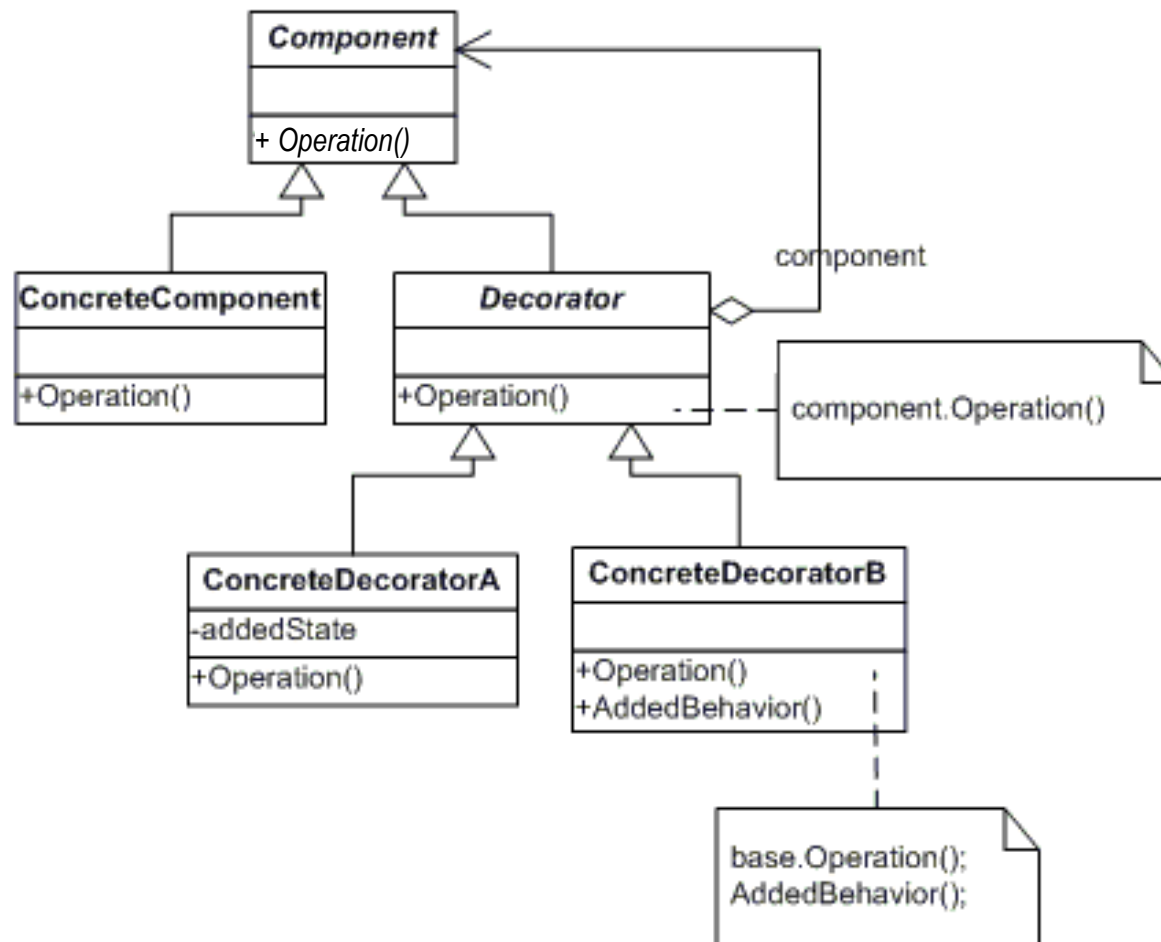
- Supongamos que tenemos una colección de objetos Java serializados y comprimidos en un archivo gzip y queremos leerlos de forma eficiente.
- 1ro hay que abrir el input stream:
`FileInputStream fis = new FileInputStream("/objects.gz");`
- 2do le incorporamos un buffer en memoria:
`BufferedInputStream bis = new BufferedInputStream(fis);`
- 3ro debemos descomprimirlo para leerlo:
`GzipInputStream gis = new GzipInputStream(bis);`
- Finalmente:
`ObjectInputStream ois = new ObjectInputStream(gis);`
`SomeObject someObject = (SomeObject) ois.readObject();`

- **Objetivo:** Agregar comportamiento a un objeto dinámicamente y en forma transparente.
- **Problema:** Cuando queremos agregar comportamiento extra a algunos objetos de una clase puede usarse herencia. El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían “mutar de clase”. El problema que tiene la herencia es que se decide estáticamente.

- Usar Decorator para:
 - agregar responsabilidades a objetos individualmente y en forma transparente (sin afectar otros objetos)
 - quitar responsabilidades dinámicamente
 - cuando subclasificar es impráctico

Patrón Decorator

- **Solución:** Definir un decorador (o “wrapper”) que agregue el comportamiento cuando sea necesario



- **Consecuencias:**

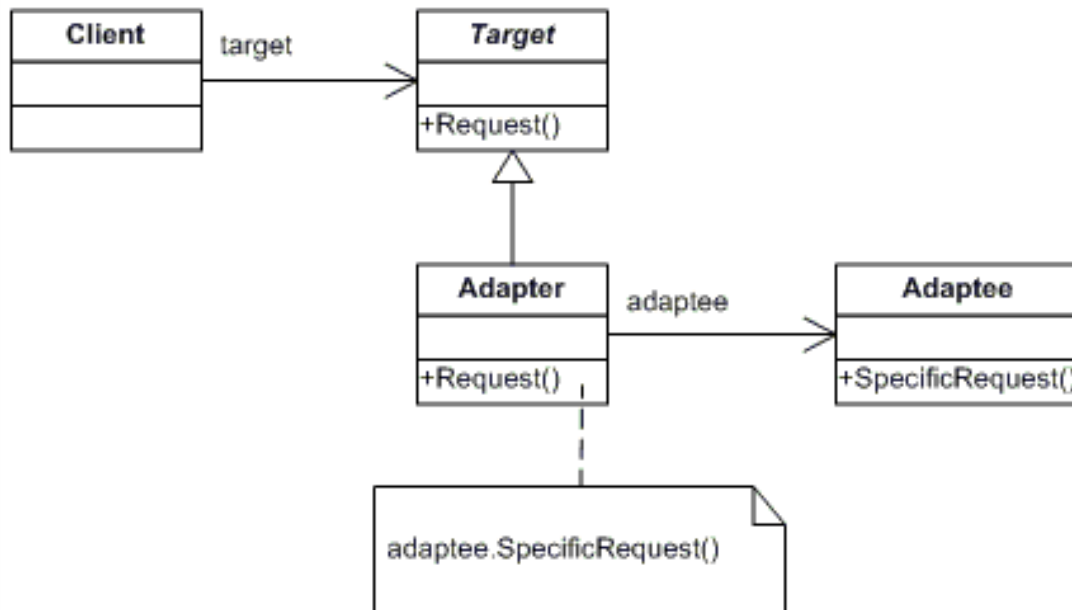
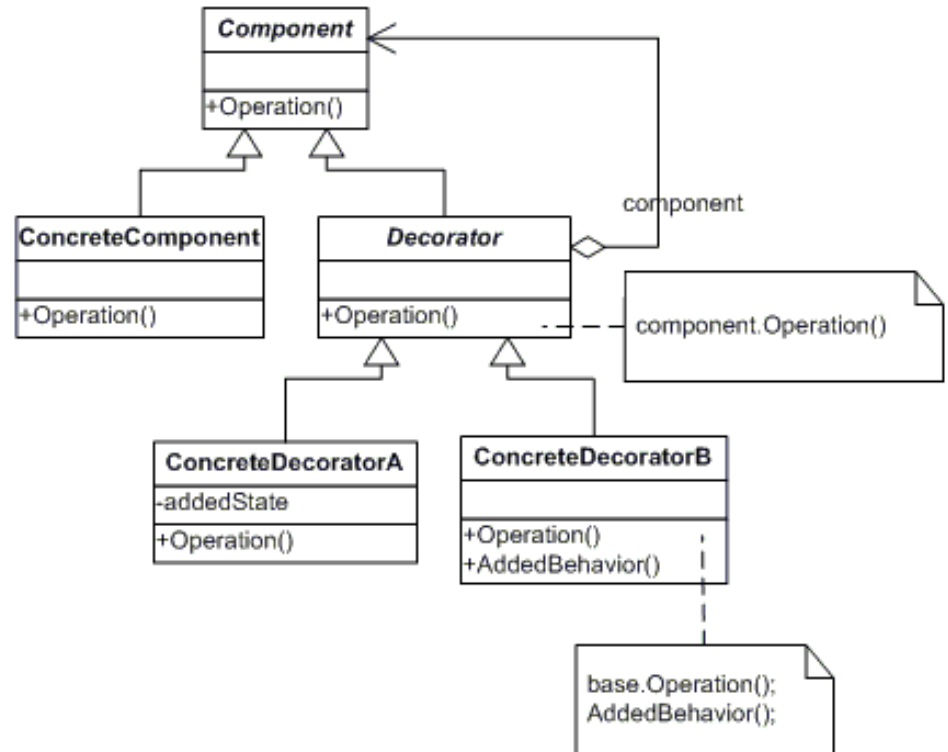
- + Permite mayor flexibilidad que la herencia.
- + Permite agregar funcionalidad incrementalmente.
- Mayor cantidad de objetos, complejo para depurar

- **Implementación:**

- Misma interface entre componente y decorador
- No hay necesidad de la clase Decorator abstracta
- Cambiar el “skin” vs cambiar sus “guts”

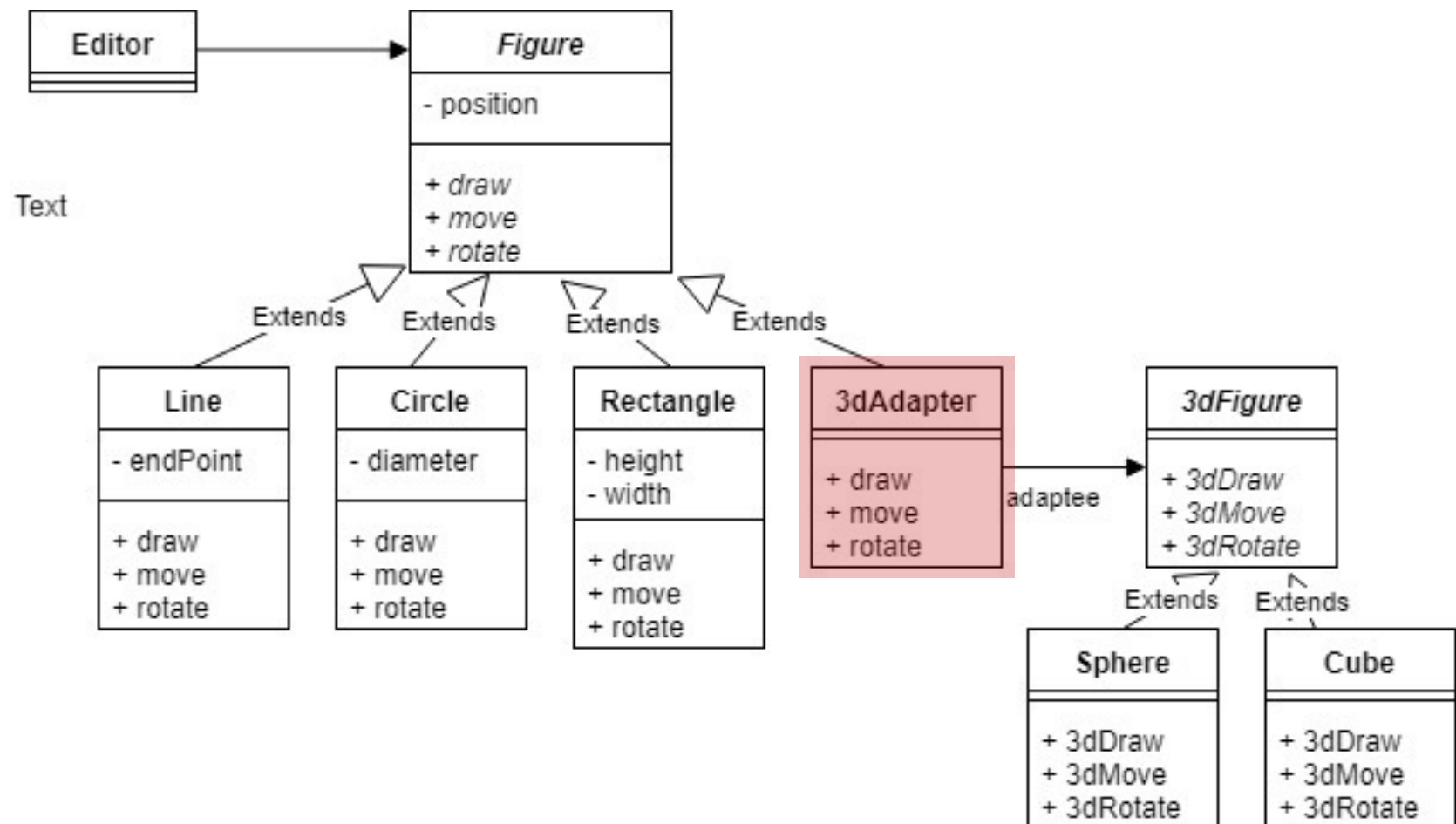
Decorator vs. Adapter

Decorator

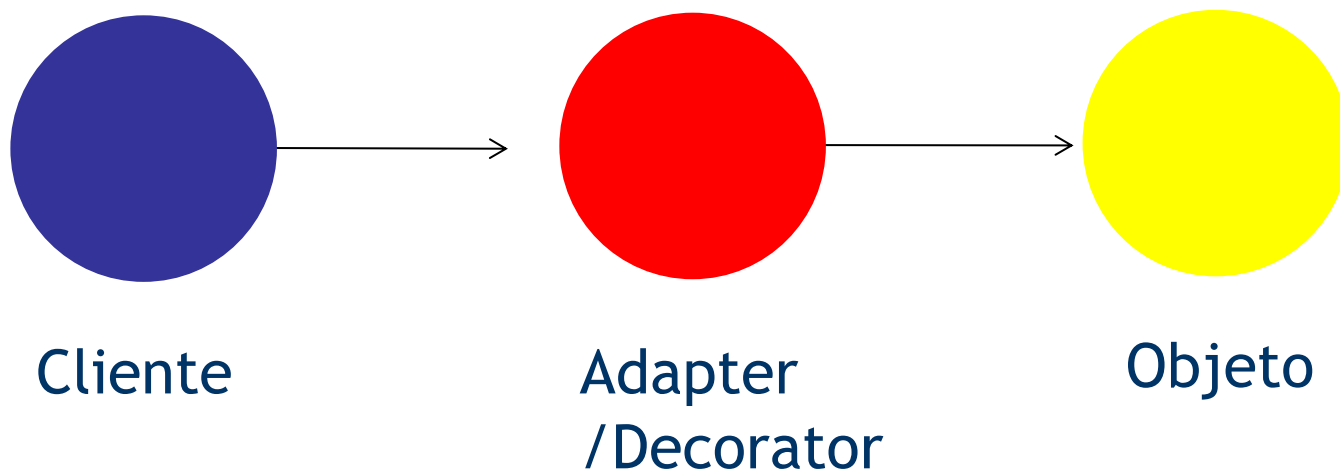


Adapter

Ejemplo de Adapter de figuras



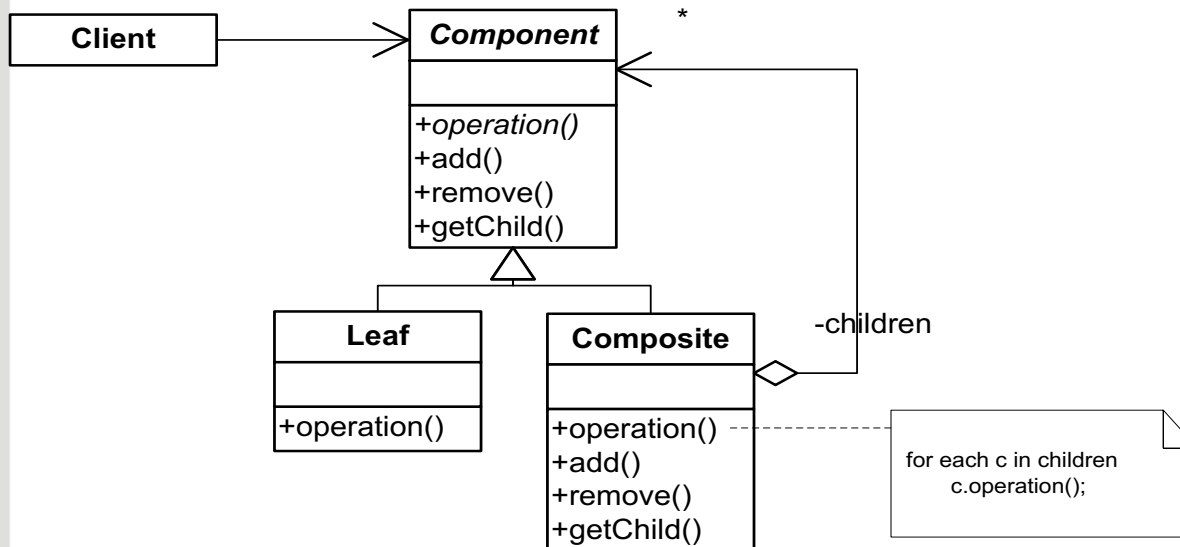
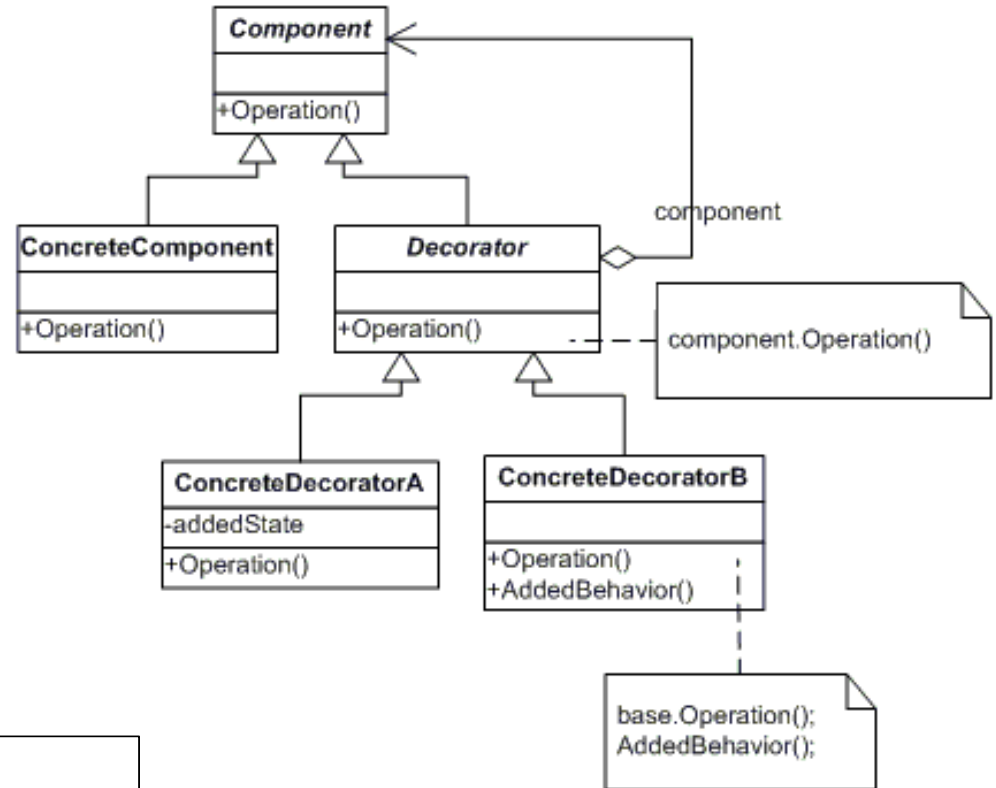
Decorator vs. Adapter (wrappers)



- Ambos patrones “decoran” el objeto para cambiarlo
- Decorator *preserva* la interface del objeto para el cliente.
- Adapter *convierte* la interface del objeto para el cliente.
- Decorators pueden y suelen anidarse.
- Adapters no se anidan.

Decorator vs. Composite

Decorator



Composite

Decorator vs. Strategy

- En qué se parecen?
 - Propósito: permitir que un objeto cambie su funcionalidad dinámicamente (agregando o cambiando el algoritmo que utiliza)
- En qué se diferencian?
 - Estructura: el Strategy cambia el algoritmo por dentro del objeto, el Decorator lo hace por fuera

