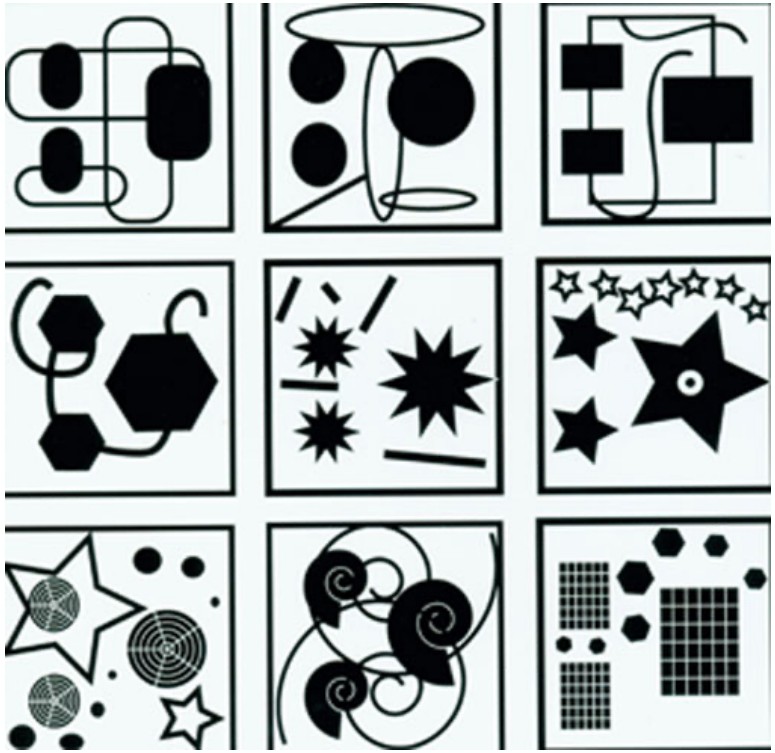
Objetos 2
Nuevos patrones: Wrappers



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

Patrones ya vistos

- 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
  - Ajuste de perspectiva
  - Tilt shift
  - 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

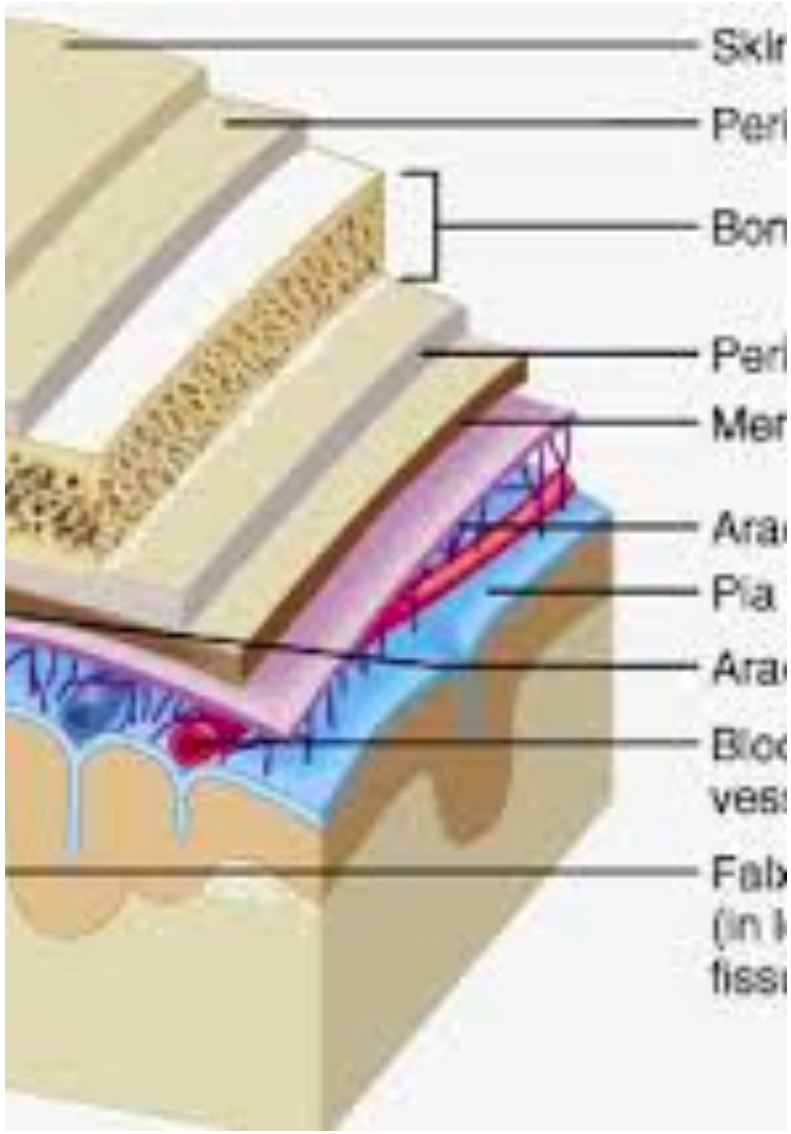
2. Crear una jerarquía separada de efectos, hacer que Foto conozca a todos sus efectos, y agregar métodos en Foto para aplicar los distintos efectos

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 solucion?

- Agregar los efectos por fuera de la foto



Ejercicio 2: Streams

- Cuando se necesita procesar una entrada o escribir a una salida, los streams resultan la mejor manera
- En Smalltalk, en Java, en .Net existe el concepto de streams y podemos encontrar una jerarquía de Stream importante, con un protocolo que permite:

  - abrir un stream (colección de acceso secuencial) para lectura o escritura
    Ej: (File named: 'myfile.pdf') readStream

  - leer / escribir el elemento de la posición actual
    #next / #nextPut: / #nextPutAll: / #peek

  - posicionar el stream
    #position / #position: / #upTo:

Ejercicio 2: Streams

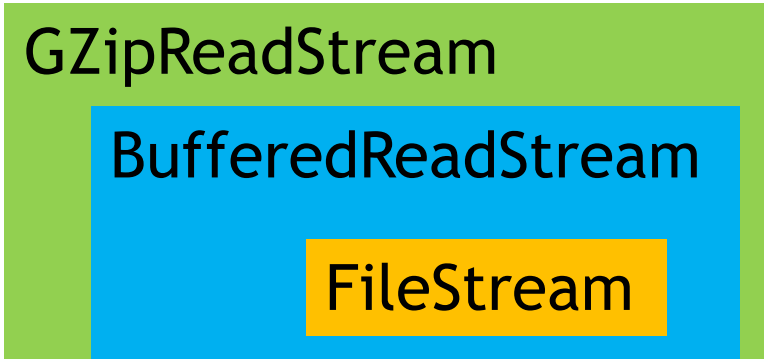
- Los streams se usan también para leer o escribir a archivos del file system.
- Los archivos pueden ser binarios o de caracteres (con distinto encoding)
  - BinaryFileStream
  - ZnCharacterReadStream - ZnCharacterWriteStream
- Deberían poder accederse de una manera eficiente usando buffering
  - ZnBufferedReadStream – ZnBufferedWriteStream
  - ... binarios o de caracteres
- Deberíamos poder comprimirlo/descomprimirlo
  - GZipReadStream – GZipWriteStream
  - ... binarios o de caracteres, buffered o no

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:
  - FileStream
  - BufferedReadStream
  - GZipReadStream



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();
```

Patrón Decorator

- **Objetivo:** Agregar comportamiento a un objeto dinámicamente y en forma transparente.

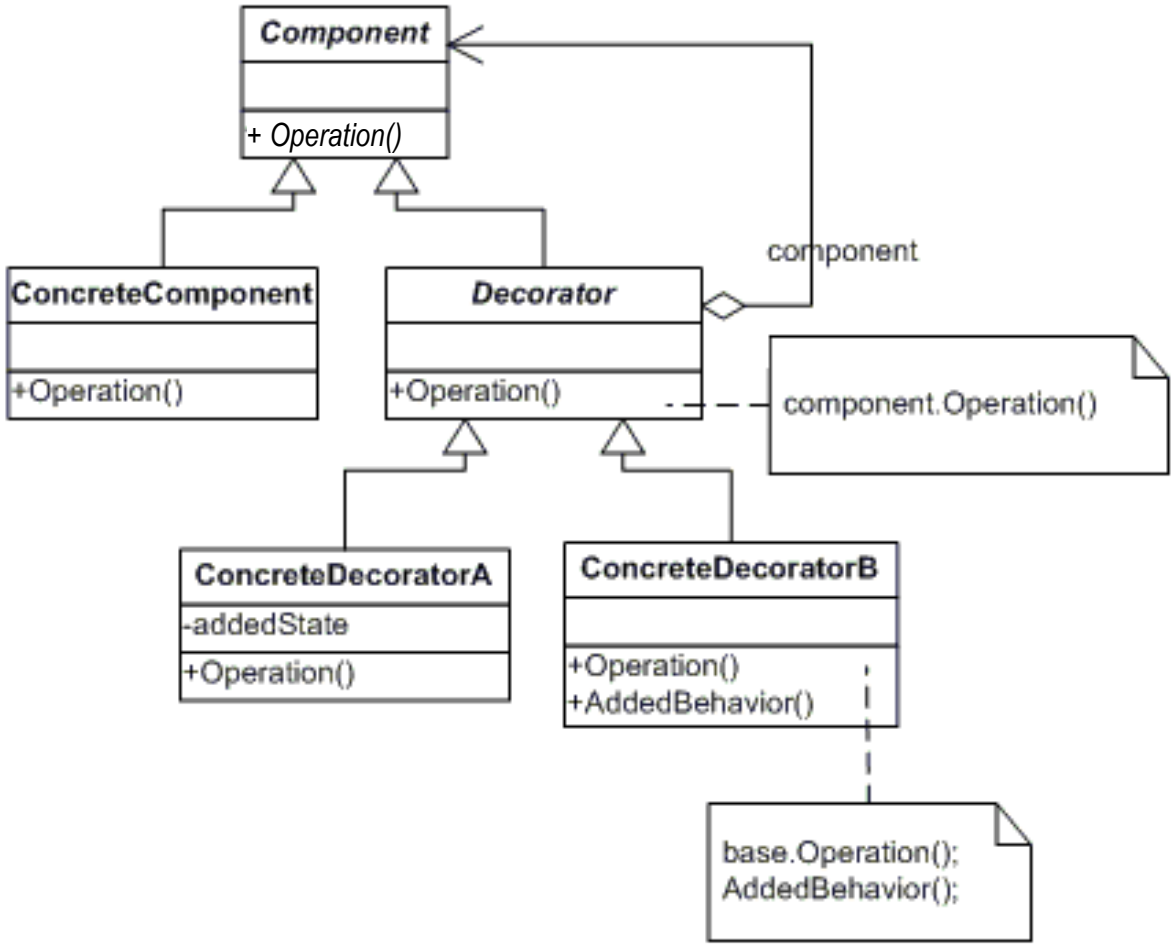
- **Problema:** Cuando queremos agregar comportamiento extra a algunos objetos de una clase puede usarse herencia. El problema es cuando necesitamos que el comportamiento se agregue o quite dinámicamente, porque en ese caso los objetos deberían “mutar de clase”. El problema que tiene la herencia es que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

Patrón Decorator - Aplicabilidad

- 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

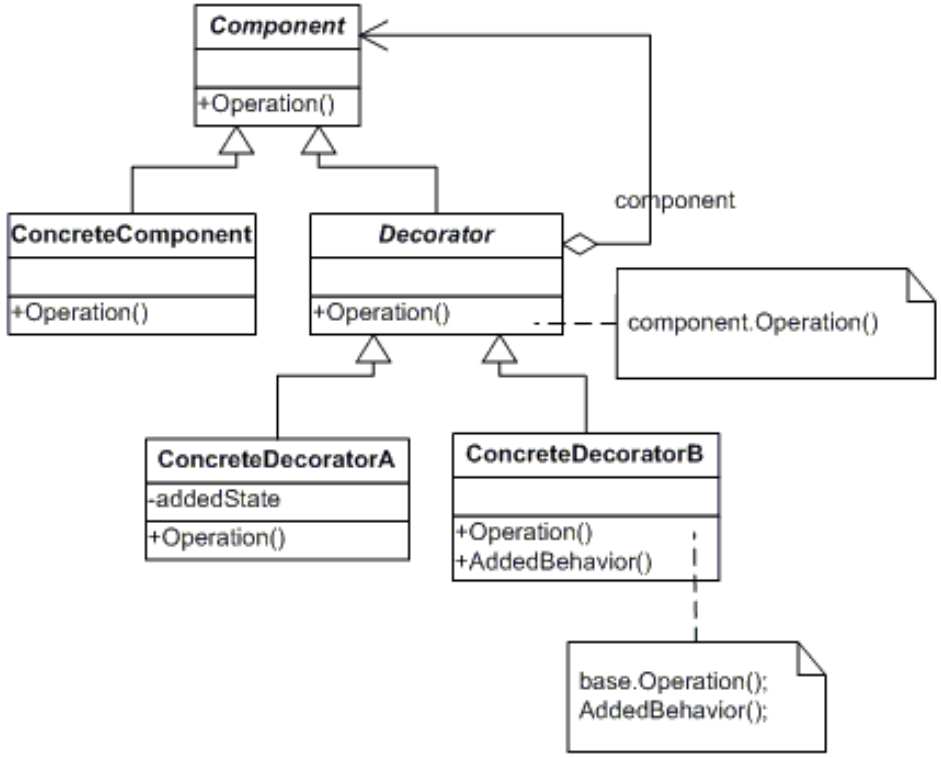


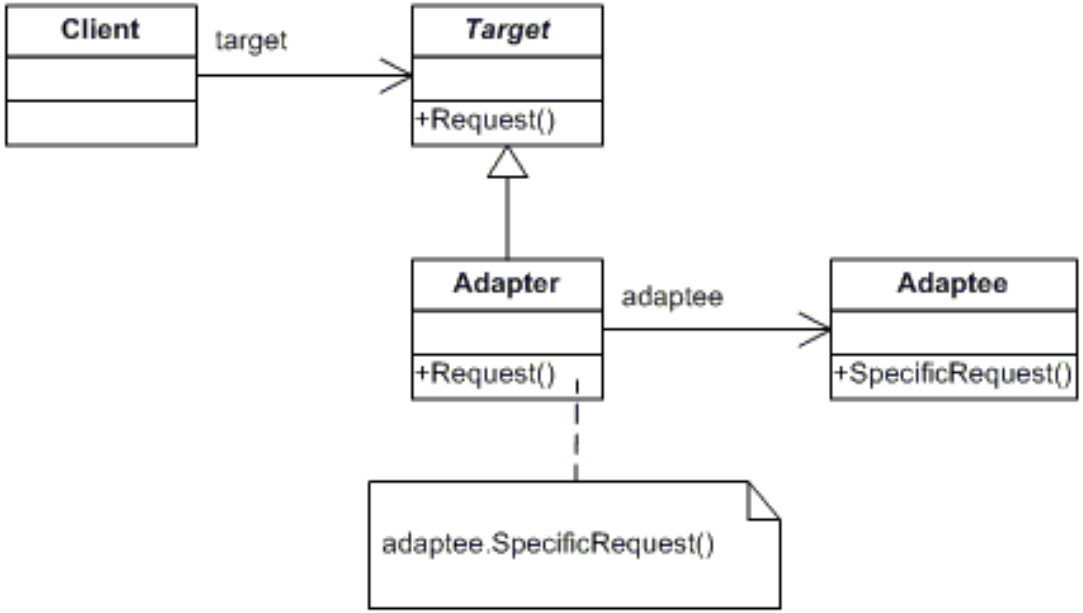
Patrón Decorator

- **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

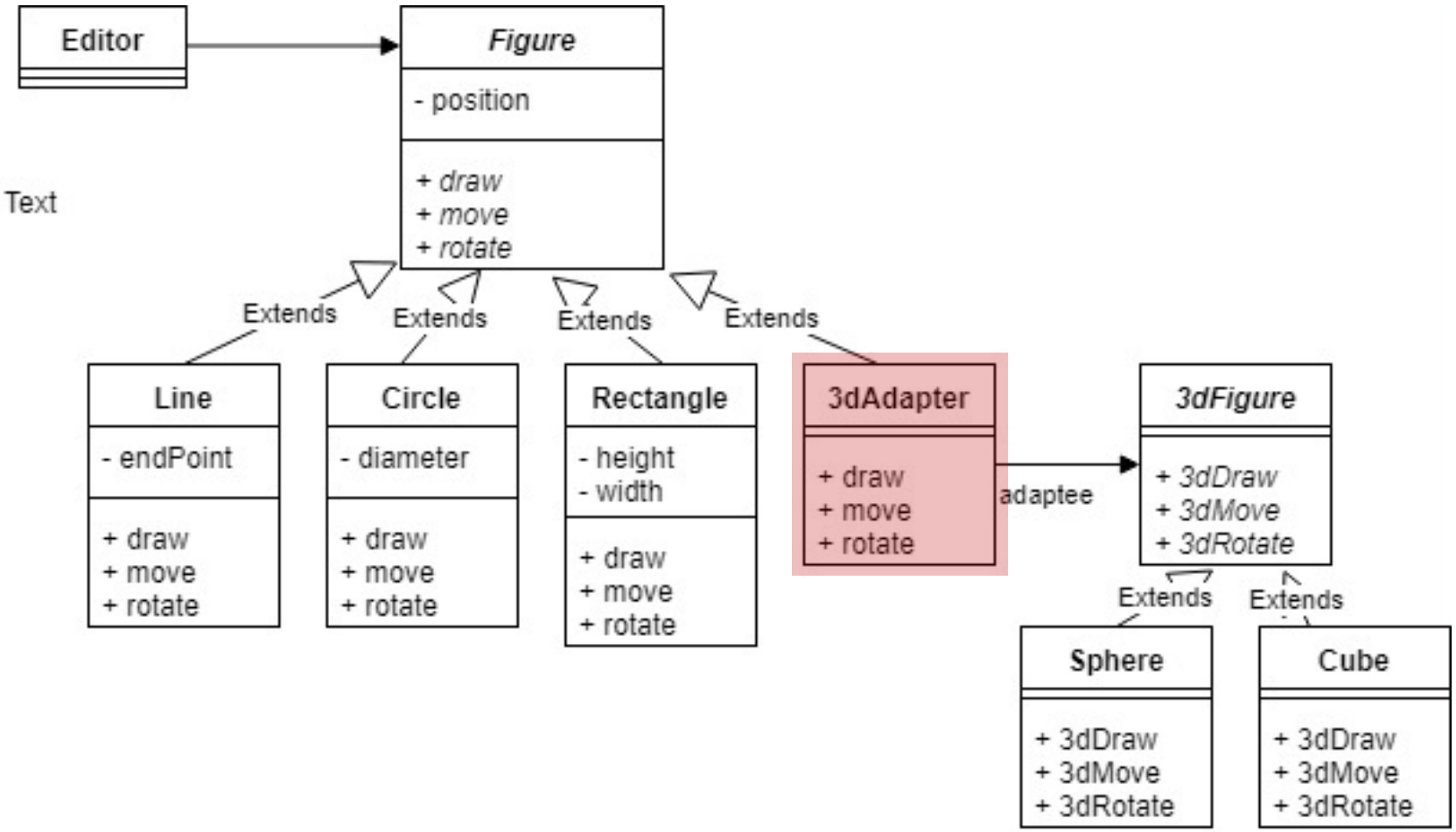
Decorator



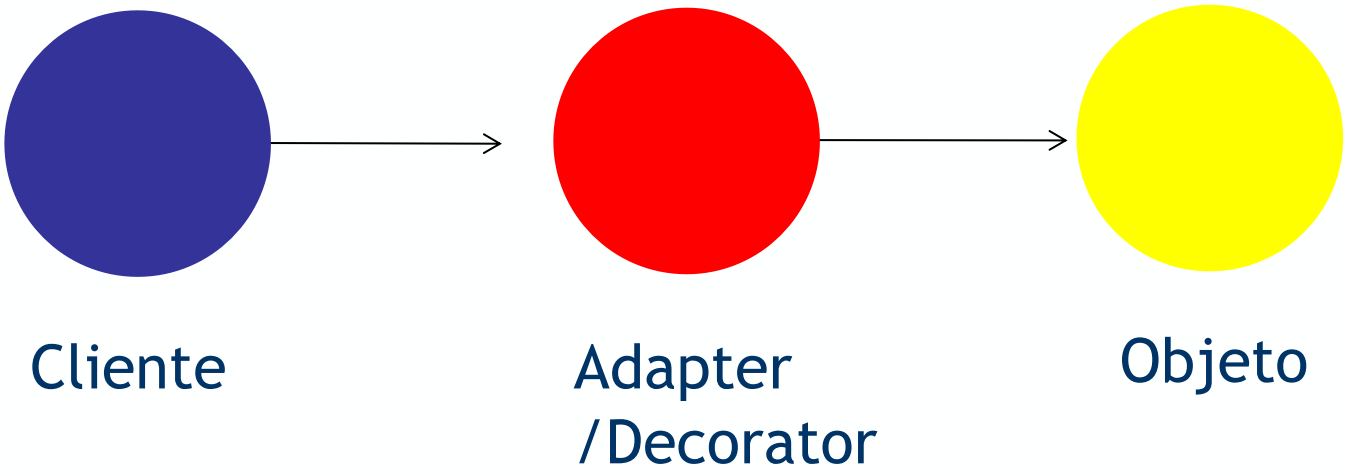


Adapter

Ejemplo de Adapter de figuras



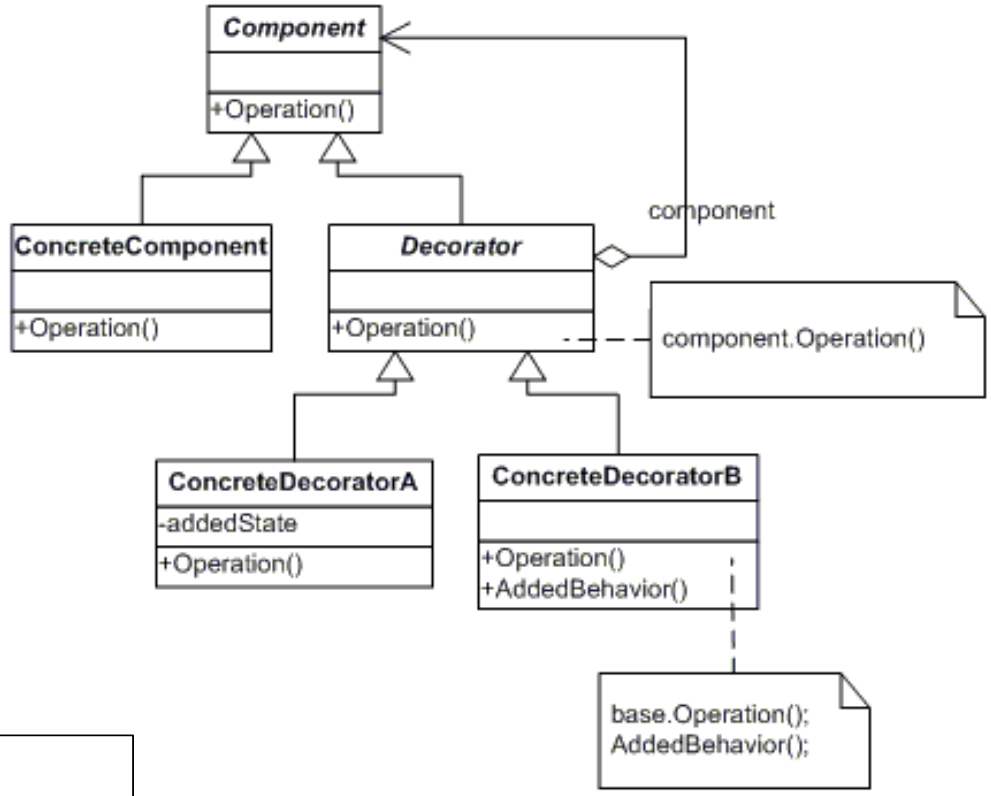
Decorator vs. Adapter (wrappers)

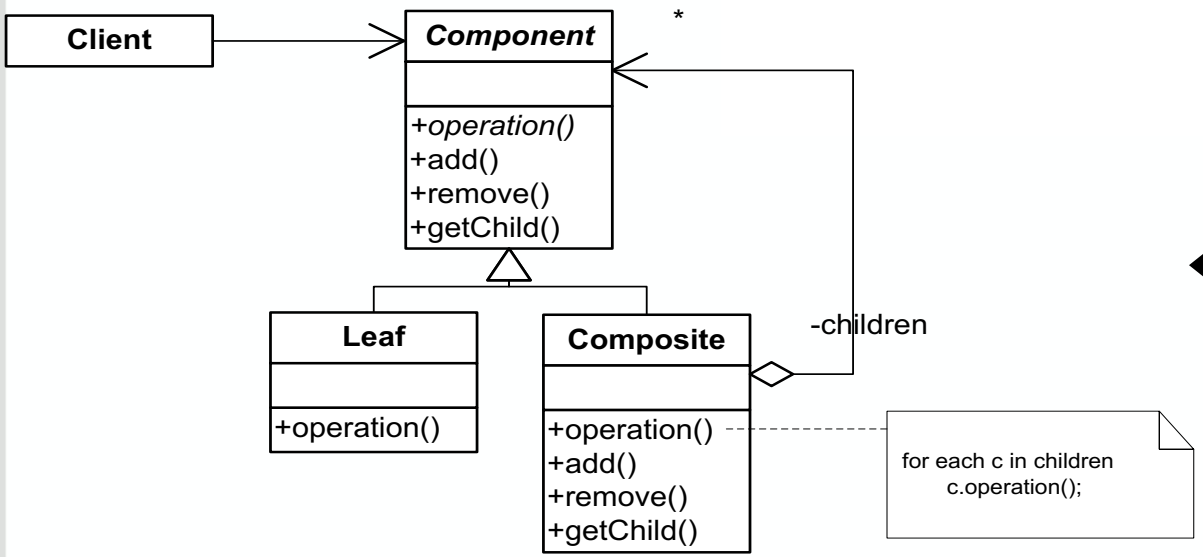


- Ambos patrones “decoran” el objeto para cambiarlo
- Decorator *preserva* la interface del objeto para el cliente.
- Adapter *convierte* la interface del objeto para el cliente.
- Decorators pueden y suelen anidarse.
- Adapters no se anidan.

Decorator vs. Composite

Decorator





Composite

Decorator vs. Strategy

- En qué se parecen?
  - Propósito: permitir que un objeto cambie su funcionalidad dinamicamente (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