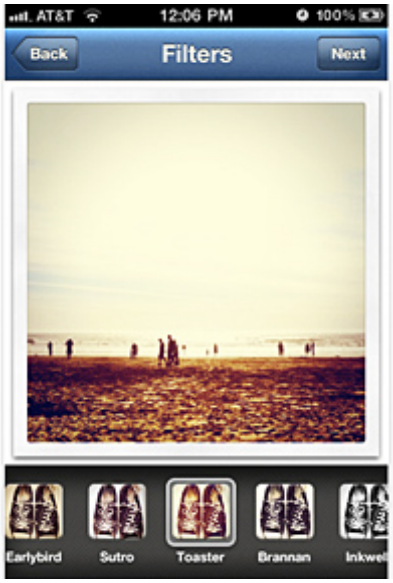
Ejercicio 1: Edición de fotos en Instagram

- 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

- Queremos agregar responsabilidades a algunos objetos individualmente y no a toda una clase

- Si usamos herencia para agregar responsabilidades solo en una subclase de objetos la solución es inflexible, porque se decide estáticamente y no podríamos quitarlas

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 podemos encontrar una jerarquía de Stream importante, y todas las subclases responden al mismo comportamiento
  - FileStream
  - BufferedReadStream
  - GZipReadStream

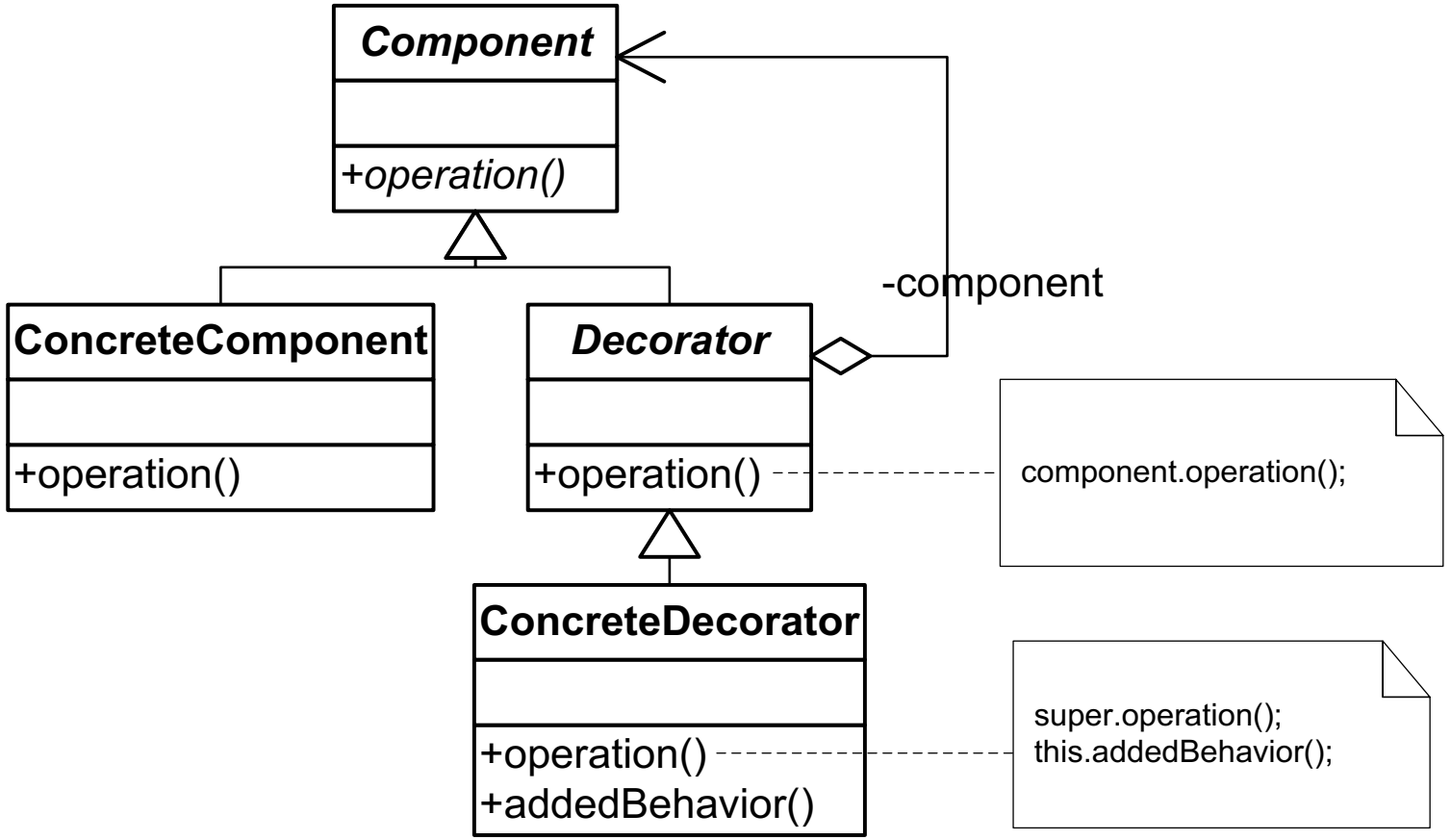
Patrón Decorator

- **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

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

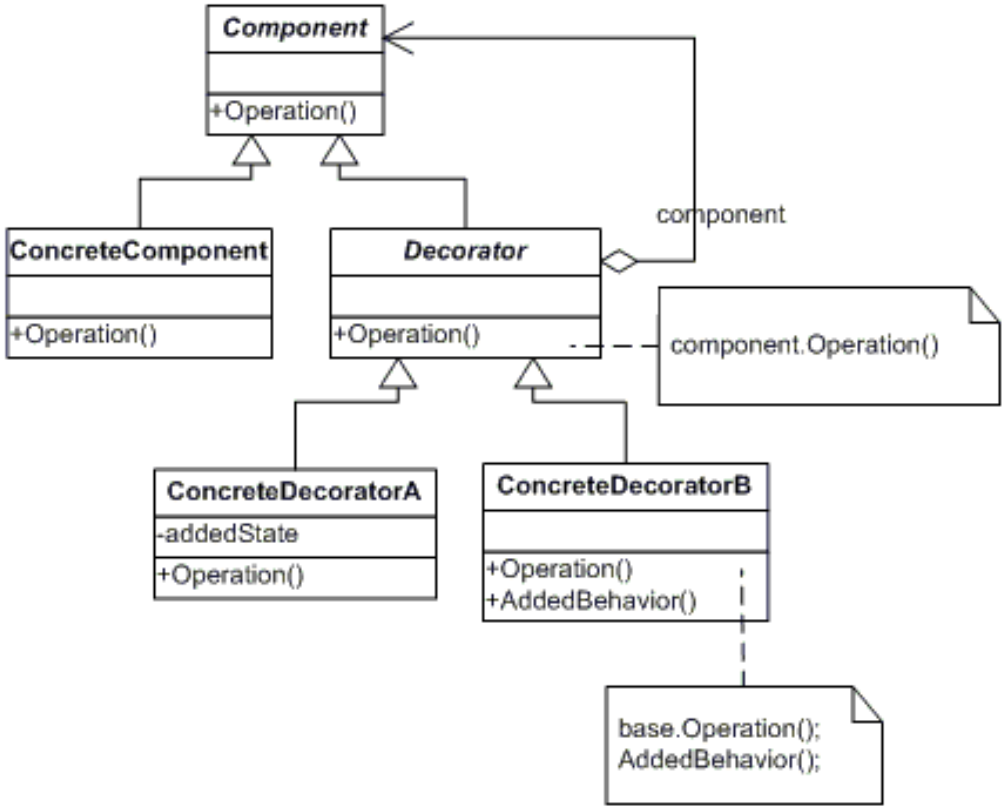


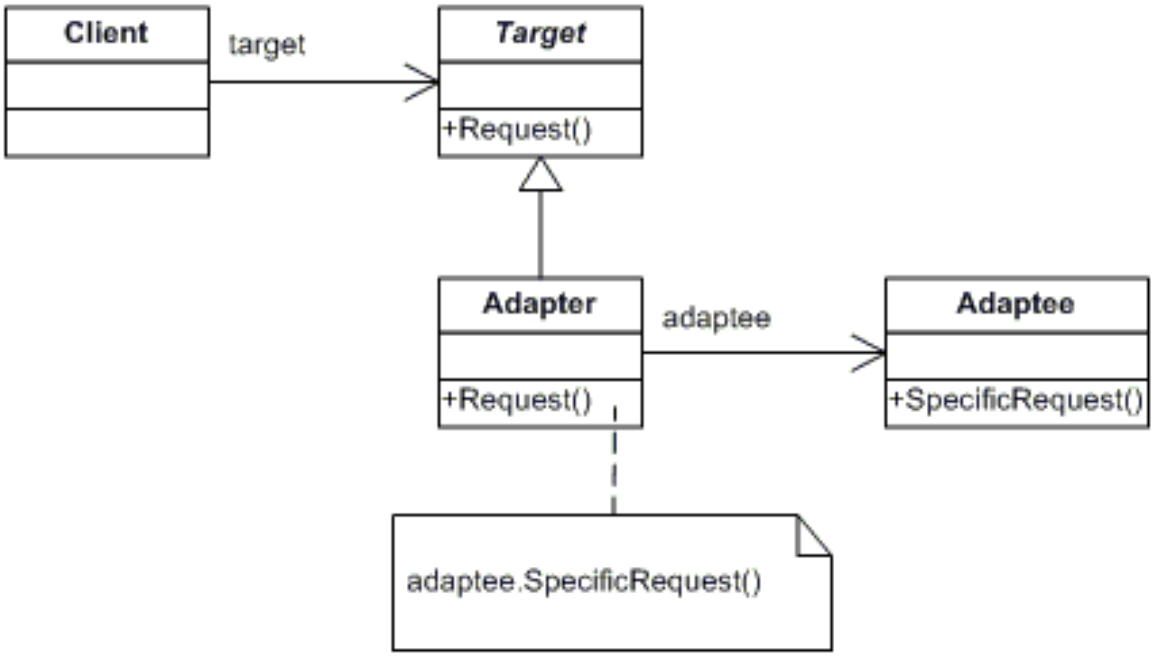
Patrón Decorator

- **Consecuencias:**
  - + Permite mayor flexibilidad que la herencia.
  - + Permite agregar funcionalidad incrementalmente.
  - – Mayor cantidad de objetos
- **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



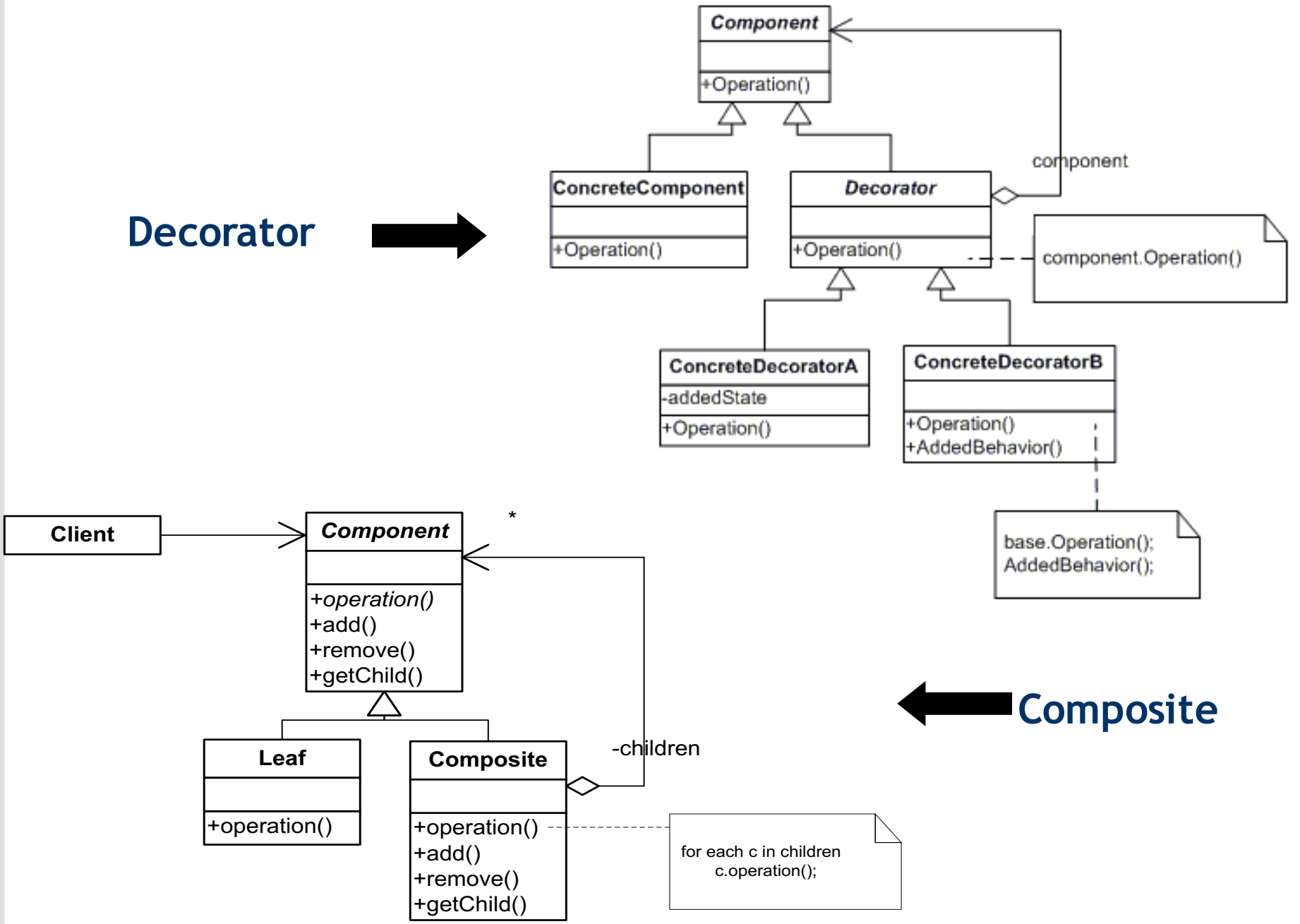


Adapter

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

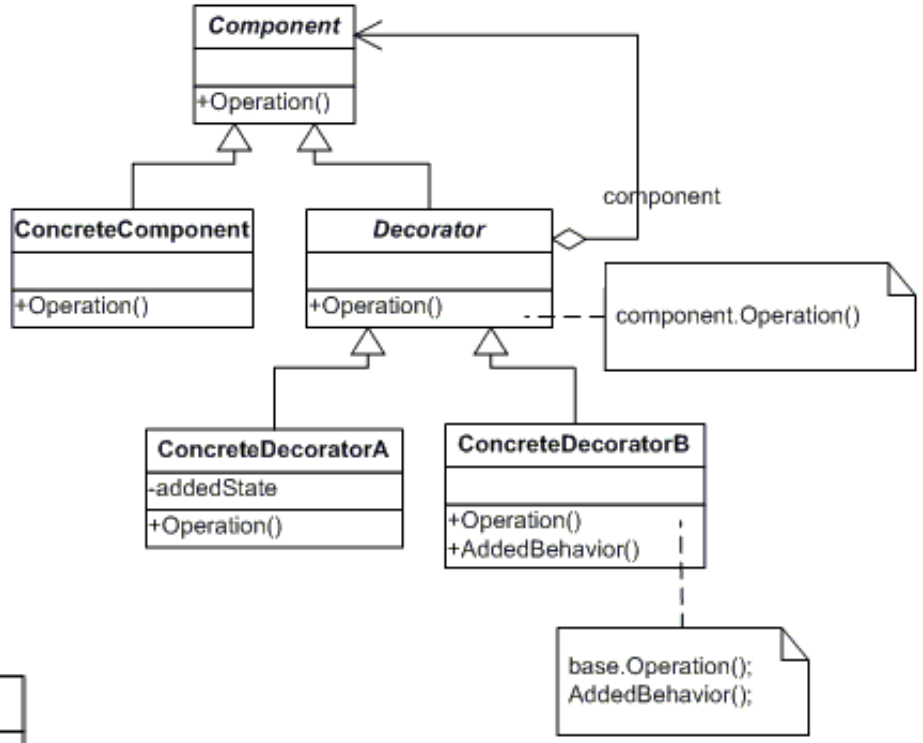


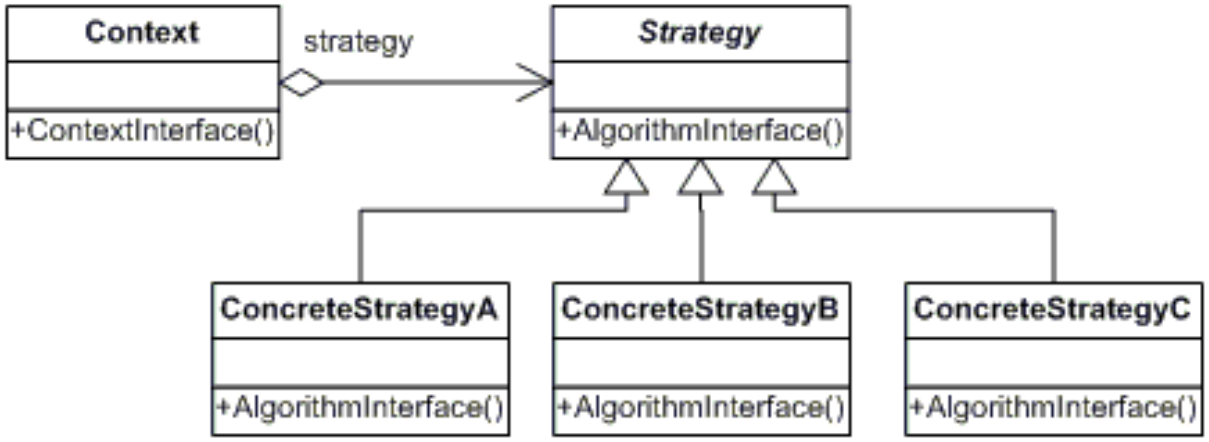
Decorator vs. Composite

- En qué se parecen?
- En qué se diferencian?

Decorator vs. Strategy

Decorator





Strategy

Decorator vs. Strategy

- Ambos permiten cambiar el comportamiento de un objeto.
- Decorator cambia el envoltorio.
- Pero
  - Decorator cambia el envoltorio vs.
  - ≠ Strategy cambia el objeto internamente.
  - El componente no conoce quien o quienes lo decoran
  - ≠ Con Strategy, el componente debe conocer a sus posibles strategies (menos extensible).
  - El protocolo de Decorator debe ser = al de su Component.
  - ≠ El Strategy puede tener otro protocolo. Entonces, Strategy conviene cuando la clase Component es muy pesada.