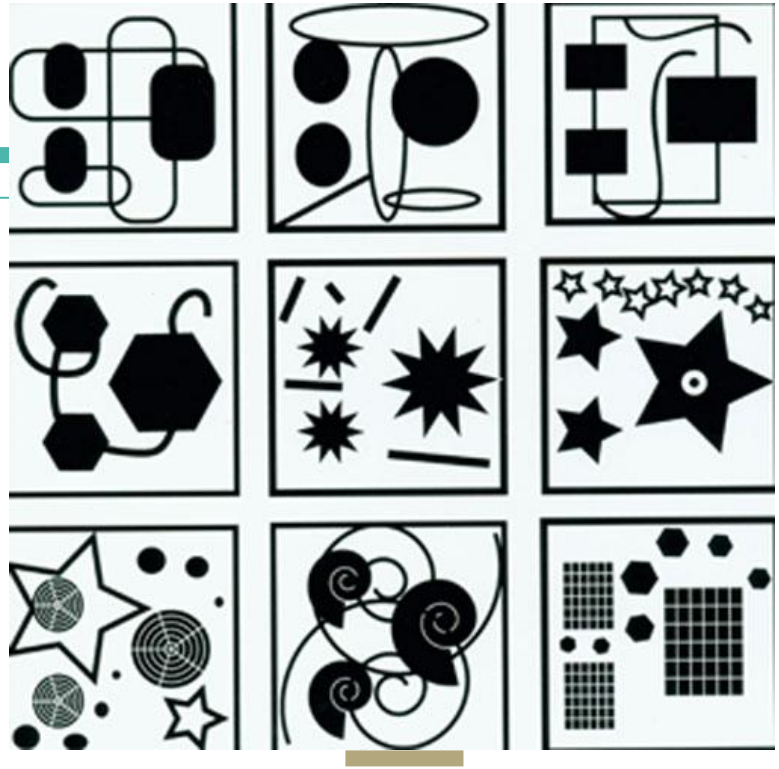
Nuevos patrones: Wrappers



Gabriela Perez:
gperez@lifia.info.unlp.edu.ar

Patrones ya vistos

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

Hoy veremos 2 nuevos patrones estructurales

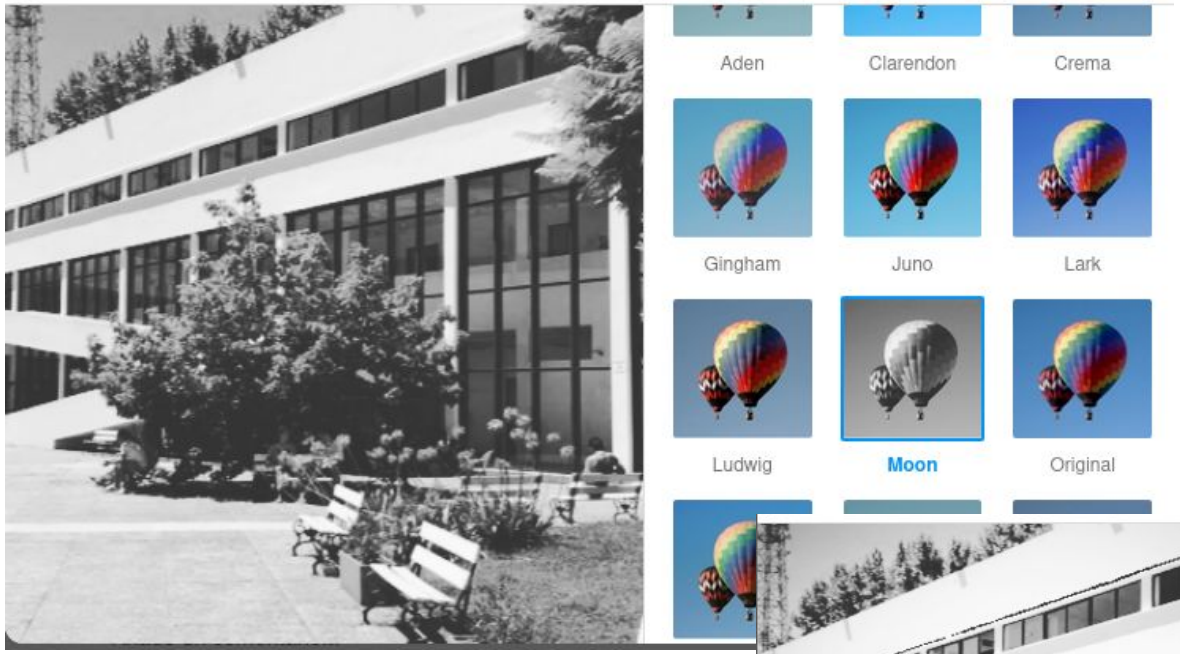
Ejemplo 1: Edición de fotos en Instagram

Cada foto puede responder información de

- fecha y hora cuando fue tomada
- resolución
- con que dispositivo
- el tamaño del archivo
- mostrarse (display)

Ejemplo 1: Edición de fotos en Instagram

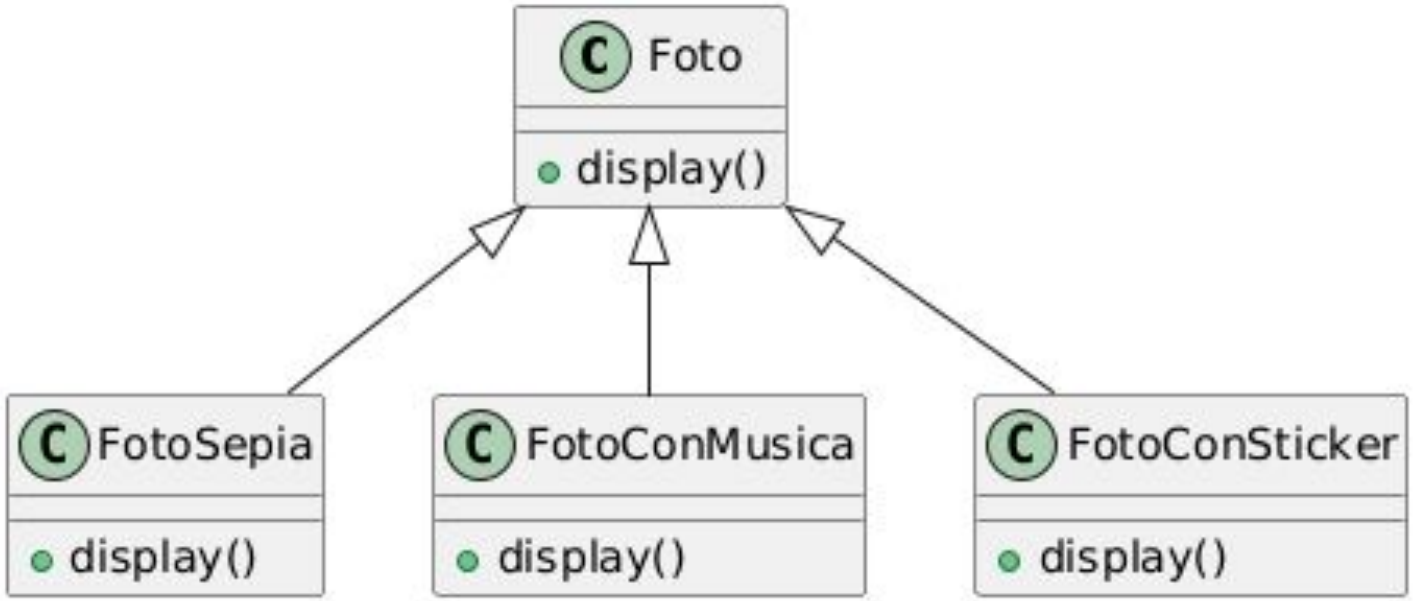
- A cada foto podemos aplicarle distintos efectos:
  - Filtros
  - Brillo, contraste, temperatura, saturación, etc. Etc.
  - Quiz, sticker, location, music



- Cada una de estas características puede agregarse o quitarse.

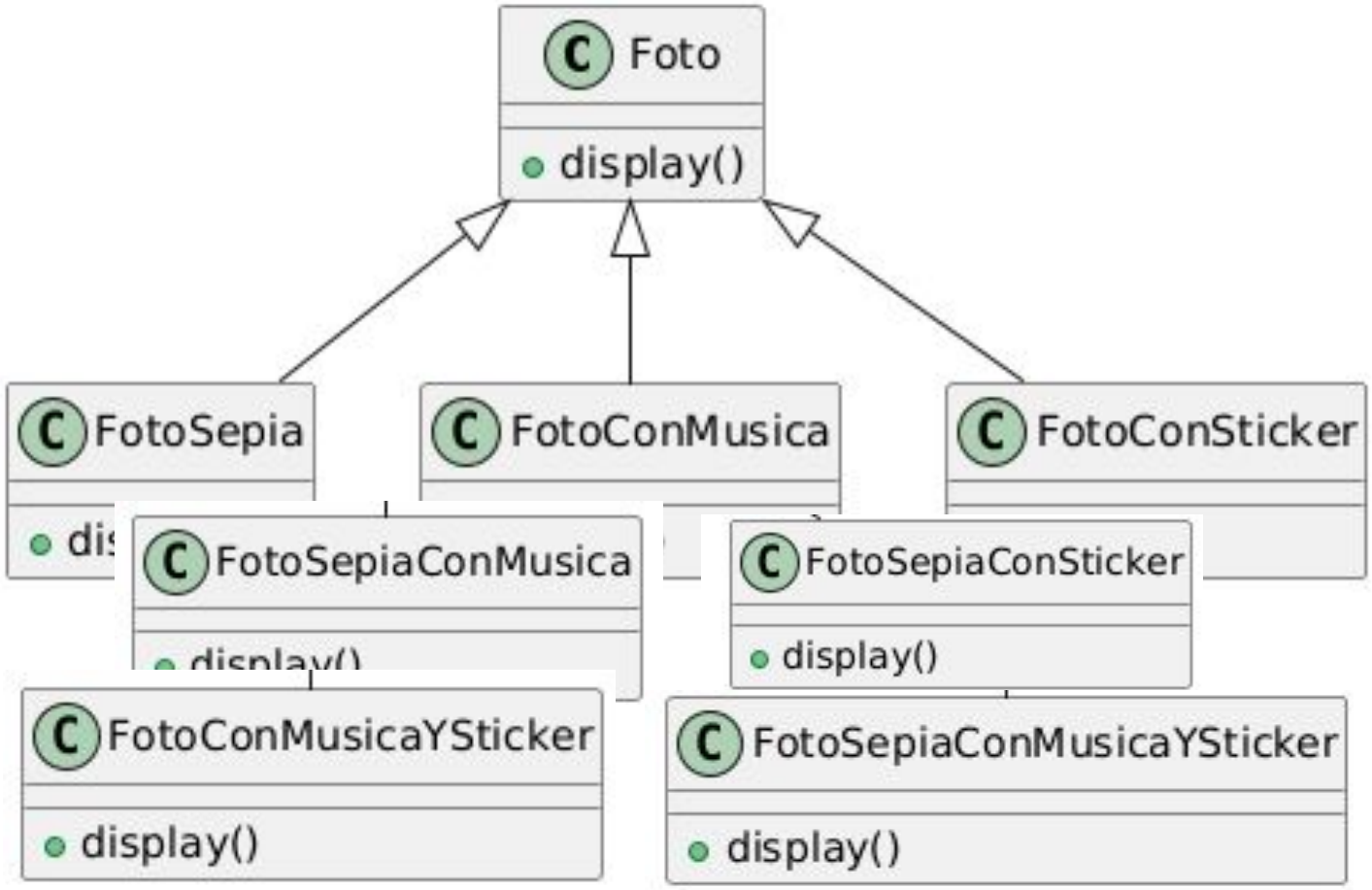
Ejemplo 1: Soluciones posibles?

1. Crear subclases de Foto para los distintos efectos



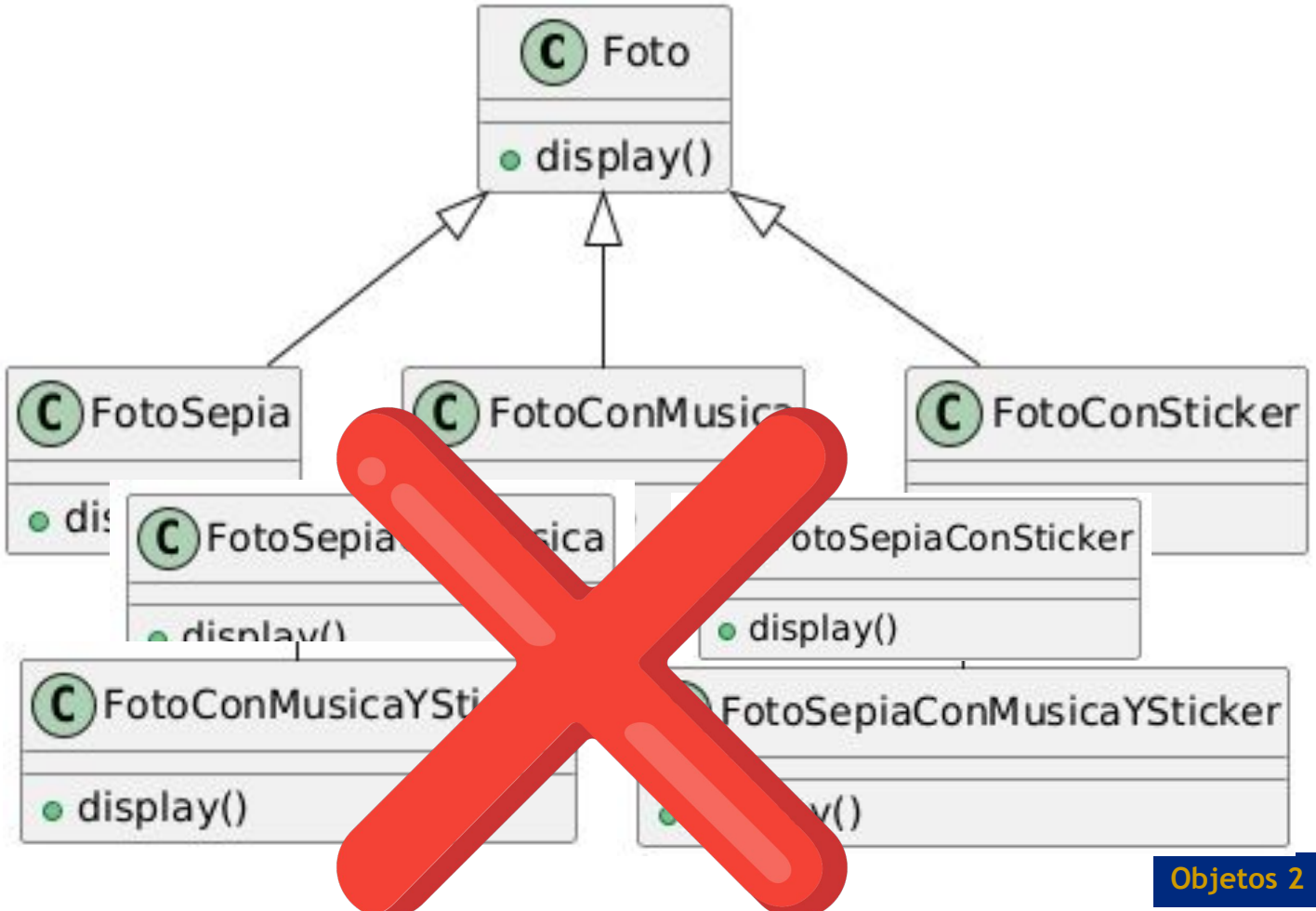
Ejemplo 1: Soluciones posibles?

1. Crear subclases de Foto para los distintos efectos



Ejemplo 1: Soluciones posibles?

1. Crear subclases de Foto para los distintos efectos



Ejemplo 1: Soluciones posibles?

1. Crear subclases de Foto para los distintos efectos

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

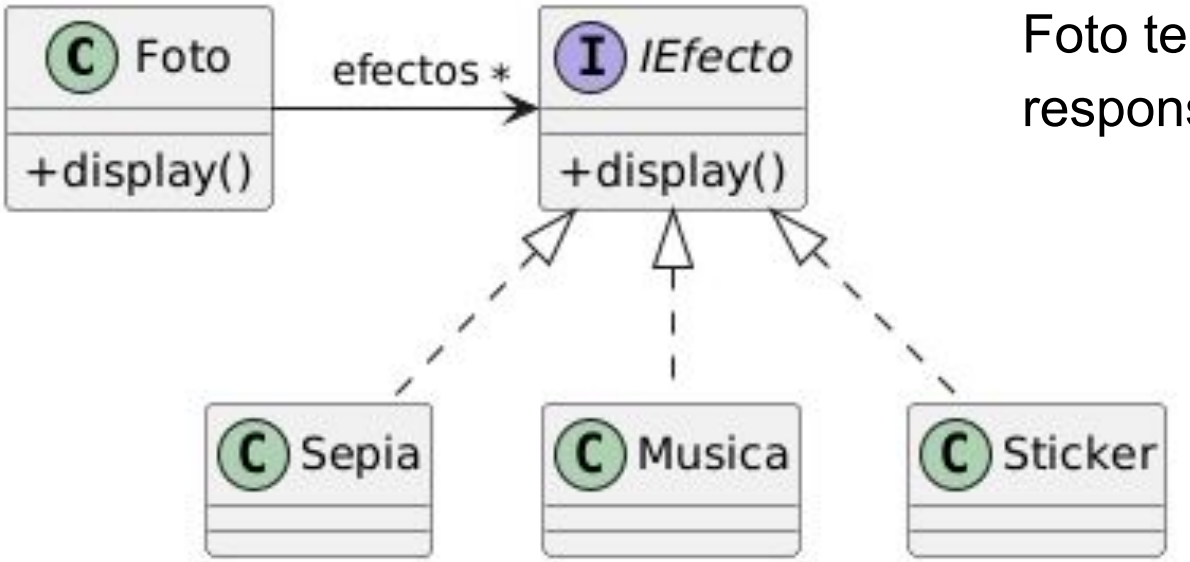
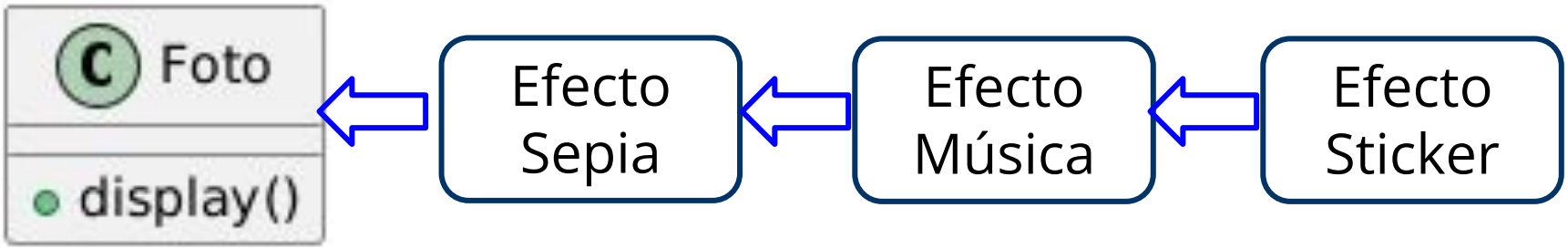


Foto tendrá más responsabilidades

Ejemplo 1: Soluciones posibles?

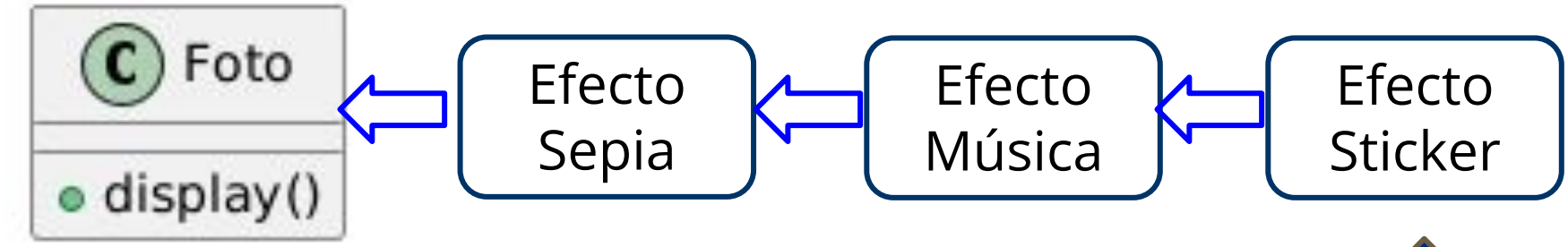
3er solución

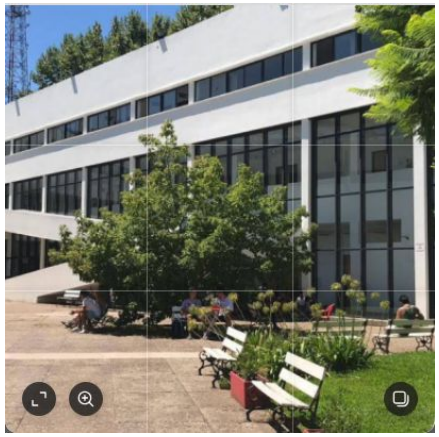


Ejemplo 1: Soluciones posibles?

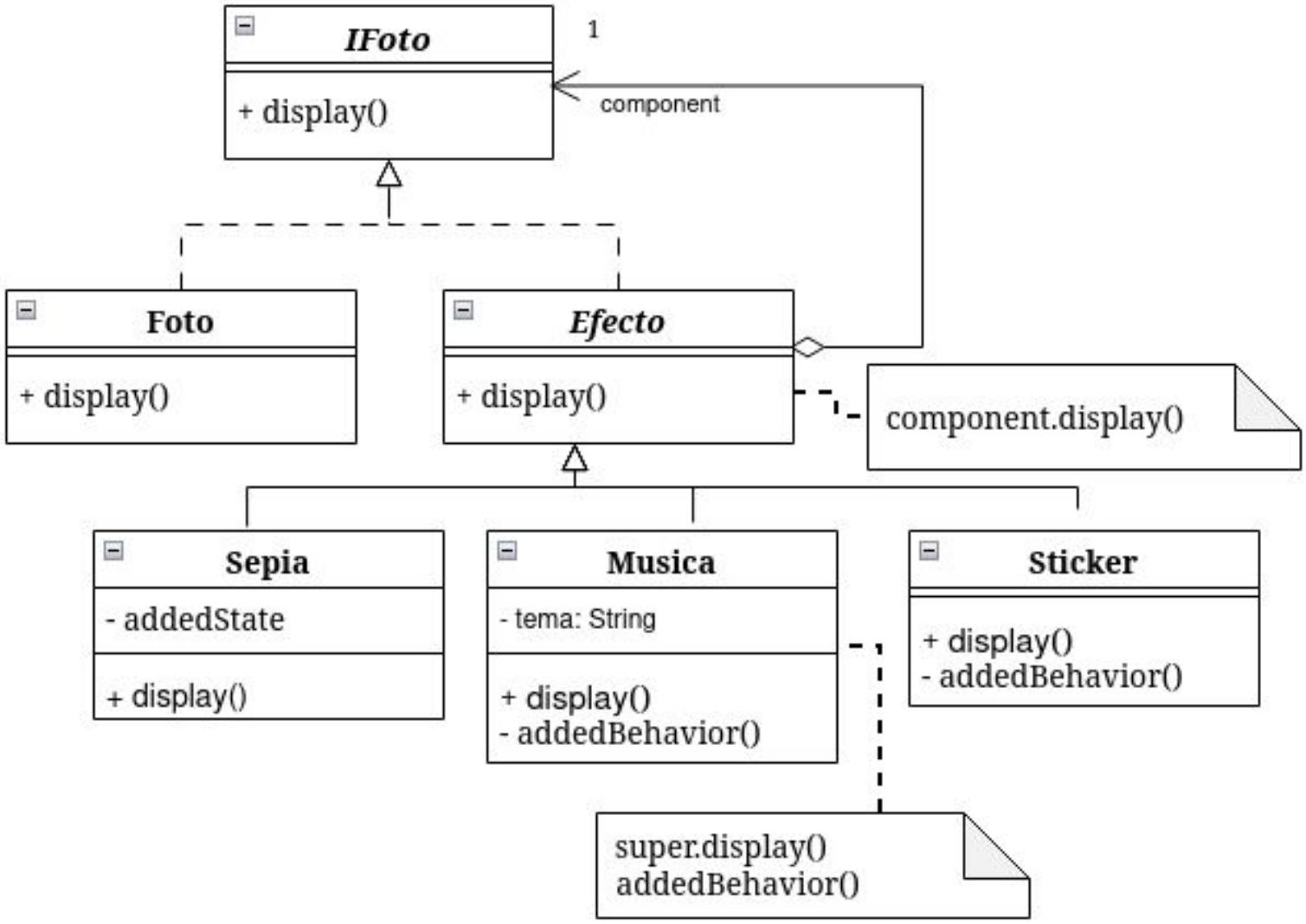
3er solución

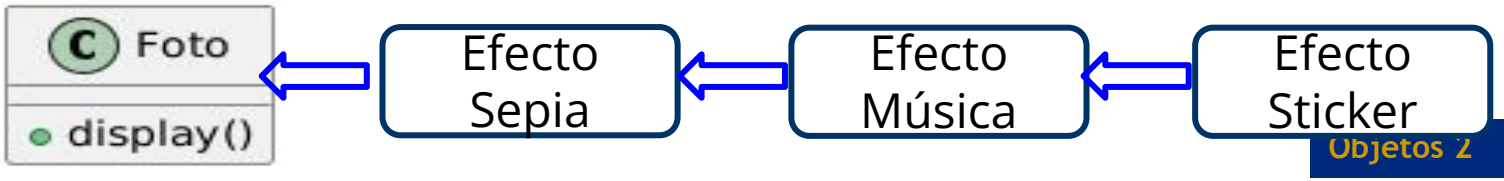
display display display display



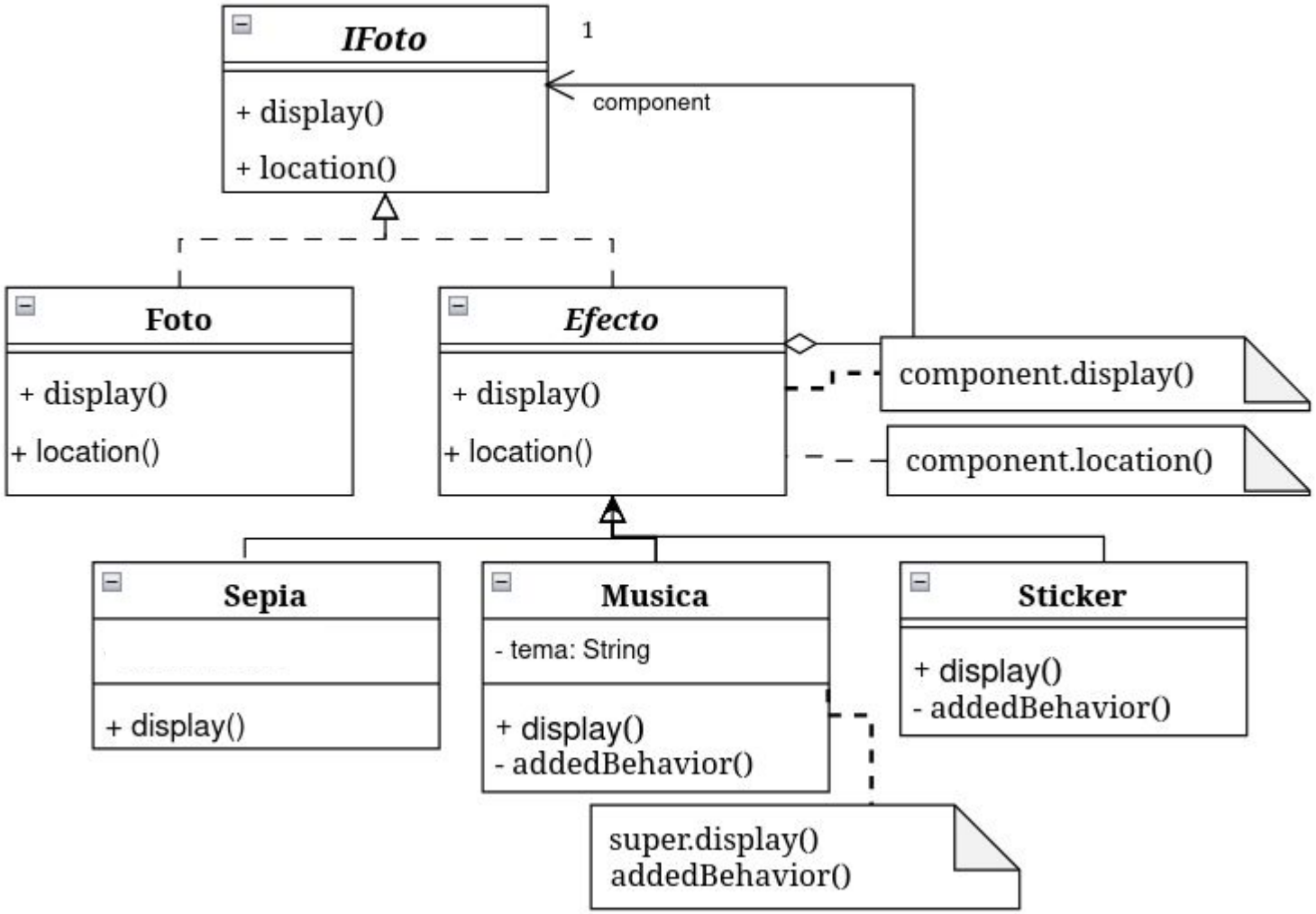


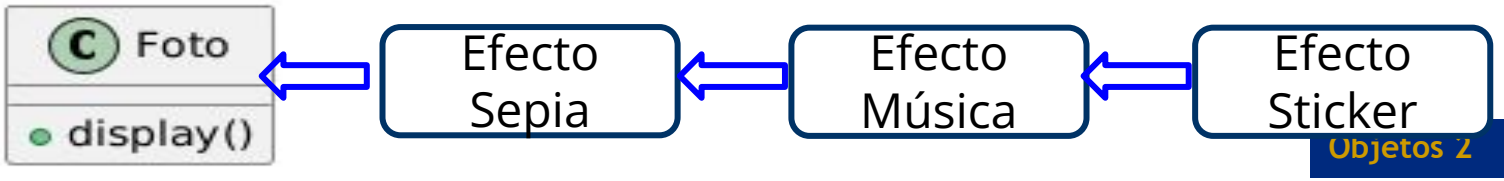
Ejemplo 1: Edición de fotos en Instagram





Ejemplo 1: Edición de fotos en Instagram





Ejemplo 1 – Fuerzas del problema

Christopher Alexander, arquitecto, define las **fuerzas del problema** como las tensiones, necesidades o condiciones en conflicto que emergen en un contexto determinado y que el diseño debe abordar.
Un buen diseño es aquel que logra equilibrar armónicamente esas fuerzas en tensión

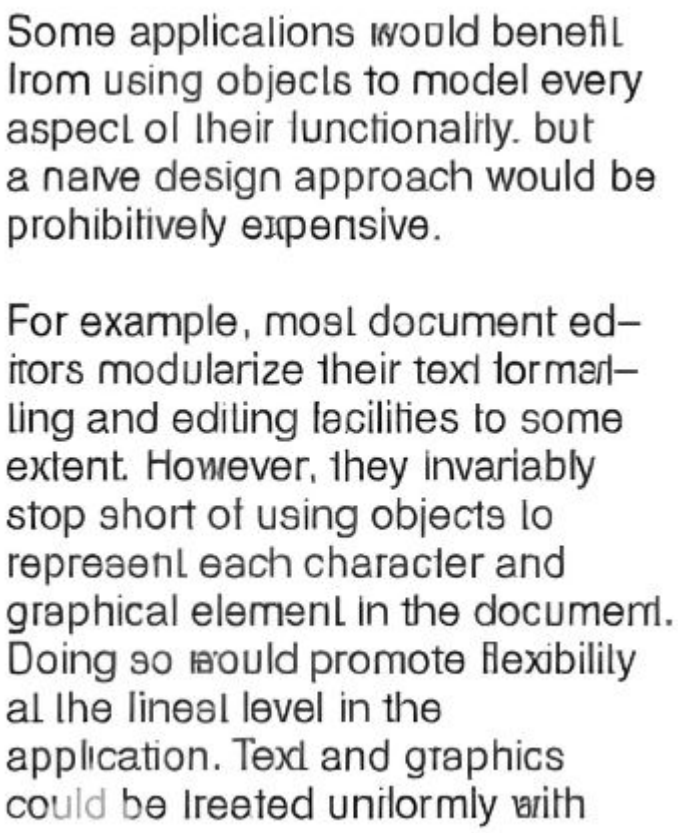
- Queremos agregar responsabilidades a algunos objetos individualmente y no a todas las responsabilidades a una clase
- Estas responsabilidades deben poder agregarse o quitarse dinámicamente
- El usuario debería poder elegir cualquier combinación de efectos, y en cualquier orden.
- Debería ser fácil agregar nuevos tipos de efectos sin necesidad de modificar los otros efectos

Ejemplo 1 – Fuerzas del problema

- Si usamos herencia (solución 1) para agregar responsabilidades en las subclases la solución es inflexible, porque se decide estáticamente y no podríamos quitarlas, ni cambiarlas, manteniendo la instancia

- Si usamos composición (solución 2) queda un protocolo y una responsabilidad muy grande para la clase original

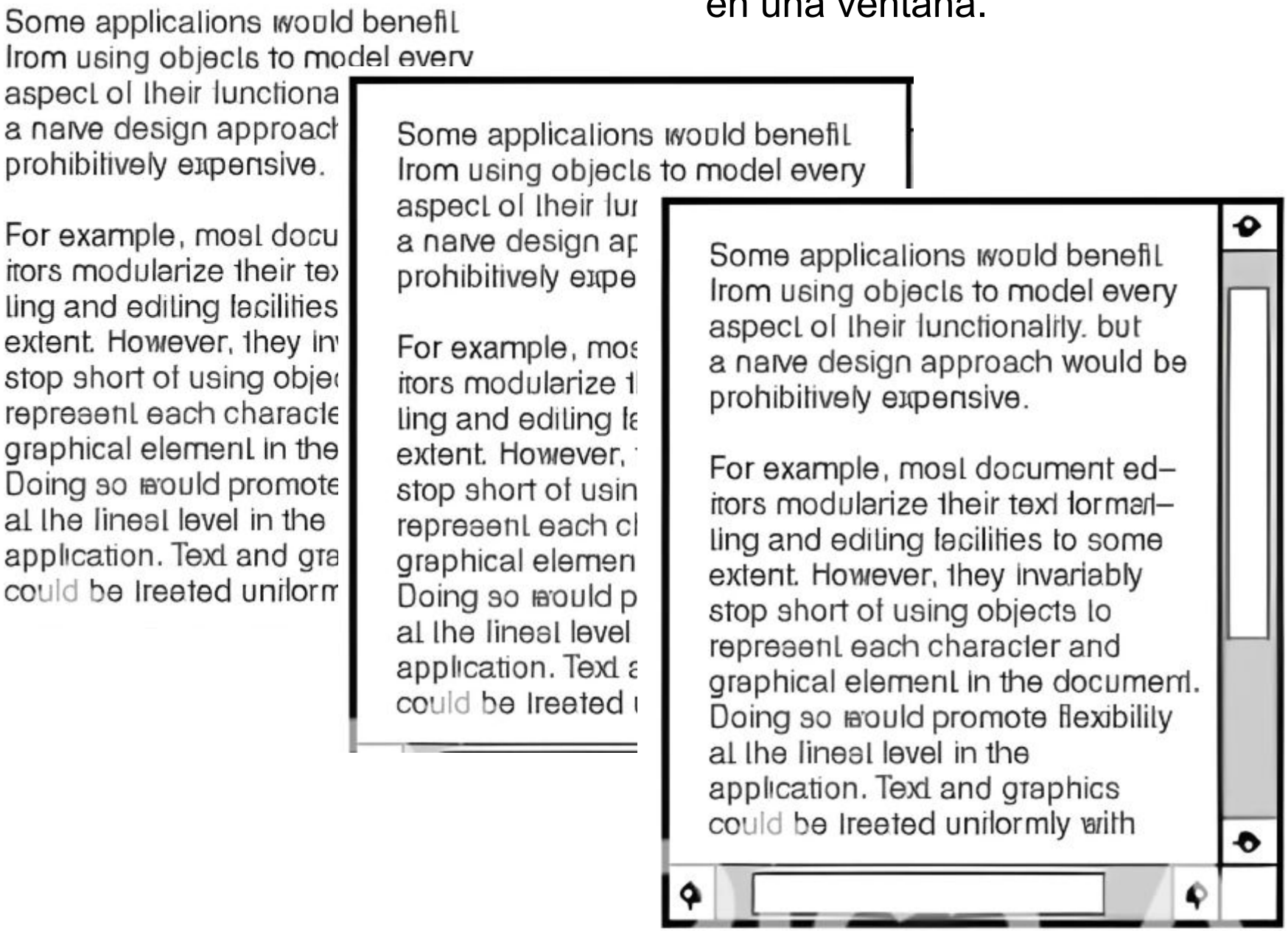
Ejemplo 2: Interfaz gráfica



Text View que muestra texto en una ventana.

Ejemplo 2: Interfaz gráfica

Text View que muestra texto en una ventana.

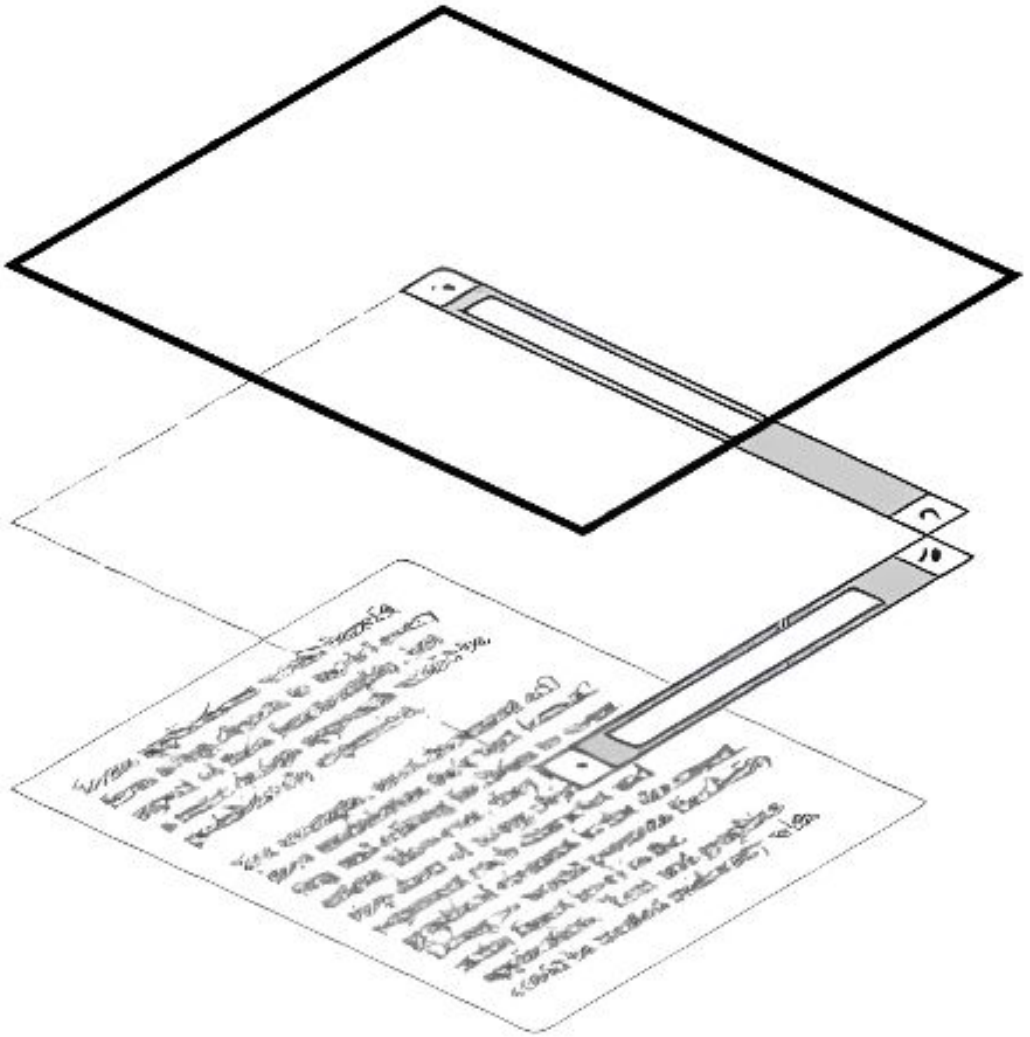


Ejemplo 2: Interfaz gráfica

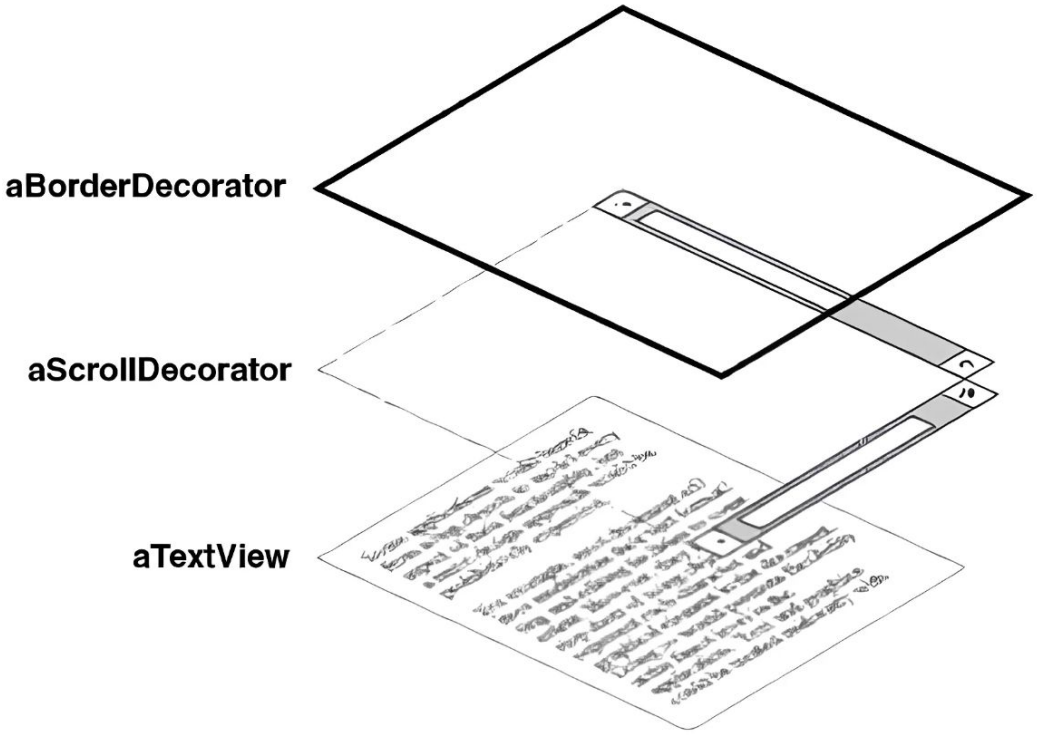
Border

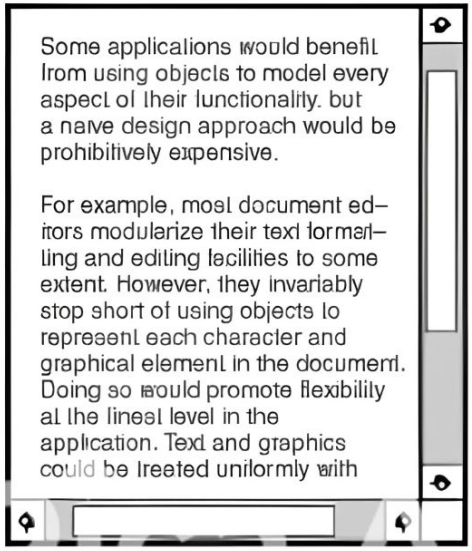
Scroll

aTextView

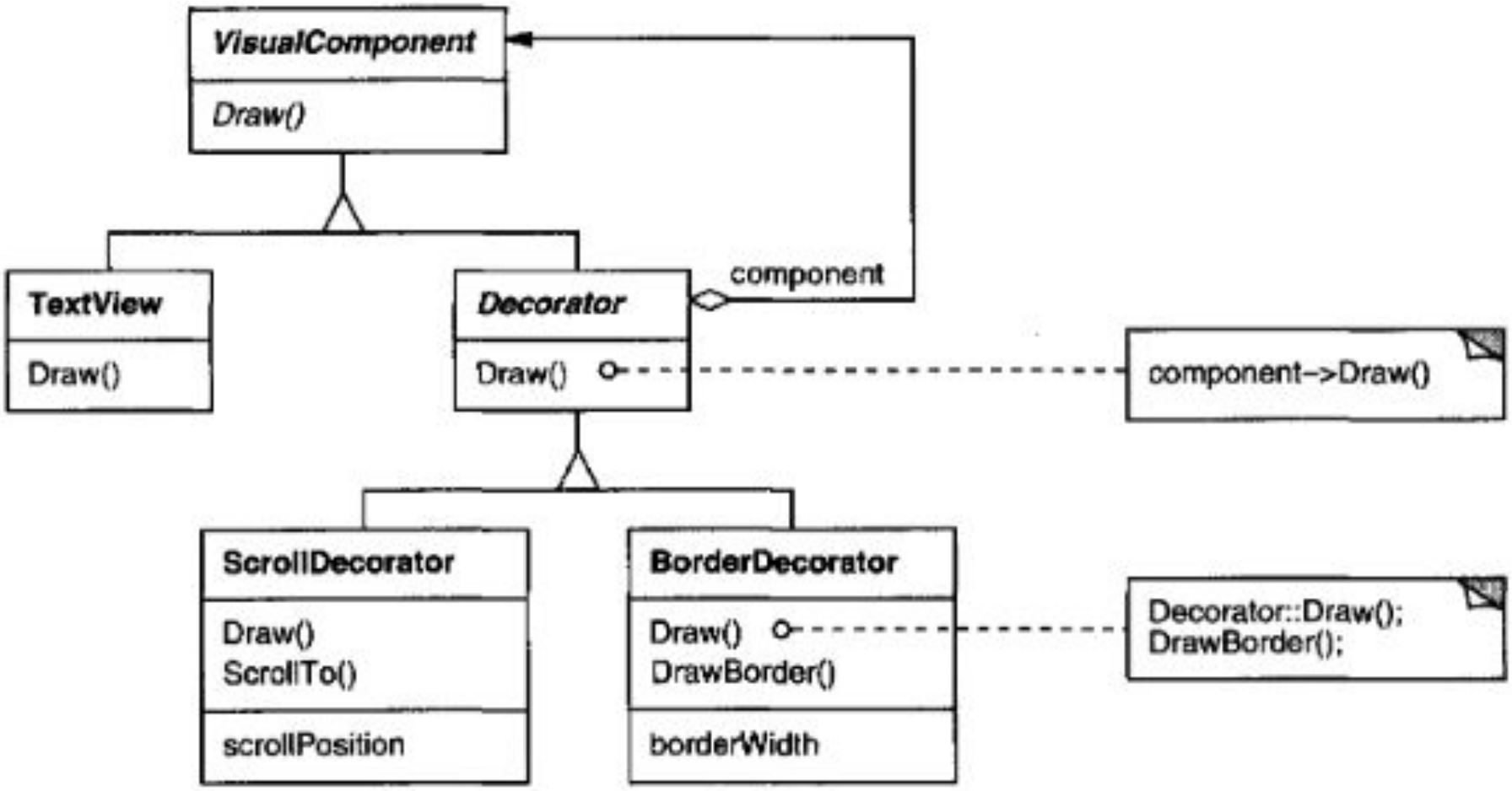


Ejemplo 2: Interfaz gráfica





Ejemplo 2: Interfaz gráfica



Ejemplo 2 → Fuerzas del problema = Ejemplo 1

- Tengo un problema recurrente:
  - 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

Patrón Decorator

- **Intent**
- **Applicability**
- **Estructura**
- **Consecuencias**
- **Implementación**
- **Relación con otros patrones**

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

- **Problema:** Cuando queremos agregar comportamiento adicional a ciertos objetos de una clase, una opción es usar herencia.

  Sin embargo, presenta limitaciones cuando necesitamos que ese comportamiento se pueda agregar o quitar dinámicamente en tiempo de ejecución.

  En esos casos, la herencia no es adecuada, ya que implica una decisión estática. Esta rigidez hace que la herencia no sea flexible para escenarios donde los objetos necesitan "mutar de clase" o modificar su comportamiento de forma dinámica.

Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

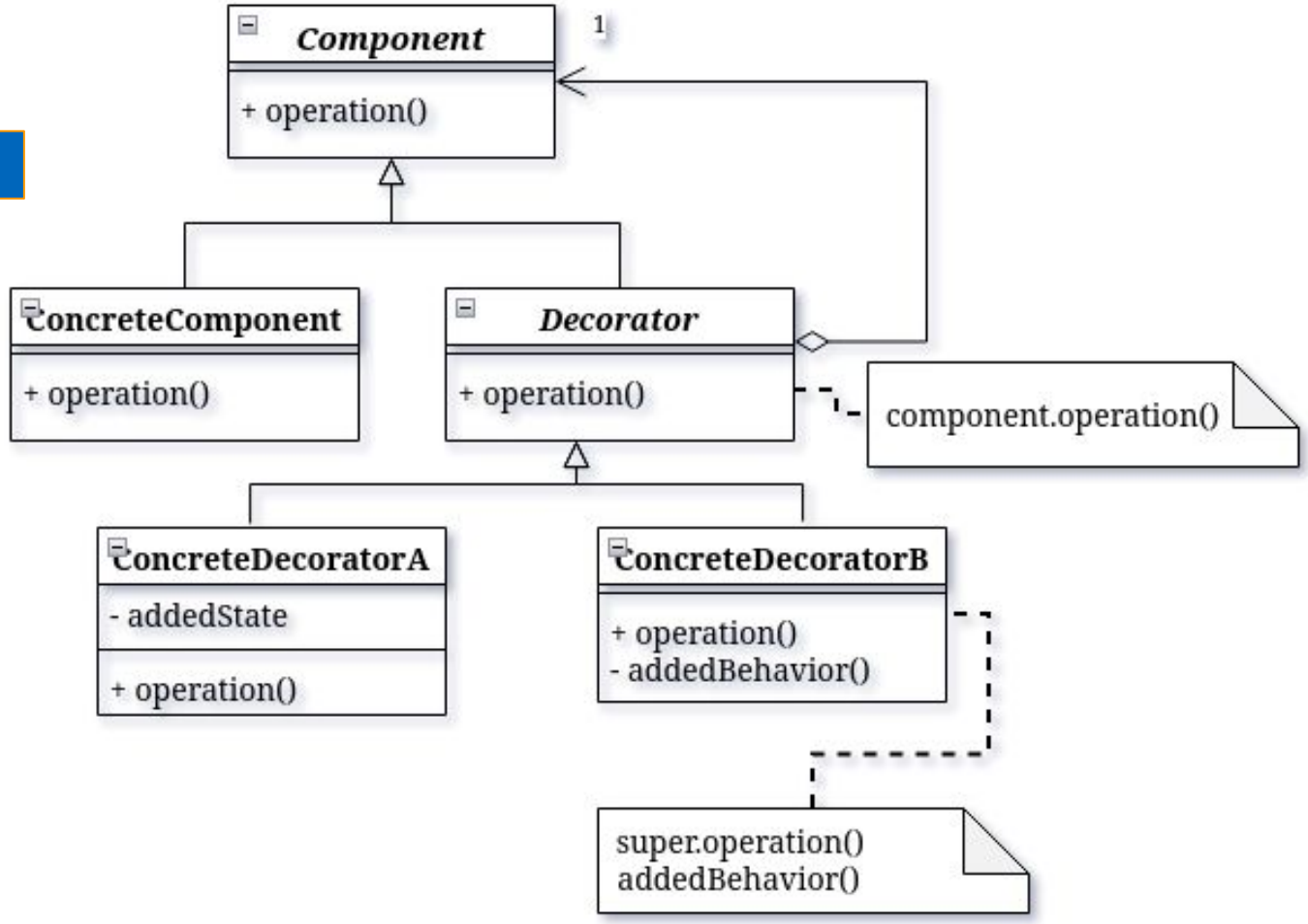
Usar Decorator para:

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

Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

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



Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

- **Puntos a favor**:
  - Permite mayor flexibilidad que la herencia.
  - Permite agregar funcionalidad incrementalmente.

- **Puntos en contra**
  - Mayor cantidad de objetos, complejo para depurar

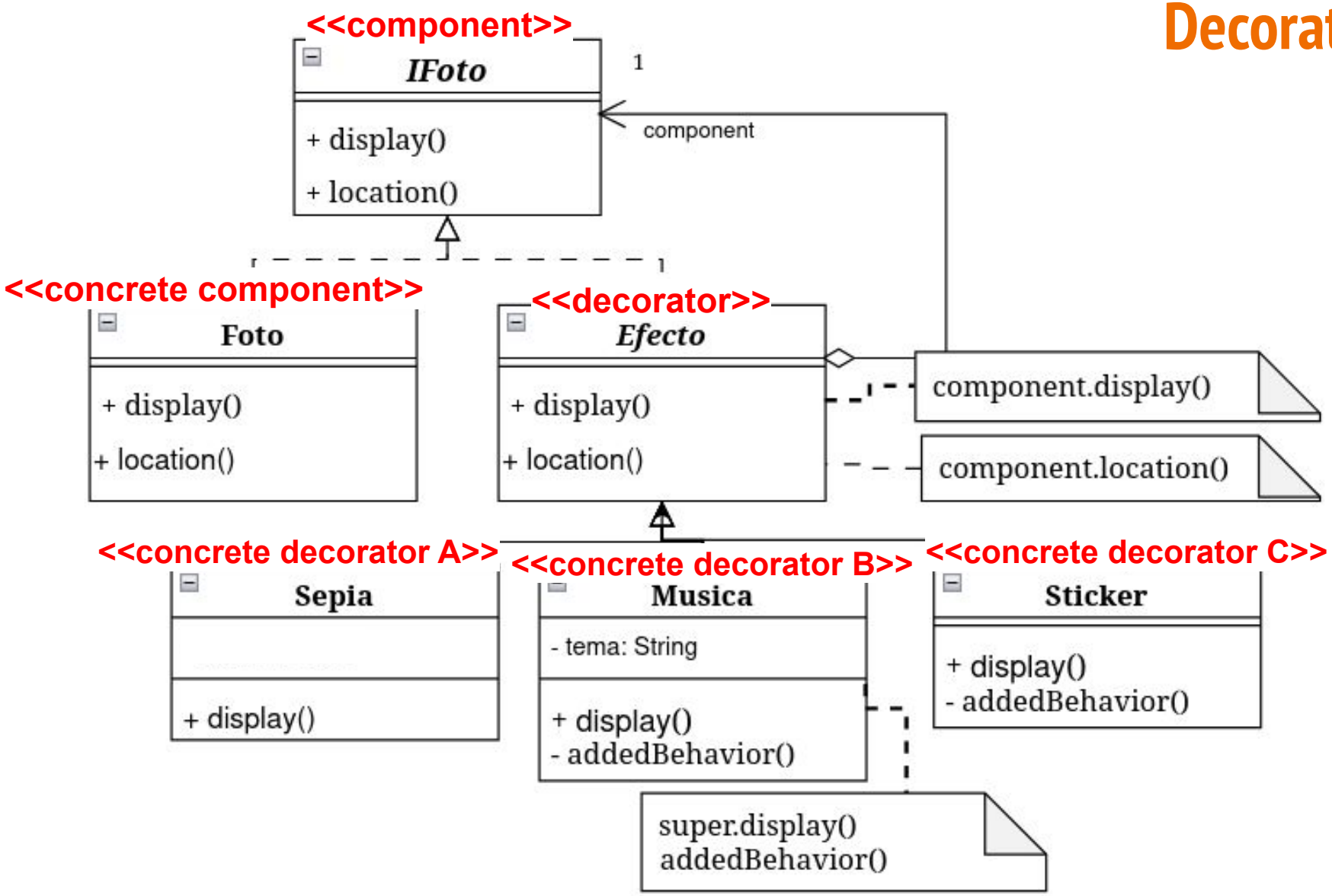
Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

- **Implementación:**
  - Misma interface entre componente y decorador. Tanto la imagen base como los decoradores implementan **la misma interfaz**
  - No hay necesidad de la clase Decorator abstracta, si se tiene un solo decorador.
  - Cambiar el “skin” vs cambiar sus “guts”
    - Decorator puede verse como una “piel” que *modifica el comportamiento externo*, no la estructura interna (los “guts”).
    - Es decir: no cambiamos su estructura interna

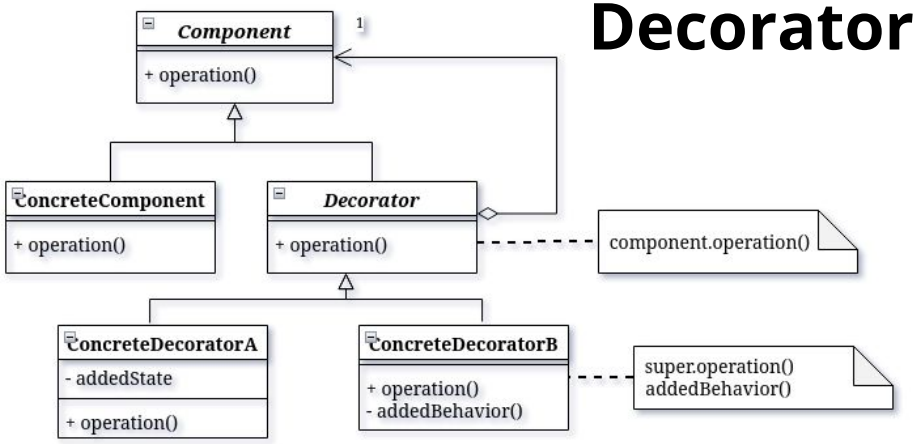
Ejemplo 1: Edición de fotos en Instagram

Decorator

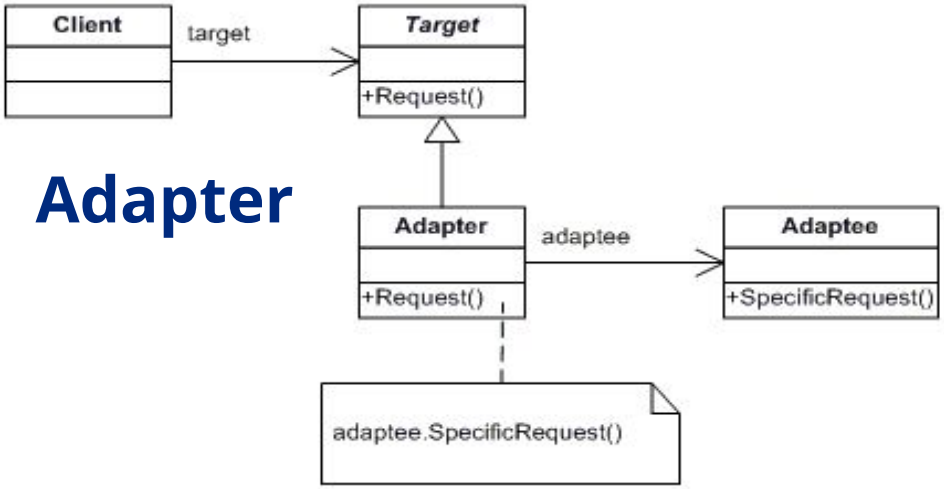


Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones



Decorator

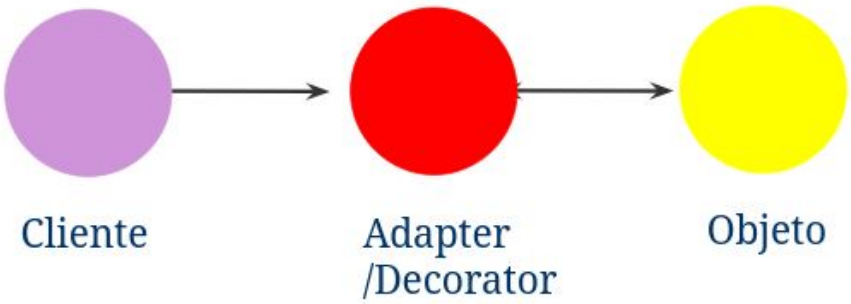


Adapter

Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Decorator vs Adapter



Ambos patrones “decoran” el objeto para cambiarlo

Decorator

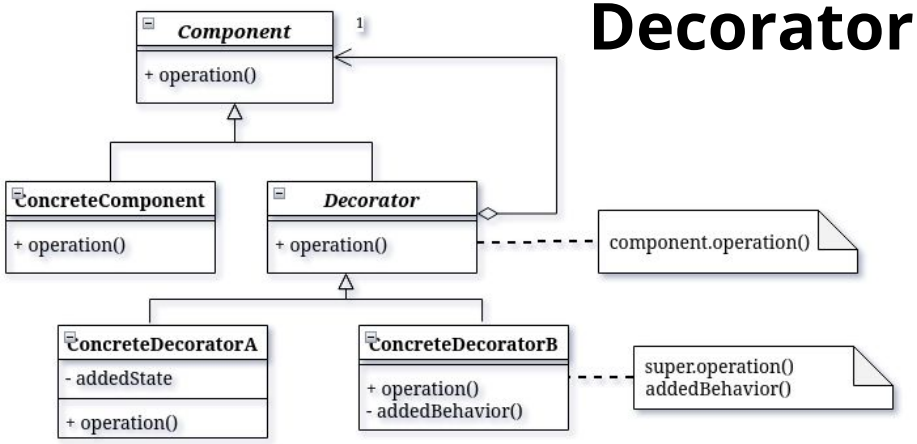
- Decorator *preserva* la interface del objeto para el cliente.
- Decorators pueden y suelen anidarse.

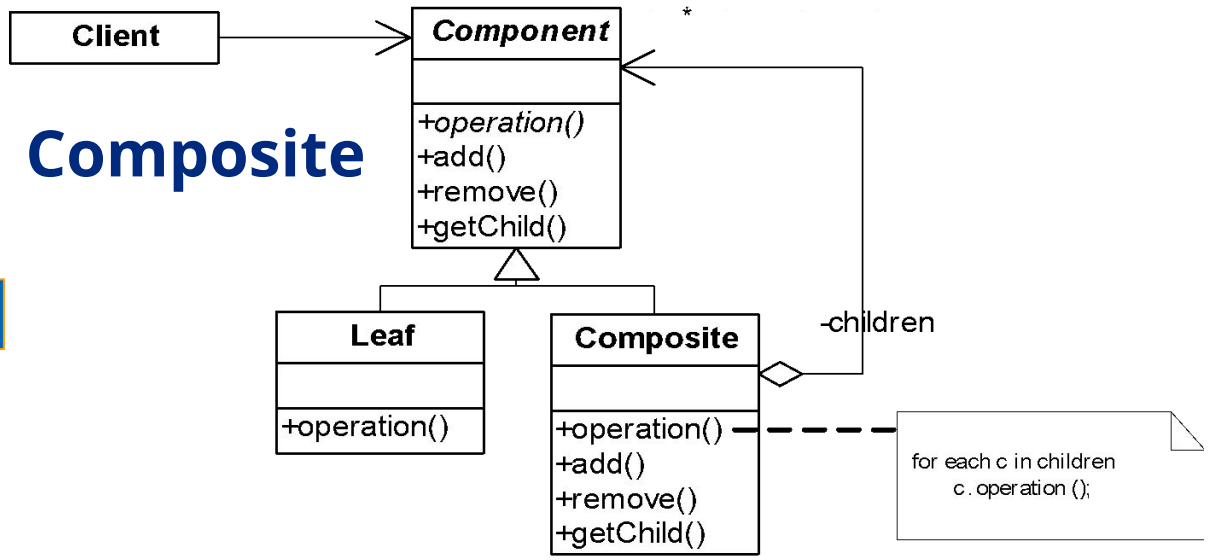
Adapter

- Adapter *convierte* la interface del objeto para el cliente.
- Adapters no se anidan.

Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones





Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Decorator

- Mantiene una estructura con sólo un siguiente
- El propósito es agregar funcionalidad dinámicamente

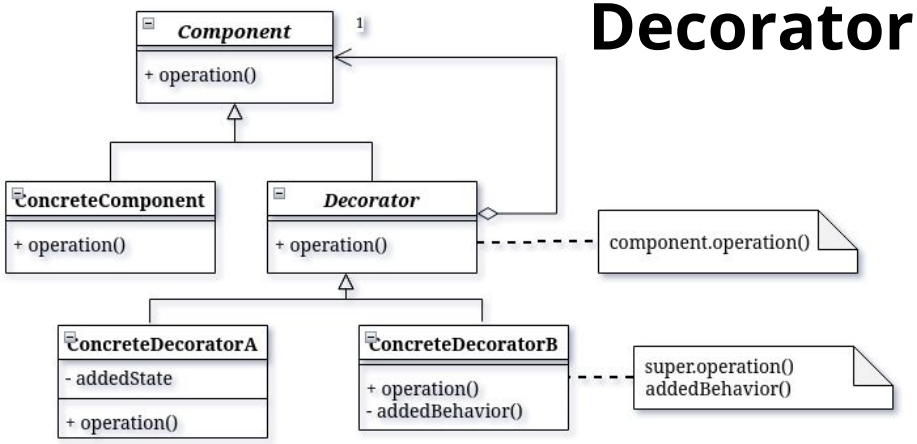
Composite

- Mantiene una estructura de tipo árbol, un composite usualmente se compone de varias partes
- El propósito es componer objetos y tratarlos de manera uniforme.

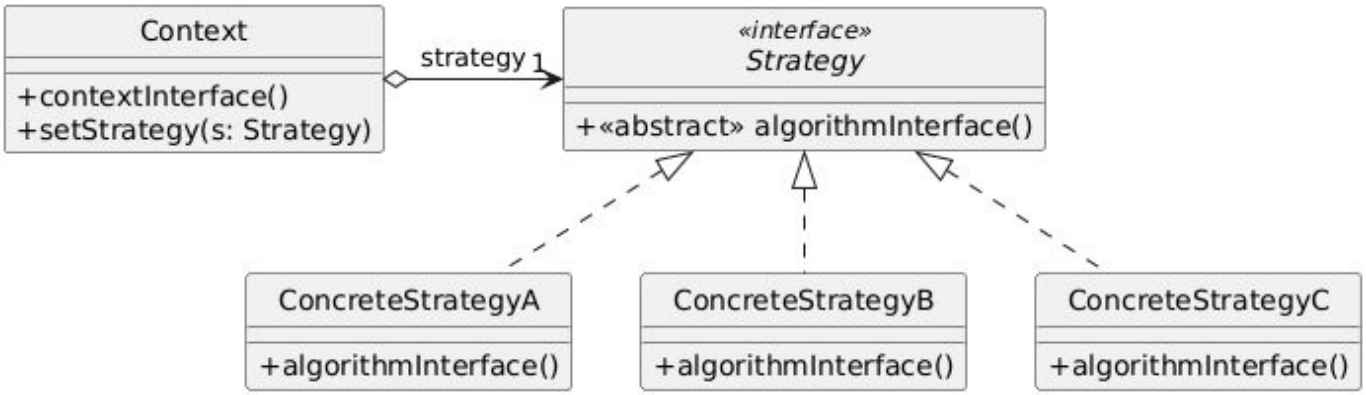
Patrón Decorator

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Decorator



Strategy



Patrón Decorator

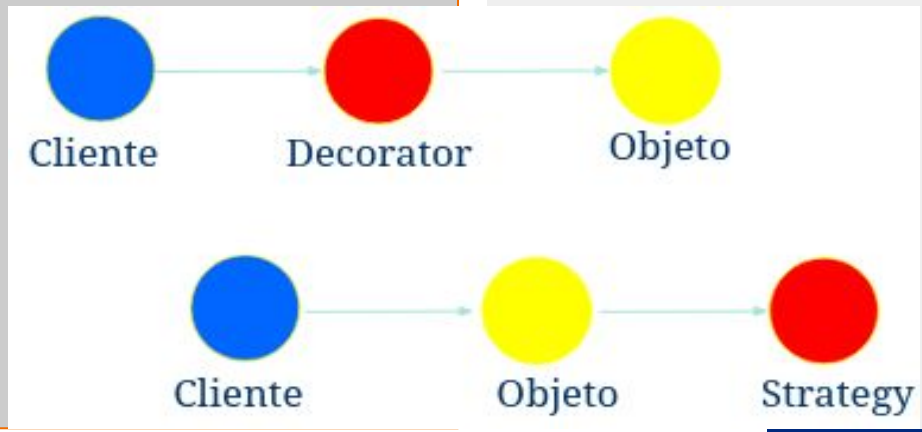
- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Decorator vs Strategy

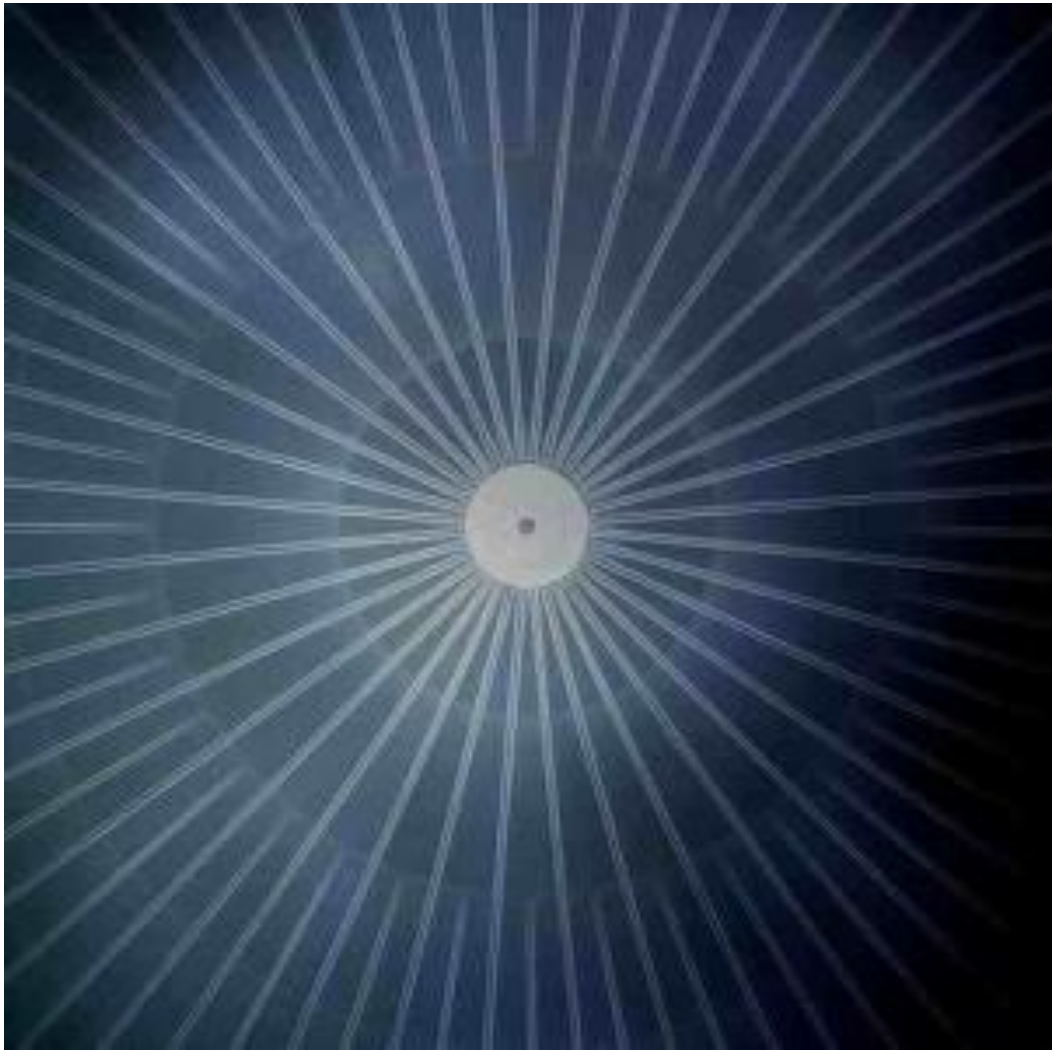
- En qué se parecen?
  - Propósito: permitir que un objeto cambie su funcionalidad dinámicamente (agregando o cambiando el algoritmo que utiliza)

- El Decorator cambia el algoritmo por fuera del objeto

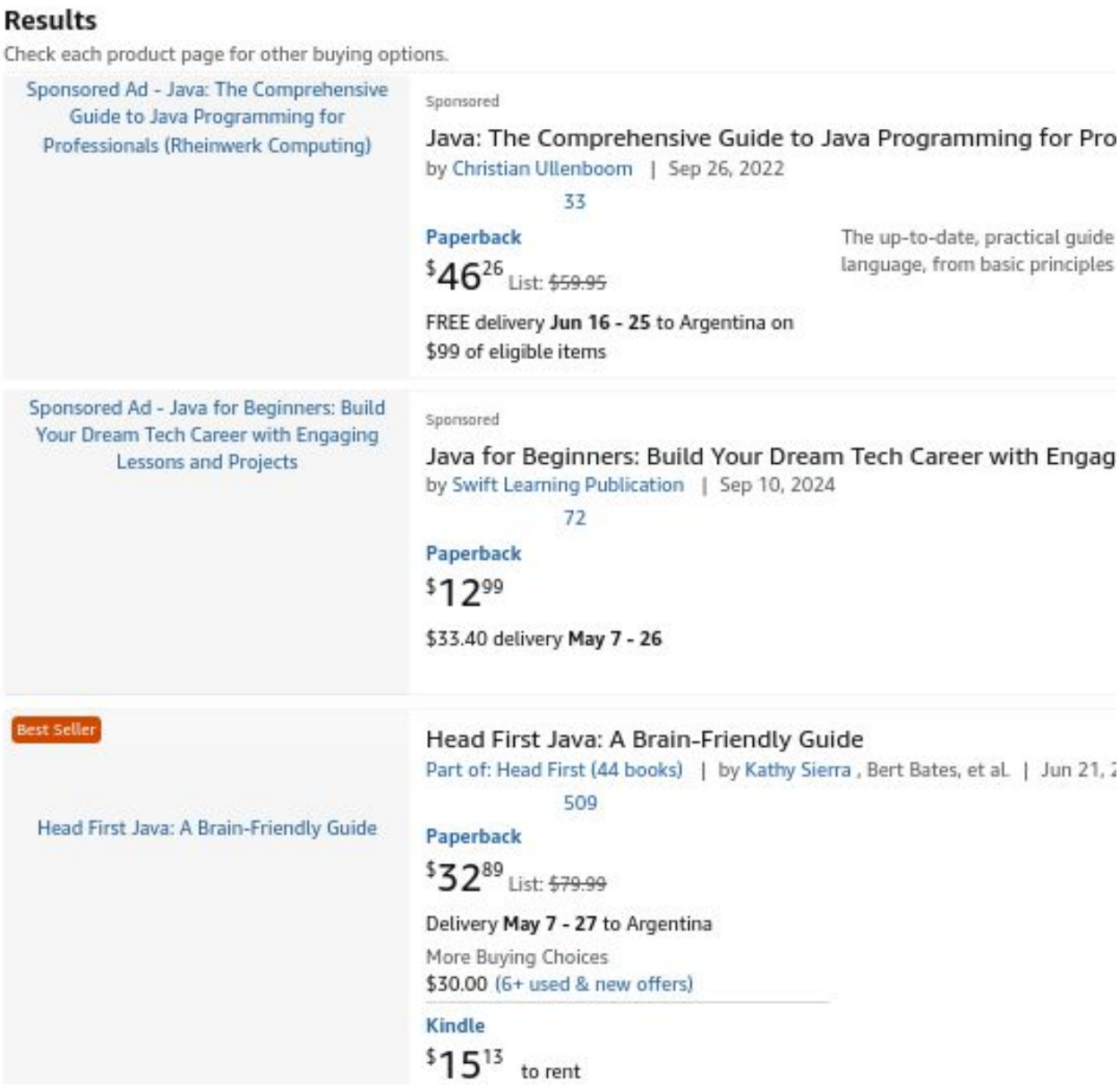
- El Strategy cambia el algoritmo por dentro del objeto,



2do nuevo patrón



Ejemplo 1 - Carga bajo demanda



Ejemplo 1 - Carga bajo demanda

- En muchos casos una página web puede tener muchas imágenes, siendo estas pesadas y lentas de cargar
- No queremos que la apertura del sitio sea lenta.
- En algunos casos las imágenes ni siquiera serán vistas.
- Queremos evitar el costo de leer la imagen hasta tanto sea necesario mostrarla

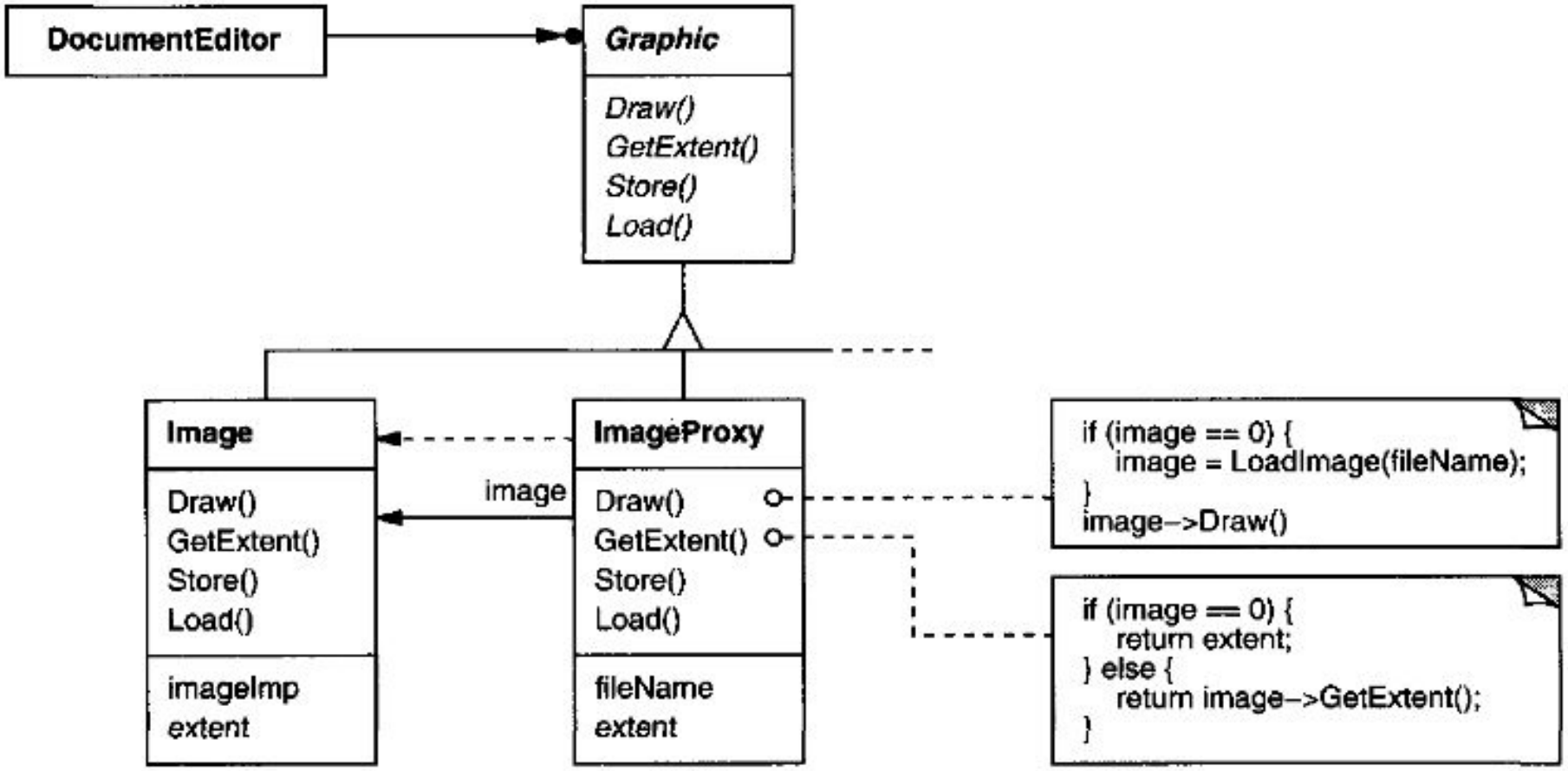
- ¿qué colocamos en el documento en lugar de la imagen?
  - necesitamos un « representante » de la imagen, de manera de darle al cliente un objeto que se vea y actúe como el cliente espera

Ejemplo 1 - Carga bajo demanda: Solución

- La idea es crear una imagen “falsa”, un impostor que
  - Debe responder a los mensajes de la imagen verdadera (mantiene el protocolo).
  - Sabe responder a algunos mensajes (tamaño de la imagen)
  - Cuando sea necesario mostrarla en pantalla, debe ir a buscar la imagen original al servidor, leerla y mostrarla.

Ejemplo 1 - Carga bajo demanda: Solución

- Cargar las imágenes bajo demanda, utilizando un objeto *que* se comporte como una imagen normal y que sea el responsable de cargar la imagen bajo demanda



Patrón Proxy

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Propósito: proporcionar un intermediario de un objeto para controlar su acceso.

- Una de las razones para controlar el acceso a un objeto es **posponer el costo completo de su creación e inicialización** hasta que realmente necesitemos usarlo.

Patrón Proxy

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Aplicabilidad: cuando se necesita una referencia más flexible hacia un objeto

Patrón Proxy

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

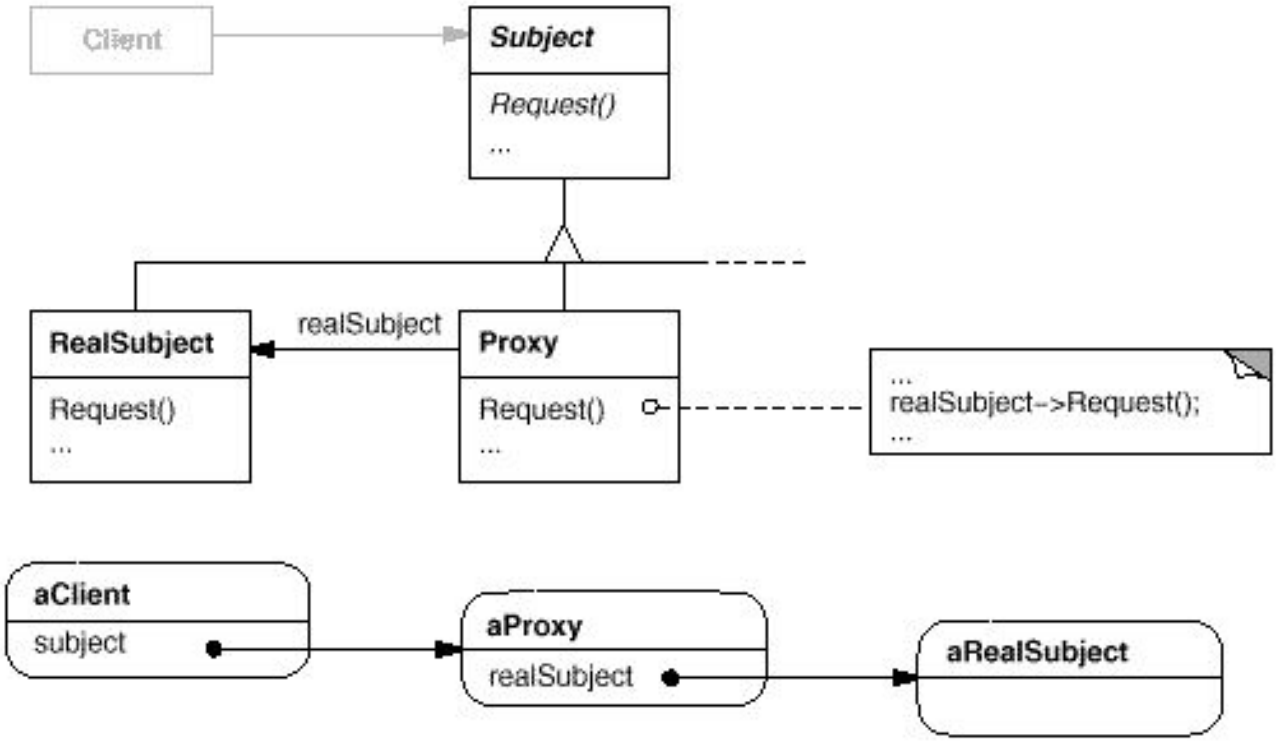
Aplicaciones del proxy:

- **Virtual proxy:** demorar la construcción de un objeto hasta que sea realmente necesario, cuando sea poco eficiente acceder al objeto real.
- **Protection proxy:** Restringir el acceso a un objeto por seguridad.
- **Remote proxy:** representar un objeto remoto en el espacio de memoria local. Es la forma de implementar objetos distribuidos. Estos proxies se ocupan de la comunicación con el objeto remoto, y de serializar/deserializar los mensajes y resultados.

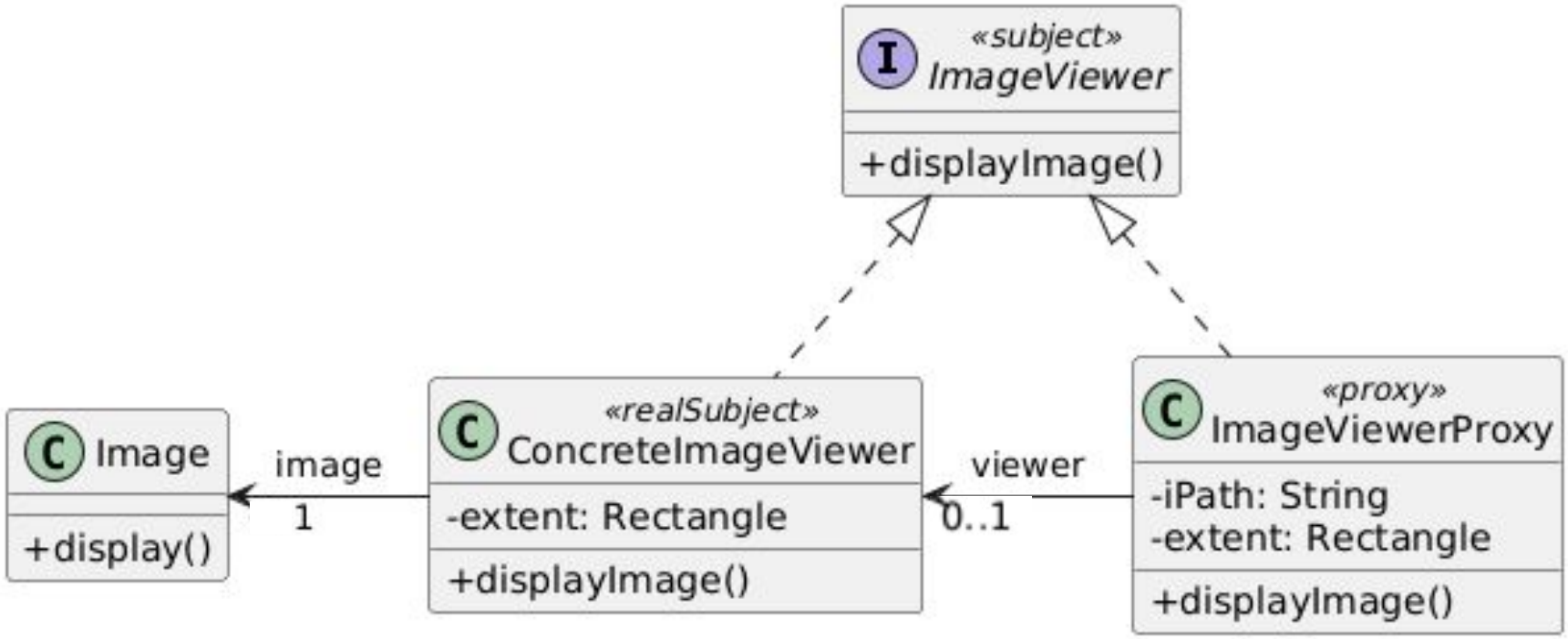
Patrón Proxy

- **Intent**
- **Applicability**
- **Estructura**
- **Consecuencias**
- **Implementación**
- **Relación con otros patrones**

- Colocar un objeto intermedio que respete el protocolo del objeto que está reemplazando.
- Algunos mensajes se delegarán en el objeto original. En otros casos puede que el proxy colabore con el objeto original o que reemplace su comportamiento.



Patrón Proxy - Proxy virtual

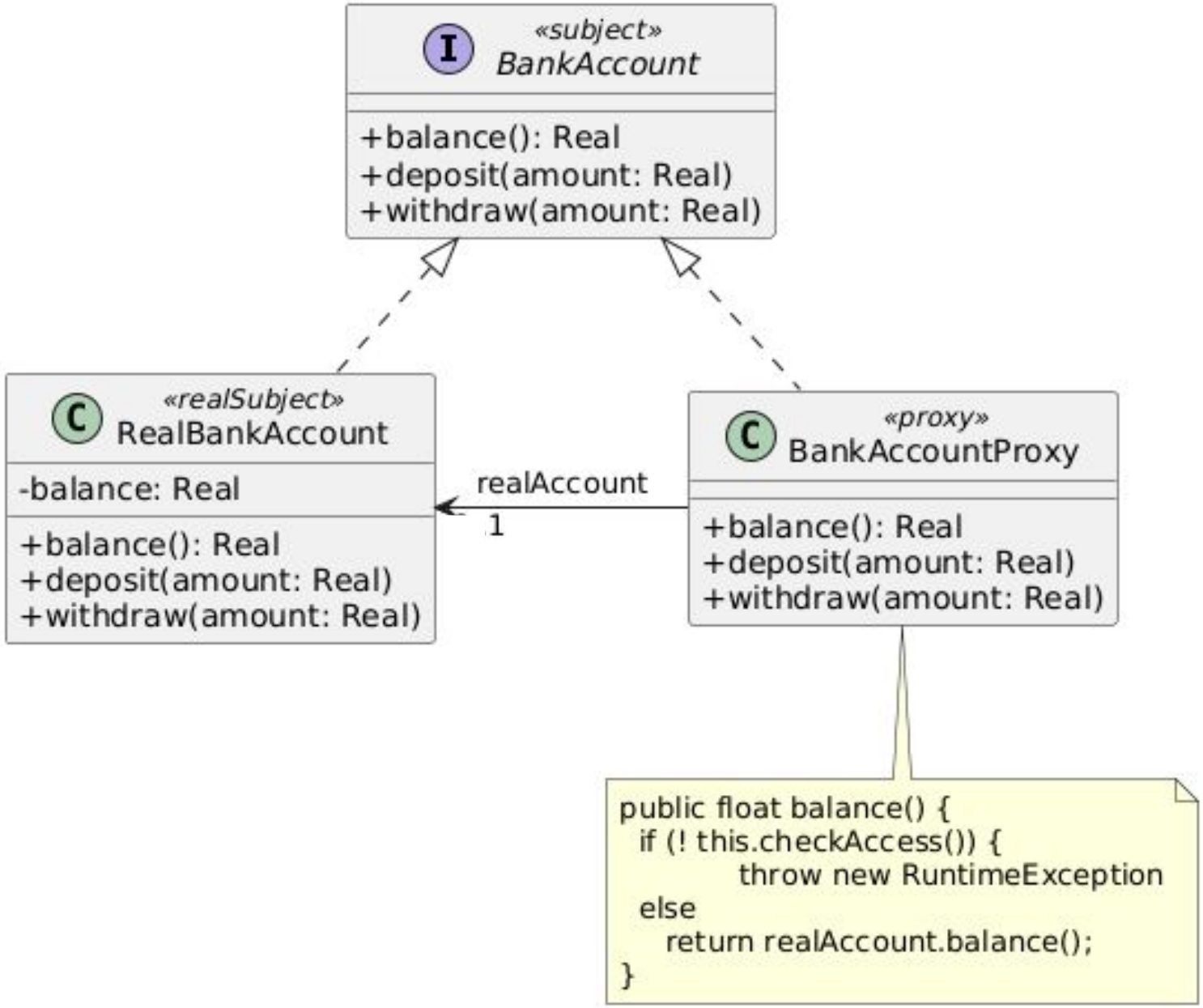


Patrón Proxy - Proxy virtual

```
ublic class ImageViewerProxy implements
mageViewer {
   private String iPath;
   private Rectangle extent;
   private ConcreteImageViewer viewer;

 public ImageViewerProxy(String path, Rectangle
ec) {
      iPath = path;
      extent = rec;

 public void displayImage() {
  if (viewer == null) {
 viewer = new ConcreteImageViewer(iPath, extent);
  }
  viewer.displayImage(); }}
```

```
public class ConcreteImageViewer
implements ImageViewer {
   private Image image;
   private Rectangle extent;


  public ConcreteImageViewer
        (String path, Rectangle rec) {
    // Costly operation
    image = Image.load(path);
   extent = rec;
}
   public void displayImage() {
        // Costly operation
       image.display();

   }}
```

Patrón Proxy - Proxy de protección



Patrón Proxy - Proxy de protección

```
public class BankAccountProxy  implements
BankAccount {

    private BankAccount realAccount;
    public BankAccountProxy(BankAccount
anAccount) {
     realAccount = anAccount; }

    public float balance() {
     if (! this.checkAccess()) {
         throw new RuntimeException("accesso
denegado"); }
     return realAccount.balance();
    }
    public void deposit(float amount) {
     if (this.checkAccess)
         realAccount.deposit(amount); }
    public void withdraw(float amount) {
     if (this.checkAccess)
         realAccount.withdraw(amount);}

    private boolean checkAccess() ...
```

```
public class RealBankAccount
implements BankAccount {

    private float balance;

public float balance() {
    return balance;
}

public void deposit(float
amount) {
    balance += amount;
}

public void withdraw(float
amount) {
    balance -= amount;
}
}
```

Proxy de acceso remoto

- Para acceder a objetos que se encuentran en otro espacio de memoria, en una arquitectura distribuida
- El proxy empaqueta el request, lo envía a través de la red al objeto real, espera la respuesta, desempaqueta la respuesta y retorna el resultado
- En este contexto el proxy suele utilizarse con otro objeto que se encarga de encontrar la ubicación del objeto real. Este objeto se denomina **Broker**, del patrón de su mismo nombre

Patrón Proxy

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Puntos a favor

- **Indirección en el acceso al objeto**: El patrón Proxy introduce un nivel de indirección que permite controlar el acceso al objeto real, lo que ofrece flexibilidad en su uso.

Puntos en contra

- Complejidad adicional

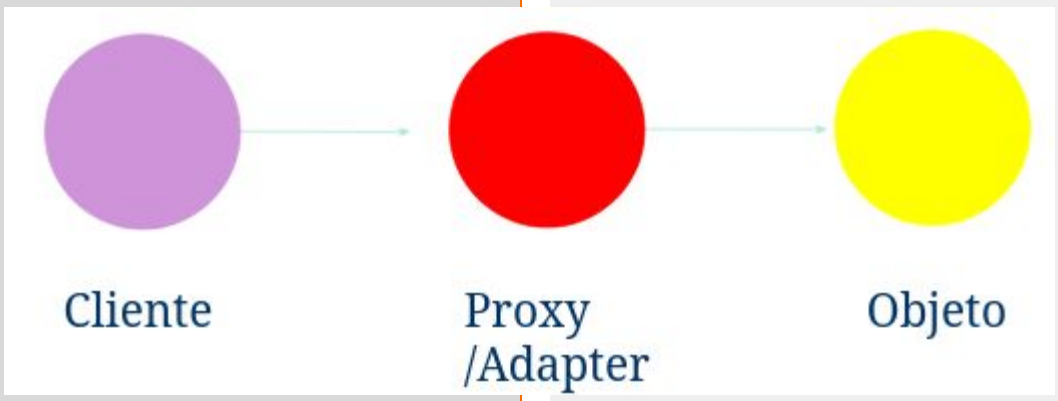
Ejemplos de Proxy

- https://java-design-patterns.com/patterns/proxy/
- https://stackabuse.com/the-proxy-design-pattern-in-java/
- ...

Patrón Proxy

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Proxy vs **Adapter**



proporciona la misma interfaz que su sujeto

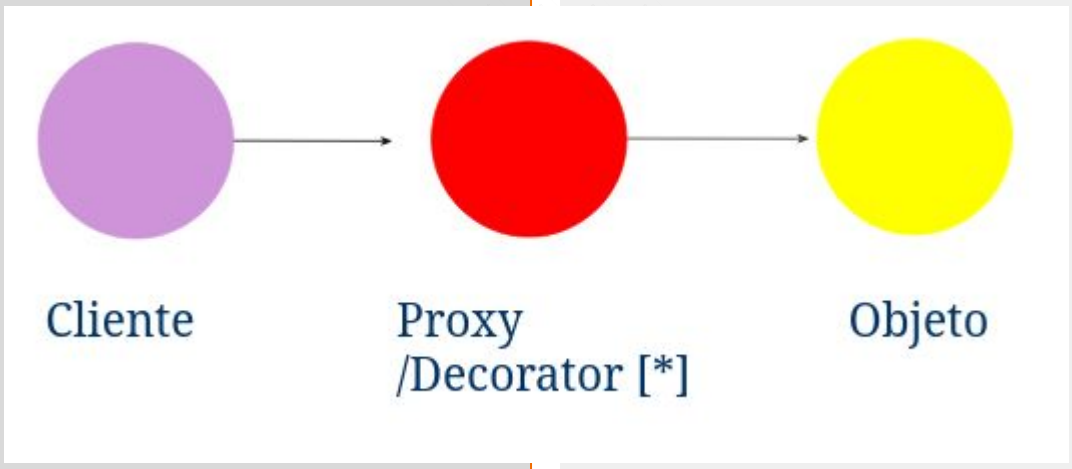
proporciona una interfaz diferente al objeto que adapta.

Patrón Proxy

- Intent
- Applicability
- Estructura
- Consecuencias
- Implementación
- Relación con otros patrones

Proxy

Decorator



controla el acceso a un objeto.

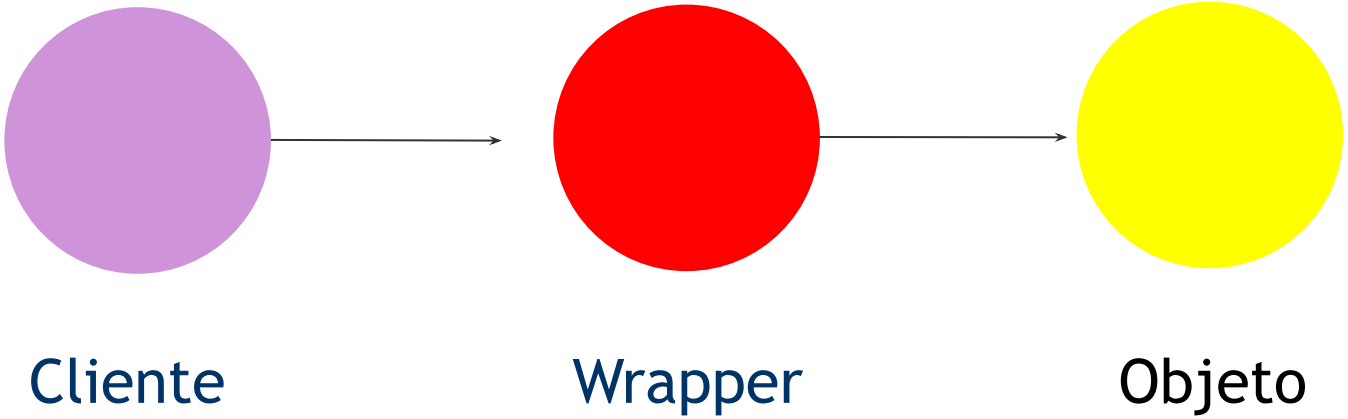
agrega una o más responsabilidades a un objeto,

Los propósitos son diferentes

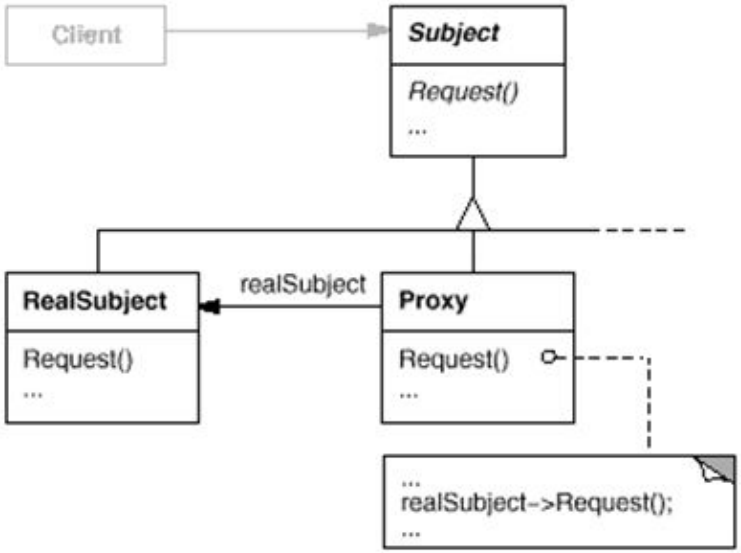
Adapter, Decorator, Proxy

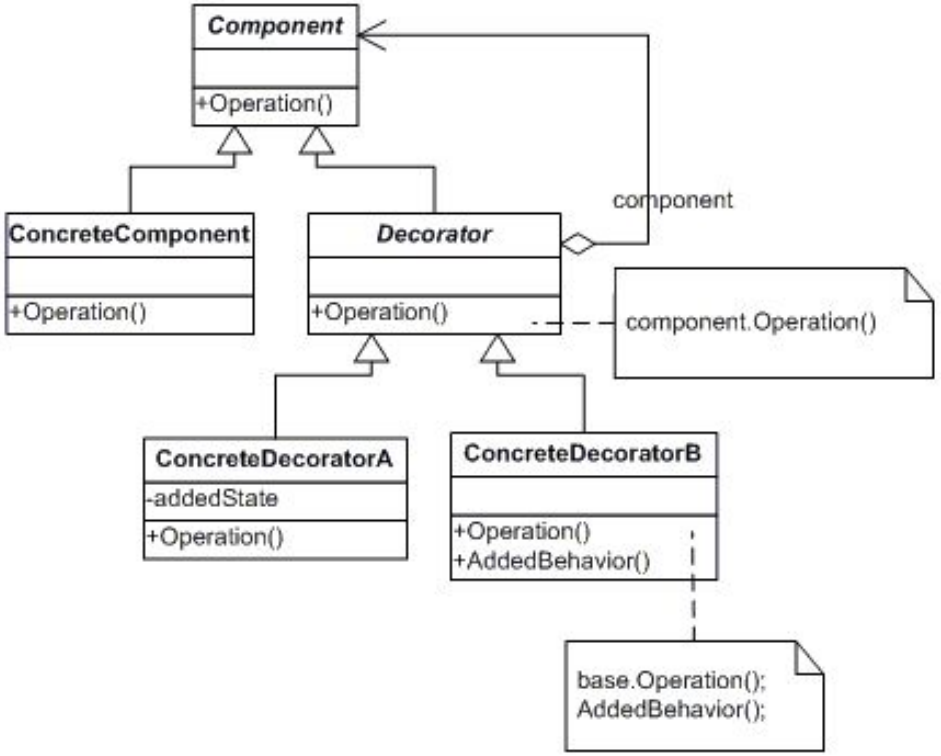
- Todos patrones estructurales
- Todos con diagramas de objetos similares
- Distinto propósito

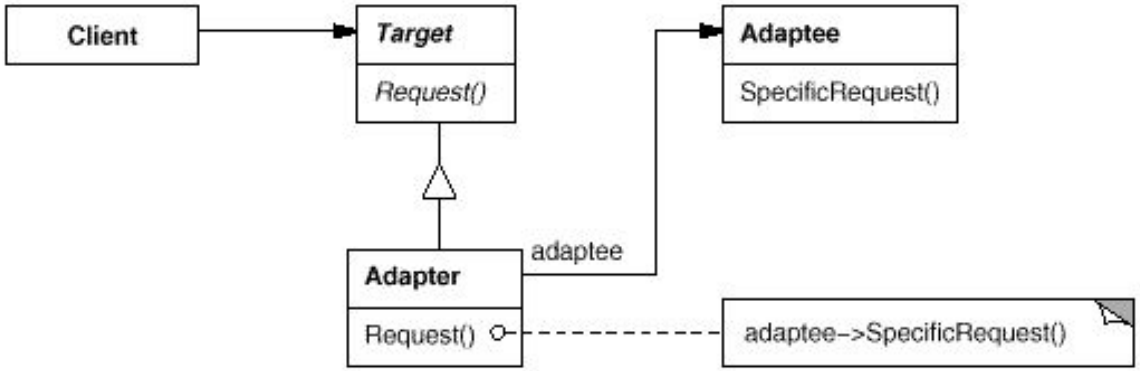
- A todos se los llama también “wrappers”



Proxy vs. Decorator vs. Adapter







¿ Preguntas ?