Orientación a Objetos II 2025

Explicación de práctica
Semana del 28 de abril

Ejercicios de la semana

Semana 21/4 - Hasta ejercicio 15: Armado de PCs

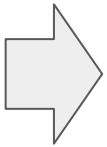
- Ejercicio 16 - Filtros de Imágenes
- Ejercicio 17 - Acceso a la base de datos
- Ejercicio 18 - File Manager
- Ejercicio 19 - Estación meteorológica
- Ejercicio 20 - Construcción de personajes de juegos de rol

Ejercicio 16 - Filtros de Imágenes

En este proyecto encontrará la librería **ImageFilter** que implementa algunos filtros básicos sobre imágenes PNG tales como: **Rainbow**, **Repeater**, **RGBShifter,** entre otros.

La librería incluye una herramienta (**PNGFilterLauncher**) que permite aplicar secuencia de filtros sobre una archivo de entrada y generar un archivo (.png) de salida.

PNG original

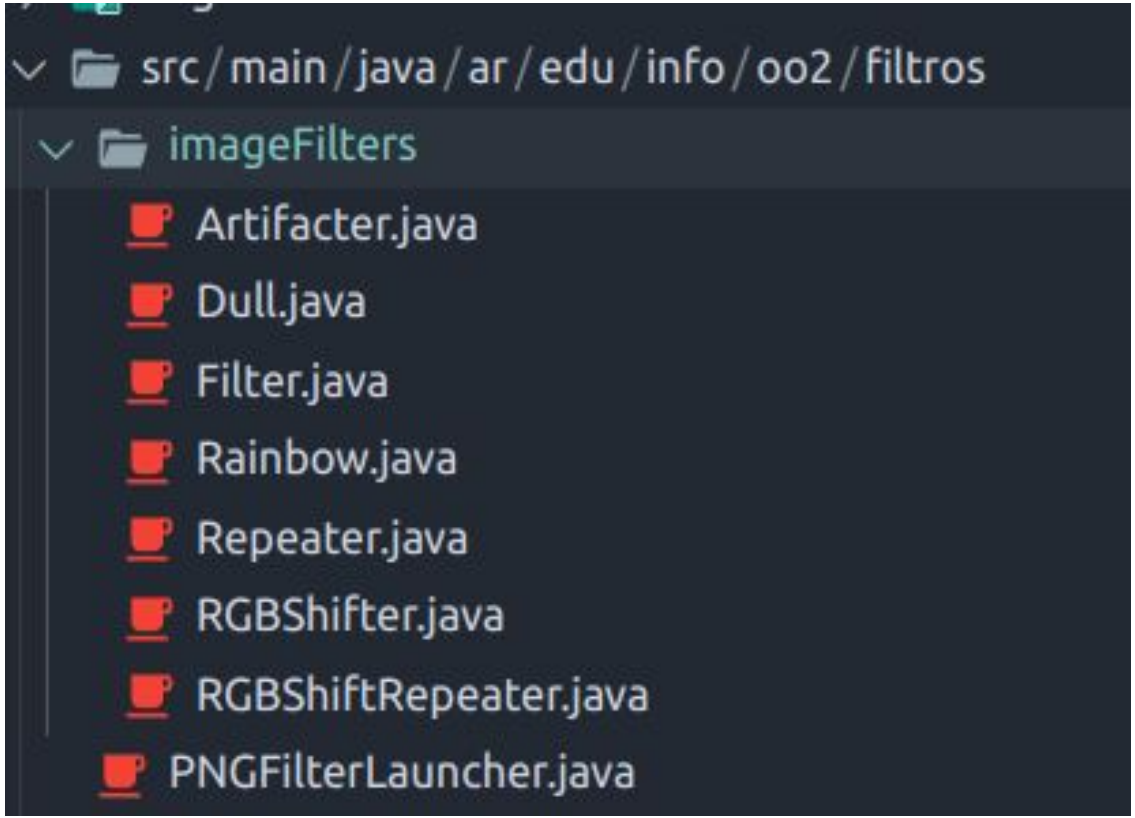


Rainbow

Repeater

RGBShifter

Ejercicio 16 - Estructura del proyecto



Dentro del directorio **ImageFilters** están los diferentes filtros disponibles que se pueden usar para transformar las imágenes.

PNGFilterLauncher es una clase con un método *main* que recibe los **argumentos** de ejecución. Esto permite indicar la imágen sobre la que se va a trabajar, qué filtros aplicar y en qué orden

Ejercicio 16 - Estructura del proyecto

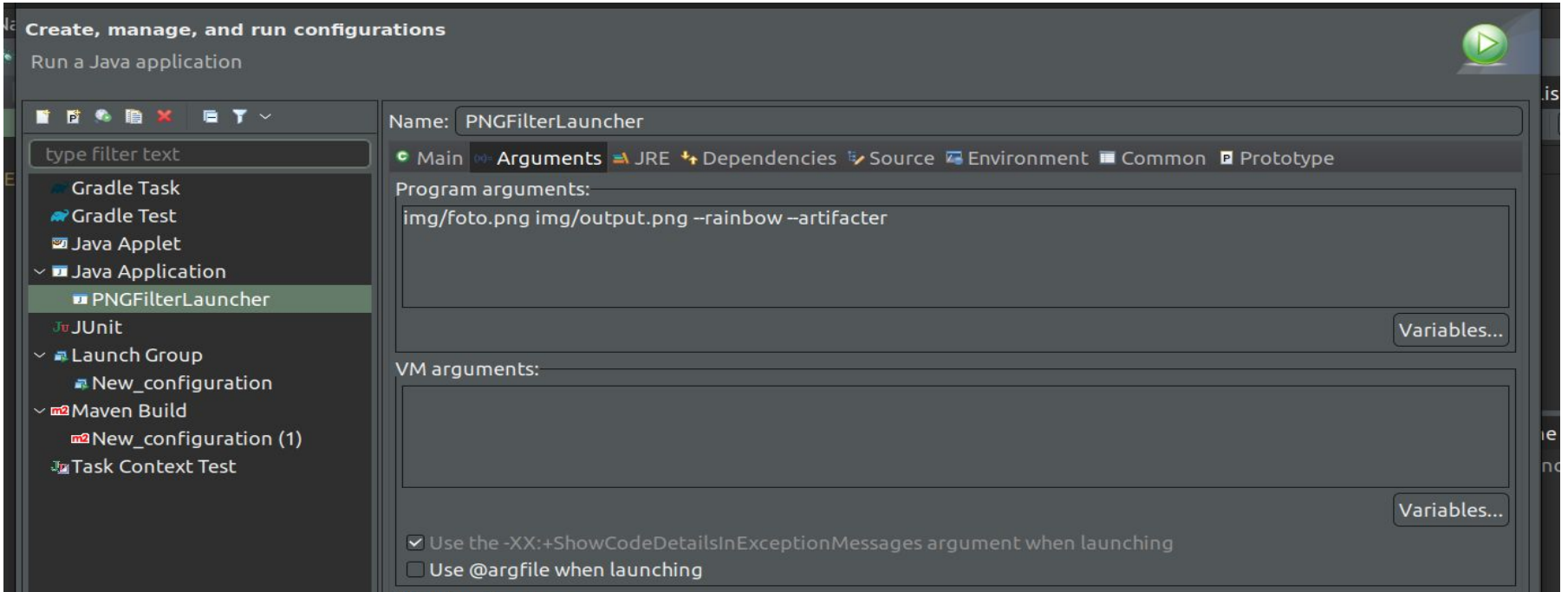
```java
public class PNGFilterLauncher {
  private static Map<String, Filter> filters = new HashMap<String, Filter>();
  private static ArrayList<Filter> activeFilters = new ArrayList<Filter>();
  private static File inputFile;
  private static File outputFile;

  Run | Debug
  public static void main(String args[]) {
    // Initialize the list of filers
    PNGFilterLauncher.initializeFilters();

    // Process the arguments
    if (args.length < 2) {
      System.out.println(x:"Usage: java Main input-file output-file [--filter]");
      return;
    }
    PNGFilterLauncher.processArgs(args);
```

Dentro del directorio **ImageFilters** están los diferentes filtros disponibles que se pueden usar para transformar las imágenes.

PNGFilterLauncher es una clase con un método *main* que recibe los **argumentos** de ejecución. Esto permite indicar la imágen sobre la que se va a trabajar, qué filtros aplicar y en qué orden

Ejercicio 16 - Parámetros de entrada

Para configurar los argumentos que recibe la aplicación, buscar en el menú principal de Eclipse: **“Run” > “Run configurations”** e indicarlos en **“arguments”**



Ejercicio 16 - Parámetros de entrada

- En **Windows**

  ```
  img\foto.png img\output.png --rainbow --artifacter
  ```

- En **Linux/Mac**:

  ```
  img/foto.png img/output.png --rainbow --artifacter
  ```

- **img/foto.png**: PNG original. Puede ser cualquier imágen PNG, el proyecto trae una de ejemplo.
- **img/output.png**: Ruta en donde queremos que deje el resultado.
- **Lista de filtros a aplicar**: Explorar el código para ver qué otras opciones existen

Forma de ejecutar alternativa

Ubicado en el directorio del proyecto, y luego de su compilación con Maven, ejecutar en consola:

```
java -cp target/classes ar.edu.info.oo2.filtros.PNGFilterLauncher
img/foto.png img/salida.png --rainbow
```

También se pueden usar maneras propias que provea el IDE

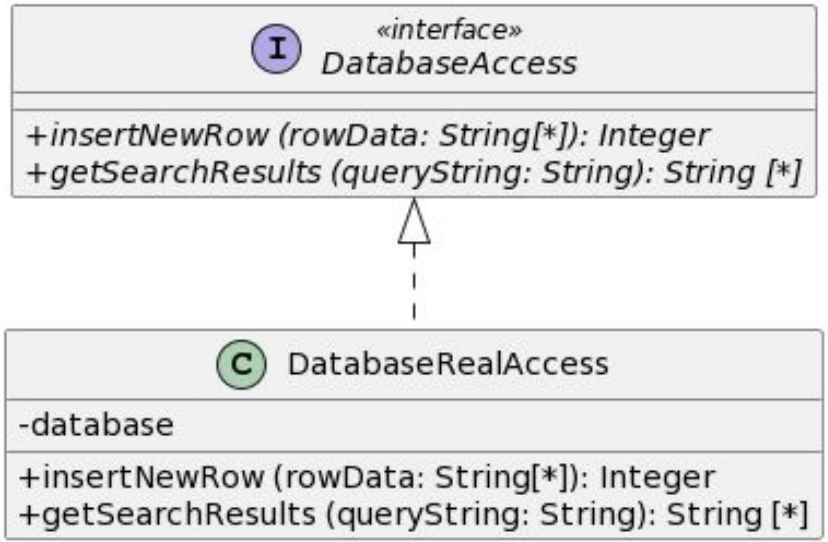
Ejercicio 16 - Tareas

1. Descargue el proyecto y pruebe los diferentes Filtros
2. Analice el código y documente el proyecto con un **Diagrama de Clases**
3. Evalúe cuál de los siguientes patrones mejor describe el diseño de los Filtros: **Template Method**, **Strategy**, **Decorator**. Para realizar la evaluación se sugiere contestar las siguientes preguntas aplicadas a cada uno de los patrones
4. Se requiere crear el filtro **Monochrome** y se presenta un pseudocódigo cómo guía

Ejercicio 17 - Acceso a la base de datos

Queremos acceder a una base de datos que contiene información sobre cómics.

Este acceso está dado por el comportamiento de la clase **DatabaseRealAccess** con el siguiente protocolo y modelado como muestra la siguiente figura.



Ejercicio 17 - Acceso a la base de datos

En este caso, ustedes recibirán una implementación prototípica de la clase **DatabaseRealAccess** que simula el uso de una base datos de la siguiente forma:

```
// Instancia una base de datos que posee dos filas
database = new DatabaseRealAccess();

// Retorna el siguiente arreglo: ['Spiderman' 'Marvel'].
database.getSearchResults("select * from comics where id=1");

// Retorna 3, que es el id que se le asigna
database.insertNewRow(Arrays.asList("Patoruzú", "La flor"));

// Retorna el siguiente arreglo: ['Patoruzú', 'La flor'], ya que lo insertó antes
database.getSearchResults("select * from comics where id=3");
```

Ejercicio 17 - Acceso a la base de datos

En esta oportunidad, usted debe proveer una solución utilizando un patrón que le permita brindar protección al acceso a la base de datos de forma que lo puedan realizar solamente usuarios que se hayan autenticado previamente.

Ejercicio 17 - Comportamiento esperado

```
1  // Una contraseña muy difícil.
2  String password = "p455w0rD"
3
4  // Instancia una base de datos que posee dos filas, y se accede con contraseña
5  database = new YourClass( ... some mysterious parameters ... , password);
6
7  // No retorna nada.
8  database.getSearchResults("select * from comics where id=1");
9
10 // Pero… si previamente realizó el login con la contraseña correcta…
11 database.login(password)
12 // Retorna el siguiente arreglo: ['Spiderman' 'Marvel'].
13 database.getSearchResults("select * from comics where id=1");
```

- Enfocarse en la protección de la base de datos
- La “implementación prototípica” sirve para poder “probar”
- Test provistos por la cátedra

Ejercicio 17 - Acceso a la base de datos

Hay que proveer una solución utilizando un patrón que le permita **brindar protección al acceso a la base de datos** de forma que lo puedan realizar solamente usuarios que se hayan autenticado previamente.

Patrón Proxy

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

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

- ✓ **Virtual proxy**: demorar la construcción de un objeto hasta que sea realmente necesario.
- ✓ **Protection proxy**: Restringir el acceso a un objeto por seguridad.
- ✓ **Remote proxy**: representar un objeto remoto en el espacio de memoria local

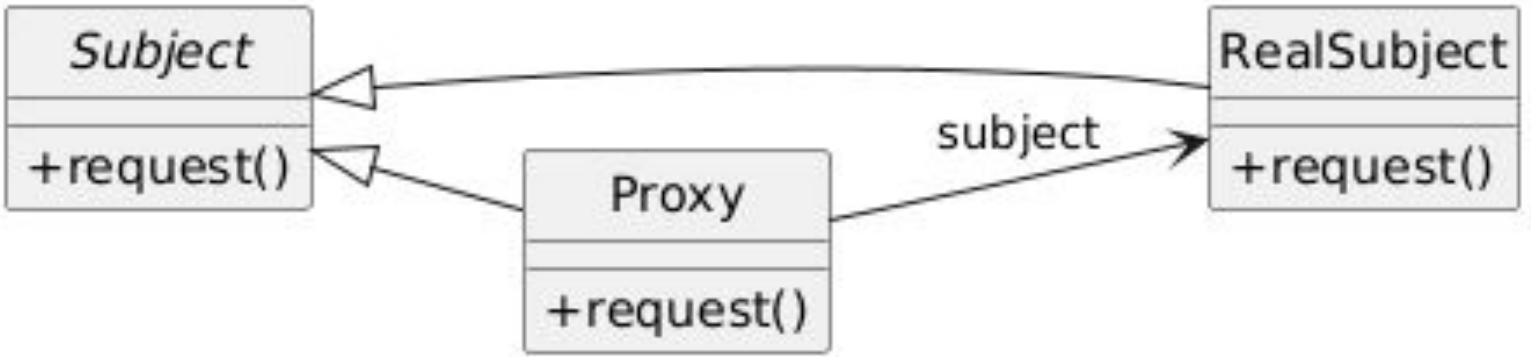
Patrón Proxy - Participantes

- **Subject**: Define la interfaz para el **Real Subject** y el **Proxy**, de modo que el **Proxy** pueda ser utilizado en cualquier lugar donde se espere un **Real Subject**
- **Real Subject:** Define el objeto real que el **Proxy** representa
- **Proxy:**
  - Mantiene una referencia que le permite acceder a un **Real Subject**
  - Controla el acceso y puede ser responsable de su creación o destrucción
  - **Diferentes responsabilidades de acuerdo al tipo de proxy**

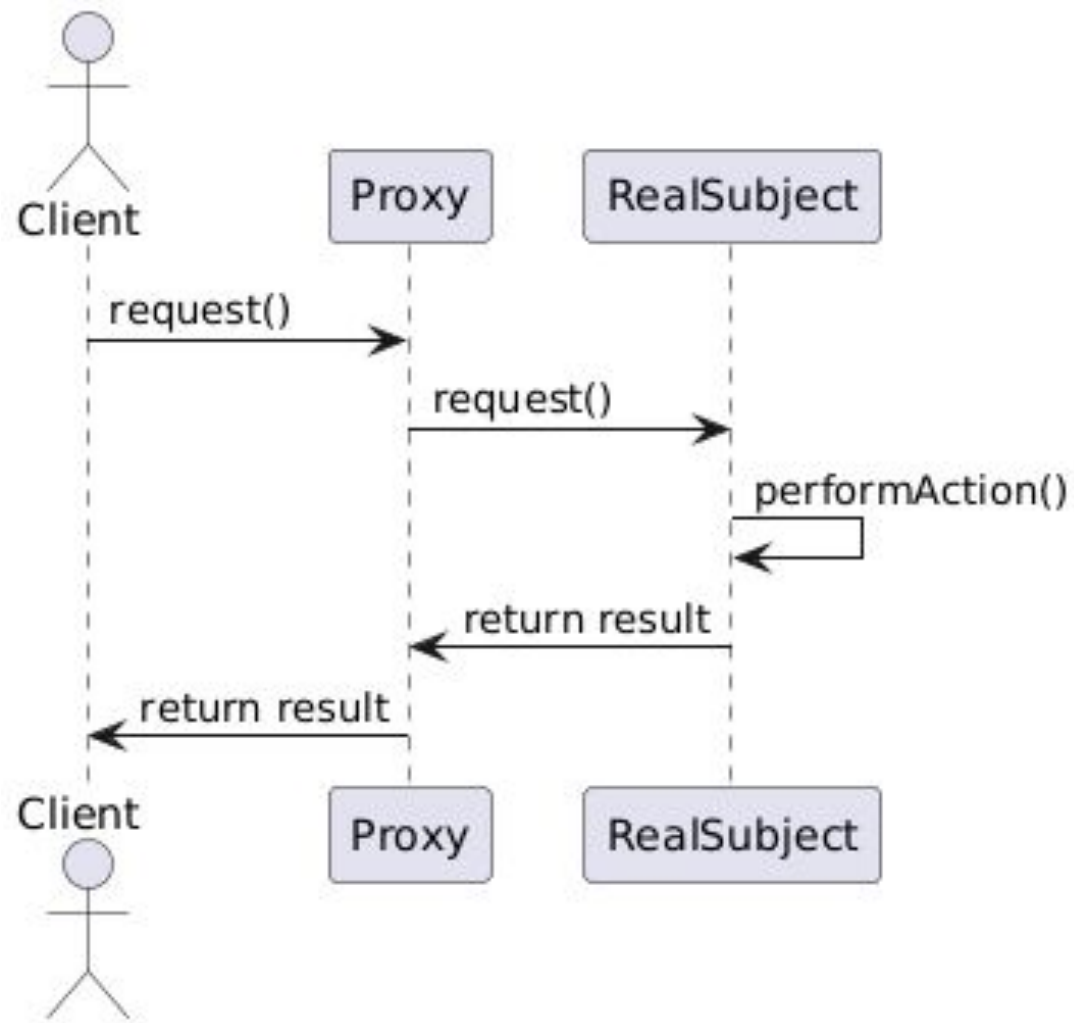
Patrón Proxy - Participantes (2)

- **Proxy:**
  - **Virtual Proxy:** crean objetos costosos bajo demanda (lazy loading)
  - **Protection Proxy:** controlan los derechos/permisos de acceso al objeto real
  - **Remote Proxy:** Envían solicitudes a través de una red. Es decir, es un objeto que no está en el espacio de memoria local. Se ocupan de comunicarse con un objeto distribuidos por la red, y de serializar/deserializar mensajes y respuestas

Patrón Proxy - Estructura



El **`Proxy`** actúa como intermediario entre el cliente y el **`RealSubject`**, enviando las solicitudes al **`RealSubject`** y luego devolviendo los resultados al cliente.

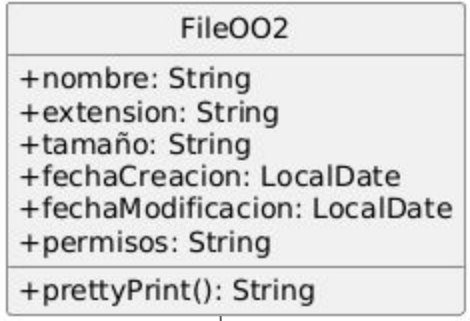


Patrón Proxy - Consecuencias

+ El cliente no debería notar si está usando el **Real Subject** o el **Proxy**
+ Indirección en el acceso al objeto -> Permite controlar el acceso al objeto real, lo que ofrece flexibilidad en su uso

- Introduce complejidad
- Dificulta depuración y trazabilidad si el **Proxy** hace muchas operaciones intermedias

Ejercicio 18 - File Manager



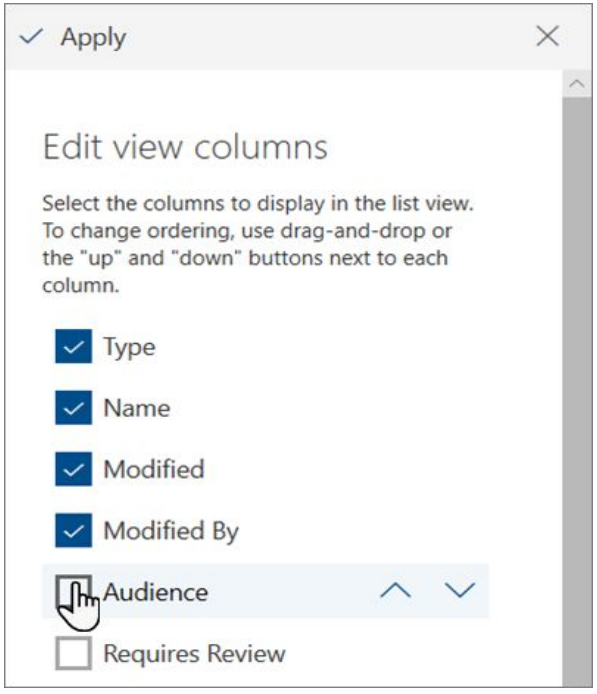
`prettyPrint()` retorna el nombre del archivo

En el File Manager el usuario debe poder elegir cómo se muestra un archivo (instancia de la clase FileOO2), es decir, cuáles de los aspectos mencionados anteriormente se muestran, y en qué orden. Esto quiere decir que un usuario podría querer ver los archivos de muchas maneras. Algunas de ellas son:

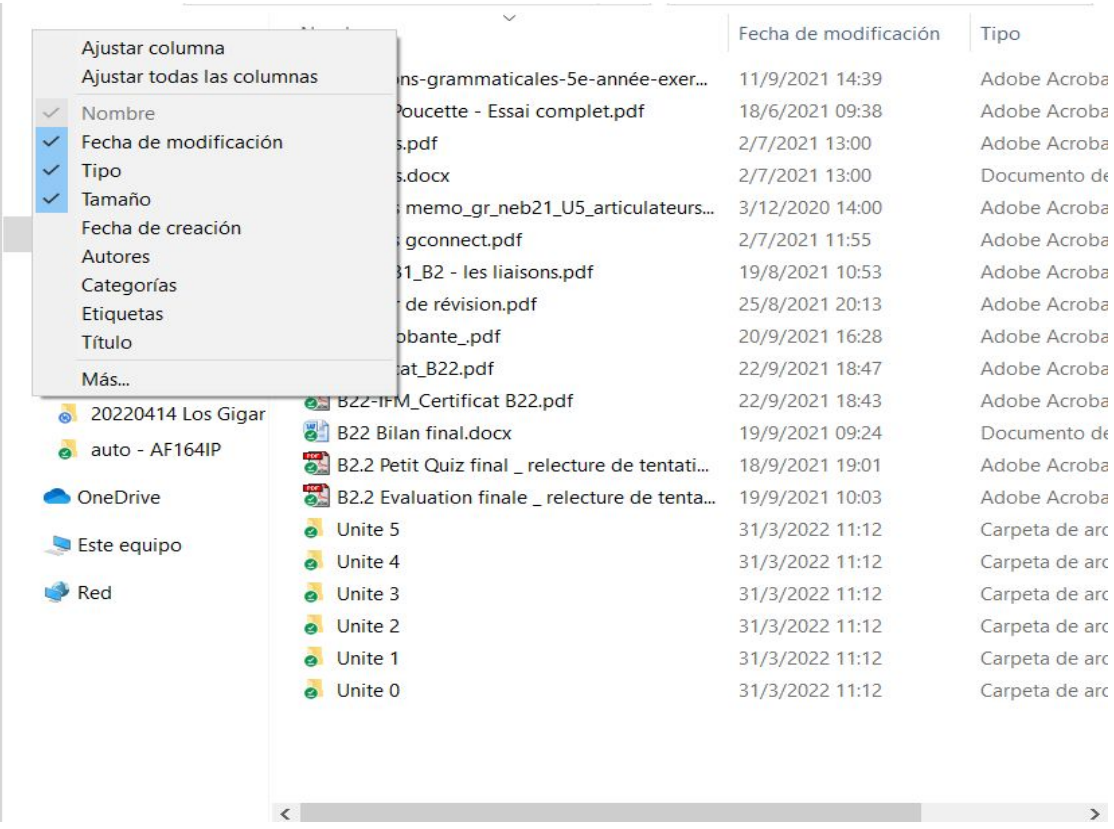
- `nombre - extensión`
- `nombre - fecha de creación - extensión`
- `nombre - tamaño - permisos - extensión`

Son sólo *algunos* ejemplos

Ejercicio 18 - File Manager (2)



Se podrían generar todas las combinaciones posibles.



Ejercicio 18 - File Manager (3)

Es decir, un objeto cliente, por ejemplo, el FileManager, le enviará al FileOO2 el mensaje prettyPrint().

De acuerdo a cómo el usuario lo haya configurado se deberá retornar un String con los aspectos seleccionados por el usuario en el orden especificado por éste.

Considere que un mismo archivo podría verse de formas diferentes desde distintos puntos del sistema, y que el usuario podría cambiar la configuración del sistema (qué y en qué orden quiere ver) en runtime.

Patrón Decorator

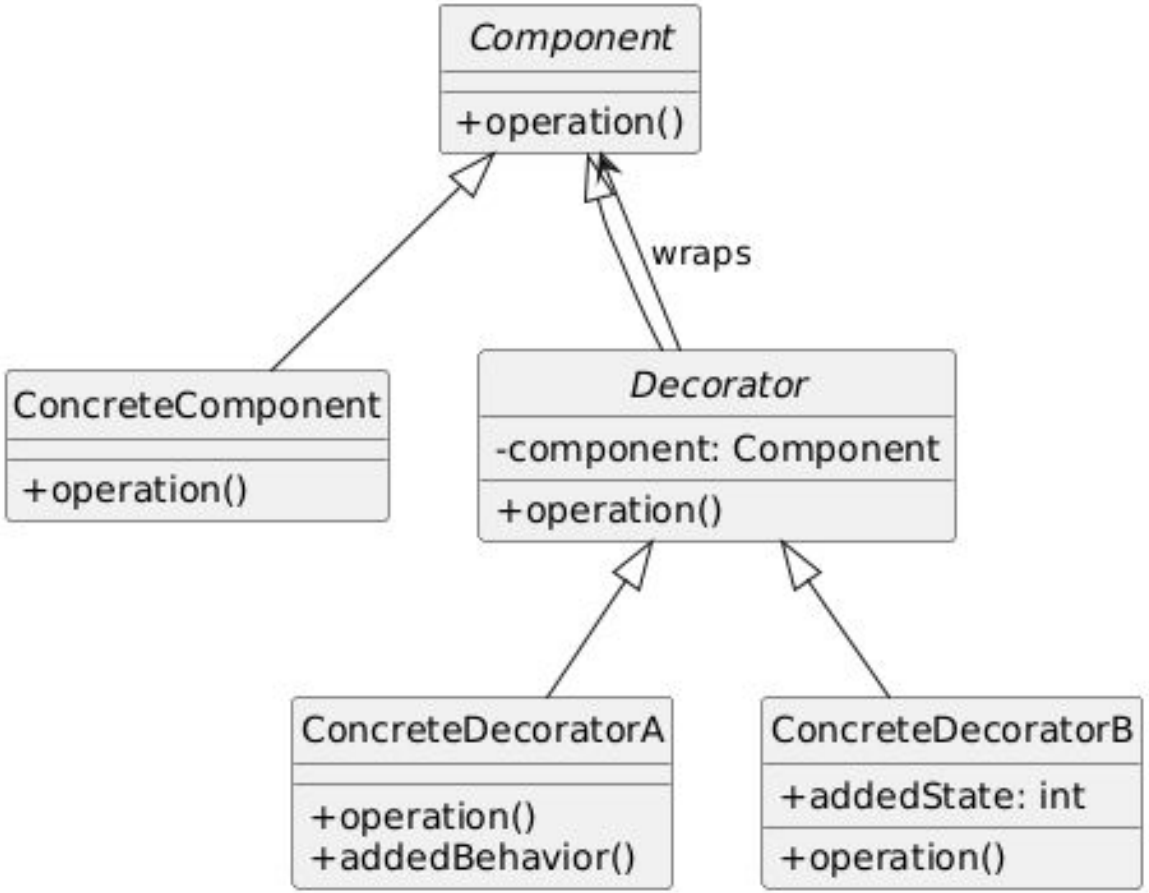
Propósito: Agregar comportamiento a un objeto dinámicamente y en forma transparente.

Aplicabilidad: Use el patrón Decorator cuando:

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

Patrón Decorador - Participantes

- **Component**: Define la interfaz para los objetos a los que se les pueden añadir responsabilidades de manera dinámica
- **Concrete Component:** Define un objeto al cual se le pueden agregar responsabilidades adicionales.
- **Decorator:** Mantiene una referencia a un objeto **Component** y define una interfaz que se ajusta a la del **Component**.
- **Concrete Decorator:** agrega responsabilidades al **Component**

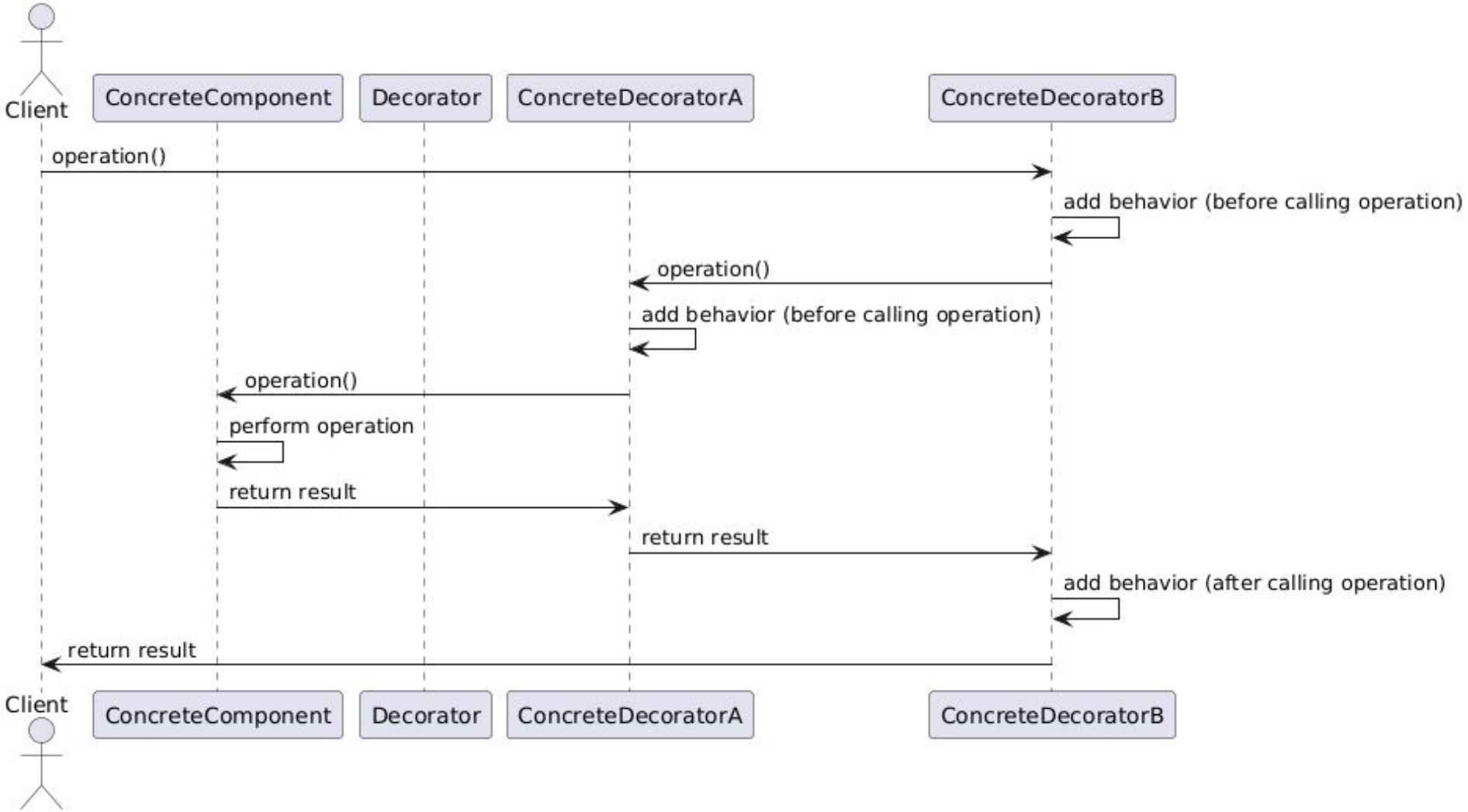


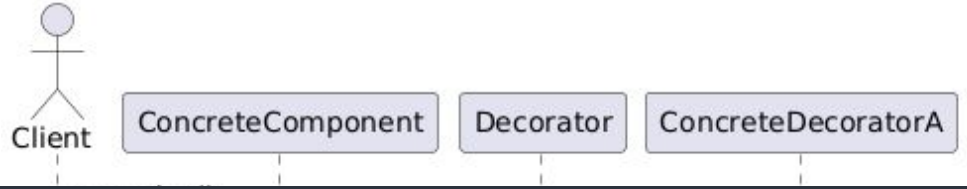
- **Component:** Interfaz base de los objetos que pueden ser decorados.
- **Concrete Component**: Objeto real que recibe decoraciones.
- **Decorator:** Clase abstracta que envuelve y delega en un **Component**
- **Concrete Decorator:** Implementa nuevas responsabilidades

Patrón Decorator - Consecuencias

+ Permite mayor flexibilidad que la herencia.
+ Permite agregar funcionalidad incrementalmente.

- Mayor cantidad de objetos.
- Mayor dificultad para depurar.


Client
ConcreteComponent
Decorator
ConcreteDecoratorA
ConcreteDecoratorB
operation()
add behavior (before calling operation)
operation()
add behavior (before calling operation)
operation()
perform operation
return result
return result
add behavior (after calling operation)
return result
Client
ConcreteComponent
Decorator
ConcreteDecoratorA
ConcreteDecoratorB




```
component = new ConcreteDecoratorB(
    new ConcreteDecoratorA(
        new ConcreteComponent()
    )
);

component.operation();
```

```
Component component = new ConcreteComponent();

component = new ConcreteDecoratorA(component);
component = new ConcreteDecoratorB(component);

component.operation();
```

Patrón Decorator - Cuestiones de implementación

- Misma interface entre componente y decorador.
- No hay necesidad de la clase Decorator abstracta, si se tiene un solo decorador.
- Cambiar el “skin” vs cambiar sus “guts”
- Decorador preserva la interface del objeto para el cliente
- Decoradores pueden y suelen anidarse