

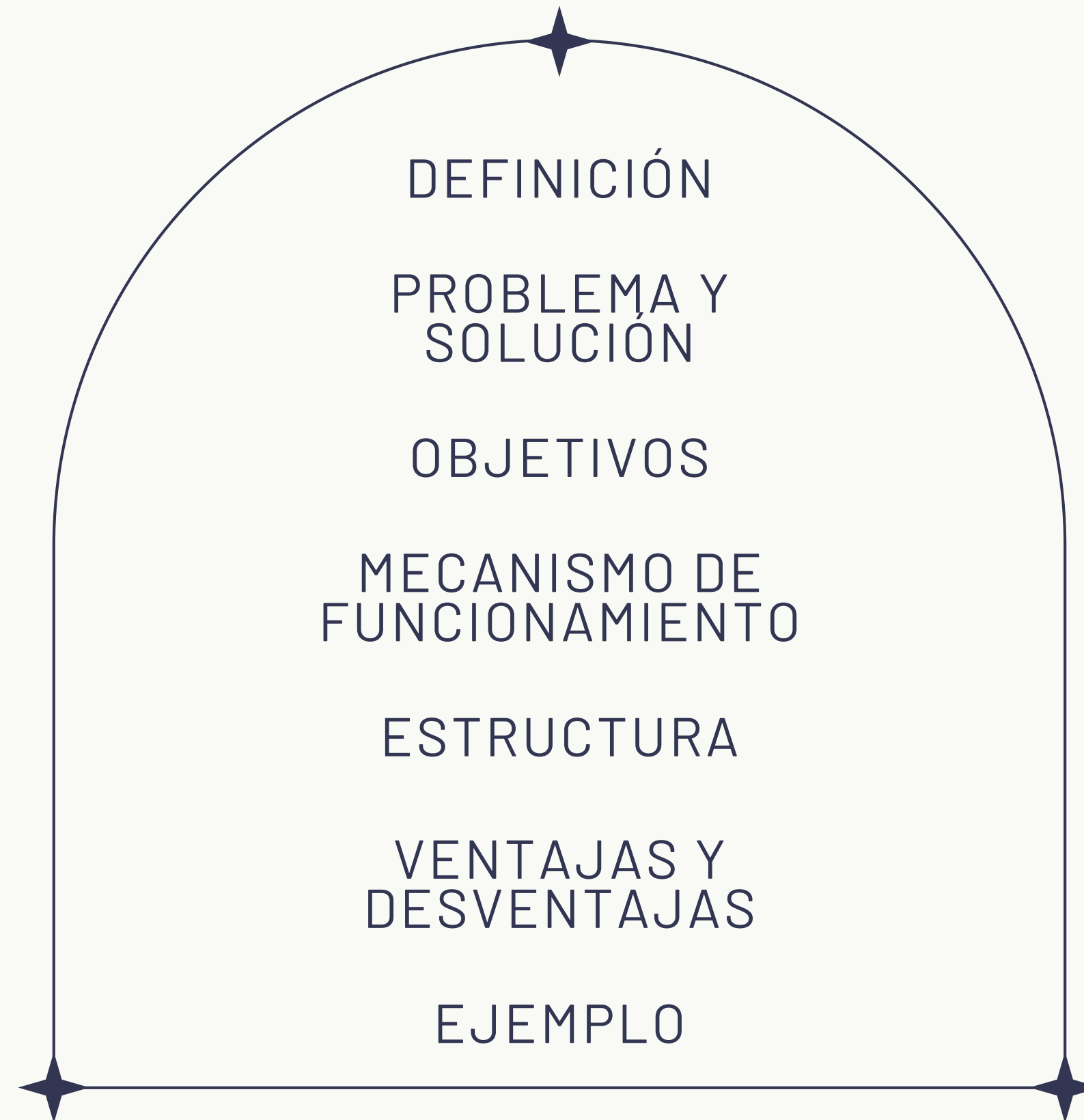
CI-0136 DISEÑO DE SOFTWARE

LABORATORIO 3

# PATRÓN DE DISEÑO **DECORATOR**

MARÍA PAULA LEUNG  
C34258

# CONTENIDOS





# DECORATOR



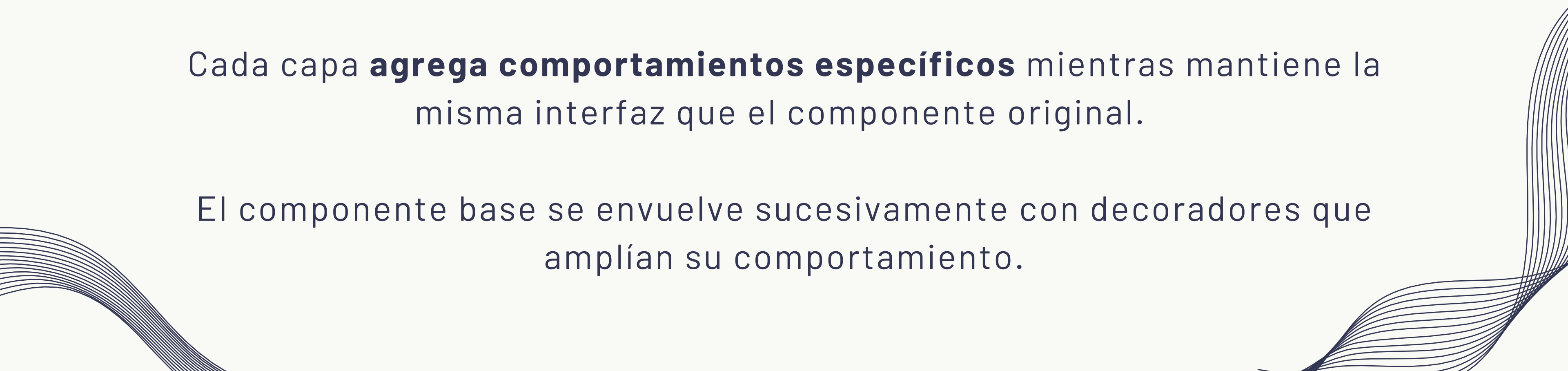
## PATRÓN ESTRUCTURAL



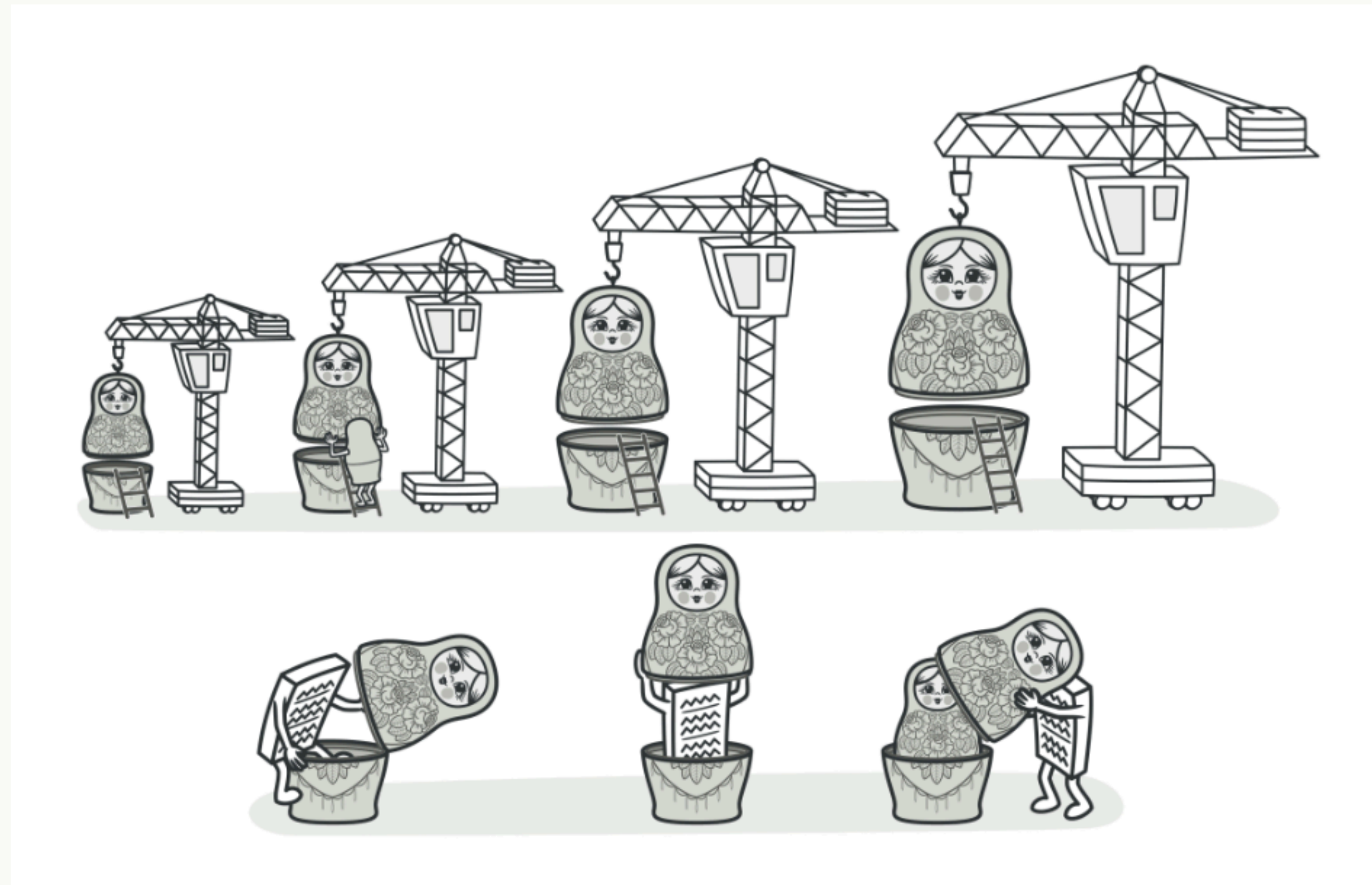
Organiza objetos en capas de funcionalidad mediante **decoradores/wrappers (envolturas)** encadenadas.

Cada capa **agrega comportamientos específicos** mientras mantiene la misma interfaz que el componente original.

El componente base se envuelve sucesivamente con decoradores que amplían su comportamiento.



# DECORATOR





# PROBLEMA

**HERENCIA TRADICIONAL RESULTA INEFICIENTE :**

**CUANDO SE NECESITA AÑADIR FUNCIONALIDADES A  
OBJETOS EXISTENTES**

Requiere crear múltiples subclases para cada combinación posible o modificar la jerarquía de clases existente, generando código rígido y difícil de mantener.



# SOLUCIÓN

## CLASES DECORADORAS:

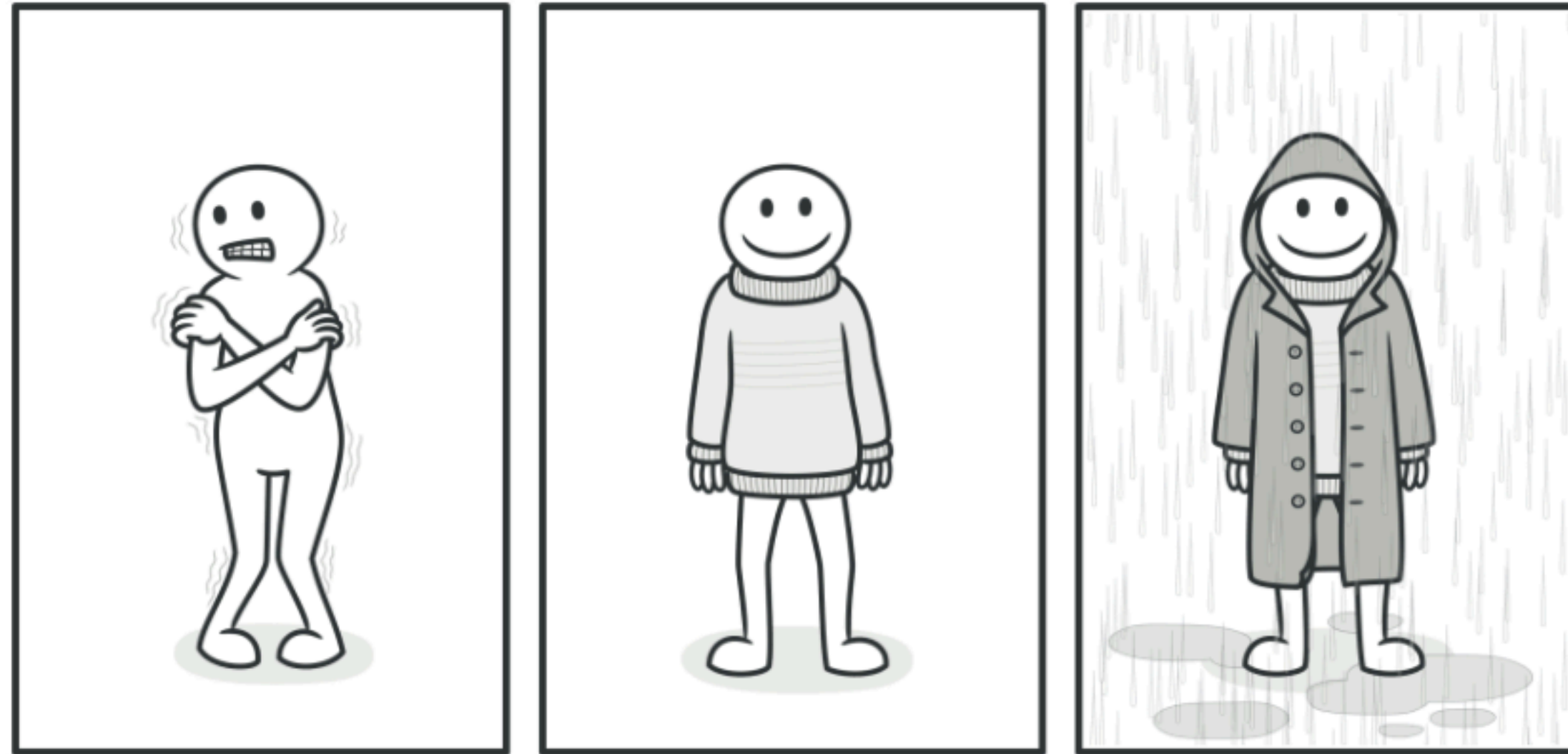
**AÑADIR NUEVAS RESPONSABILIDADES A OBJETOS  
DE FORMA DINÁMICA**

**Implementan la misma interfaz**

Facilitando la combinación de funcionalidades sin alterar la estructura original de las clases.



# ANALOGÍA



“ Todas estas prendas “extienden” tu comportamiento básico pero no son parte de ti, y puedes quitarte fácilmente cualquier prenda cuando lo desees.”

# OBJETIVOS

1.

Permitir la **composición flexible de comportamientos** a través de múltiples decoradores.

2.

**Añadir responsabilidades** a objetos individuales sin afectar a otros objetos de la misma clase.

3.

**Mantener el principio Open/Closed:** abierto para extensión pero cerrado para modificación.

4.

Facilitar la **reutilización modular de funcionalidades** en diferentes contextos.



# MECANISMO DE FUNCIONAMIENTO

- **Composición/Agregación:** el decorador permite **delegar llamadas al objeto original** mientras agrega su propio comportamiento.
- **Wrapper:** mantiene la **misma interfaz** que el objeto envuelto y permite **modificar o extender comportamiento**.
- **Encadenamiento:** **combina varios decoradores** agregando funcionalidades de manera dinámica y flexible.

# ESTRUCTURA

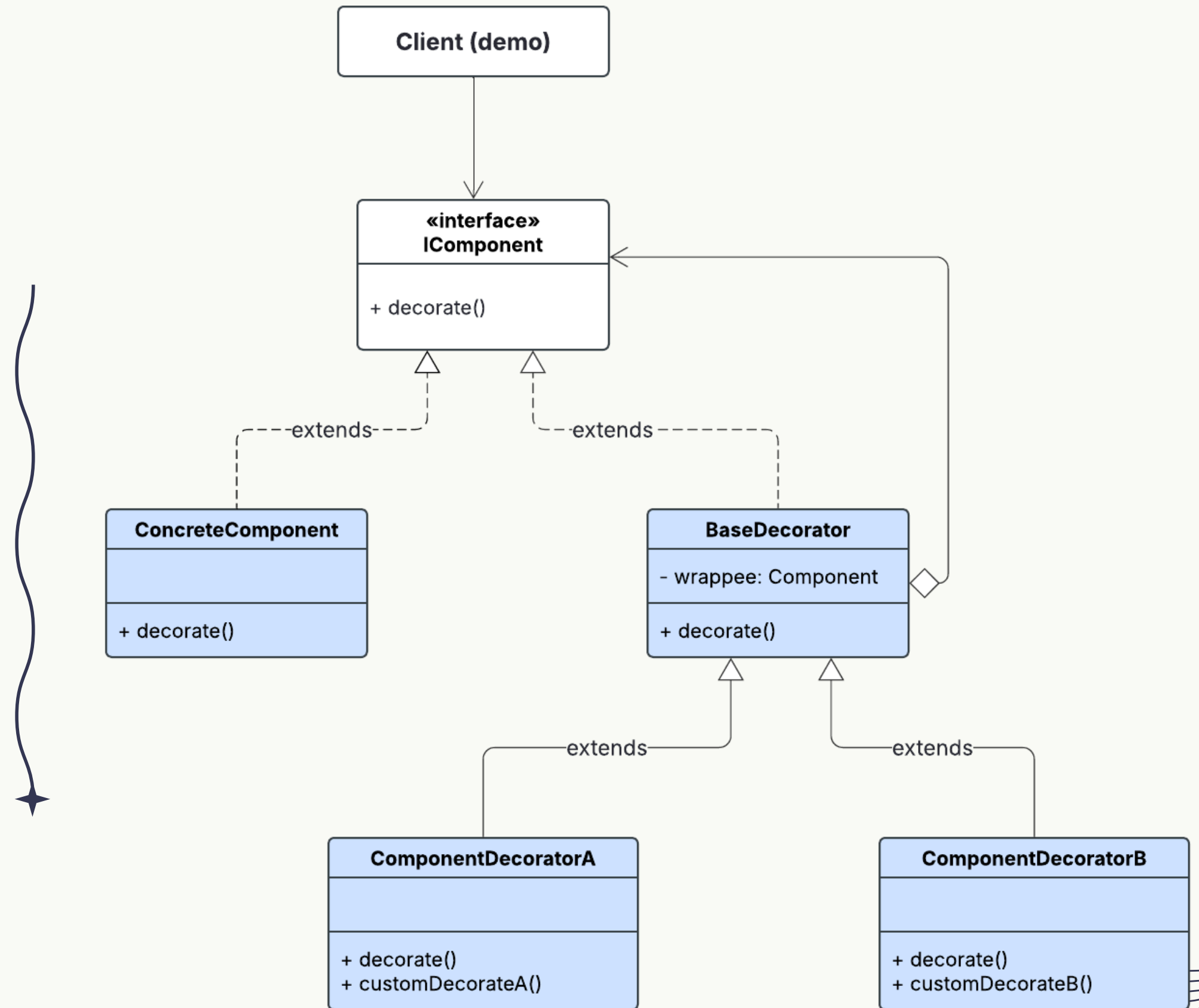
**Componente:** Define interfaz común para objetos envueltos y decoradores.

**Componente Concreto:** Representa el objeto base que los decoradores pueden extender.

**Decorator Base:** Contiene un campo que referencia al objeto envuelto y delega las operaciones.

**Decoradores Concretos:** Extienden la clase base y agregan funcionalidades adicionales.

**Cliente:** Envuelve los componentes en varias capas de decoradores.






# VENTAJAS

- **Flexibilidad:** agregar funciones sin modificar el código original.
- **Ampliación de funciones sin herencia**  
**rígida:** funcionalidades pueden añadirse o quitarse en tiempo de ejecución.
- **Composición de funcionalidades:** combina múltiples decoradores para crear objetos complejos.
- **Responsabilidad única:** cada decorador tiene una función específica y bien definida.

# DESVENTAJAS

- **Complejidad de software:** puede generar muchas clases pequeñas.
  - **Difícil de entender al inicio:** con varias capas de decoradores.
  - **Alto número de objetos:** requiere un buen control para evitar confusión con muchos objetos decoradores.
  - **Depuración complicada:** las cadenas de llamadas pueden ser difíciles de rastrear.
- 

The image features decorative wavy lines in the corners, composed of many thin, overlapping lines that create a sense of movement and depth. These lines are located in the top-left, top-right, bottom-left, and bottom-right corners of the slide.

# EJEMPLO

A decorative horizontal wavy line with a small star at its right end, positioned between the two main text blocks.

**SISTEMA DE USUARIOS CON  
ROLES Y PERMISOS**



# SISTEMA DE PERMISOS

## user\_permissions.py

Basado en niveles que determina las acciones permitidas según los roles del usuario:

```
ROLE_LEVEL = {  
    "user": 1,  
    "admin": 2,  
    "superadmin": 3  
}
```

```
PERMISSION_LEVEL = {  
    "view": 1,  
    "edit": 2,  
    "delete": 3  
}
```

Establece un sistema de jerarquía donde cada rol y permiso tiene un nivel numérico asociado. Esto permite comparaciones cuantitativas para determinar el acceso.

# COMPORTAMIENTO SEGÚN EL ROL

## USER

- **Roles:** ["user"]
- **Nivel máximo:** 1
- **Permisos:** nivel 1 ("view")
- **Sin permisos:** nivel 2 o superior ("edit", "delete")

## ADMIN

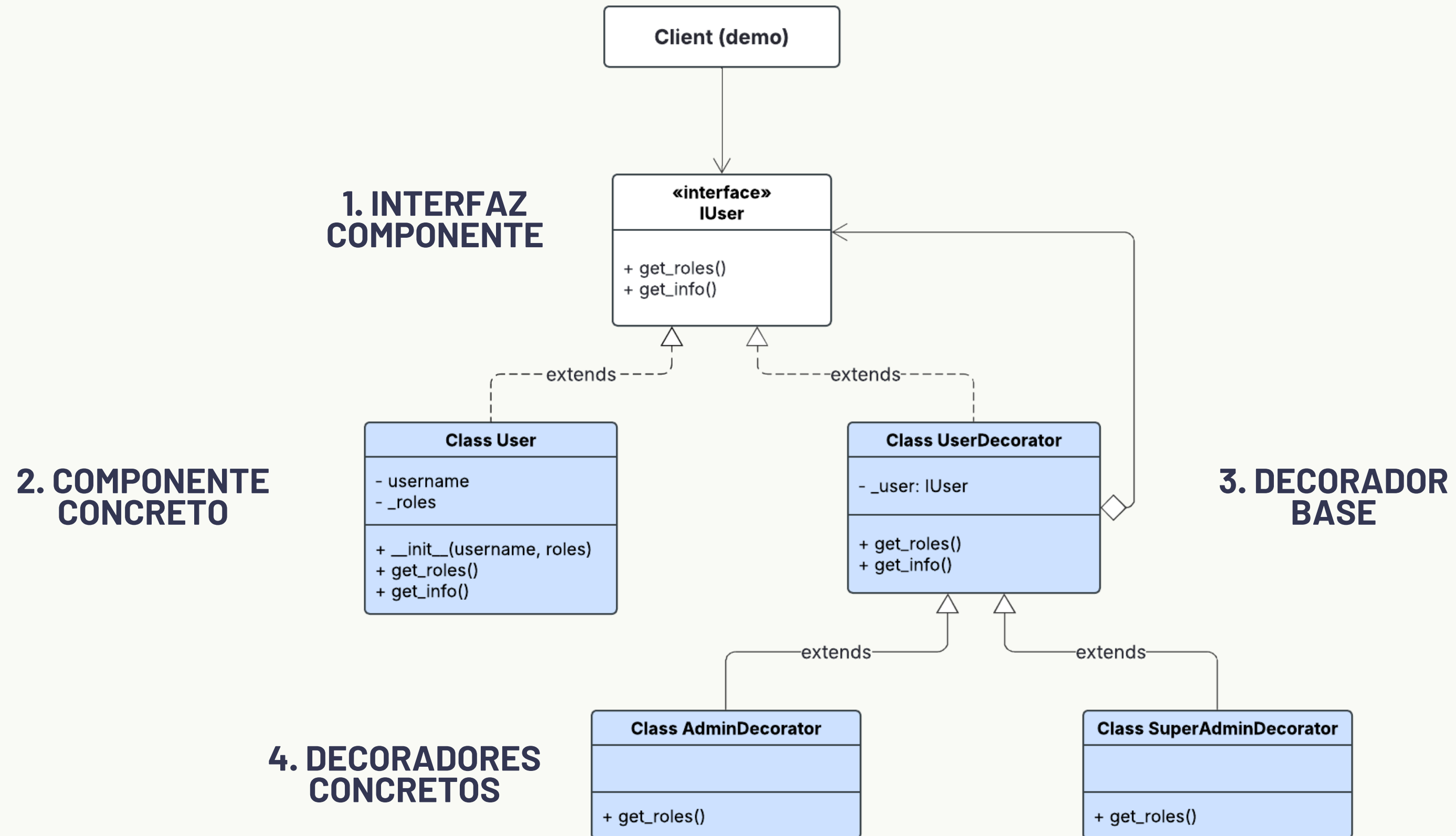
- **Roles:** ["user", "admin"]
- **Nivel máximo:** 2
- **Permisos:** nivel 1-2 ("view", "edit")
- **Sin permisos:** nivel 3 ("delete")

## SUPER ADMIN

- **Roles:** ["user", "admin", "superadmin"]
- **Nivel máximo:** 3
- **Permisos:** nivel 3 ("view", "edit", "delete")

# DECORADORES

✦  
user\_decorators.py



# 1. IUser - INTERFAZ COMPONENTE

```
class IUser(ABC):  
  
    @abstractmethod  
    def get_roles(self):  
        """Returns list of roles assigned to user"""  
        pass  
  
    @abstractmethod  
    def get_info(self):  
        """Returns user information"""  
        pass
```

- Define la interfaz que todos los usuarios y decoradores deben implementar.
- Obliga a que cualquier clase que implemente **IUser** tenga los métodos **get\_roles()** y **get\_info()**.



## 2. USER - COMPONENTE CONCRETO

```
class User(IUser):  
  
    def __init__(self, username, roles=None):  
        self.username = username  
        self._roles = roles or ["user"] # Default role is "user"  
  
    def get_roles(self):  
        return self._roles  
  
    def get_info(self):  
        return {"username": self.username,  
                "roles": self.get_roles()}
```

- Representa el usuario base que puede ser decorado con roles adicionales.
- Almacena información básica del usuario y su lista de roles.

# 3. USERDECORATOR - DECORADOR BASE

```
class UserDecorator(IUser):  
  
    def __init__(self, user: IUser):  
        self._user = user # Reference to the component being decorated  
  
    def get_roles(self):  
        return self._user.get_roles()  
  
    def get_info(self):  
        info = self._user.get_info()  
        info["roles"] = self.get_roles() # Update roles with decorated roles  
        return info
```

- Contiene una **referencia a un IUser** y delega todas las llamadas a los métodos del objeto envuelto.
- Permite **envolver un User o cualquier otro decorador** y extender su comportamiento sin modificar la clase original.

## 4. ADMINDECORATOR - DECORADOR CONCRETO

```
class AdminDecorator(UserDecorator):  
  
    def get_roles(self):  
        roles = self._user.get_roles()  
        if "admin" not in roles:  
            roles.append("admin")  
        return roles
```

- Añade el rol **admin** a un usuario existente.
- Sobrescribe el método `get_roles()` para incluir 'admin' si aún no está presente.

## 4. SUPERADMINDECORATOR - DECORADOR CONCRETO

```
class SuperAdminDecorator(UserDecorator):  
  
    def get_roles(self):  
        roles = self._user.get_roles()  
        if "superadmin" not in roles:  
            roles.append("superadmin")  
        return roles
```

- Añade el rol **superadmin** a un usuario existente.
- Sobrescribe el método `get_roles()` para incluir `superadmin` si aún no está presente.



# 5. CLIENTE

```
def show_permissions(usuario):
    for action in ["view", "edit", "delete"]:
        print(f"- Permission: {action} ->", has_permission(usuario, action))

# 5. CLIENT
def _demo():
    user = User("Maria")

    print("User (base):", user.get_info())
    show_permissions(user)

    admin = AdminDecorator(user)
    print("\nAdmin decorator applied to User:", admin.get_info())
    show_permissions(admin)

    super_admin1 = SuperAdminDecorator(admin)
    print("\nSuperAdmin decorator applied on Admin (User -> Admin -> SuperAdmin):", super_admin1.get_info())
    show_permissions(super_admin1)

    super_admin2 = SuperAdminDecorator(user)
    print("\nSuperAdmin decorator applied on User (User -> SuperAdmin):", super_admin2.get_info())
    show_permissions(super_admin2)

if __name__ == "__main__":
    _demo()
```

1. **Obtención de roles:** El sistema recupera los roles asociados al usuario.
2. **Determinación de nivel:** Calcula el nivel máximo entre los roles del usuario.
3. **Consulta de requisitos:** Identifica el nivel necesario para la acción solicitada.
4. **Decisión de acceso:** Compara niveles para determinar si se concede el permiso.

# REFERENCIAS

- [1] «Decorator». <https://refactoring.guru/es/design-patterns/decorator>
- [2] «Patrón Decorator: explicación, gráfico UML y ejemplos», IONOS Digital Guide, 19 de febrero de 2021. <https://www.ionos.com/es-us/digitalguide/paginas-web/desarrollo-web/que-es-el-patron-de-diseno-decorator/>
- [3] A. Lavasani, «Design Patterns in Python: Decorator», Medium, 14 de febrero de 2024. <https://medium.com/@amirm.lavasani/design-patterns-in-python-decorator-c882c0db650>

The image features a light gray background with decorative wavy lines in the corners. These lines are composed of many thin, dark blue lines that curve and overlap, creating a sense of movement and depth. The lines are most prominent in the top-left, top-right, and bottom-right corners, with some extending into the bottom-left corner.

**iMUCHAS  
GRACIAS!**