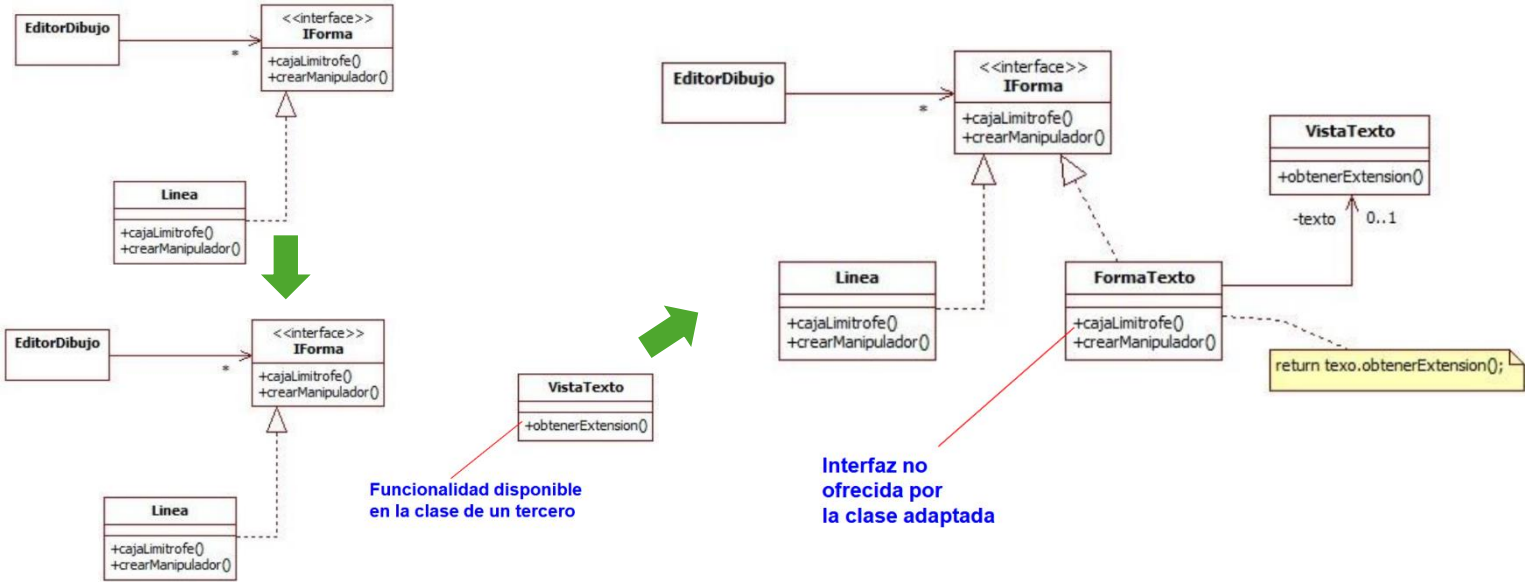
DISSO 9 – PATRONES ESTRUCTURALES

ADAPTER

- ADAPTADOR ⟶ convierte la **interfaz** de una clase en otra que es la que realmente esperan sus clientes.
- Permite cooperar a clases con **interfaces incompatibles**.

Se usa cuando:

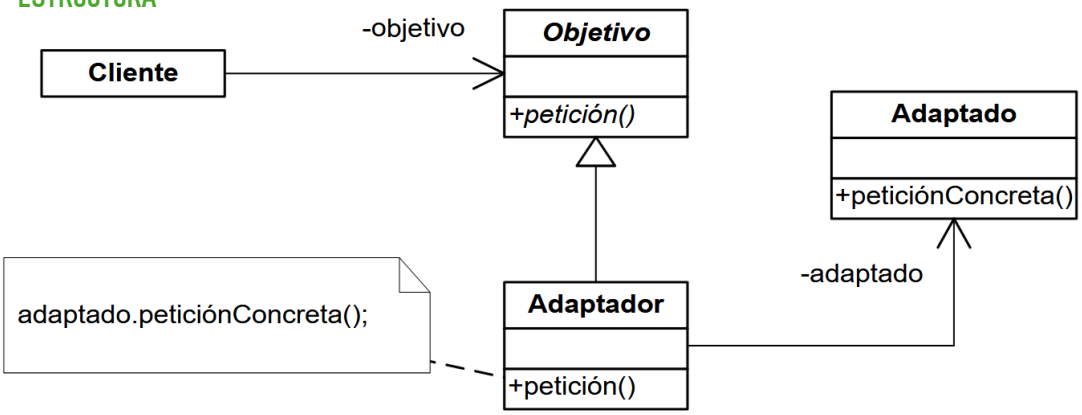
- Se quiere usar una **clase existente** y su **interfaz no concuerda** con la necesaria.
- Se quiere crear una **clase reutilizable** que coopere con clases no previstas.
- Es necesario usar varias subclases existentes y **no es práctico heredar** para adaptar su interfaz.



PARTICIPANTES

- OBJETIVO ⟶ define **interfaz específica** del dominio que usa el Cliente.
- CLIENTE ⟶ colabora con objetivos que se ajustan a la interfaz Objetivo.
- ADAPTADO ⟶ define la **interfaz** existente que necesita ser **adaptada**.
- ADAPTADOR ⟶ **adapta** las **interfaces** Adaptado y Objetivo.

ESTRUCTURA



VENTAJAS E INCONVENIENTES

- ✓ Permite al **mismo Adaptador** funcionar con **muchos Adaptados** y añadirles **funcionalidad**.
- ✗ Más difícil **redefinir** comportamiento de **Adaptado** ⟶ crear subclases de Adaptado y hacer que Adaptador se refiera a ellas.

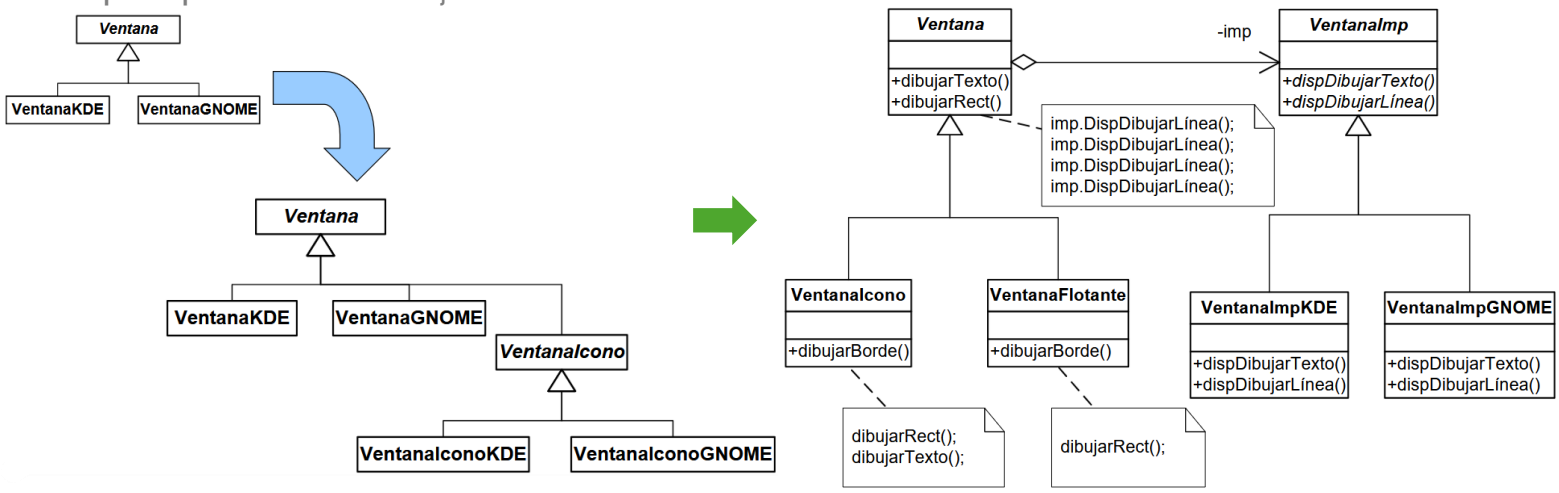
BRIDGE

- PUENTE → **separa una abstracción de su implementación** para que ambas puedan variar independientemente.
- A diferencia de Adaptador, está pensado para **cambiar la interfaz** de un **objeto existente**.

➢ **Motivación** → necesidad de **varias implementaciones** posibles para la misma abstracción, pero el enfoque habitual mediante **herencia** es **poco flexible**.

Se usa para:

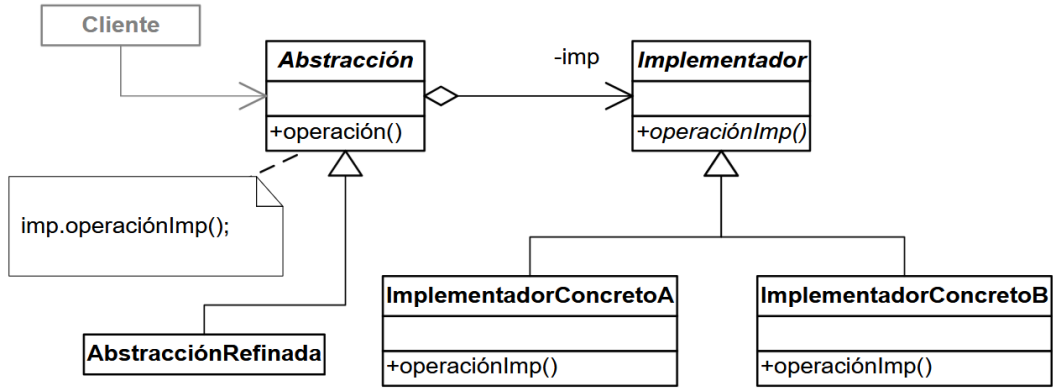
- **Evitar** un **vínculo permanente** entre abstracción e implementación (si debe decidirse en **tiempo de ejecución**).
- Cuando abstracción e implementación deban ser **ampliables** mediante **herencia**.
- Impedir que los **cambios** en la **implementación** tengan impacto en los clientes (evita **recompilación**).
- **Compartir implementación** entre varios objetos ocultándola a los clientes.



PARTICIPANTES

- ABSTRACCIÓN → define **interfaz de la abstracción** y referencia a un objeto de tipo Implementador.
- ABSTRACCIÓN REFINADA → **implementa** la interfaz definida por Abstracción.
- IMPLEMENTADOR → define **interfaz** para **clases de implementación** (es habitual que sus operaciones sean de más bajo nivel que las de Abstracción).
- IMPLEMENTADOR CONCRETO → **implementa** la interfaz Implementador.

ESTRUCTURA



VENTAJAS E INCONVENIENTES

✓ **Desacopla** interfaz e implementación:
  - La **implementación** de una abstracción puede configurarse en **tiempo de ejecución**.
  - Elimina **dependencias de compilación**.

✓ Permite **desarrollar independientemente** Abstracción e Implementador.

✓ Oculta **detalles de implementación** a los clientes.

ADAPTER VS BRIDGE

La diferencia fundamental está en su **propósito**:

- **Bridge** → une **abstracción e implementaciones**, ofrece interfaz estable a los clientes y permite que **cambie** su implementación.
- **Adapter** → resuelve **incompatibilidades** entre **interfaces existentes** sin preocuparse de cómo podrían **evolucionar** independientemente.

Se usan en diferentes puntos del **ciclo de vida**:

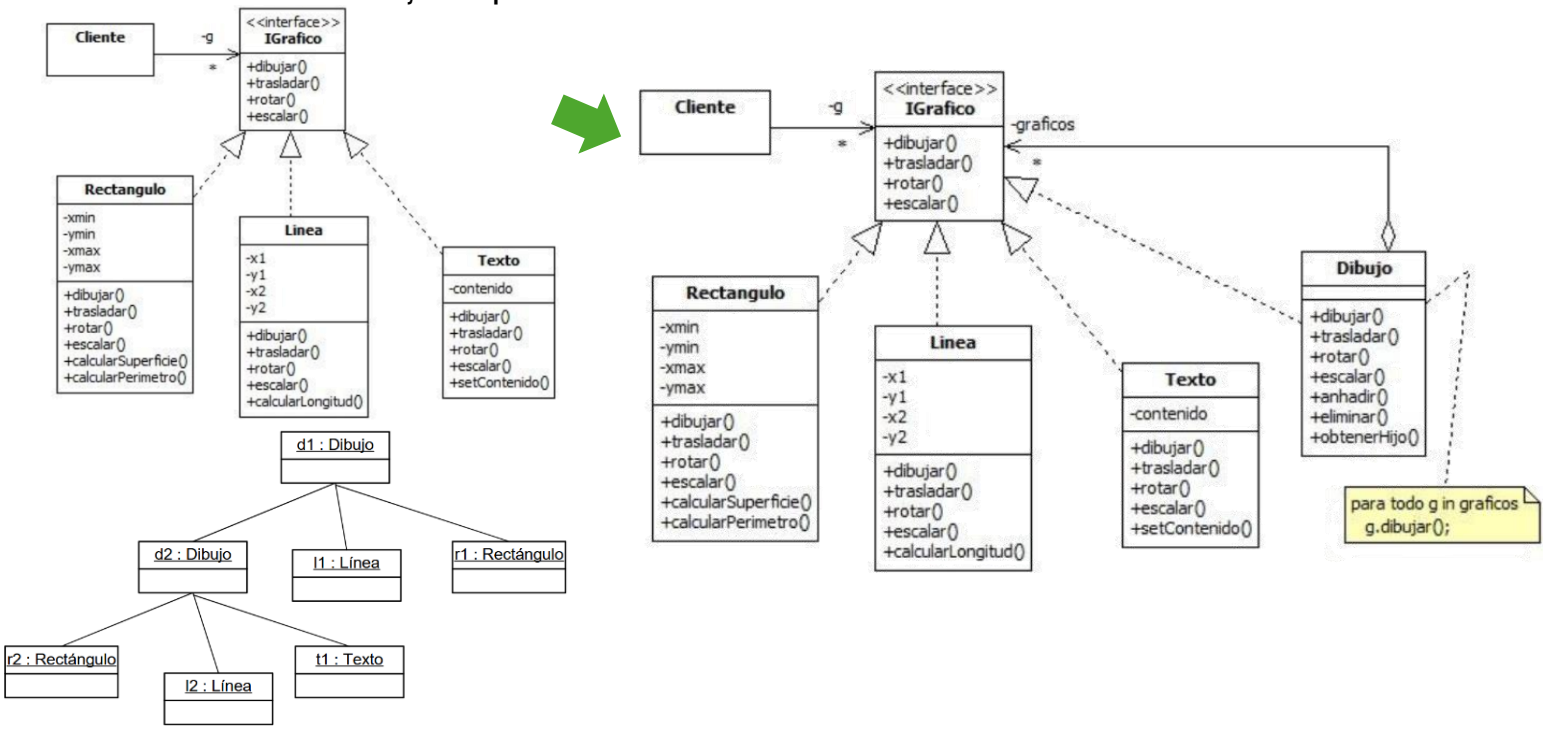
- **Bridge** → hace que las cosas funcionen **antes** de ser diseñadas.
- **Adapter** → hace que las cosas funcionen **después** de ser diseñadas.

COMPOSITE

- COMPOSITE ⟶ organiza objetos en **estructuras de árbol** para representar jerarquías.
- Permite tratar de manera **uniforme** a **individuos** y **grupos**.

Se usa para:

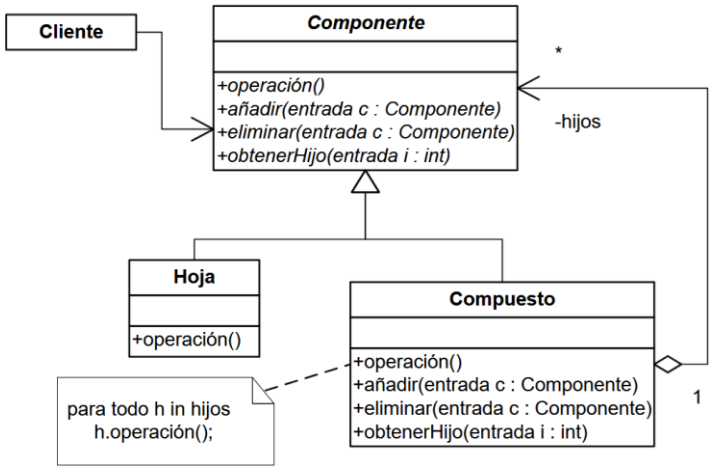
- Representar jerarquías de objetos **parte-todo**.
- Obviar diferencias entre los **individuos** y sus **composiciones**.



PARTICIPANTES

- COMPONENTE ⟶ declara **interfaz** para los objetos de la agregación e implementa **comportamiento predeterminado**.
- HOJA ⟶ establece **comportamiento** de **objetos primitivos** (hojas).
- COMPUESTO ⟶ define **comportamiento** de componentes **con hijos** e implementa las **operaciones de acceso** a los hijos.
- CLIENTE ⟶ manipula objetos a través de la interfaz Componente.

ESTRUCTURA



VENTAJAS E INCONVENIENTES

- ✓ Permite **agregación recursiva**.
- ✓ **Uniforma** el acceso a los componentes: los clientes son simples porque no saben si tratan con hojas o compuestos.
- ✓ Facilita la adición de **nuevos tipos de componentes**.
- ✗ Su diseño general dificulta **restringir los componentes** permitidos en un grupo.

IMPLEMENTACIÓN

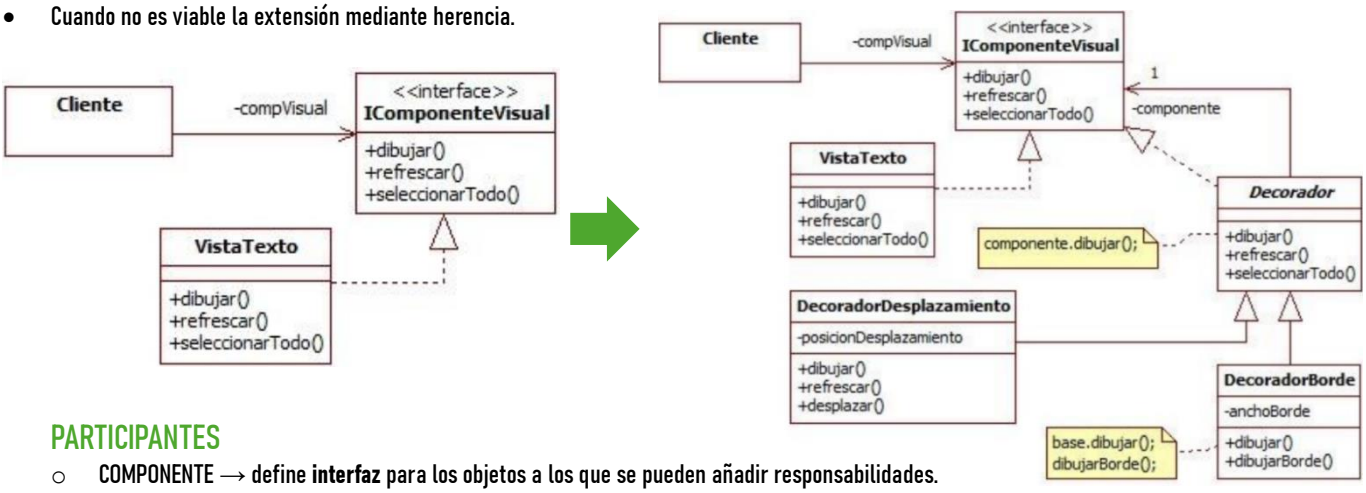
- Mantener referencias de **hijos a padres** ⟶ simplifica el recorrido por la jerarquía.
- **Maximizar** interfaz **Componente**:
  - Definir tantas **operaciones comunes** a Hoja y Compuesto como sea posible.
  - Componente **implementa** las operaciones **no aplicables a los hijos**.

DECORATOR

- DECORADOR → asigna/retira **responsabilidades adicionales** a un objeto particular **dinámicamente**.
- Proporciona una **alternativa flexible** a la **herencia** para ampliar funcionalidad.
- Suele combinarse con **Composite**.

Se usa para:

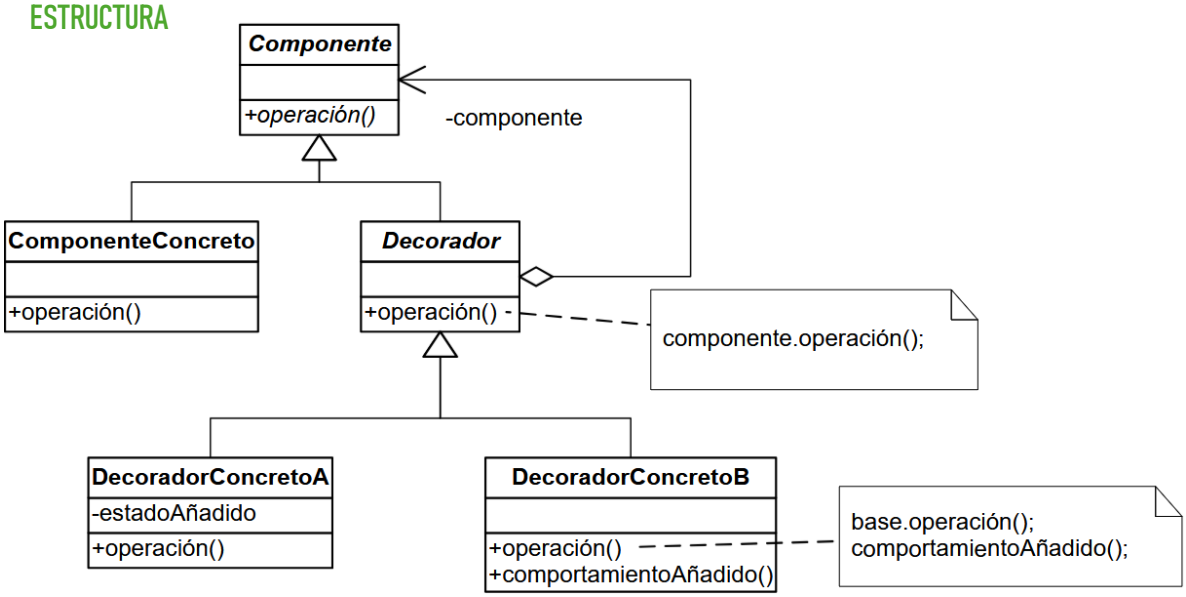
- Otorgar o revocar responsabilidades a objetos individuales de manera dinámica y transparente.
- Cuando no es viable la extensión mediante herencia.



PARTICIPANTES

- COMPONENTE → define **interfaz** para los objetos a los que se pueden añadir responsabilidades.
- COMPONENTE CONCRETO → define el **objeto original** al que se puede decorar.
- DECORADOR → **referencia** a un Componente y se **ajusta** a su interfaz.
- DECORADOR CONCRETO → **añade responsabilidades** al componente.

ESTRUCTURA



VENTAJAS E INCONVENIENTES

- ✓ Proporciona mayor **flexibilidad** que la **herencia** (que es estática).
- ✓ **Evita** clases **cargadas** de funciones en la **parte superior** de la jerarquía.
- ✗ Su uso suele dar lugar a muchos **objetos pequeños** y muy **parecidos** entre sí.

RELACIÓN CON OTROS PATRONES

- Se diferencia de **Adapter** en que cambia las **responsabilidades** de un **objeto**, **no** su **interfaz**.
- Puede ser visto como un **Composite** desvirtuado con **un solo componente** que, además, **añade responsabilidades**.

COMPOSITE VS DECORATOR

Ambos se basan en la **agregación recursiva**, pero con objetivos distintos.

- **Decorator** → añade **responsabilidades** a ciertos objetos **sin** crear **subclases**.
- **Composite** → **uniformiza** el acceso a los individuos y a sus composiciones.

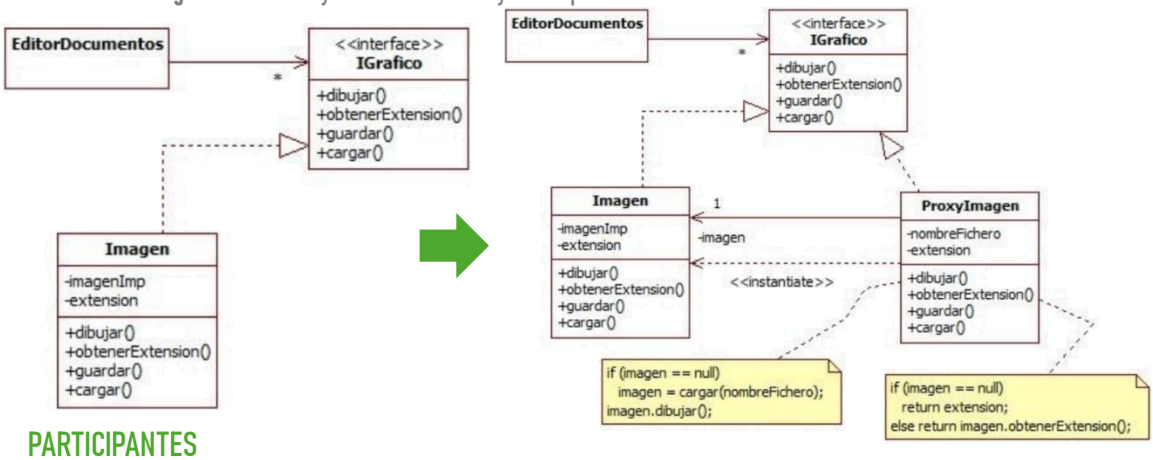
▸ Ambos propósitos son **complementarios**.

PROXY

- APODERADO ⟶ suministra un **representante** o **sustituto** de otro objeto para controlar el **acceso** a este.
- La **interfaz** del Proxy es **idéntica** a la del **objeto** representado.

Aplicación:

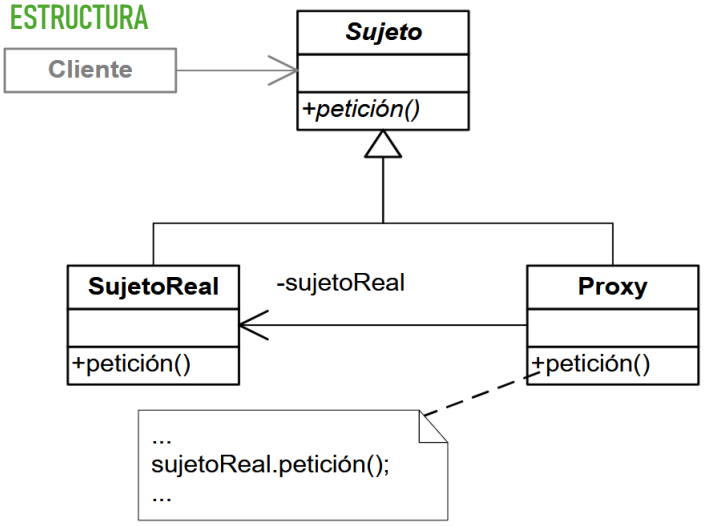
- Proxy **remoto** ⟶ representa localmente a un objeto en otro espacio de direcciones.
- Proxy **virtual** ⟶ crea objetos de alto coste por encargo.
- Proxy **de protección** ⟶ controla el acceso al objeto original.
- **Referencia inteligente** ⟶ sustituye a una referencia y hace operaciones adicionales.



PARTICIPANTES

- PROXY ⟶ controla el acceso al **objeto real** y puede ser responsable de su creación y borrado.
  - ↳ Otras responsabilidades dependen del tipo: conversión de direcciones, acceso diferido, comprobación de permisos de acceso.
- SUJETO ⟶ define la **interfaz común** para Sujeto Real y Proxy.
- SUJETO REAL ⟶ es el objeto real representado.

ESTRUCTURA



VENTAJAS E INCONVENIENTES

✓ La **indirección adicional** introducida tiene varios usos posibles:

- Proxy **remoto** ⟶ oculta que un objeto resida en un espacio de direcciones distinto.
- Proxy **virtual** ⟶ crea un objeto por encargo.
- Proxy **de protección** ⟶ regula el acceso al objeto representado.

❖ Creación diferida de copias: uso de contadores de referencias.

RELACIÓN CON OTROS PATRONES

- El **Adaptador** proporciona una **interfaz diferente** para el objeto adaptado.
- El **Proxy** tiene la **misma interfaz** que su representado, pero puede rechazar peticiones soportadas por éste.
- El **Decorador** puede representar una implementación parecida a la de un Proxy, pero el propósito es distinto.

DECORATOR VS PROXY

Ambos proporcionan una **interfaz idéntica** a la de un objeto original, pero se diferencian en su **propósito**.

- **Decorator** ⟶ asigna propiedades dinámica y recursivamente.
- **Proxy** ⟶ proporciona un sustituto de un sujeto cuando no es deseable el acceso directo.