

# apuntes\_patrones

## Note

Estos apuntes están basados en el libro Patrones de Diseño de Erich Gamma junto con los apuntes de DS que nos proporciona la Universidad.

En cada patrón, los apartados de Motivación y Estructura utilizan el segundo diagrama (es decir, el que nos da César).

Al final de cada patrón hay un ejemplo que nos da César para entender el patrón (que no deberías de estudiarlo como tal, sino más bien entenderlo)

## Strategy

### Propósito

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan

### También conocido como

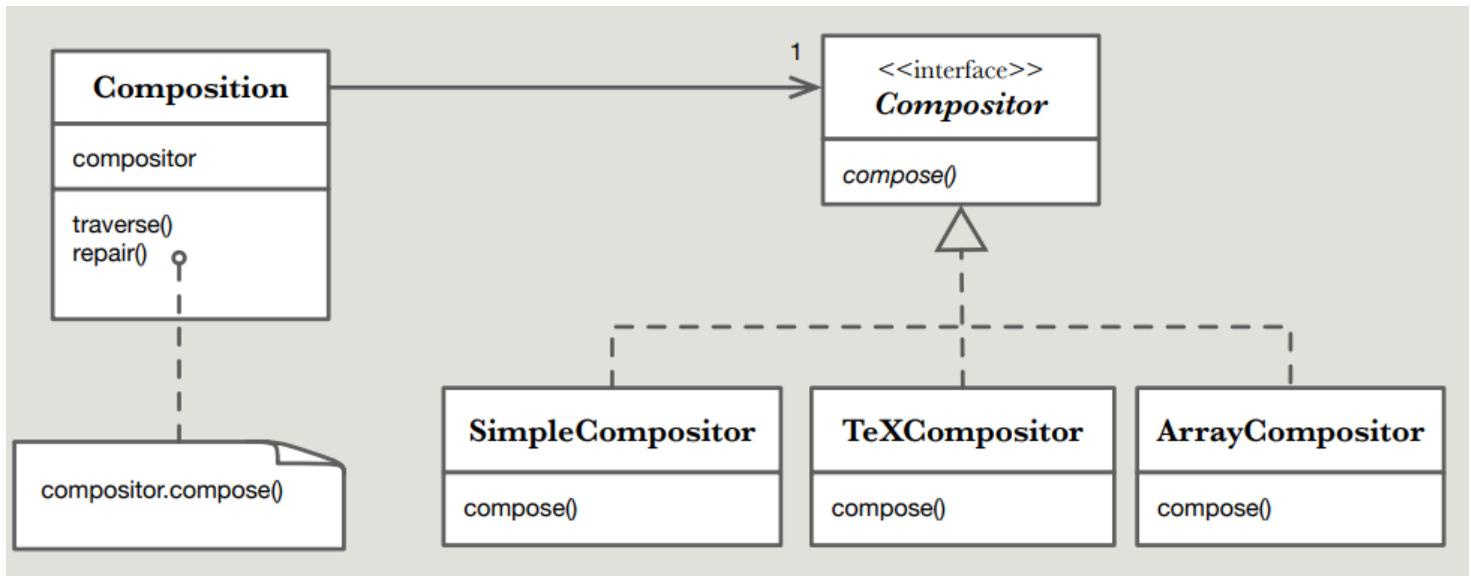
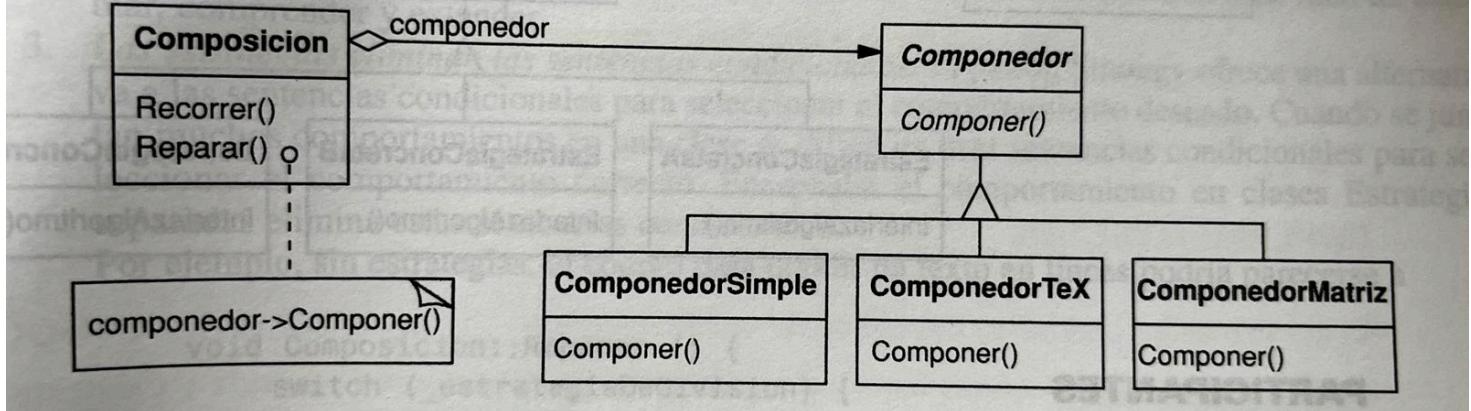
*Policy* (política)

### Motivación

Existen muchos algoritmos para dividir en líneas un flujo de texto. No resulta una buena práctica por varias razones:

- Los clientes que necesitan dividir el texto en líneas se vuelven más complejos si tienen que incluir dicho código, lo que los hace más grandes y más difíciles de mantener
- No tenemos por qué permitir múltiples algoritmos si no los vamos a usar todos
- Es difícil añadir nuevos algoritmos o modificar los existentes

Estos problemas pueden evitarse definiendo clases que encapsulen los diferentes algoritmos de división en líneas. Un algoritmo así encapsulado se denomina estrategia.

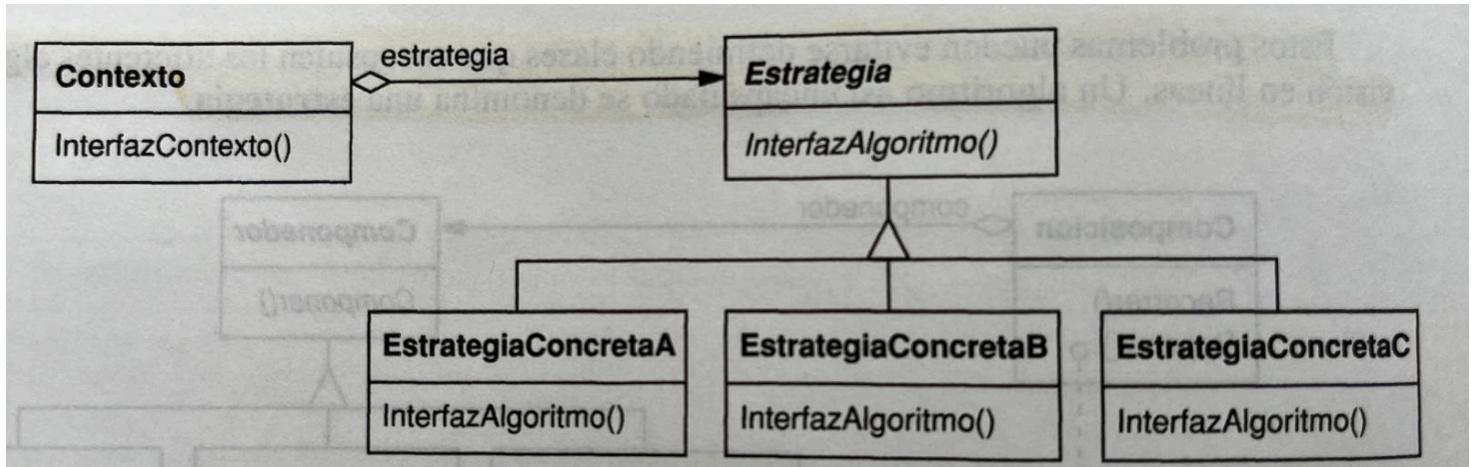


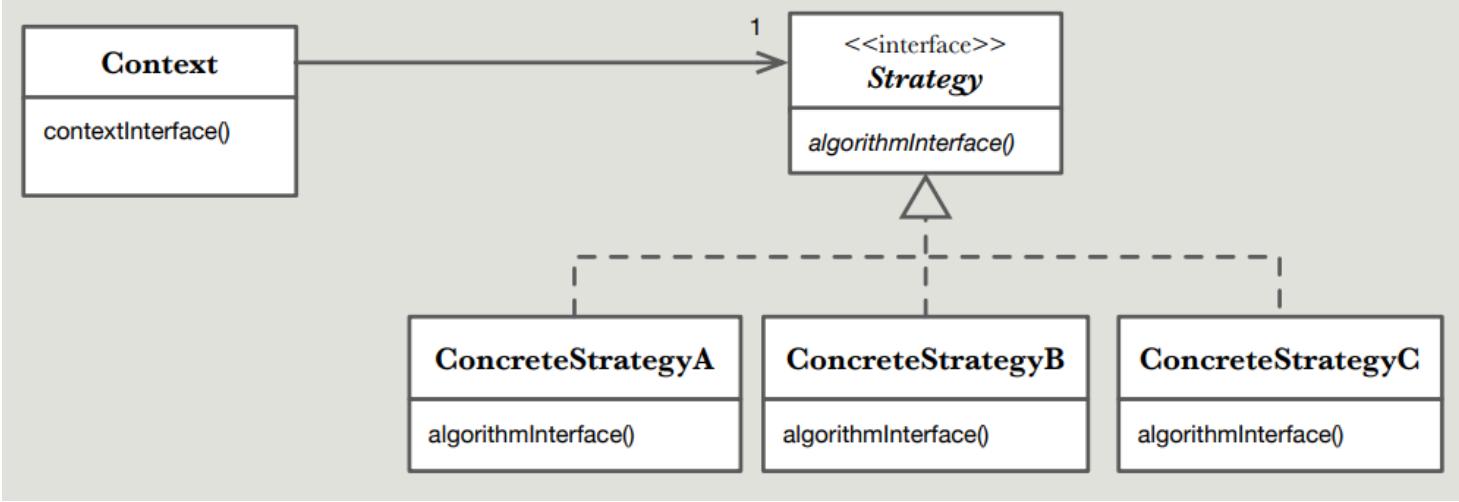
## Aplicabilidad

Usar el patrón Strategy cuando:

- Muchas clases relacionadas difieren sólo en su comportamiento
- Se necesitan distintas variantes de un algoritmo
- Un algoritmo usa datos que los clientes no deberían conocer
- Una clase define muchos comportamientos, y éstos se representan como múltiples sentencias condicionales en sus operaciones

## Estructura





## Participantes

- **Estrategia** (Componedor)
  - Declara una interfaz común a todos los algoritmos permitidos. El Contexto usa esta interfaz para llamar al algoritmo definido por una EstrategiaConcreta
- **EstrategiaConcreta** (ComponedorSimple, ComponedorTeX, ComponedorMatriz)
  - Implementa el algoritmo usando la interfaz Estrategia
- **Contexto** (Composición)
  - Se configura con un objeto EstrategiaConcreta
  - Mantiene una referencia a un objeto Estrategia
  - Puede definir una interfaz que permita a la Estrategia acceder a sus datos

## Colaboraciones

- La estrategia y el contexto colaboran para implementar el algoritmo escogido
  - El contexto puede pasar todos los datos que necesita al llamar a la estrategia concreta
  - O se puede pasar a sí mismo como referencia para que ésta llame a los métodos que necesite
- Los clientes colaboran con el contexto
  - Pueden pasársela la estrategia concreta

## Consecuencias

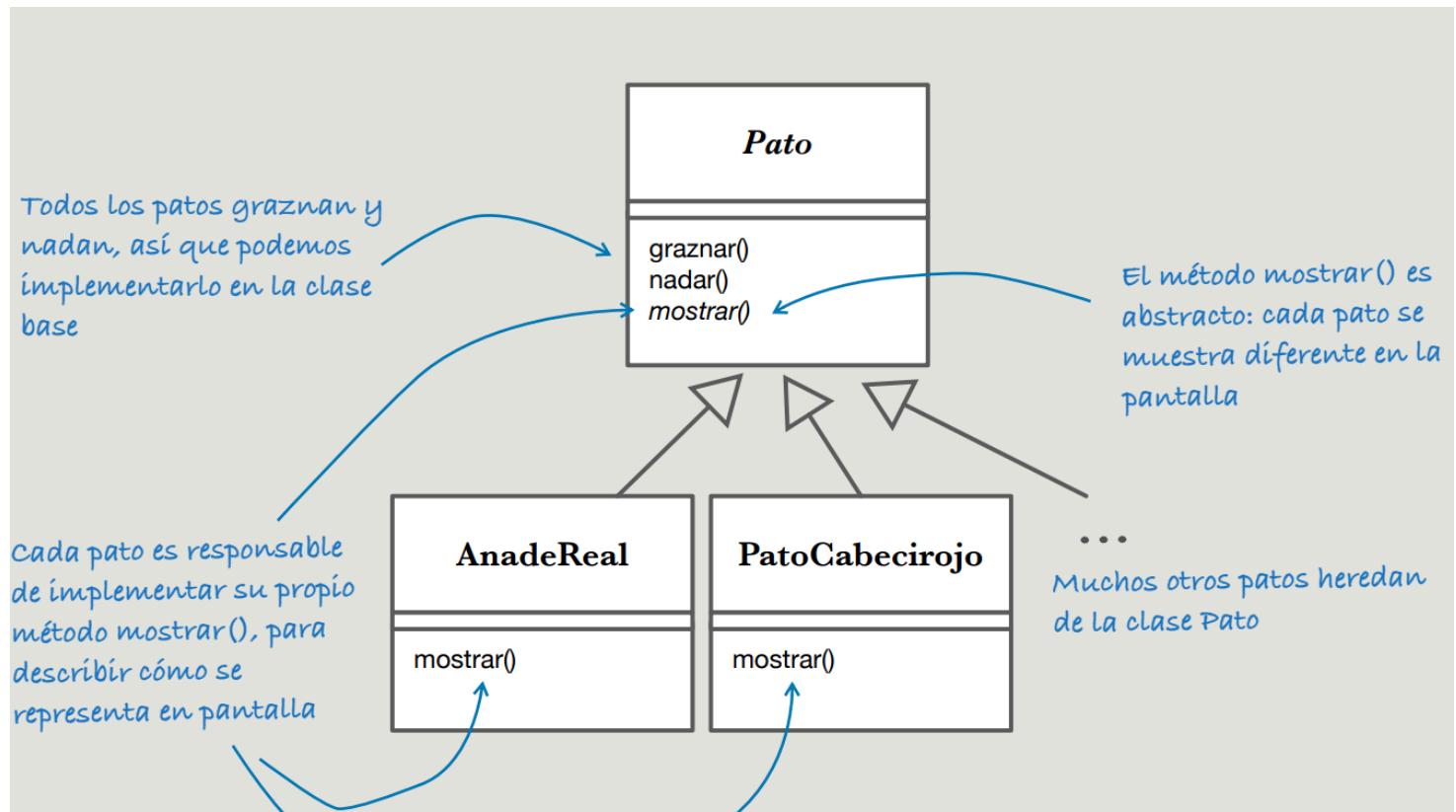
- Define familias de algoritmos relacionados
- Es una alternativa a la herencia
  - Hace que el contexto sea más fácil de entender, modificar y mantener
  - Evita la duplicación de código
  - Evita la explosión de subclases
  - Se puede cambiar dinámicamente
- Elimina las múltiples sentencias condicionales
- El cliente puede elegir entre varias implementaciones (incluso dinámicamente)
- Los clientes deben conocer las diferentes estrategias
- Puede complicarse la comunicación entre el contexto y las estrategias

- Crece el número de objetos

## Posibles usos

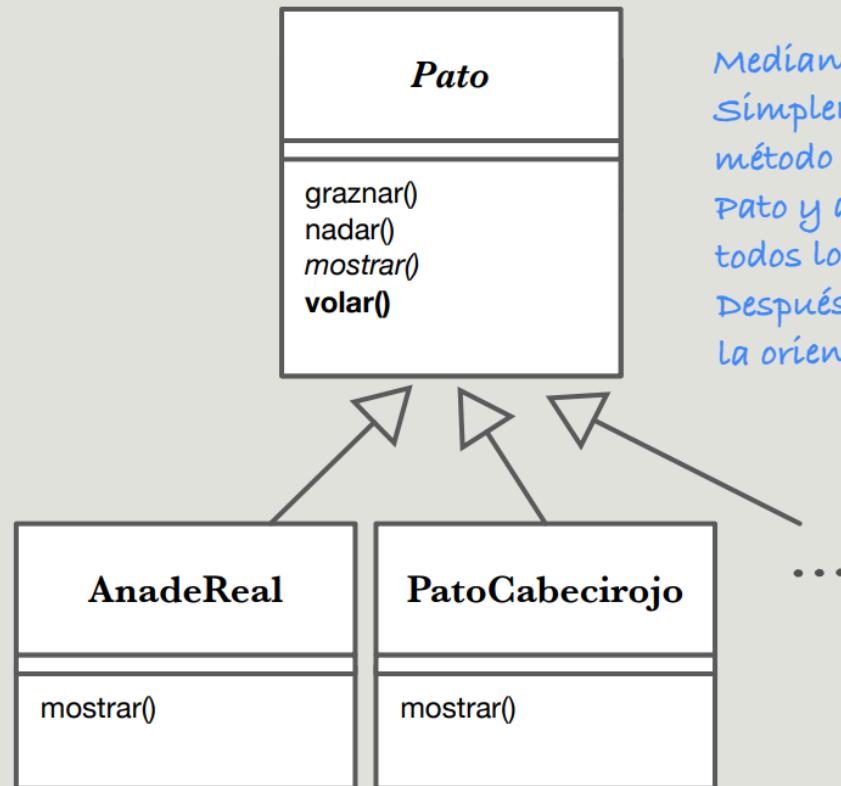
- Validadores de campos de formularios
- Distintas modalidades de juego
  - En un juego de póquer, dominó, etc
  - Niveles de dificultad en el ajedrez o un simulador de coches

## Ejemplo que nos da César



¿Cómo hacemos para que ahora vuelen?

# Solución tradicional



Mediante la herencia.  
Simplemente añadimos un  
método «volar» a la clase  
**Pato** y automáticamente  
todos los patos lo heredarán.  
Después de todo, ¿no era eso  
la orientación a objetos?

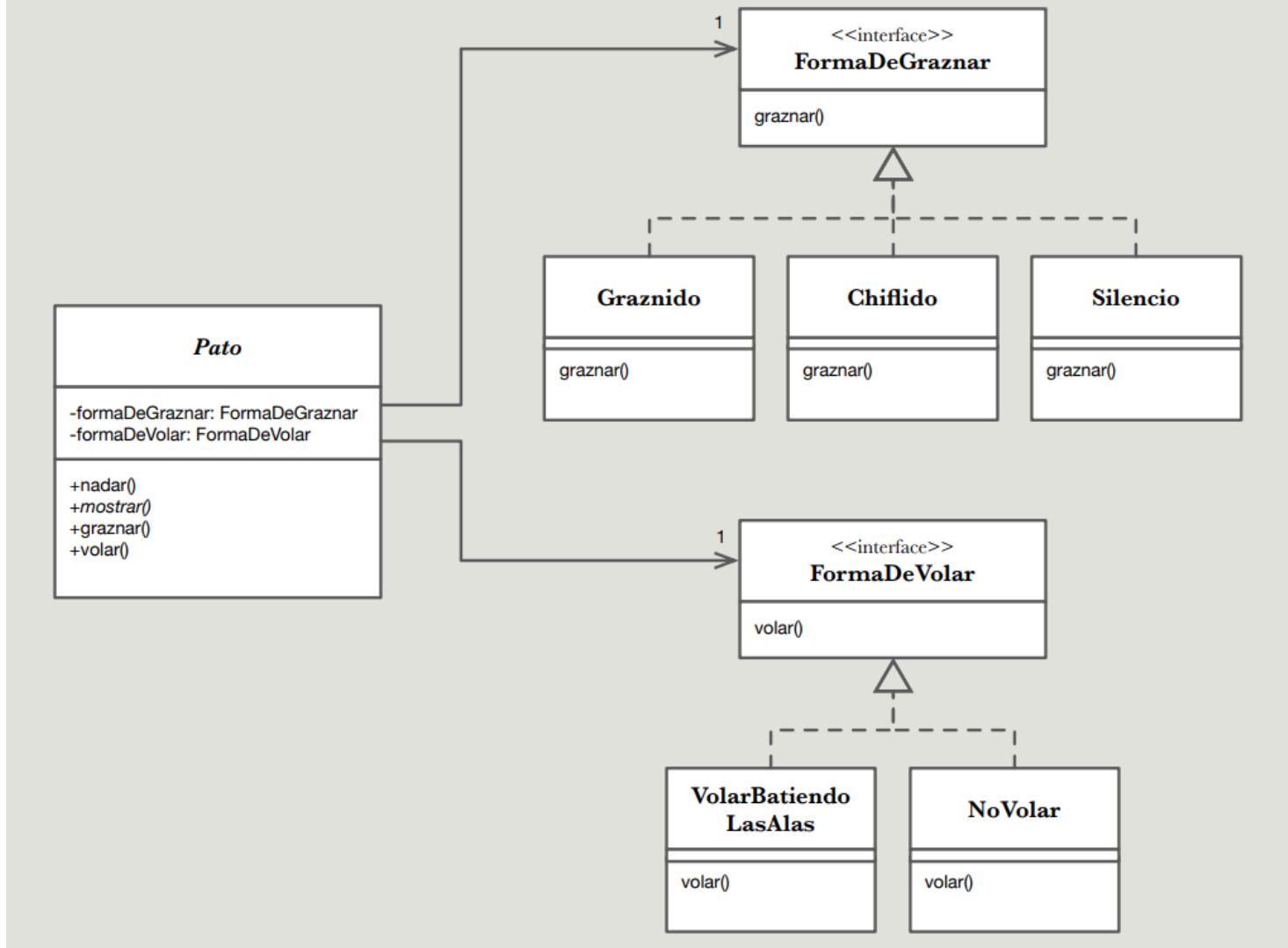
## ⚡ Error

Esto conlleva un problema

- Ahora tenemos un montón de patos de goma volando por la pantalla
- Al implementar `volar` en la superclase, hemos dado esa habilidad a todos los patos, incluyendo aquellos que no deberían

Una posibilidad sería que el `volar` en `PatoDeGoma` no hiciera nada (estuviese vacío)

**Solución:**



## Factory Method

### Propósito

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos

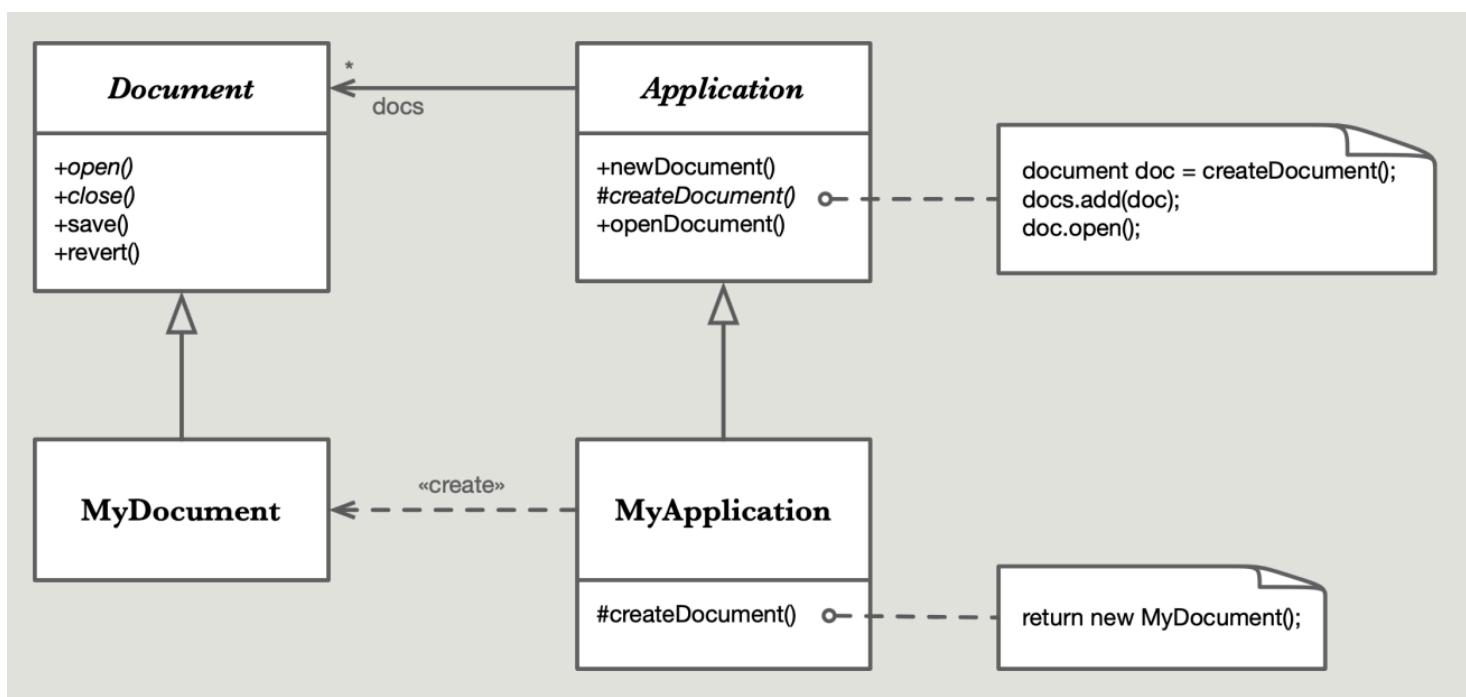
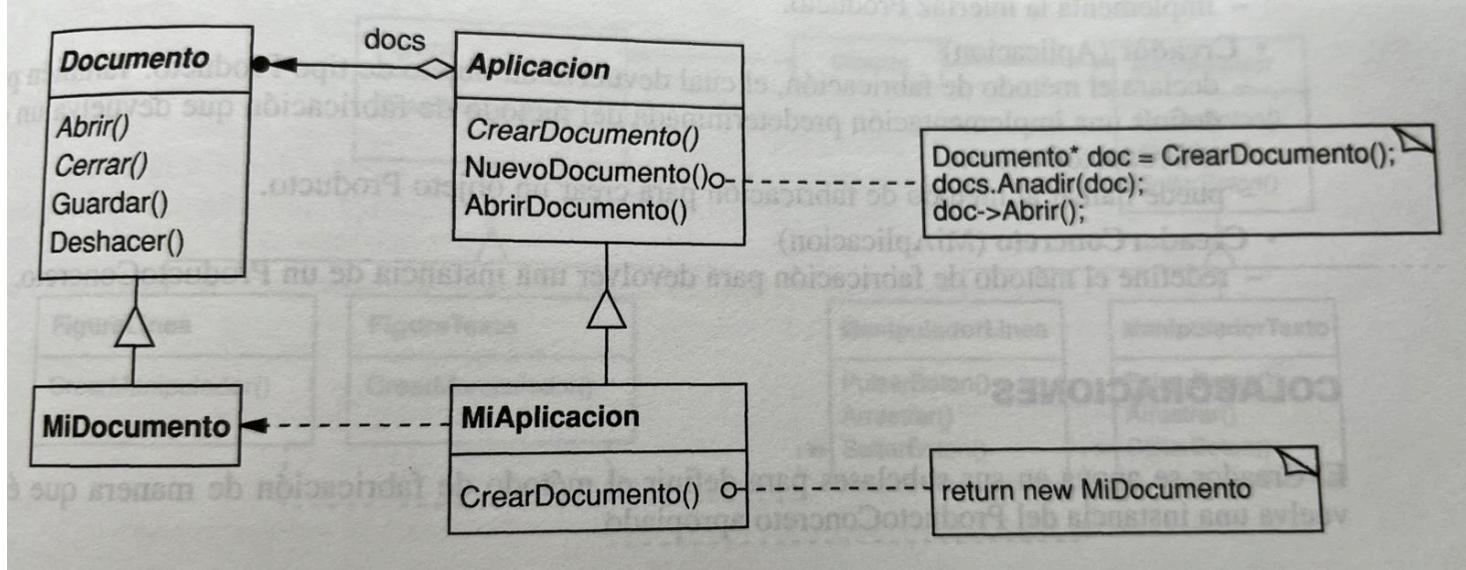
### También conocido como

*Virtual Constructor* (*Constructor Virtual*)

### Motivación

- Sea un framework para la construcción de editores de documentos de distintos tipos
  - Clases abstractas `Application` y `Document`
    - Los clientes deberán heredar de ellas para implementar los detalles de cada aplicación concreta
    - P. ej. `DrawingApplication` y `DrawingDocument`
  - ¿Cómo se implementa la opción `New` del menú?
  - ¿Cómo sabe la clase `Application` qué tipo concreto del documento debe crear?

Llamaremos a `CrearDocumento` un método de fabricación porque es el responsable de "fabricar" un objeto

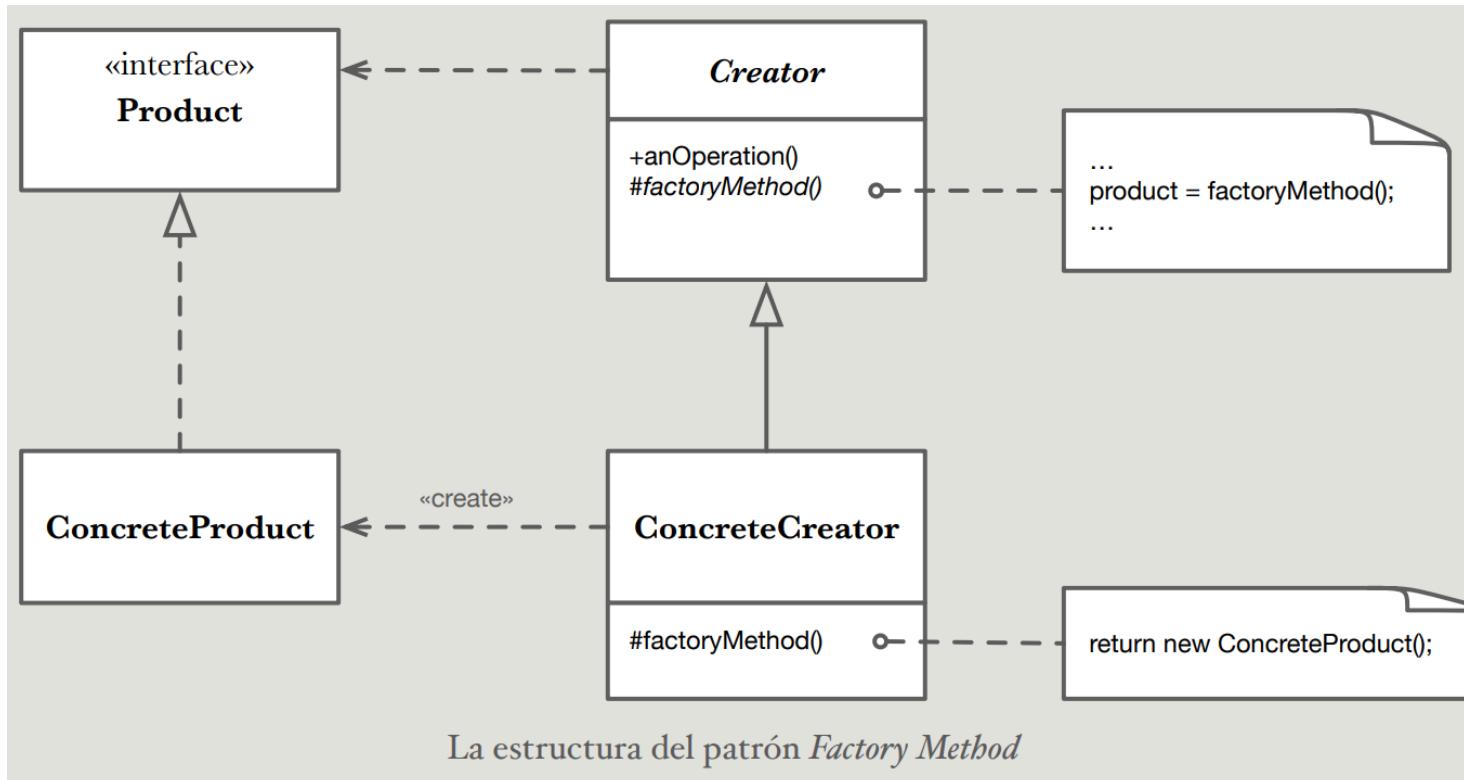
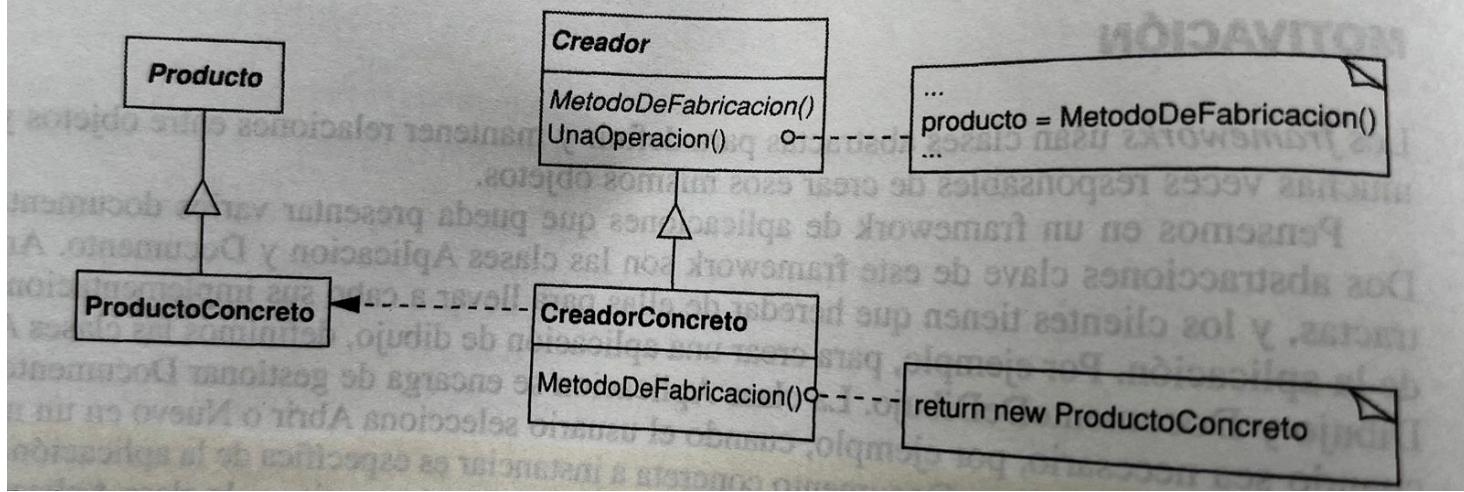


## Aplicabilidad

Úsese el patrón Factory Method cuando:

- Una clase no puede anticipar la clase de objetos que debe crear
- Una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea
- Las clases delegan la responsabilidad en una de entre varias subclases, y queremos localizar qué subclase concreta es en la que se delega

## Estructura



## Participantes

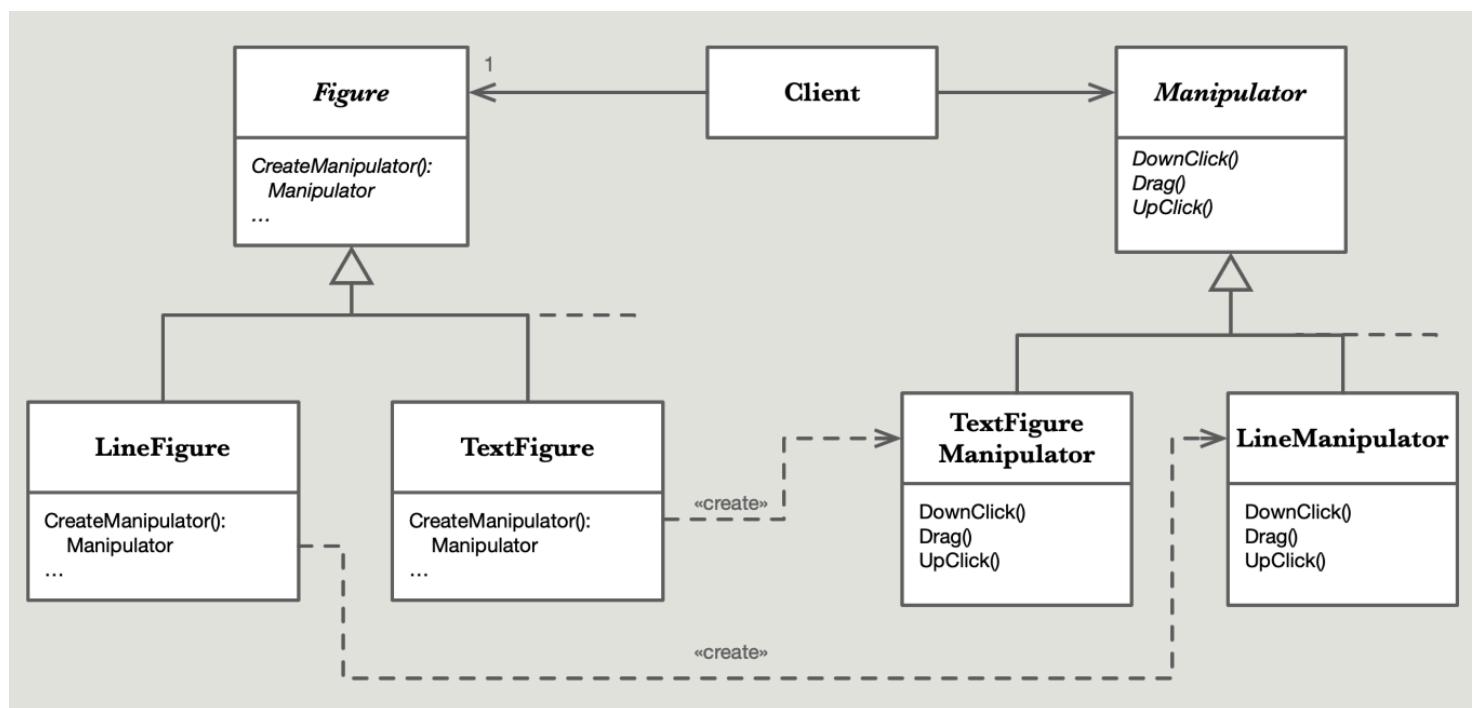
- **Producto** (Documento)
    - Define la interfaz de los objetos que crea el método de fabricación
  - **ProductoConcreto** (MiDocumento)
    - implementa la interfaz del Producto
  - **Creador** (Aplicacion)
    - Declara el método de fabricación, el cual devuelve un objeto de tipo Producto.
    - Puede definir una implementación predeterminada del método de fabricación que devuelva un objeto ProductoConcreto
    - Puede llamar al método de fabricación para crear un objeto Producto
  - **CreadorConcreto** (MiAplicacion)
    - Redefine el método de fabricación para devolver una instancia de un ProductoConcreto

## Colaboraciones

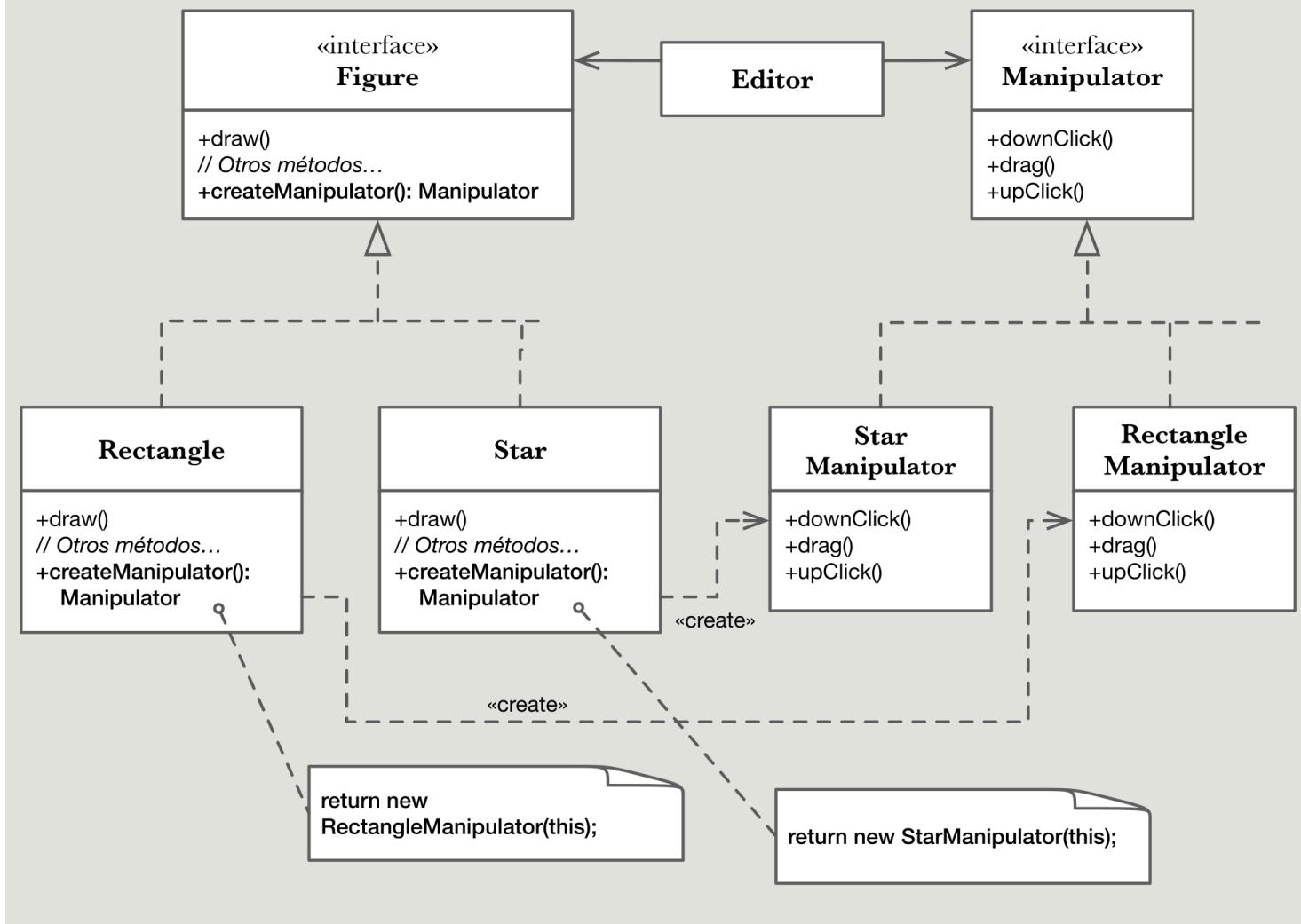
- El Creador se apoya en sus subclases para definir el método de fabricación que devuelve una instancia del ProductoConcreto apropiado

## Consecuencias

- Elimina la necesidad de enlazar clases específicas de la aplicación en el código
  - Sólo maneja la interfaz `Product`
  - Por lo que permite añadir cualquier clase `ConcreteProduct` definida por el usuario
- **Inconveniente:**
  - Tener que crear una subclase de `Creator` en los casos en los que ésta no fuera necesaria de no aplicar el patrón
- En los ejemplos anteriores, el método de fabricación era llamado únicamente por el creador y sus subclases. No tiene por qué ser siempre así: hay veces en que puede ser el cliente quien se encargue de ello
  - Por ejemplo en los casos de **jerarquías paralelas**



Como en nuestro editor:



¿Qué diferencia a las jerarquías paralelas de la versión normal del patrón?

- Que es el cliente quien llama al método de creación.
- Que pasa a ser público, en vez de protegido.

## Abstract Factory

### Propósito

Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas

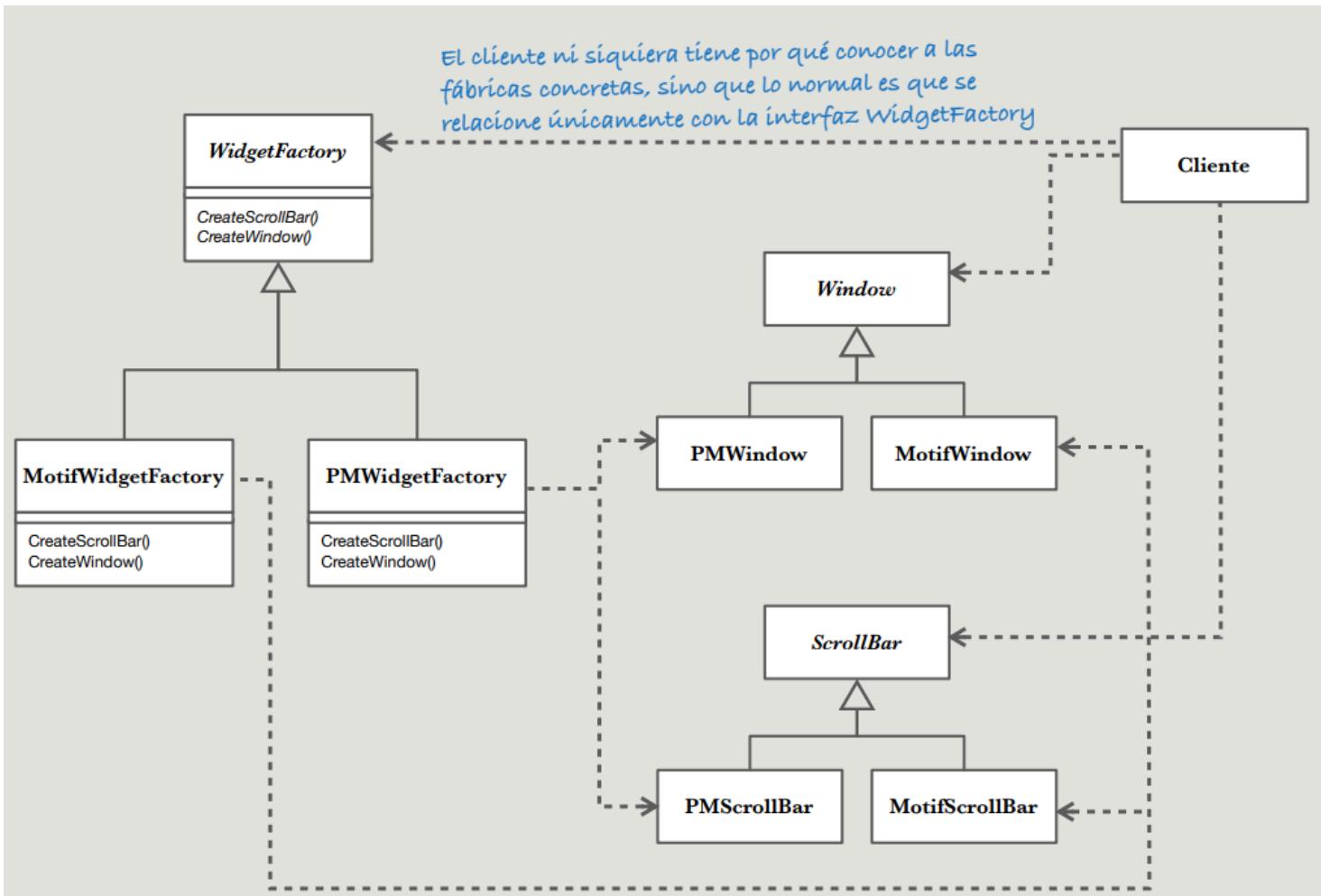
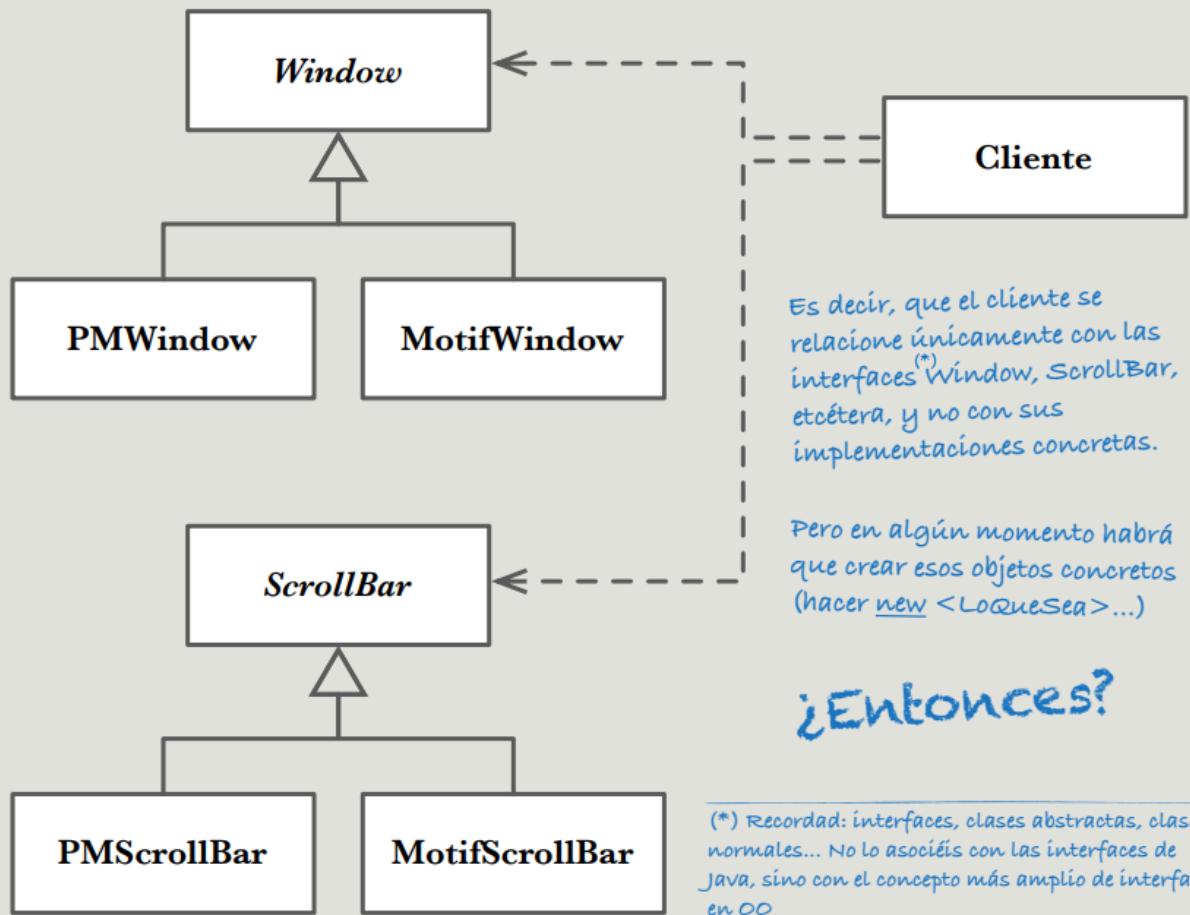
### También conocido como

*Kit*

### Motivación

- Sea una biblioteca gráfica que permita generar interfaces para diferentes entornos de ventanas
  - Cada uno de ellos tendrá una clase distinta para representar una ventana, una barra de desplazamiento, un botón...
- Si queremos que una aplicación se aproveche de ello y sea portable, no podrá crear directamente objetos de esas clases específicas

Queremos esto:

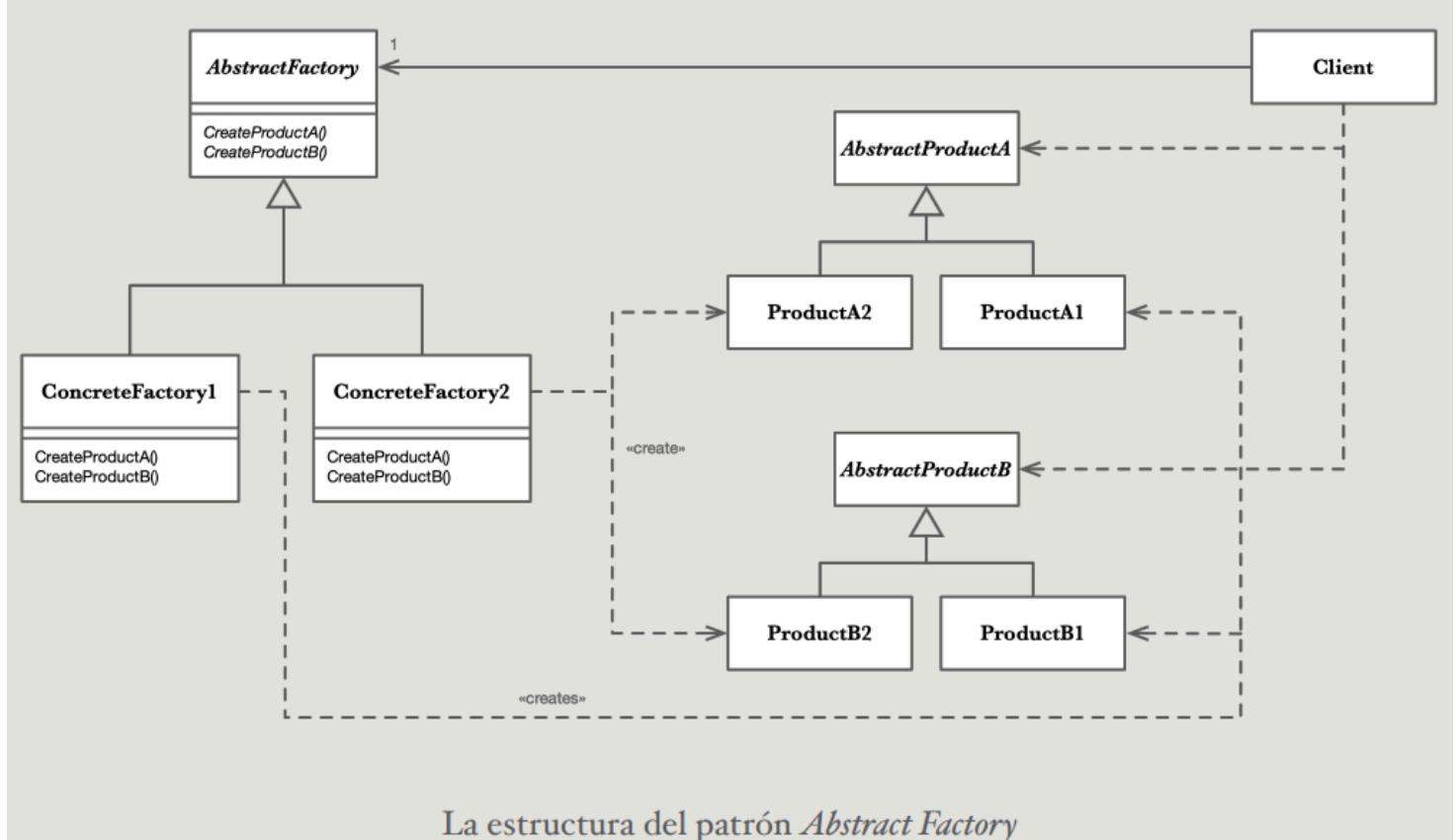


Aplicabilidad

Úsese el patrón Abstract Factory cuando:

- Un sistema debe ser independiente de cómo se crean, componen y representan sus productos
- Un sistema debe ser configurado con una familia de productos de entre varias
- Una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción
- Quiere proporcionar una biblioteca de clases de productos, y sólo quiere revelar sus interfaces, no sus implementaciones

## Estructura



## Participantes

- **FabricaAbstracta** (FabricaDeUtiles)
  - Declara una interfaz para operaciones que crean objetos producto abstractos
- **FabricaConcreta** (FabricaDeUtilesMotif, FabricaDeUtilesPM)
  - Implementa las operaciones para crear objetos producto concretos
- **ProductoAbstracto** (Ventana, BarraDeDesplazamiento)
  - Declara una interfaz para un tipo de objeto producto
- **Producto Concreto** (VentanaMotif, BarraDeDesplazamientoMotif)
  - Define un objeto producto para que sea creado por la fábrica correspondiente
  - Implementa la interfaz ProductoAbstracto
- **Cliente**
  - Sólo usa interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto

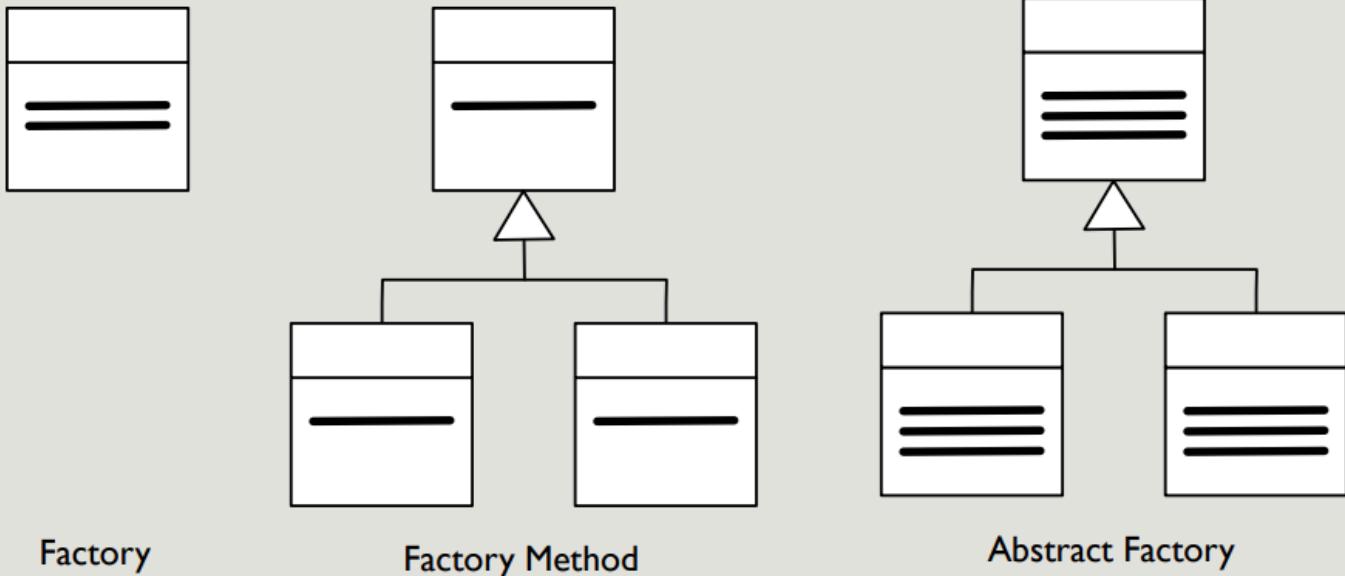
## Consecuencias

- Aísla las clases concretas
  - Los clientes manipulan los productos únicamente a través de sus interfaces abstractas, gracias a que las clases de productos concretos están encapsuladas en cada fábrica concreta, no aparecen en el código
- Permite intercambiar fácilmente familias de productos
  - Basta con cambiar una única clase, en un único sitio: la fábrica concreta
- Promueve la consistencia entre los productos
  - Sólo se pueden usar conjuntamente los objetos de cada familia
- Dificulta añadir nuevos tipos de productos
  - Hay que cambiar la interfaz de la fábrica abstracta y por tanto implementar el nuevo método en todas sus subclases

## Implementación

- Las fábricas pueden ser `Singletons`
- Crear los productos
  - Normalmente, el `Abstract factory` emplea a su vez un `Factory Method` para cada producto
    - Sencillo, pero requiere crear una subclase por cada familia
  - Otra posibilidad es emplear el patrón `Prototype`: una única clase para la fábrica abstracta y tener distintos objetos de la misma configurados mediante prototipos
- Fábricas extensibles
  - Pretenden resolver el problema de poder añadir nuevos tipos de productos
  - Un único método de creación especificando el tipo de producto como parámetro
  - Más flexible, pero menos seguro (se pierde la comprobación estática de tipos)

## Factory Method vs Abstract Factory



Las líneas en negrita representan métodos que crean objetos. Esta figura, debida a Kerievsky (2005), ilustra de ese modo, muy esquemáticamente, las diferencias más significativas entre una simple clase de creación y los patrones de diseño *Factory Method* y *Abstract Factory*.

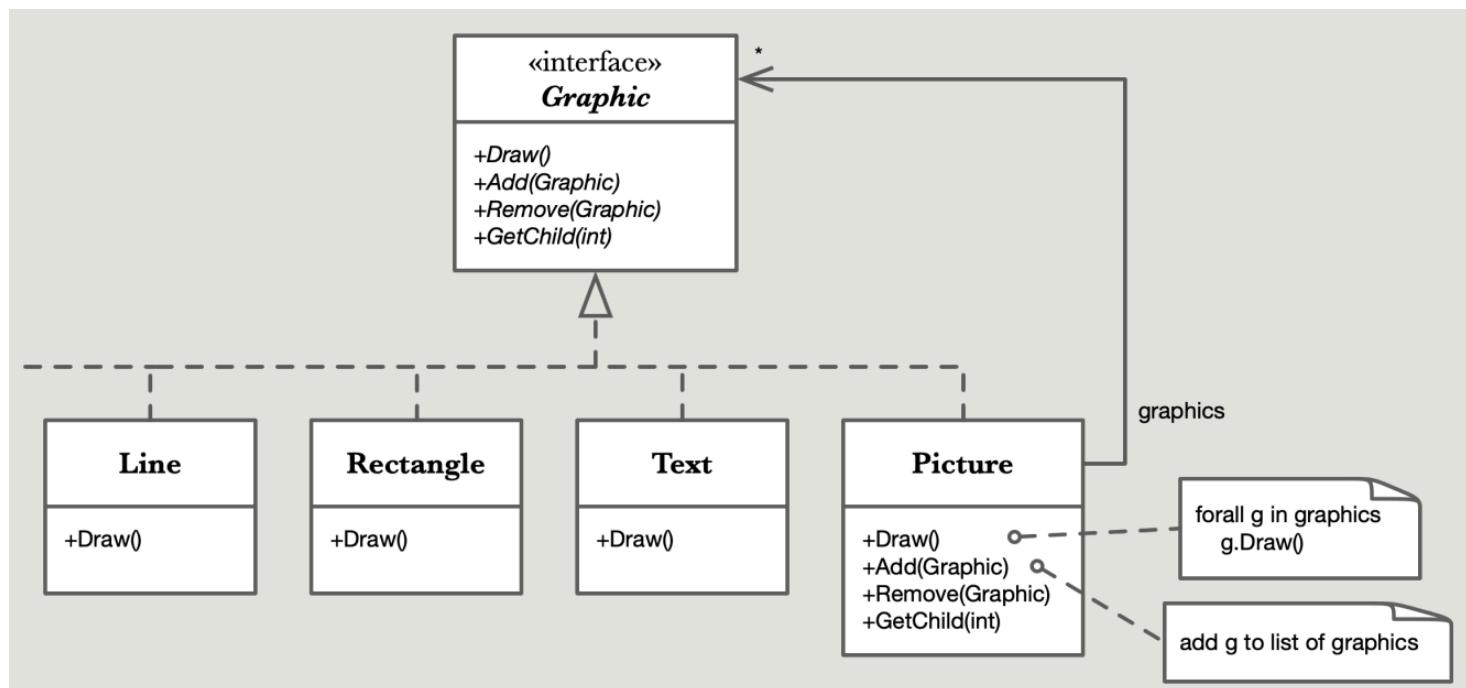
## Composite

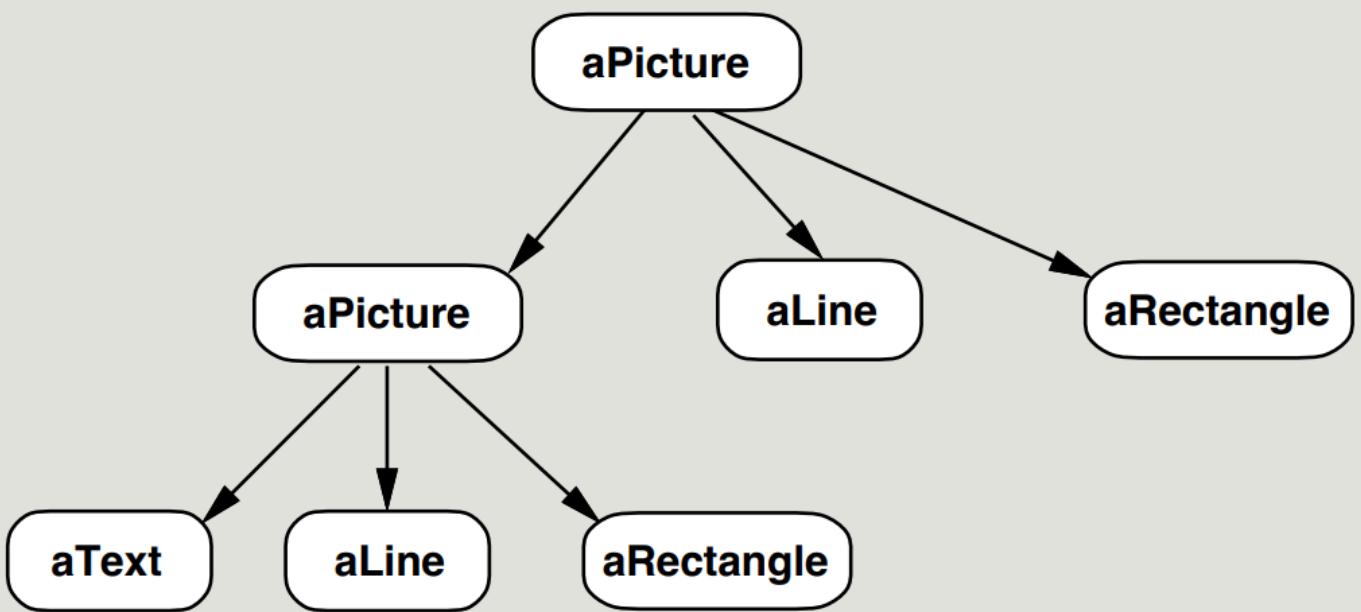
### Propósito

Compone objetos en estructuras de árbol para representar jerarquías de parte/todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos

### Motivación

- Un editor de dibujo permite realizar dibujos compuestos de elementos simples (líneas, rectángulos...) u otros dibujos
  - ¿Cómo evitamos que los clientes tengan que distinguir entre unos y otros?



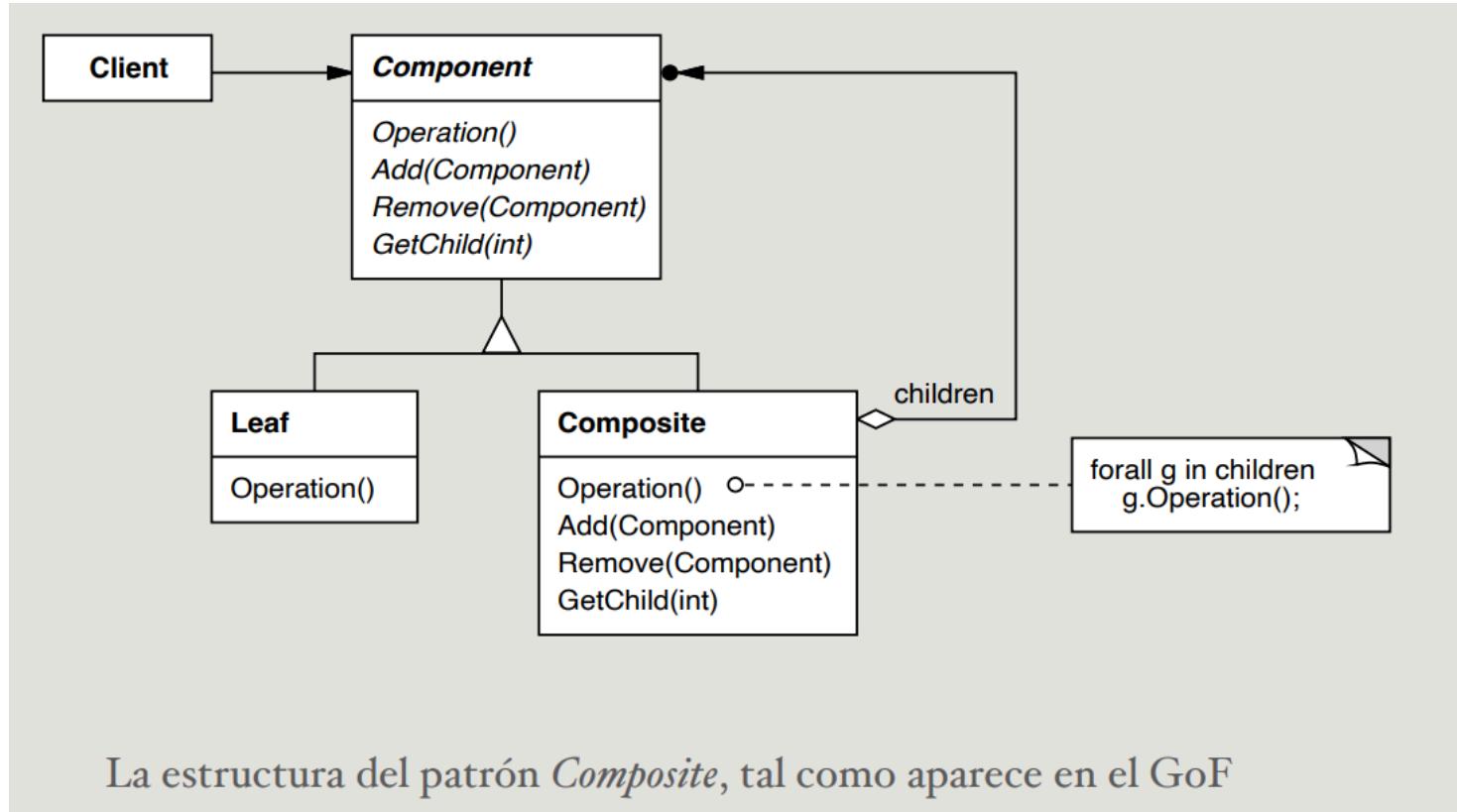


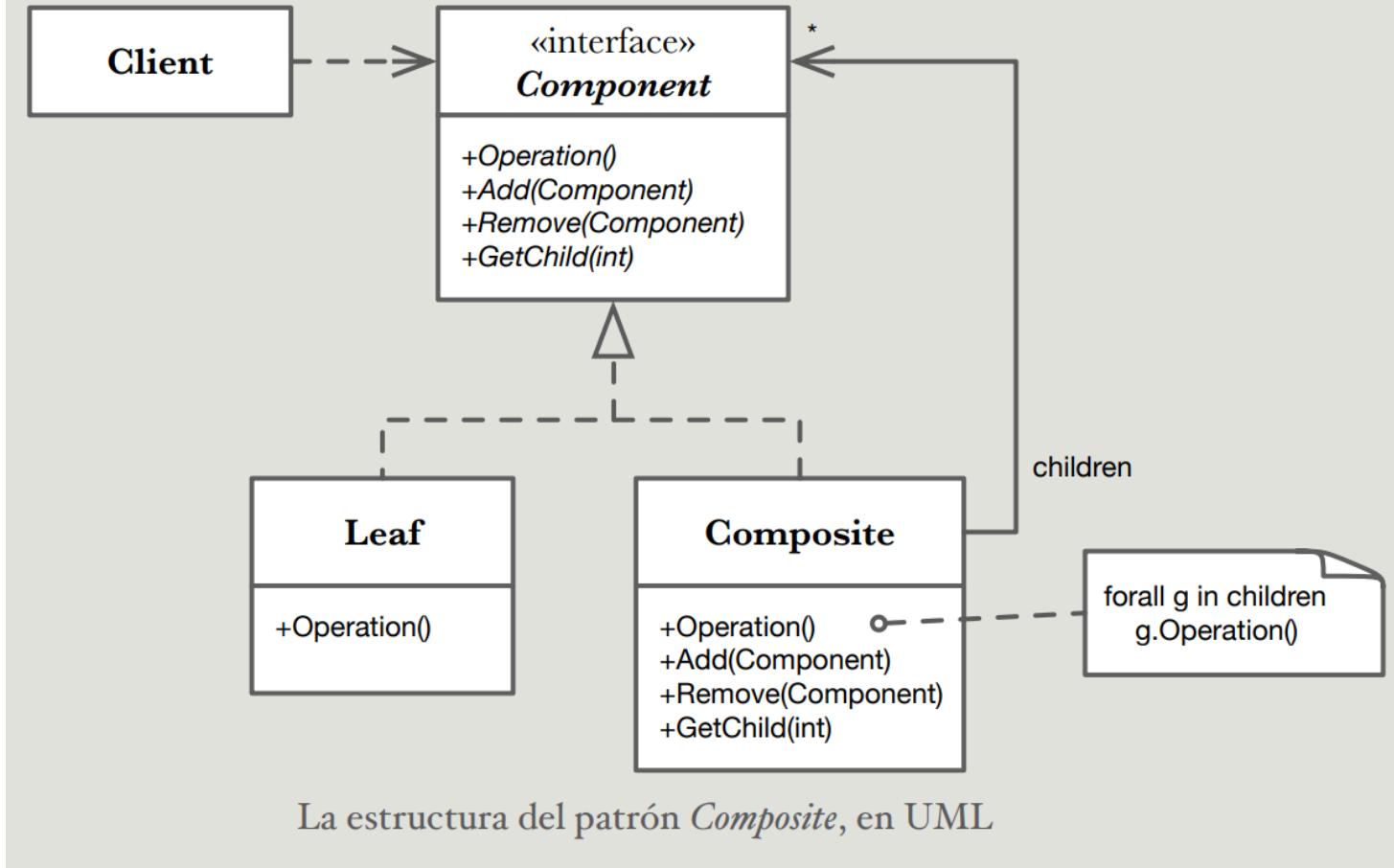
## Aplicabilidad

Usar el patrón Composite cuando:

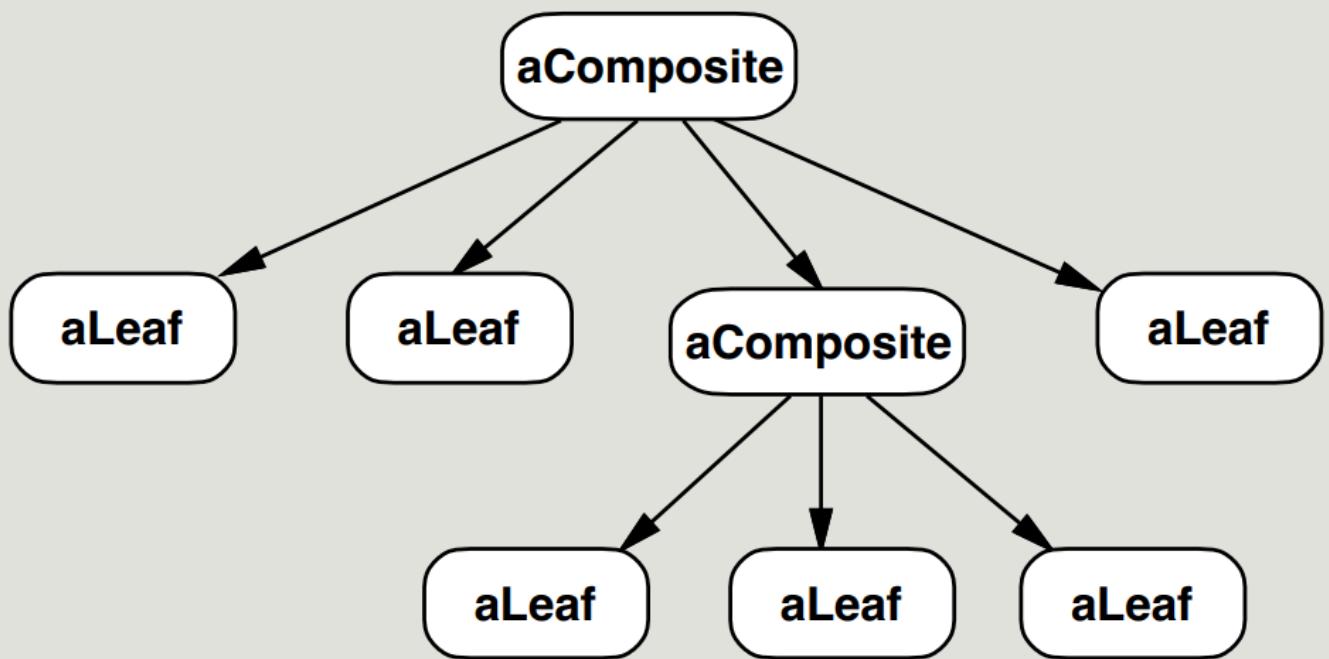
- Quiera representar jerarquías de objetos parte-todo
- Quiera que los clientes traten por igual los objetos y los objetos individuales

## Estructura





- Una estructura de objetos típica del patrón *Composite* se parece a ésta:



## Participantes

- Componente (Grafico)
  - Declara la interfaz común

- Implementa el comportamiento predeterminado de la interfaz que es común a todas las clases
- Declara operaciones para acceder a sus hijos (aunque no necesariamente, pues pueden declararse en el compuesto)
- (opcional) Define una interfaz para acceder al padre
- **Hoja** (Rectangulo, Linea, Texto...)
  - Representa objetos hoja en la composición
  - Una hoja no tiene hijos
- **Compuesto** (Dibujo)
  - Almacena sus componentes hijos
  - Implementa las operaciones relacionadas con los hijos
- **Cliente**
  - Manipula los objetos de la composición a través de la interfaz de Componente

## Consecuencias

- Permite jerarquías de objetos tan complejas como se quiera
  - Allá donde el cliente espere un objeto primitivo, podrá recibir un compuesto y no se dará cuenta
- Simplifica el cliente
  - Al eliminar el código para distinguir entre unos y otros
- Se pueden añadir nuevos componentes fácilmente
- Como desventaja, podría hacer el diseño demasiado general

## Implementación

- Referencias explícitas al parent
- Maximizar la interfaz de *Component*
  - El componente puede proporcionar implementaciones predeterminadas que luego las clases hoja y las compuestas redefinan
  - Problema: puede haber operaciones que tengan sentido en unas pero no en otras
    - Ejemplo: las operaciones de gestión de los hijos
- Orden de los hijos
  - Si es significativo, hay que diseñar las interfaces de acceso y gestión de los hijos cuidadosamente

## State

### Propósito

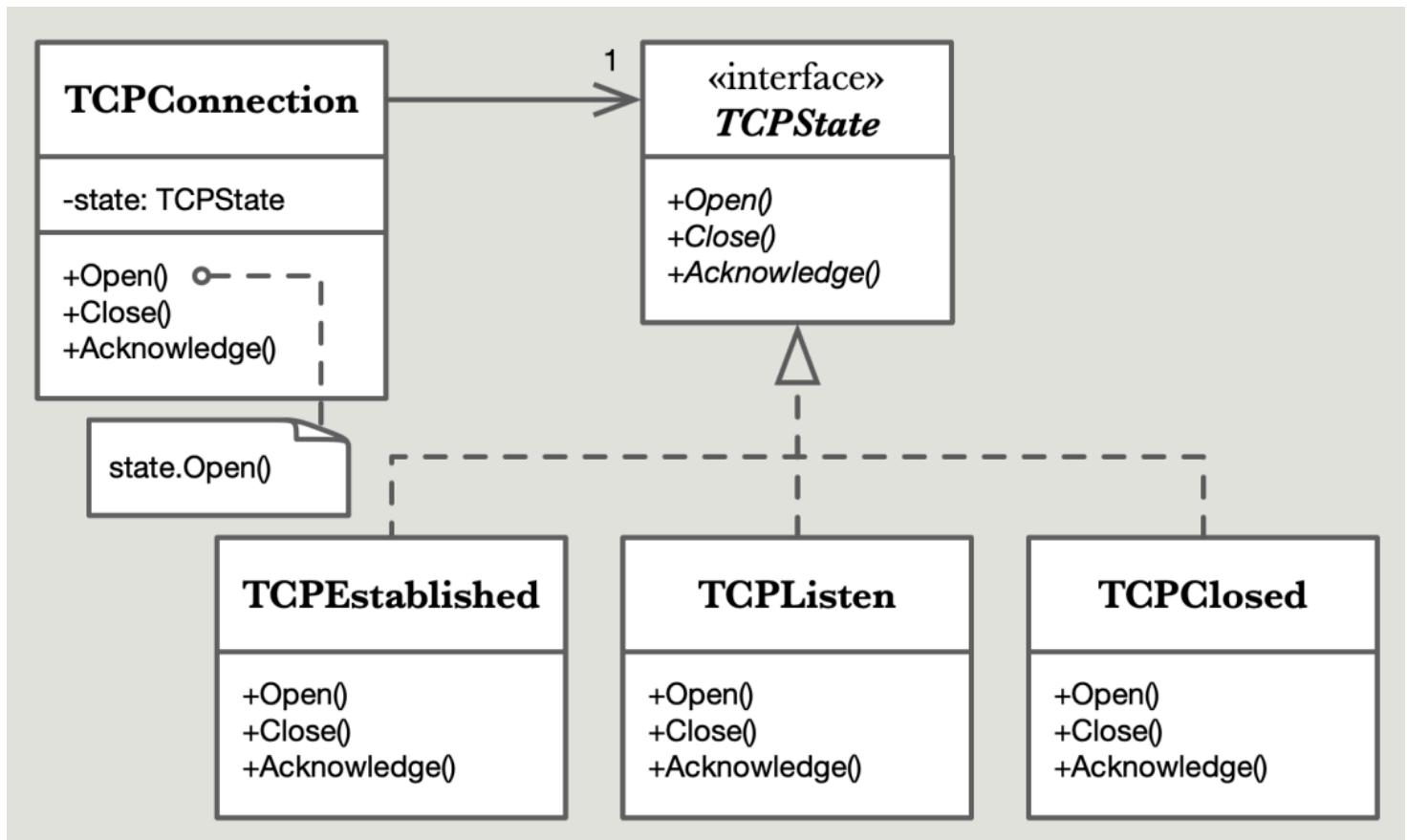
Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto

### También conocido como

*Objects for States* (Estados como Objetos)

# Motivación

- Una conexión de red es representada en una implementación TCP como TCPConnection
- La conexión puede estar en uno de los siguientes estados:
  - Abierta
  - Escuchando
  - Cerrada

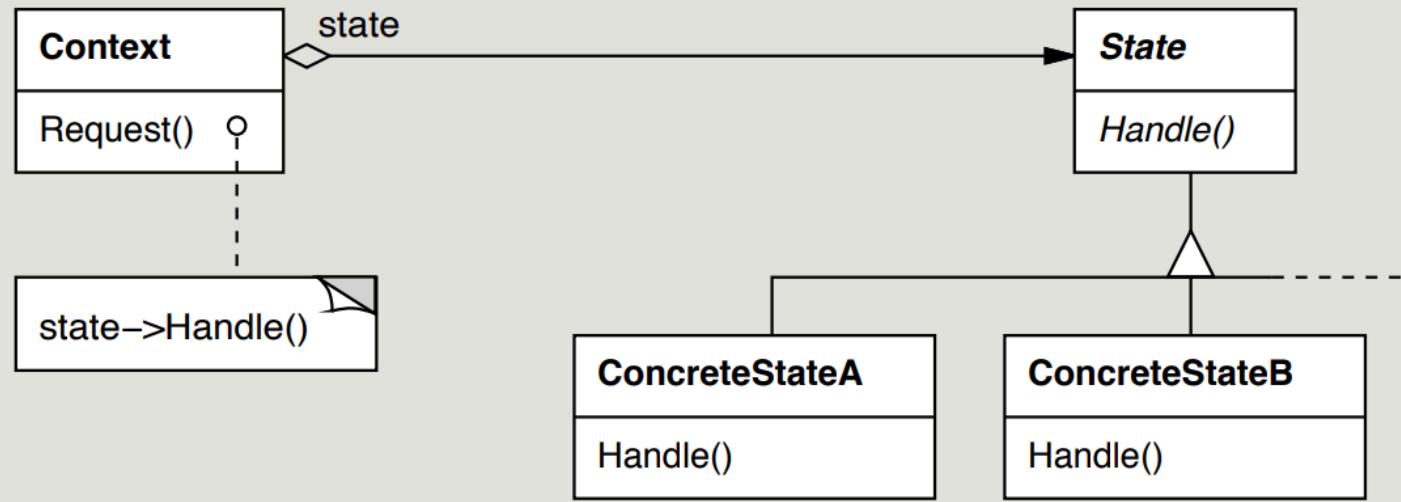


## Aplicabilidad

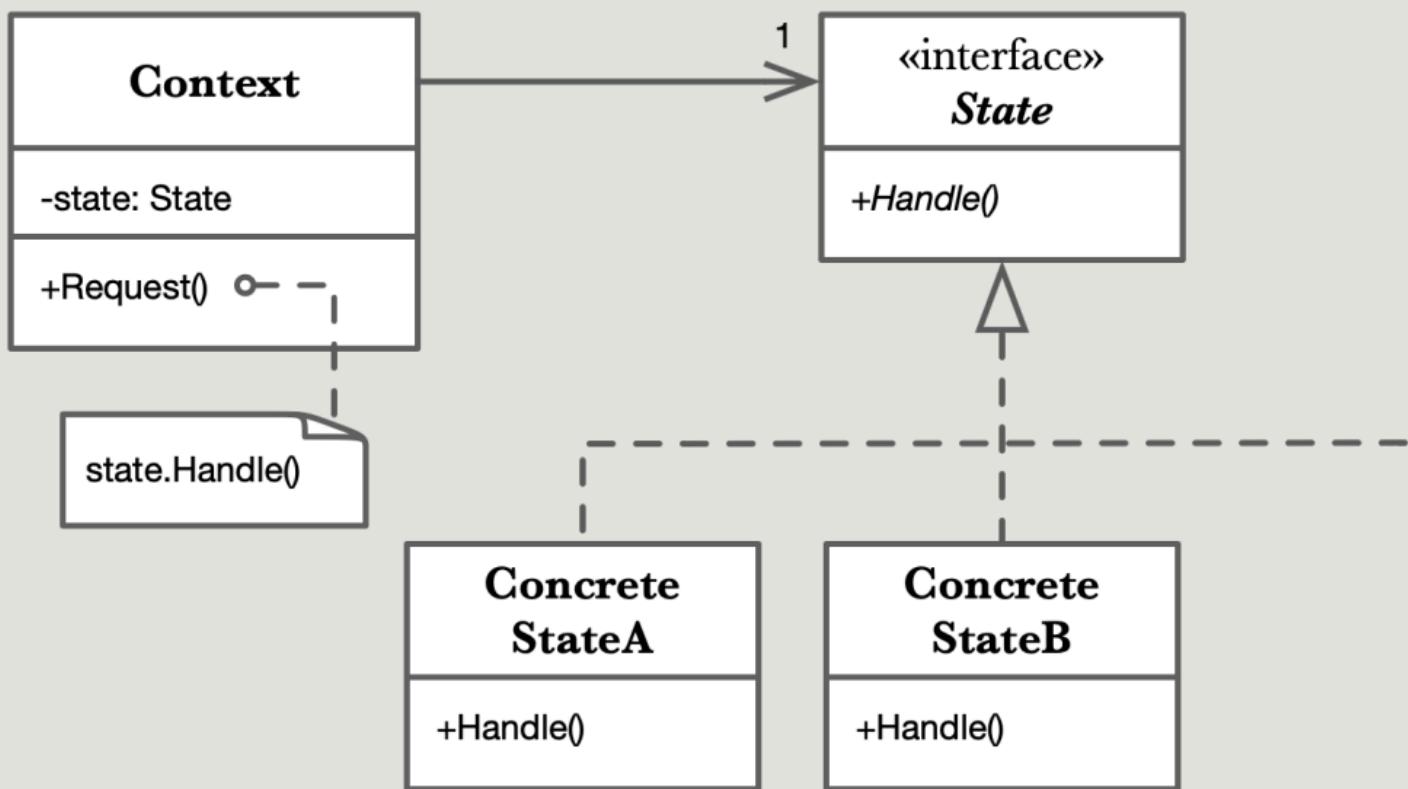
Úsese el patrón State en cualquiera de los siguientes casos:

- El comportamiento de un objeto depende de su estado, y éste puede cambiar en tiempo de ejecución
- Las operaciones tienen largas sentencias condicionales anidadas que tratan con los estados
  - Siendo el estado normalmente una constante
  - Muy frecuentemente, son varias las operaciones en las que se repite esa misma estructura condicional
  - Este patrón mueve cada rama de la lógica condicional a una clase aparte
    - Lo que nos permite tratar al estado del objeto como un objeto de pleno derecho, que puede variar independientemente de otros objetos

## Estructura



La estructura del patrón *State*, tal como aparece en el GoF



La estructura del patrón *State*, en UML

## Participantes

- **Contexto** (ConexionTCP)
  - Define la interfaz de interés para los clientes
  - Mantiene una instancia de una subclase de EstadoConcreto que define el estado actual
- **Estado** (EstadoTCP)
  - Define una interfaz para encapsular el comportamiento asociado con el estado del Contexto

- **subclases del EstadoConcreto** (TCPEstablecida, TCPEscuchando, TCPCerrarada)
  - Cada subclase implementa un comportamiento asociado con un estado del Contexto

## Colaboraciones

- El Contexto delega las operaciones dependientes del estado al objeto que representa el estado actual
- El contexto podría pasarse a sí mismo como parámetro
  - Para que el estado acceda al contexto si es necesario
- una vez que el Contexto es inicializado en un determinado estado, los clientes no necesitan tratar directamente con los estados
- O bien el Contexto o bien los EstadosConcretos deciden cuándo se pasa de un estado a otro

## Consecuencias

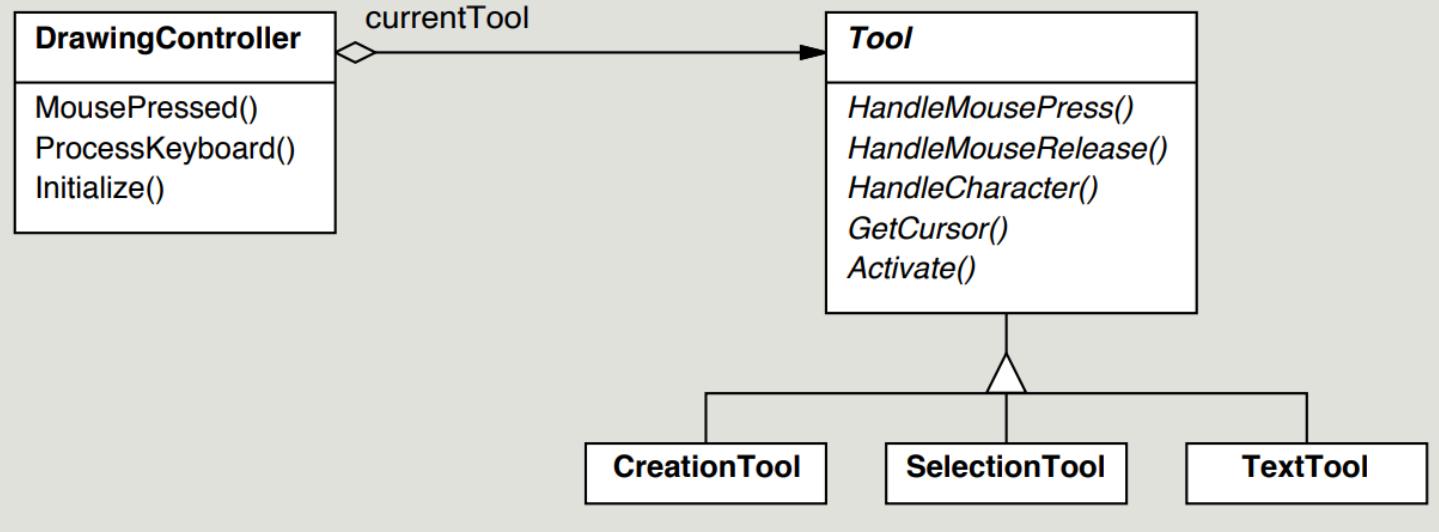
- Localiza el comportamiento específico del estado y lo aísla en un objeto
  - Se pueden añadir nuevos estados y transiciones fácilmente simplemente definiendo nuevas subclases de `State`
- Hace explícitas las transiciones entre estados

## Implementación

- ¿Quién define las transiciones entre estados?
  - Si el criterio es siempre el mismo, puede ser el propio contexto
  - Normalmente es más flexible y apropiado que sean las subclases de los estados
    - Esto requiere añadir una operación al contexto para cambiar su estado de forma explícita
    - **Ventaja:** flexibilidad
    - **Inconveniente:** dependencias de implementación (acoplamiento) entre las subclases que representan los estados concretos
- Uso de la herencia dinámica
  - Este patrón no sería necesario en lenguajes que permiten cambiar la clase de un objeto en tiempo de ejecución o que proveen mecanismos para delegar peticiones automáticamente en otros objetos
- Acciones de entrada/salida
  - Cuando hay que realizar alguna acción al entrar o salir de un estado puede venir bien añadir un método `entry`, `exit` o ambos a la interfaz `State`

## Posibles usos

# ● Editor gráfico HotDraw



## State vs Strategy

- **Strategy**: es seleccionada por un agente externo o por el contexto. Una estrategia tiende a tener un método único de "inicio" que llama a todos los demás. Hay mucha cohesión entre los métodos de un Strategy
- **State**: un State generalmente selecciona el siguiente estado de su contexto. Un estado tiende a tener muchos métodos no relacionados, por lo que hay poca cohesión entre los métodos de un State
  - Los métodos del State suelen llamarse igual y tener la misma firma que las que tenían dicho subconjunto de operaciones en el contexto

## Template Method

### Propósito

Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

### Motivación

- Sea un framework de aplicaciones que proporciona unas clases `Application` y `Document`
  - La primera es la responsable de abrir los documentos almacenados en ficheros
  - La segunda representa la información en sí del documento, una vez que ya ha sido leído del fichero
- Las aplicaciones construidas con el framework redefinirán ambas clases para adaptarlas a sus necesidades concretas
- ¿Cómo implementar, de manera genérica, el método `openDocument` de la clase `Application`?

```

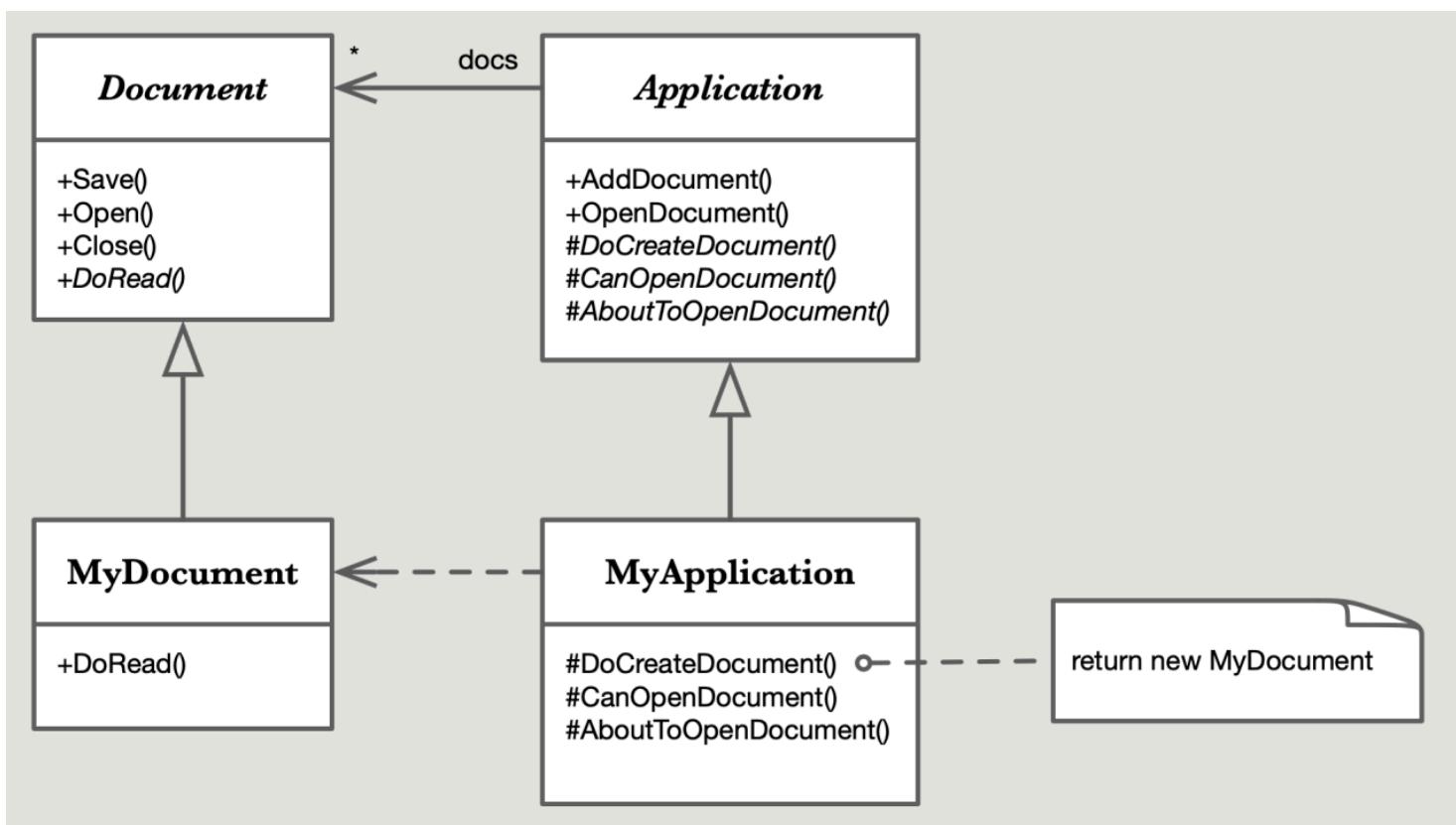
void Application::OpenDocument (const char* name)
{
    if (!CanOpenDocument(name))
        return;

    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open(); doc->DoRead();
    }
}

```

- El método abstracto anterior define todos los pasos necesarios para abrir un documento
  - Comprueba si se puede abrir, crea un objeto `Document`, lo añade al conjunto de documentos y finalmente lo lee
  - Lo hace en términos de operaciones abstractas

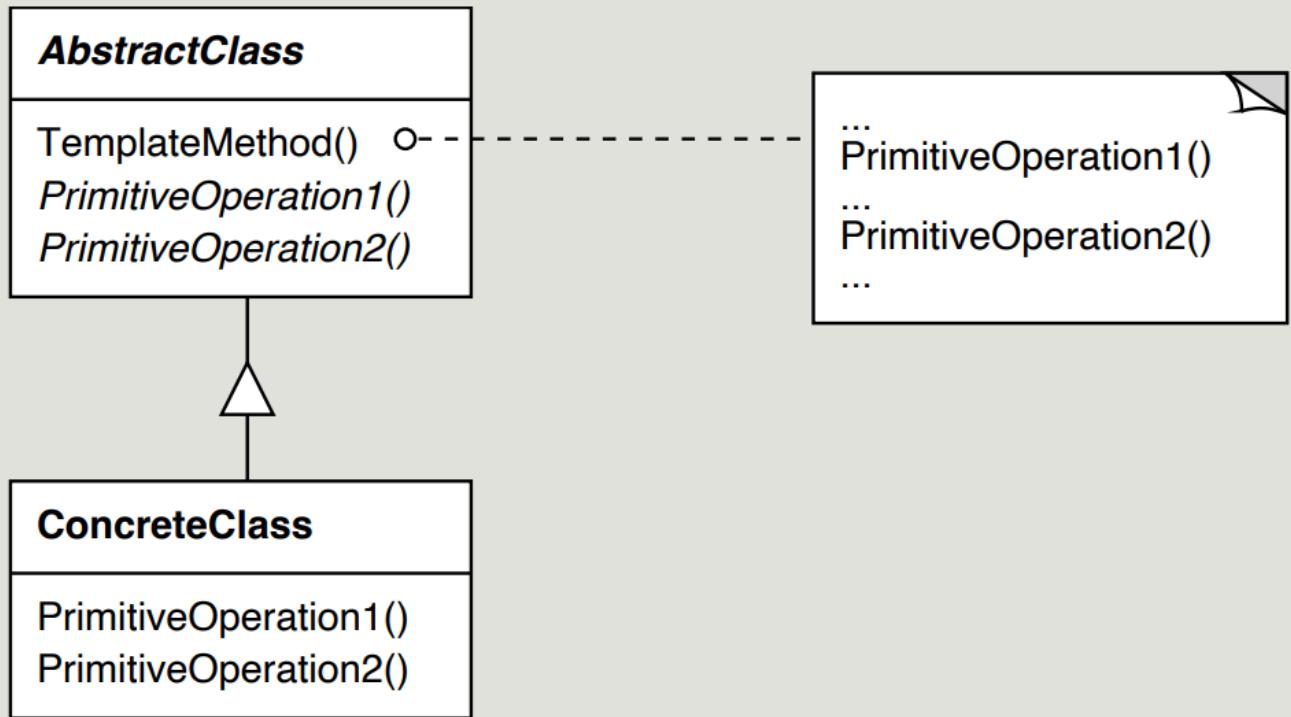


## Aplicabilidad

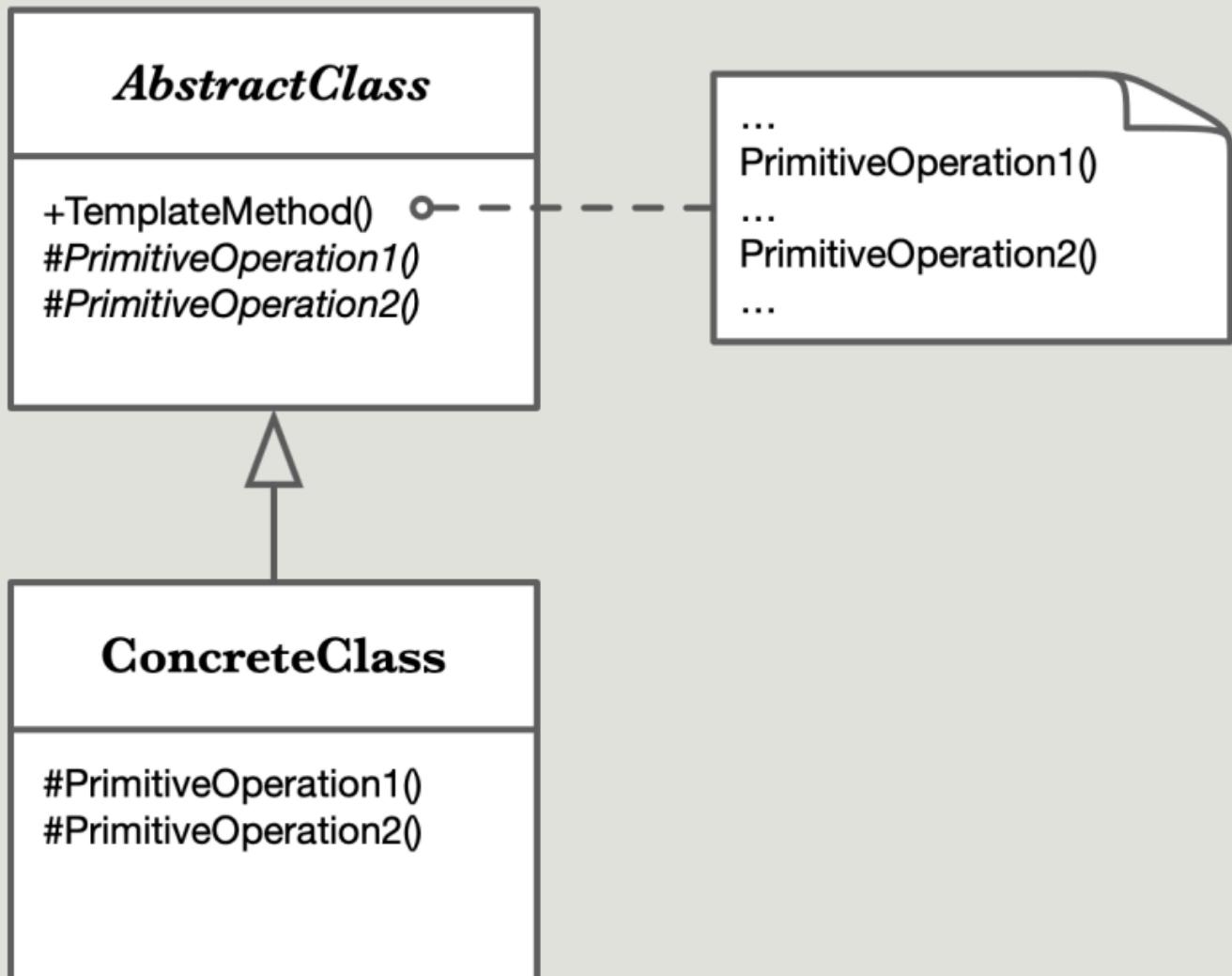
El patrón Template Method debería usarse:

- Para implementar las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento que puede variar
- Como motivo de factorizar código, cuando movemos cierto código a una clase base común para evitar código duplicado
- Para controlar el modo en que las subclases extienden la clase base
  - Dejando que sea sólo a través de unos métodos plantilla dados

## Estructura



La estructura del patrón *Template Method*, tal como aparece en el GoF



La estructura del patrón *Template Method*, en UML

## Participantes

- **ClaseAbstracta** (Application)
  - Define las operaciones primitivas abstractas que redefinirán las subclases
  - Implementa un método de plantilla con el esqueleto del algoritmo
- **ClaseConcreta** (MiApplication)
  - Implementa las operaciones primitivas para realizar los pasos del algoritmo específicos de las subclases

## Consecuencias

- Los métodos de plantilla son una técnica fundamental para la reutilización de código
- Inversión de control
  - Es la clase padre quien llama a operaciones en los hijos
- Los métodos de plantilla pueden llamar a los siguientes tipos de operaciones:
  - Operaciones concretas de otras clases
  - Operaciones concretas en la propia clase base abstracta
    - Proporcionan comportamiento predeterminado que las subclases pueden redefinir si es necesario

- Operaciones primitivas (es decir, abstractas)
  - Que las subclases deberán implementar
- Métodos de fabricación
- Operaciones de enganche (*hook*)
  - Normalmente protegidas, tienen una implementación vacía en la clase abstracta, que las subclases podrán redefinir. Son como operaciones opcionales

Operaciones de enganche:

- Una subclase puede extender una operación de la clase padre llamando explícitamente a dicha operación:

```
void operation()
{
    super.operation();
    // DerivedClass extended behavior
}
```

- Problema

- Que nos podemos olvidar de llamar a esa operación del padre

- Podemos hacerlo a la inversa: un método de plantilla en la clase padre que llame a una operación en la subclase:

```
void operation()
{
    // ParentClass behavior
    hookOperation();
}

void hookOperation()
{
}
```

## Implementación

- Hacer las operaciones primitivas llamadas por el método de plantilla `protected`

- Y, a aquéllas que deban ser obligatoriamente redefinidas, abstractas
- Minimizar el número de operaciones primitivas abstractas
- Convenios de nombrado
  - Es conveniente identificar las operaciones que deben ser redefinidas anteponiendo un prefijo al nombre (ej: `Do-`)

## Patrones relacionados

- *Factory Method*
  - Muchas veces los métodos de fabricación son llamados desde métodos plantilla
- *Strategy*
  - El *Template Method* usa la herencia para modificar parte de un algoritmo; *Strategy* usa delegación para cambiar el algoritmo entero

## Adapter

### Propósito

Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes. Permite que cooperen clases que de otra forma no podrían tener interfaces compatibles

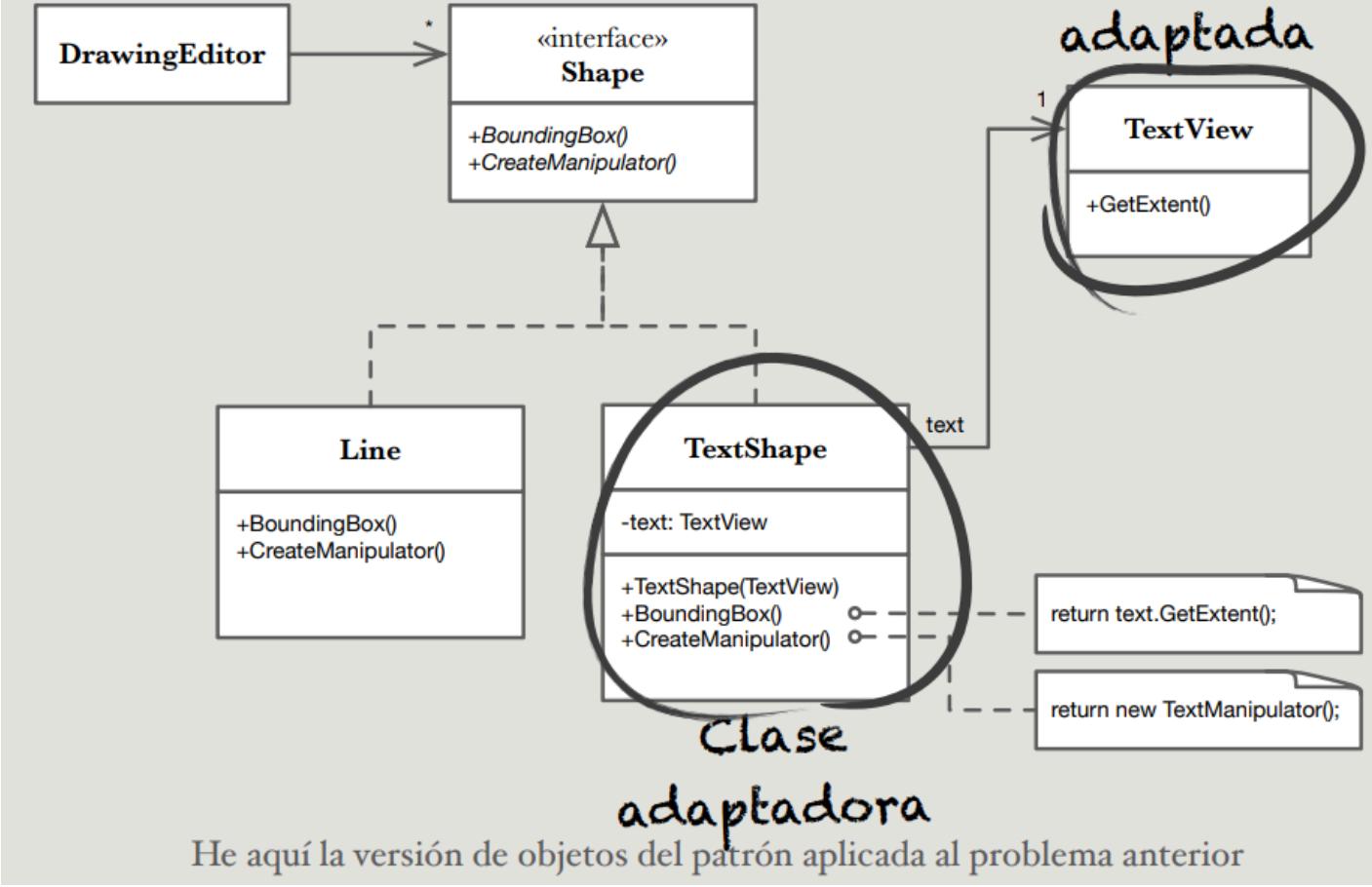
### También conocido como

*Wrapper* (Envoltorio)

### Motivación

- Supongamos que estamos haciendo un editor de dibujo
  - La abstracción fundamental es el objeto gráfico (`Shape`), que puede dibujarse a sí mismo
  - Define una subclase por cada tipo de objeto gráfico: `LineShape`, `PolygonShape` ...
- Supongamos que, para implementar una subclase `TextShape` (bastante más compleja que las anteriores) queremos echar mano de una clase `TextView` que nos proporciona la biblioteca gráfica
- La interfaz de `TextView` no tendrá nada que ver con la de `Shape`
  - Le faltarán algunas operaciones, otras las tendrá con otro nombre, o bien recibirán parámetros de otro tipo...
- Crearemos una clase `TextShape` que adapte la interfaz `TextView` a la de `Shape`
- Dos opciones:
  - Heredando la interfaz de `Shape` y la implementación de `TextView` (versión de dos clases)
  - Mediante composición de objetos, haciendo que `TextShape` delegue en una instancia de `TextView` (versión de objetos)

# Solución



## Aplicabilidad

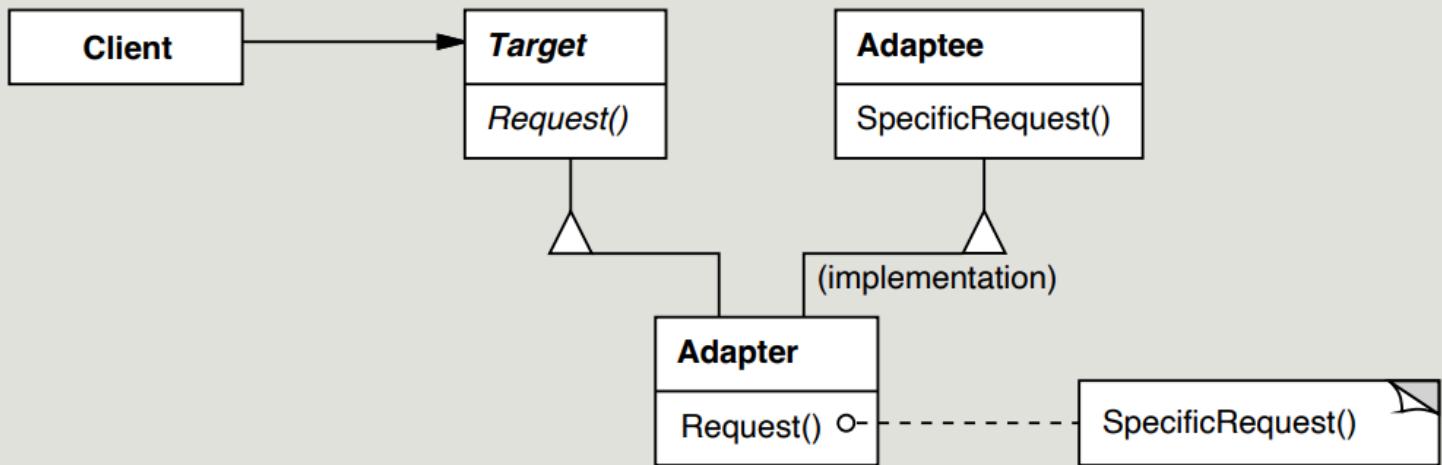
Debería usarse el patrón Adapter cuando:

- Se quiere usar una clase existente y su interfaz no concuerda con la que necesita
- Se quiere crear una clase reutilizable que coopere con clases con las que no está relacionada (que no tendrán interfaces compatibles)
- (Sólo la versión de objetos) Necesitamos usar varias subclases existentes pero sin tener que adaptar su interfaz creando una nueva subclase de cada una

## Estructura

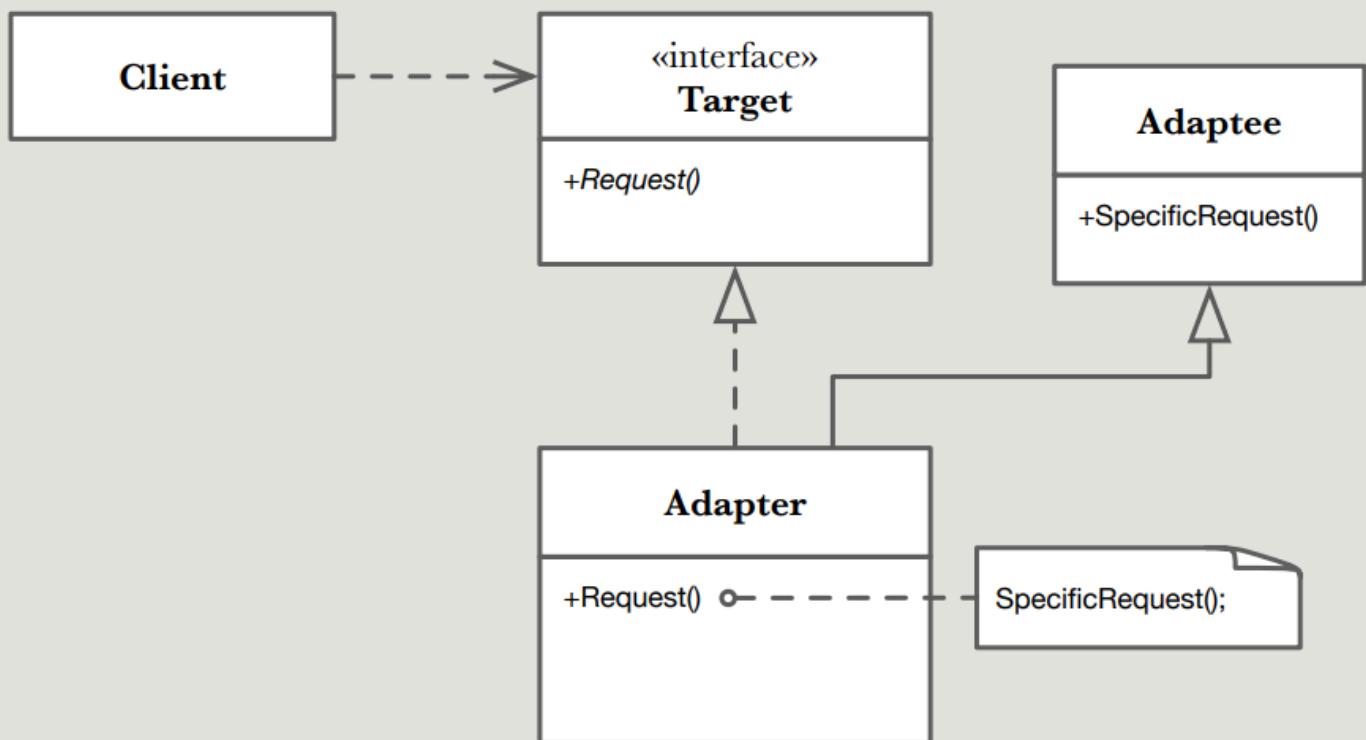
# (versión de clases)

- Un adaptador de clases usa herencia múltiple:



La versión de clases del patrón *Adapter*, tal como aparece en el GoF

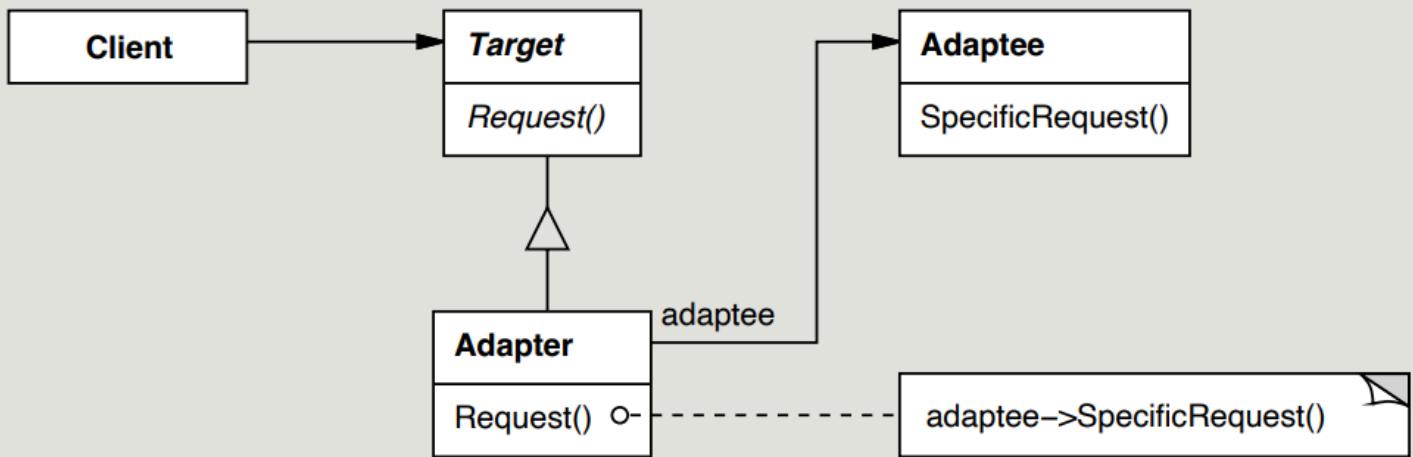
## Estructura (versión de clases)



La versión de clases del patrón *Adapter*, en UML

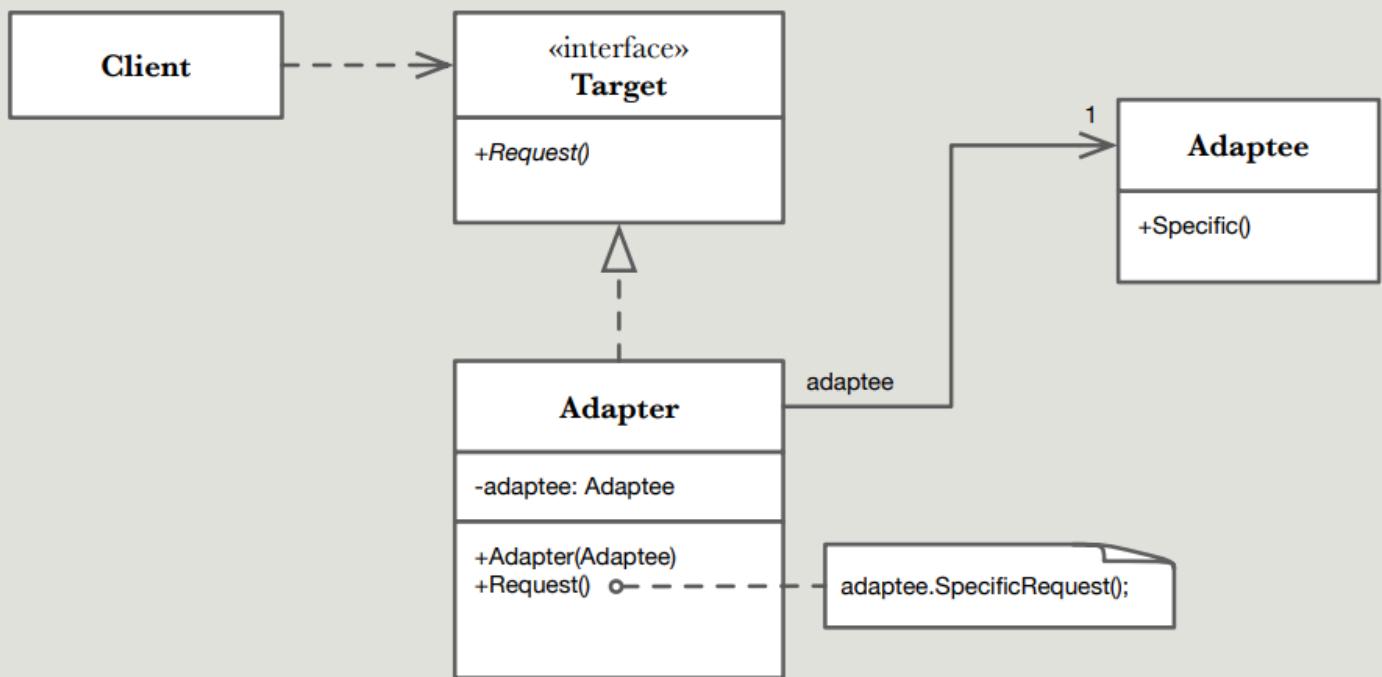
# Estructura (versión de objetos)

- Un adaptador de objetos se basa en composición de objetos:



La estructura del patrón *Adapter*, tal como aparece en el GoF

# Estructura (versión de objetos)



Un adaptador de objetos se basa en composición de objetos

## Participantes

- **Objetivo (Forma)**
  - Define la interfaz específica del dominio que usa el Cliente
- **Cliente (EditorDeDibujo)**
  - Colabora con objetos que se ajustan a la interfaz Objetivo
- **Adaptable (VistaTexto)**
  - Define una interfaz existente que necesita ser adaptada
- **Adaptador (FormaTexto)**
  - Adapta la interfaz de Adaptable a la interfaz Objetivo

## Consecuencias

Las versiones de clases y de objetos de este patrón tienen diferentes ventajas e inconvenientes:

- **Un adaptador de clases:**
  - Adapta una clase concreta a una interfaz (no se puede usar cuando queremos adaptar una clase y todas sus subclases)
  - Permite que el adaptador redefina parte del comportamiento de la clase adaptada (es una subclase de aquélla)
  - Introduce un solo objeto adicional, sin indirección

- **Un adaptador de objetos**
  - Permite que un único adaptador funcione no sólo con un objeto de la clase adaptada, sino de cualquiera de sus subclases
  - Permite adaptar objetos existentes
  - No es del tipo de objeto adaptado

## Command

### Propósito

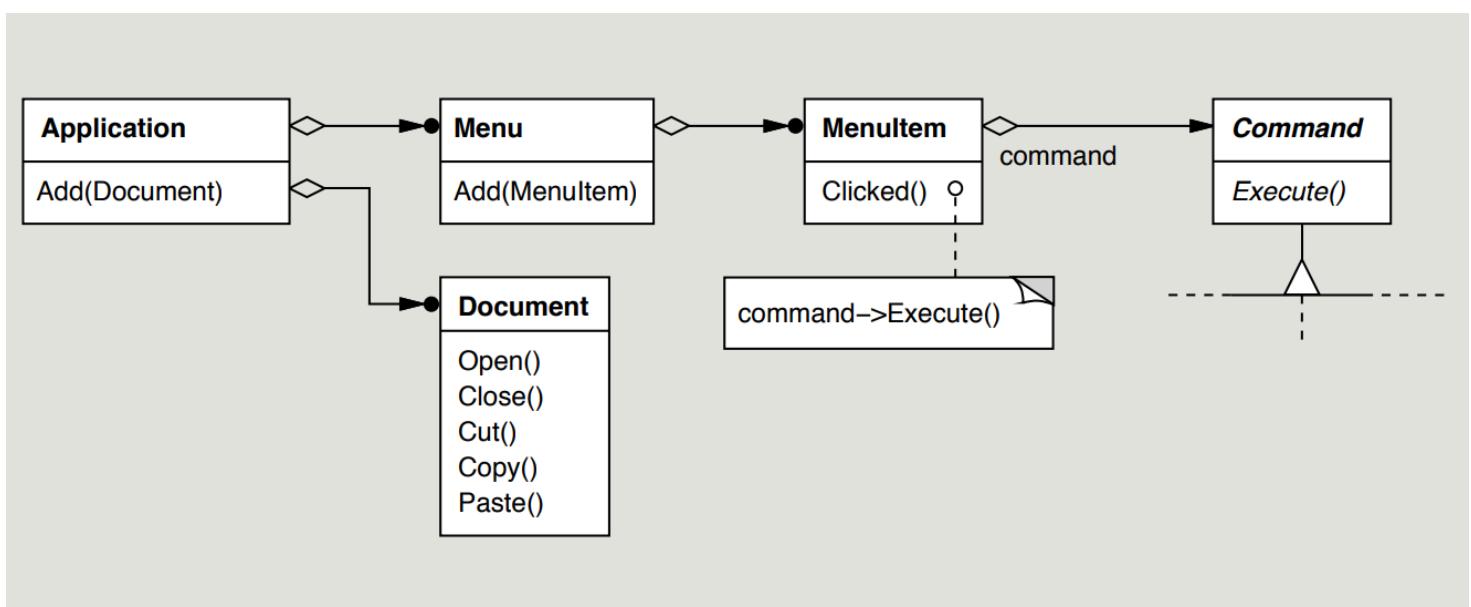
Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones, y poder deshacer las operaciones

### También conocido como

*Action* (Acción), *Transaction* (Transacción)

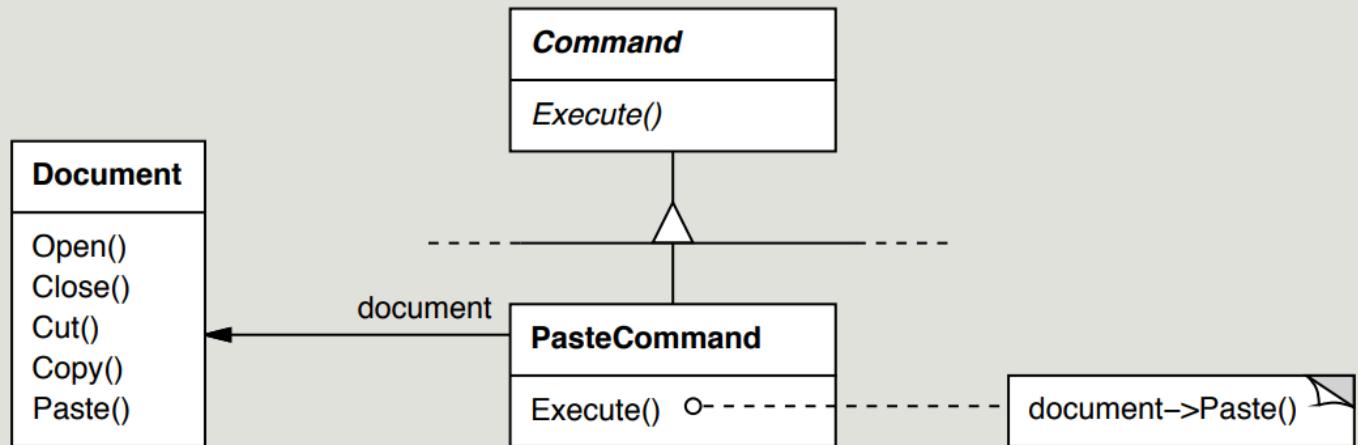
### Motivación

- Una biblioteca de clases para interfaces de usuario tendrá objetos como botones y elementos de menú responsables de realizar alguna operación en respuesta a una entrada del usuario
- La biblioteca no puede implementar dichas operaciones directamente en el botón o el menú
  - Sólo las aplicaciones que usan la biblioteca saben qué hay que hacer y a qué operaciones de otros objetos hay que llamar

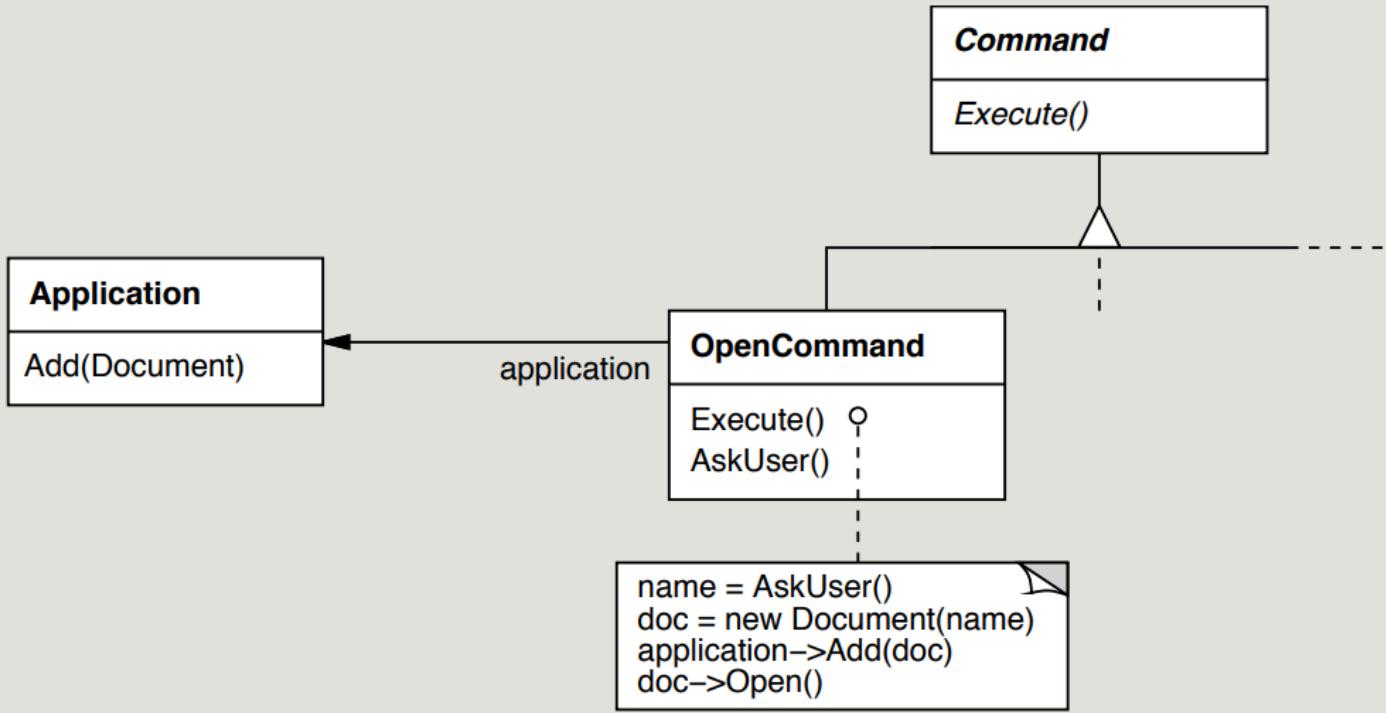


- La clave de este patrón es una interfaz `Command` que define una operación `execute`
- Son las subclases concretas quienes implementan la operación y especifican el receptor de la orden
- Podemos configurar cada elemento del menú, `MenuItem`, con un objeto `Command`
- Los elementos del menú no saben qué objeto concreto están usando (simplemente llaman a su método `execute`)

# Por ejemplo, «pegar»



## ○ «abrir documento»



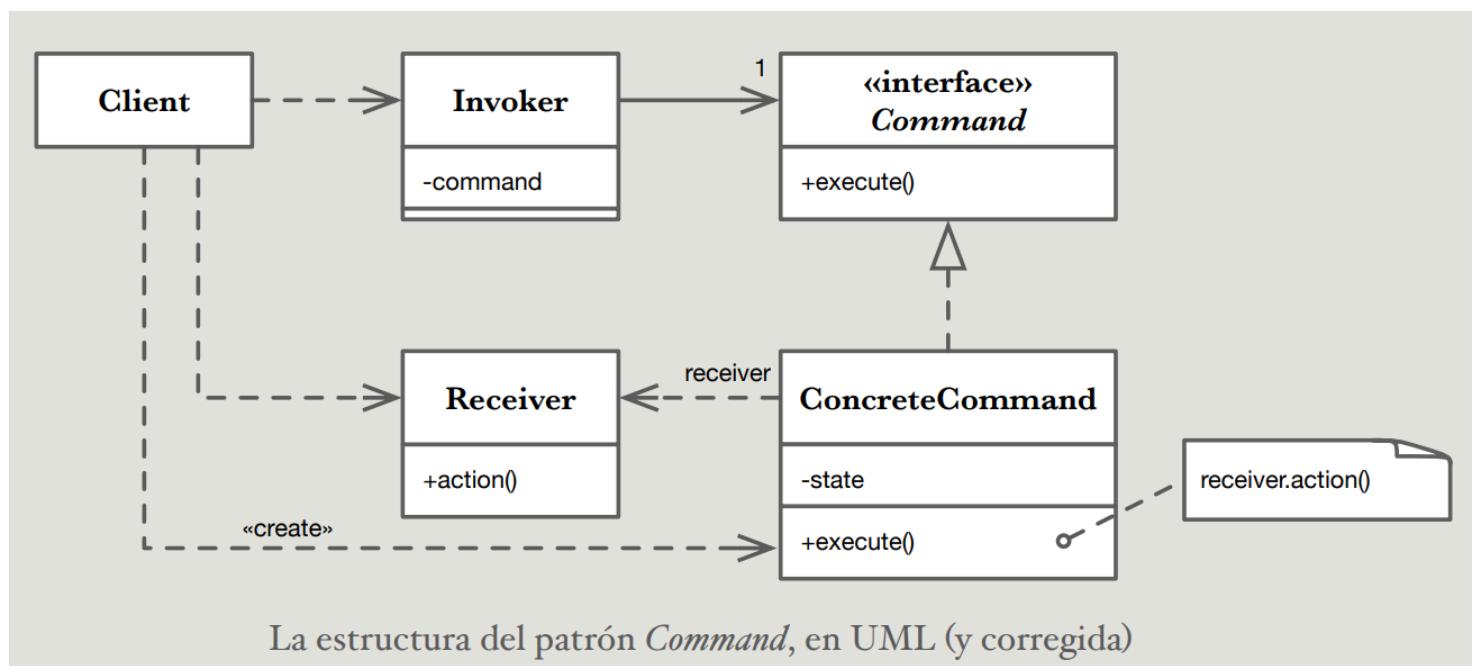
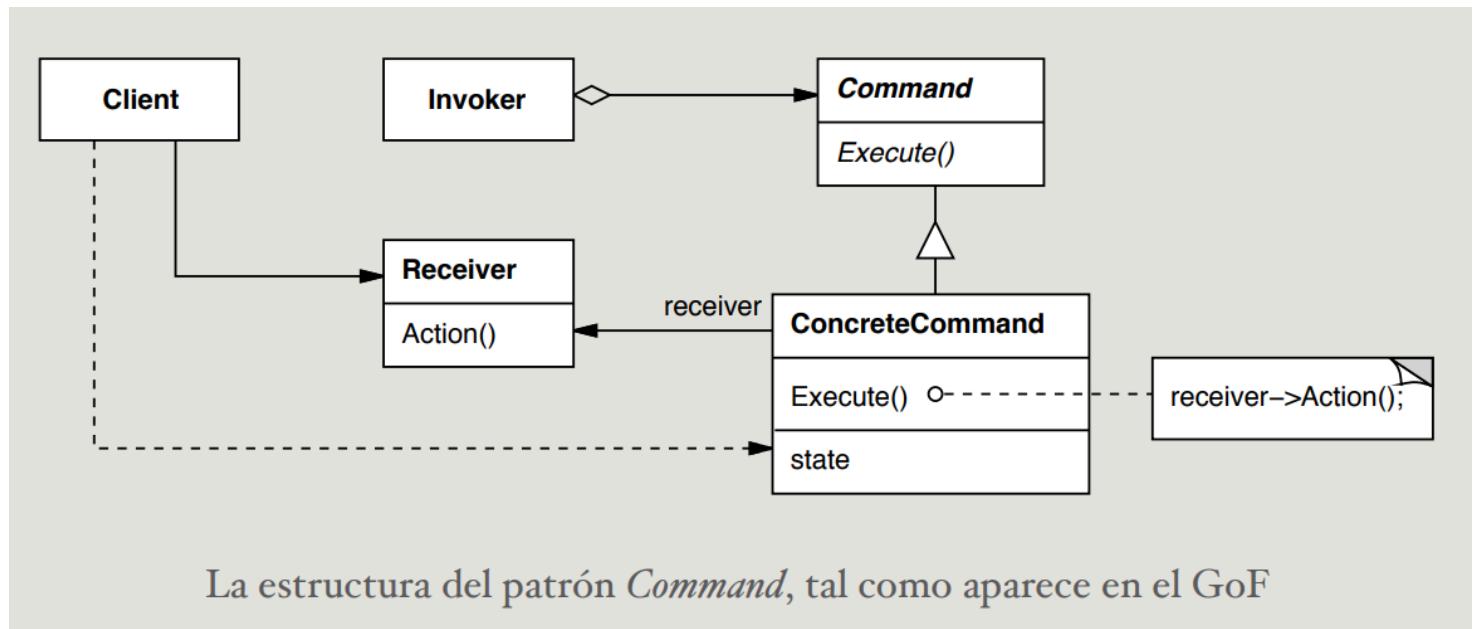
## Aplicabilidad

Úsese el patrón Command cuando se quiera:

- Parametrizar objetos con una acción a realizar
- Especificar, guardar y ejecutar peticiones en distintos momentos
  - Es decir, que la acción a realizar y el objeto que la crea tengan ciclos de vida distintos (desacoplamiento temporal)

- Permitir deshacer/repetir (undo/redo)
  - En ese caso, execute deberá guardar el estado para poder revertir los efectos de ejecutar la operación
  - Y hará falta una operación añadida, unexecute
- Guardar todas las operaciones ejecutadas en un registro (log)
  - Proporcionando un par de operaciones store y load
- Usar transacciones

## Estructura



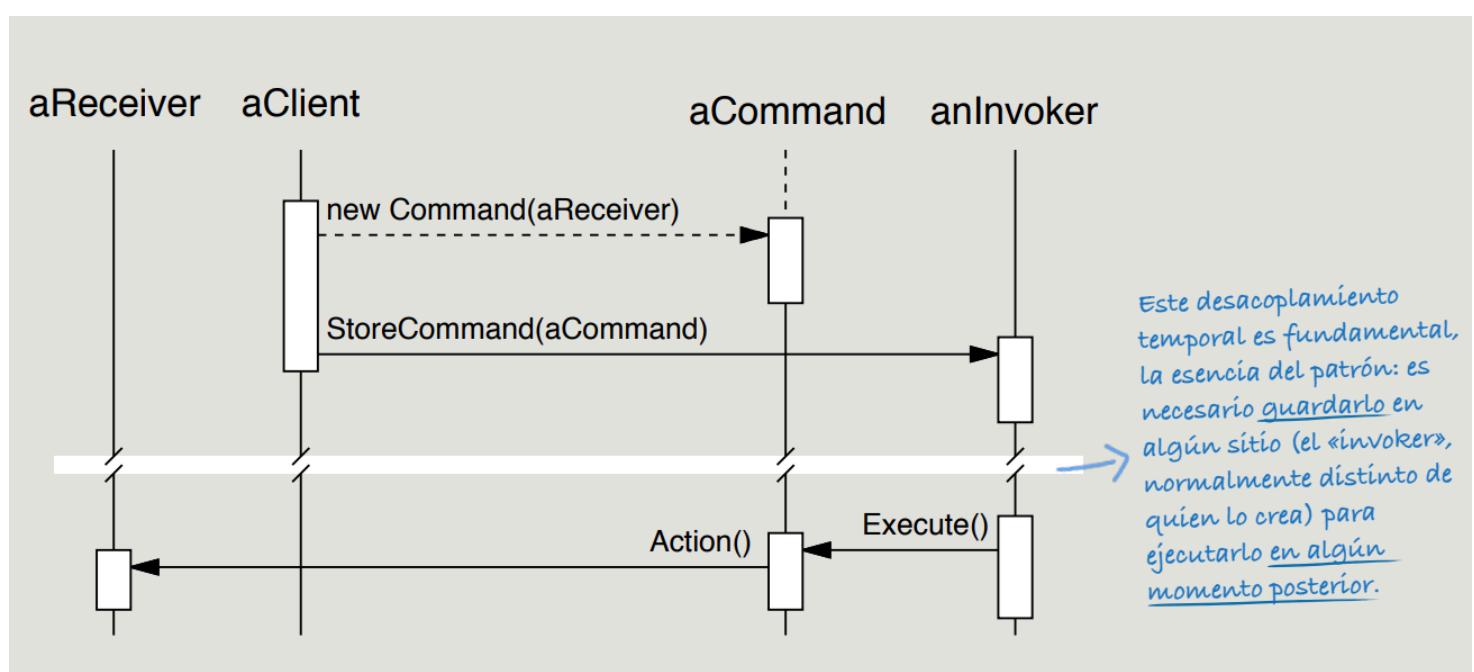
## Participantes

- **Orden** (Command)
  - Define una interfaz para ejecutar una operación
- **OrdenConcreta** (ConcreteCommand, OpenCommand...)
  - Define un enlace entre un objeto receptor y una acción
  - Implementa `execute` llamando a las operaciones de dicho receptor

- **Cliente** (Aplicación)
  - Crea un objeto OrdenConcreta (ConcreteCommand) y establece su receptor
- **Invocador** (ElementoDeMenu)
  - Le pide a la orden que ejecute la petición
- **Receptor** (Documento, Aplicacion)
  - Quien realmente lleva a cabo la acción.

## Colaboraciones

- El cliente crea un objeto ConcreteCommand y especifica su receptor
- Un objeto Invoker guarda el objeto ConcreteCommand
- Aquél llama a la operación de este último
  - Quien antes guarda el estado para luego poder deshacer la operación (si son operaciones que se pueden deshacer)
- El objeto ConcreteCommand se vale de las operaciones de su receptor para llevar a cabo la acción



## Consecuencias

- Desacopla el objeto que llama a la operación del que sabe cómo llevarla a cabo
- Son ciudadanos de primera clase (objetos)
- Se pueden ensamblar (Composite)
- Resulta sencillo añadir nuevas acciones, al no tener que tocar las clases existentes

## Implementación

- ¿Cómo de inteligente debería ser?
  - Desde un mero enlace entre el receptor y las operaciones a realizar en él hasta implementarlo todo él solo sin especificar un receptor
- Diferentes niveles de deshacer/repetir
  - A veces será necesario crear una copia del objeto antes de guardarlo en el historial

- En este caso, los Command serían también Prototype

## Ejemplo de César

### Sin Command

```
public void actionPerformed(ActionEvent e)
{
    Object o = e.getSource();
    if (o = fileNewMenuItem)
        doFileNewAction();
    else if (o = fileOpenMenuItem)
        doFileOpenAction();
    else if (o = fileOpenRecentMenuItem)
        doFileOpenRecentAction();
    else if (o = fileSaveMenuItem)
        doFileSaveAction();
    // and more ...
}
```

Mal

### Usando «Action»

```
// create our actions
cutAction = new CutAction("Cut", cutIcon, "Cut stuff onto the clipboard",
                         new Integer(KeyEvent.VK_CUT));
copyAction = new CopyAction("Copy", copyIcon, "Copy stuff to the clipboard",
                           new Integer(KeyEvent.VK_COPY));
pasteAction = new PasteAction("Paste", pasteIcon,
                             "Paste whatever is on the clipboard",
                             new Integer(KeyEvent.VK_PASTE));

// create our main menu
JMenu fileMenu = new JMenu("File");
JMenu editMenu = new JMenu("Edit");

// create our menu items, using the same actions the toolbar buttons use
JMenuItem cutMenuItem = new JMenuItem(cutAction);
JMenuItem copyMenuItem = new JMenuItem(copyAction);
JMenuItem pasteMenuItem = new JMenuItem(pasteAction);

// add the menu items to the Edit menu
editMenu.add(cutMenuItem);
editMenu.add(copyMenuItem);
editMenu.add(pasteMenuItem);
```

<http://alvinalexander.com/java/java-action-abstraction-actionlistener>

# Decorator

## Propósito

Asigna responsabilidades adicionales a un objeto dinámicamente , proporcionando una alternativa flexible a la herencia para extender la funcionalidad

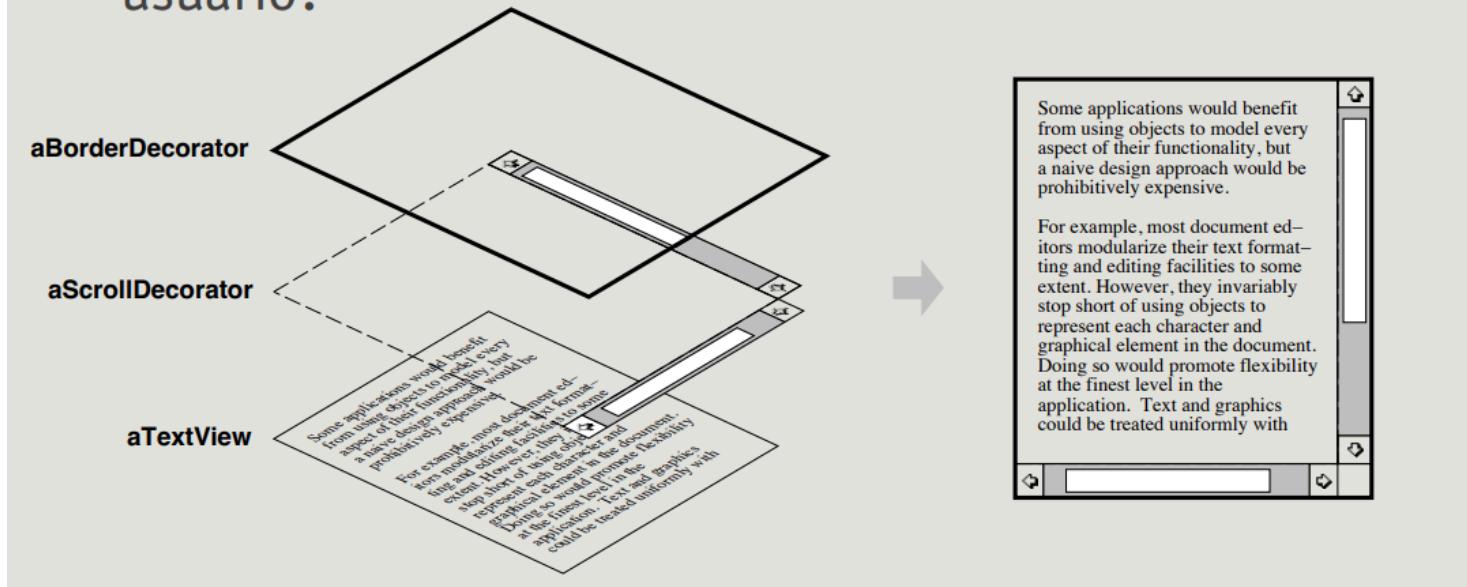
## También conocido como

*Wrapper* (Envoltorio)

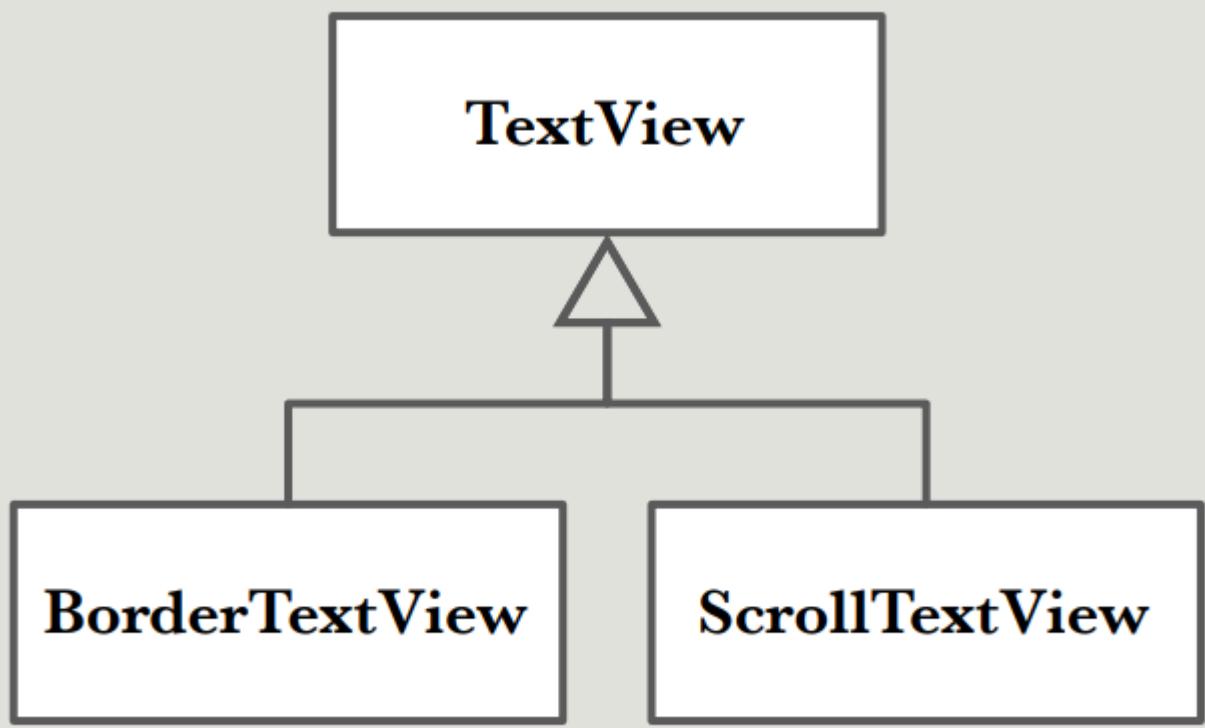
## Motivación

- A veces queremos añadir responsabilidades a objetos individuales, no a toda una clase

- Por ejemplo, ¿cómo añadiríamos un borde o una barra de desplazamiento a un componente de interfaz de usuario?



- Primera alternativa: mediante la herencia

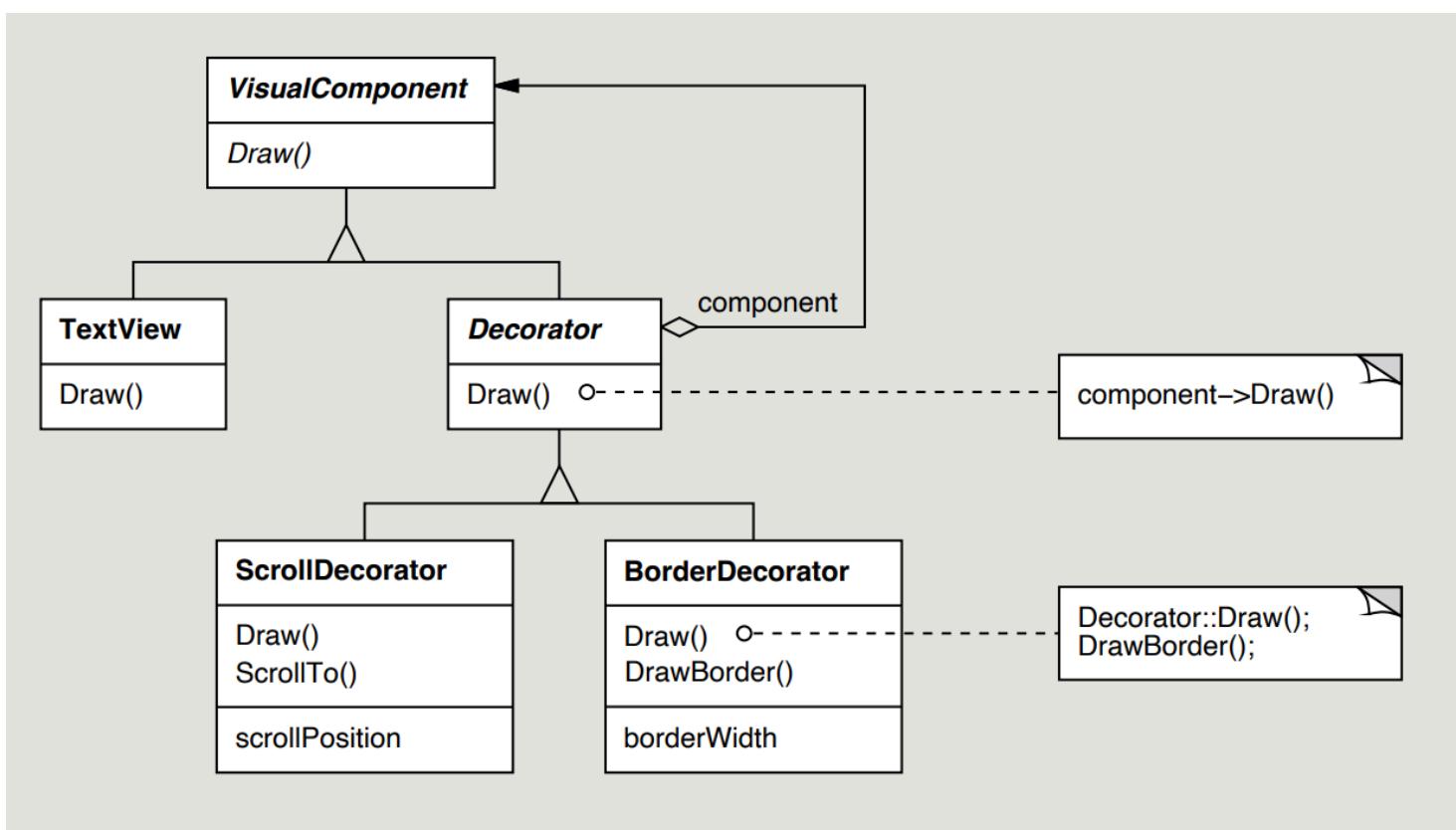
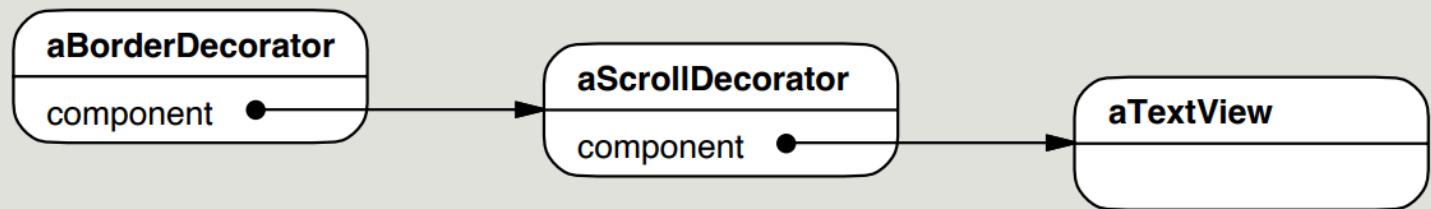


### Problem

- **Inflexible:** la elección del borde se hace estáticamente, no la puede hacer el cliente
  - **Explosión de clases:** ¿qué pasaría si queremos un `TextView` con borde y barra de desplazamiento?
- 
- Una solución más flexible es envolver el componente en otro objeto que sea quien añada el borde
    - Este objeto envoltorio es el decorador
  - El decorador sigue cumpliendo la interfaz del objeto original, así que su presencia es transparente para los clientes del componente
    - El decorador delega las peticiones al componente y puede llevar a cabo acciones adicionales
    - La transparencia permite anidar decoradores de forma recursiva

# Motivación: ejemplo

- El siguiente diagrama de objetos muestra un `TextView` con borde y barra de desplazamiento

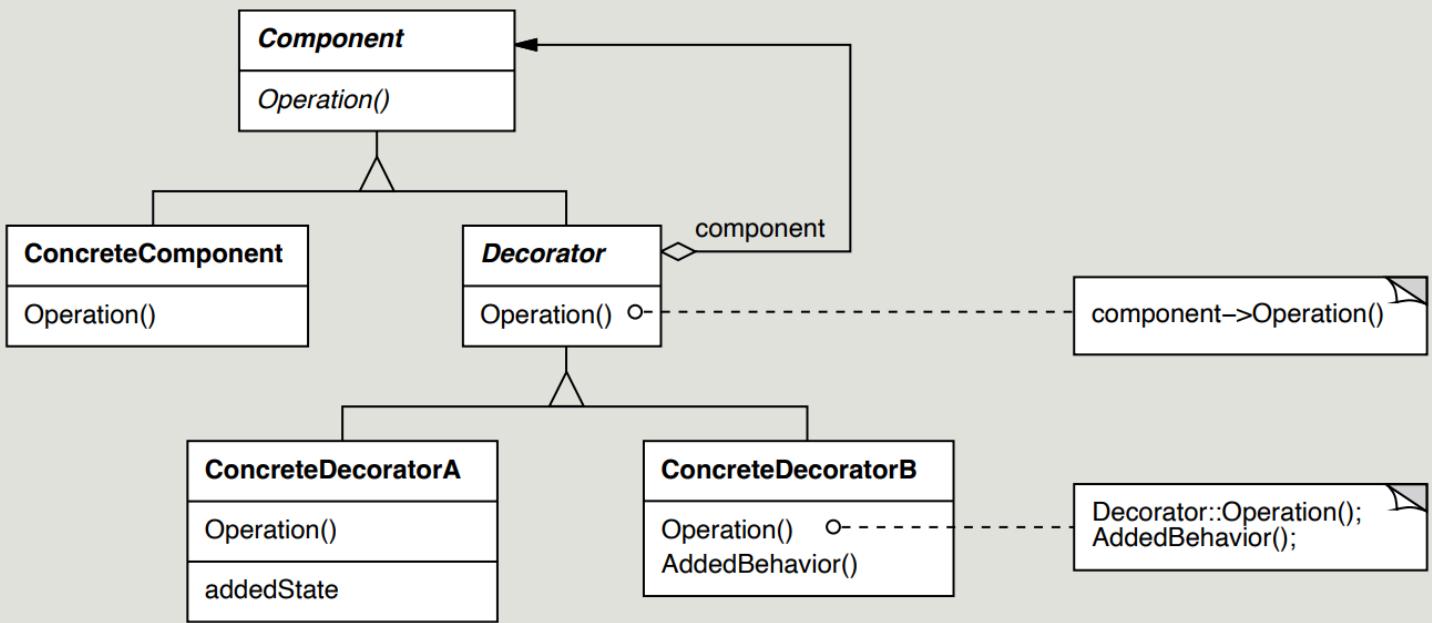


## Aplicabilidad

Use el Decorator:

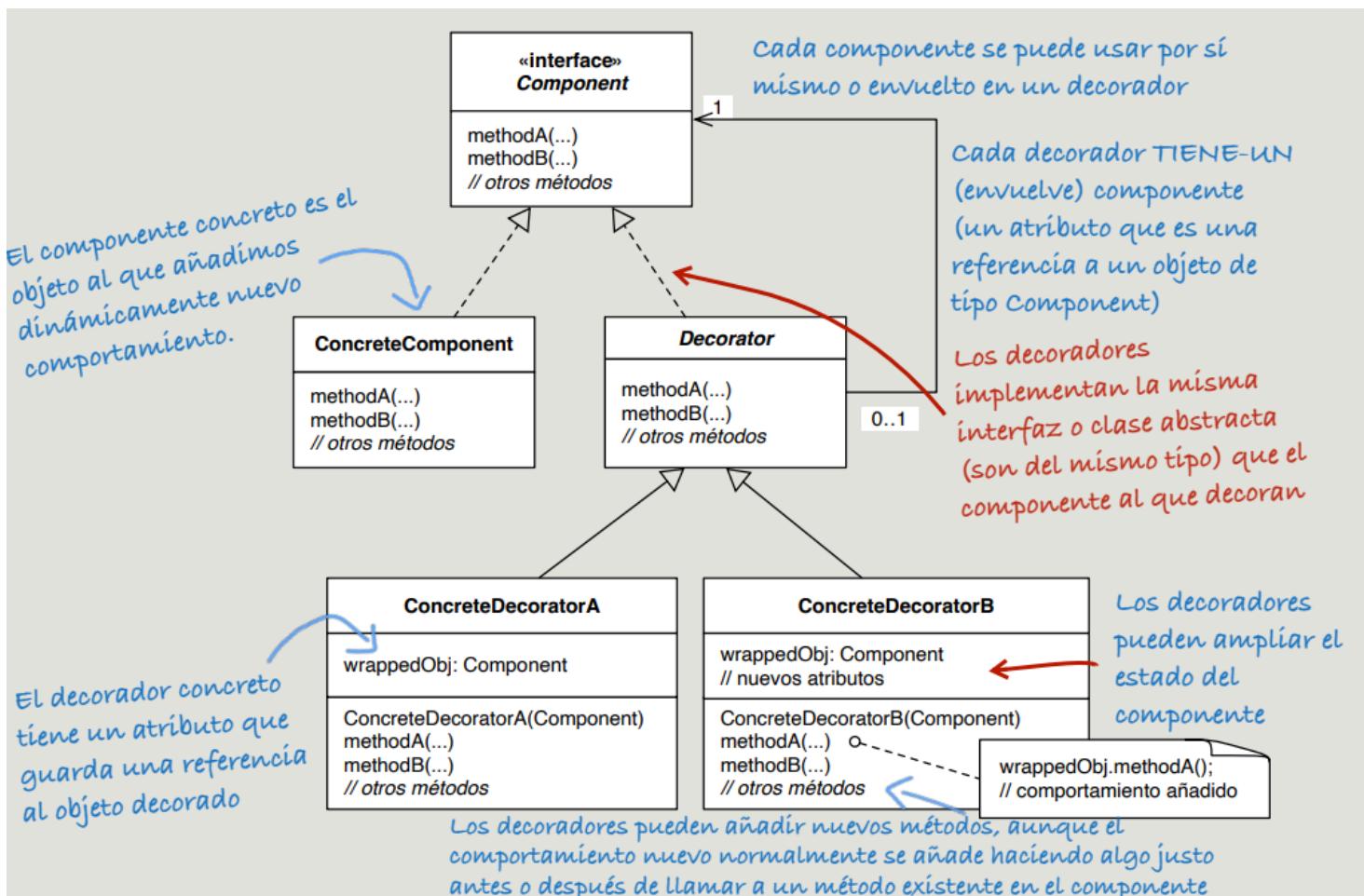
- Para añadir responsabilidades a otros objetos dinámicamente y de forma transparente
- Cuando no se puede heredar o no resulta práctico (explosión de subclases para permitir cada combinación posible)

## Estructura



La estructura del patrón *Decorator*, tal como aparece en el GoF (en UML, y más detallada y explicada, la hemos visto ya en la diapositiva 8)

En UML:



## Participantes

- **Componente** (ComponenteVisual)
  - Define la interfaz de los objetos a los que se les puede añadir responsabilidades dinámicamente
- **ComponenteConcreto** (VistaTexto)

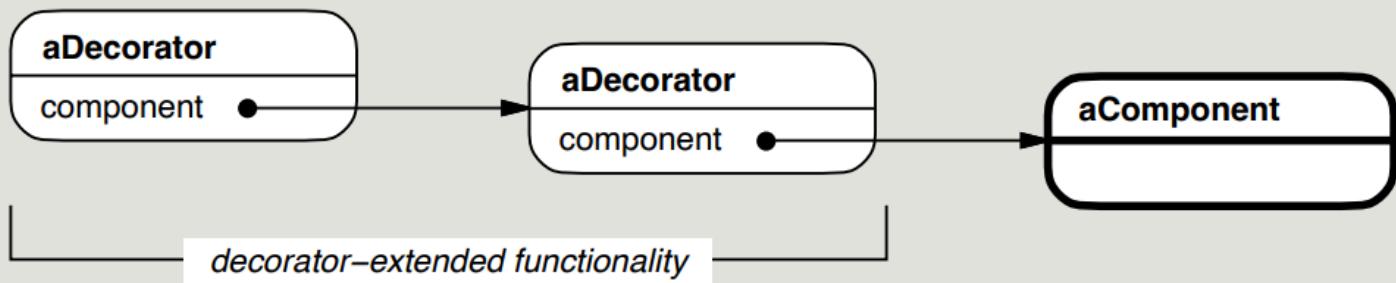
- Define un objeto al que se le pueden añadir responsabilidades adicionales
- **Decorador**
  - Mantiene una referencia a un objeto Componente y tiene su misma interfaz
- **DecoradorConcreto** (DecoradorBorde, DecoradorDesplazamiento)
  - Añade responsabilidades al componente

## Consecuencias

- **Ventajas:**
  - Más flexibilidad que la herencia estática
  - Evita que las clases de arriba de la jerarquía estén repletas de funcionalidades
- **Inconvenientes:**
  - Un decorador y sus componentes no son idénticos
    - Desde el punto de vista de la identidad de objetos
  - Muchos objetos pequeños
    - Mayor dificultad para depurar

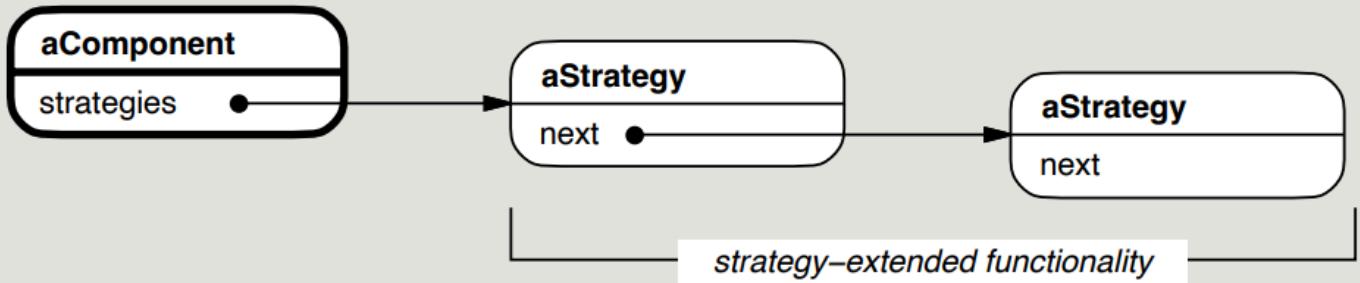
## Implementación

**Cambiar la «piel» del objeto en vez de sus «tripas»**



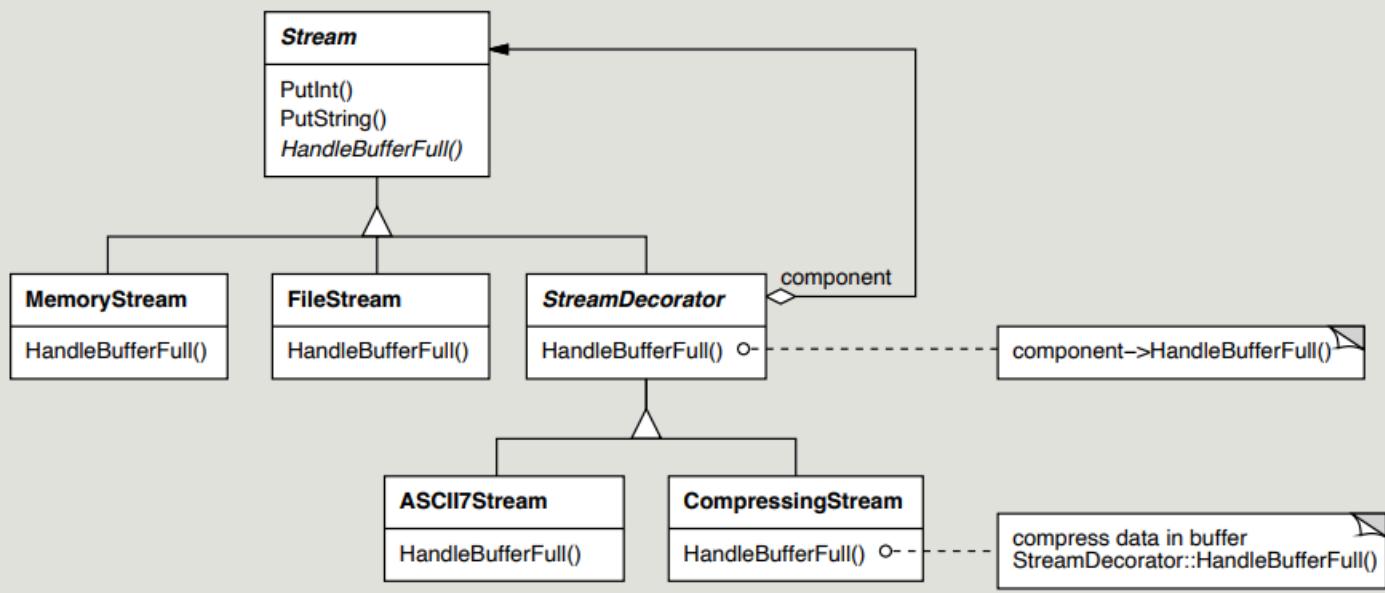
- ◎ Una alternativa sería emplear el patrón *Strategy*

- Por ejemplo, el componente podría permitir distintos tipos de borde delegando el dibujado de éste a un objeto **Border** aparte (la estrategia)



## Posibles usos

- Muchas bibliotecas de interfaces gráficas de usuario
- Los flujos en las bibliotecas de entrada/salida



## Patrones relacionados

- **Adapter:**
  - El Decorador sólo cambia las responsabilidades del objeto, no su interfaz
- **Composite:**
  - Un Decorador puede verse como un Composite de un solo componente
  - Pero el decorador añade responsabilidades adicionales (no está pensado para la agregación de objetos)
- **Strategy:**
  - El decorador cambia la piel del objeto; una estrategia cambia sus tripas

## Ejemplo real

# Java I/O

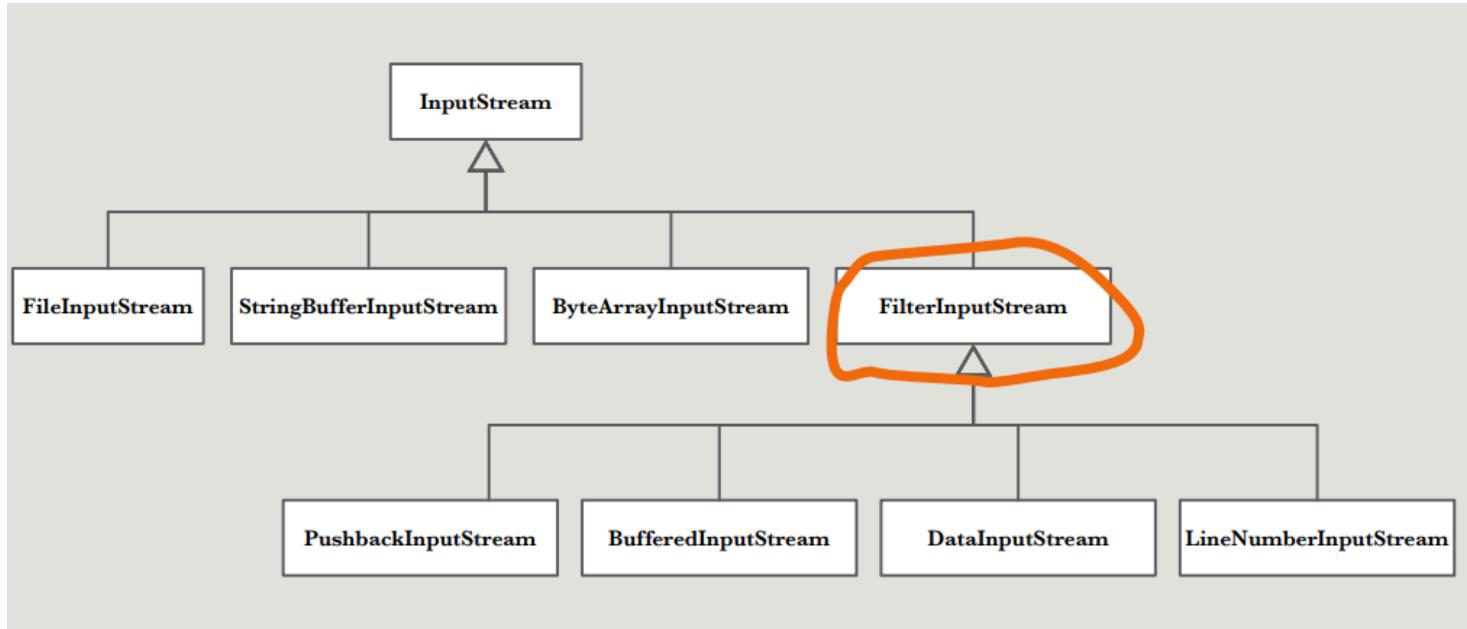
```
Reader in = new BufferedReader  
(new InputStreamReader(System.in));
```

## ● Ejemplos:

- BufferedInputStream
- LineNumberInputStream
- DataInputStream
- PushbackInputStream

¿Qué tienen en común estas clases?

Sí, son decoradores concretos.  
Pero, ¿qué clase de la API de Java haría las veces del decorador en sí?



## Observer

### Propósito

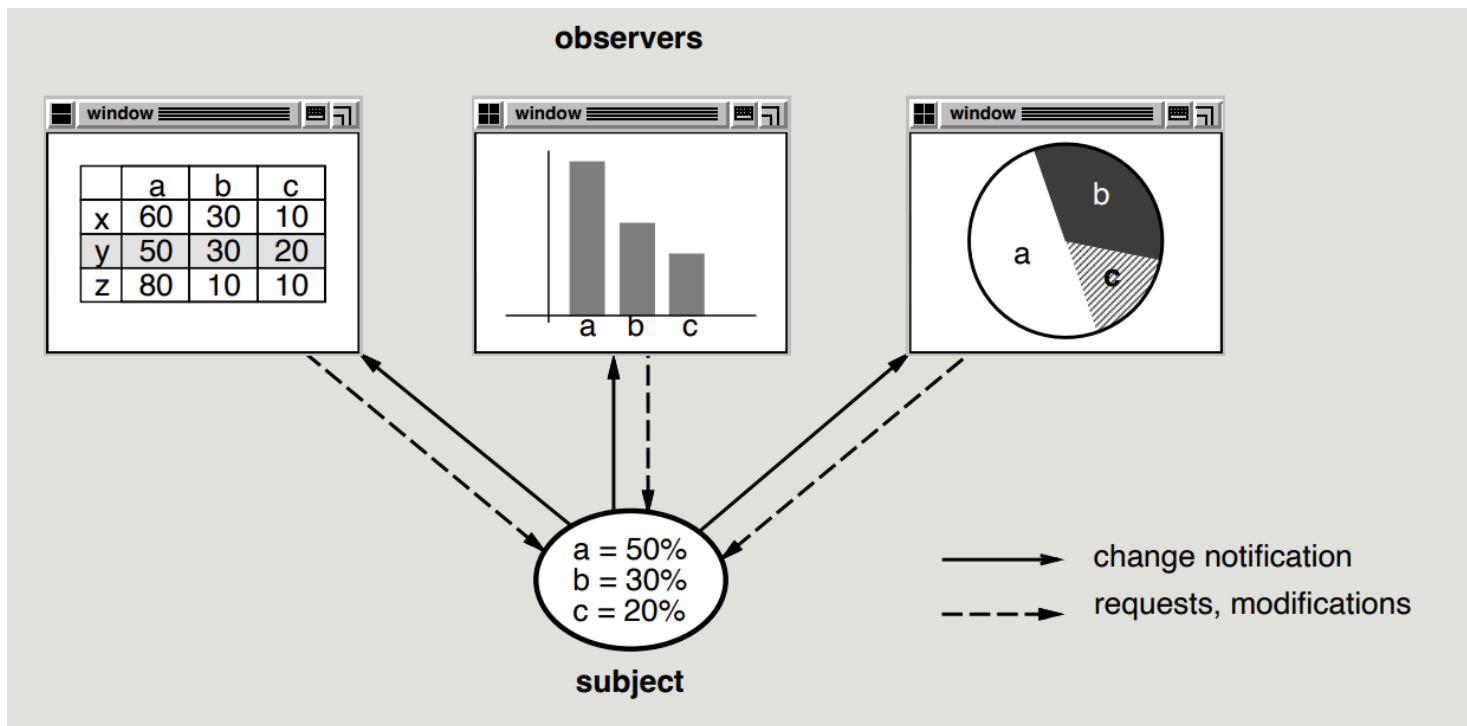
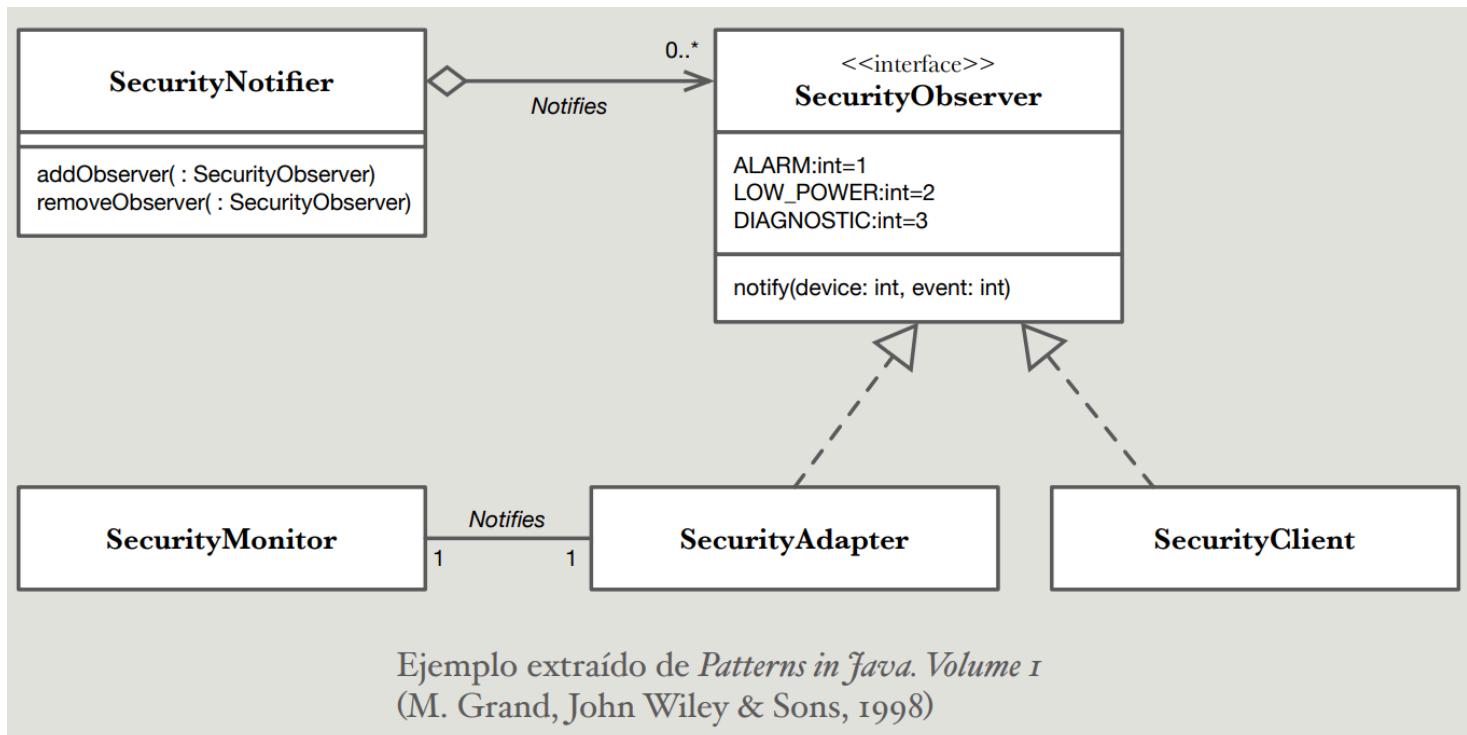
Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él

### También conocido como

*Dependents* (Dependientes), *Publish-Subscribe* (Publicar-Suscribir)

# Motivación

- Muchas veces un efecto lateral de partir un sistema en una colección de objetos relacionados es que necesitamos mantener la consistencia entre dichos objetos
  - ¿Cómo hacerlo sin que sus clases estén fuertemente acopladas?



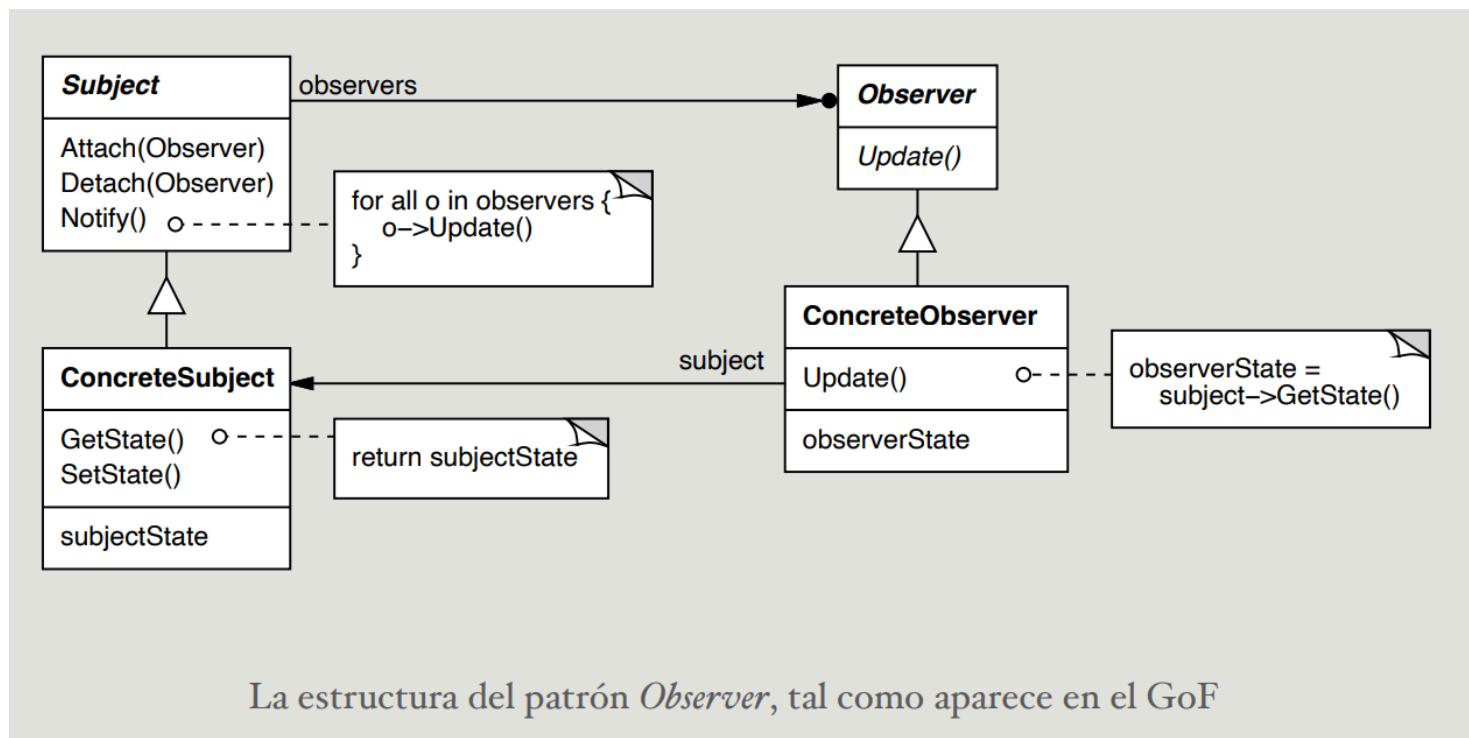
# Aplicabilidad

Úsese el patrón Observer:

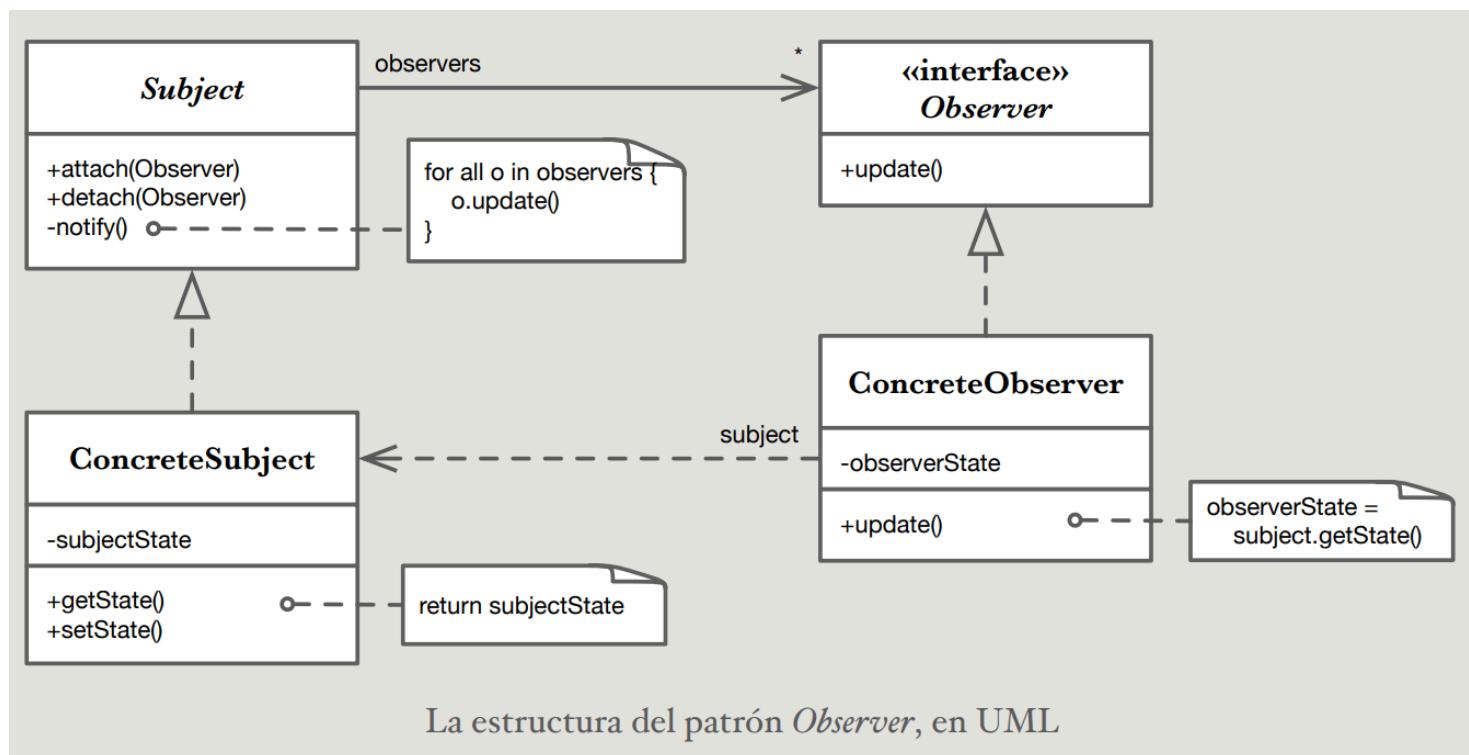
- Cuando una abstracción tiene dos aspectos, uno de los cuales depende del otro
  - Encapsular estos aspectos en objetos separados permite que los objetos varíen (y puedan ser reutilizados) de forma independiente
- Cuando un cambio en un objeto requiere que cambien otros

- Y no sabemos a priori cuáles ni cuántos
- Cuando un objeto necesita notificar a otros cambios en su estado sin hacer presunciones sobre quiénes son dichos objetos
  - Es decir, cuando no queremos que estén fuertemente acoplados

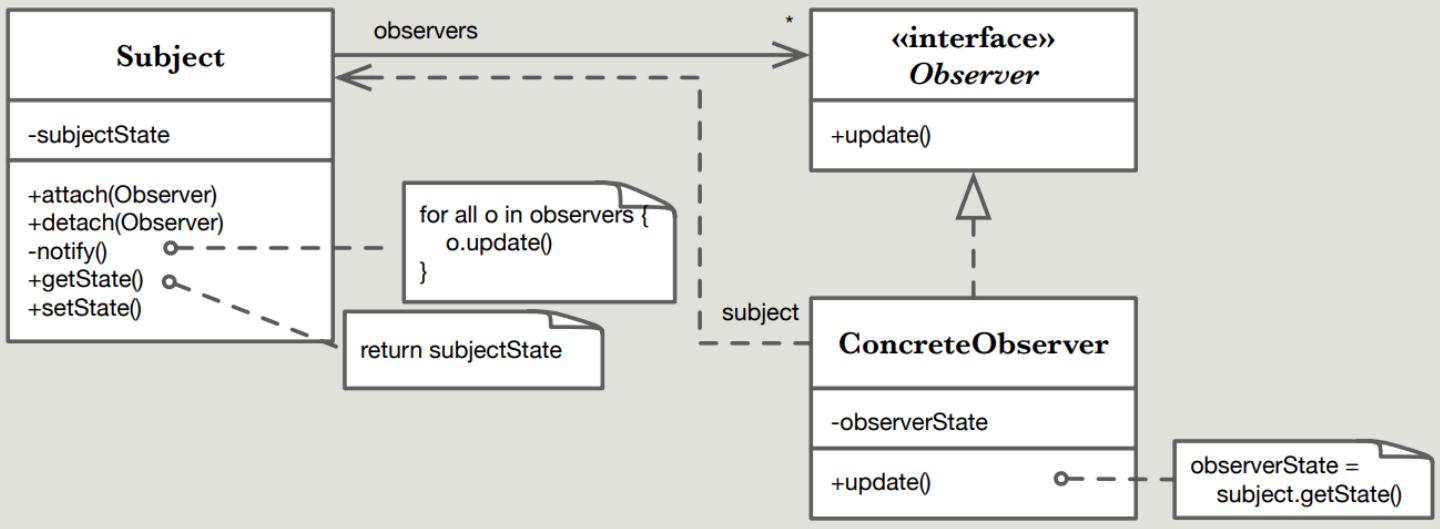
## Estructura



La estructura del patrón *Observer*, tal como aparece en el GoF



La estructura del patrón *Observer*, en UML



La estructura del patrón *Observer*, como suele ser más habitual en la práctica (sin una interfaz o clase abstracta aparte específicas sólo para el «Subject», sino añadiéndole dichas responsabilidades directamente al objeto observado)

## Participantes

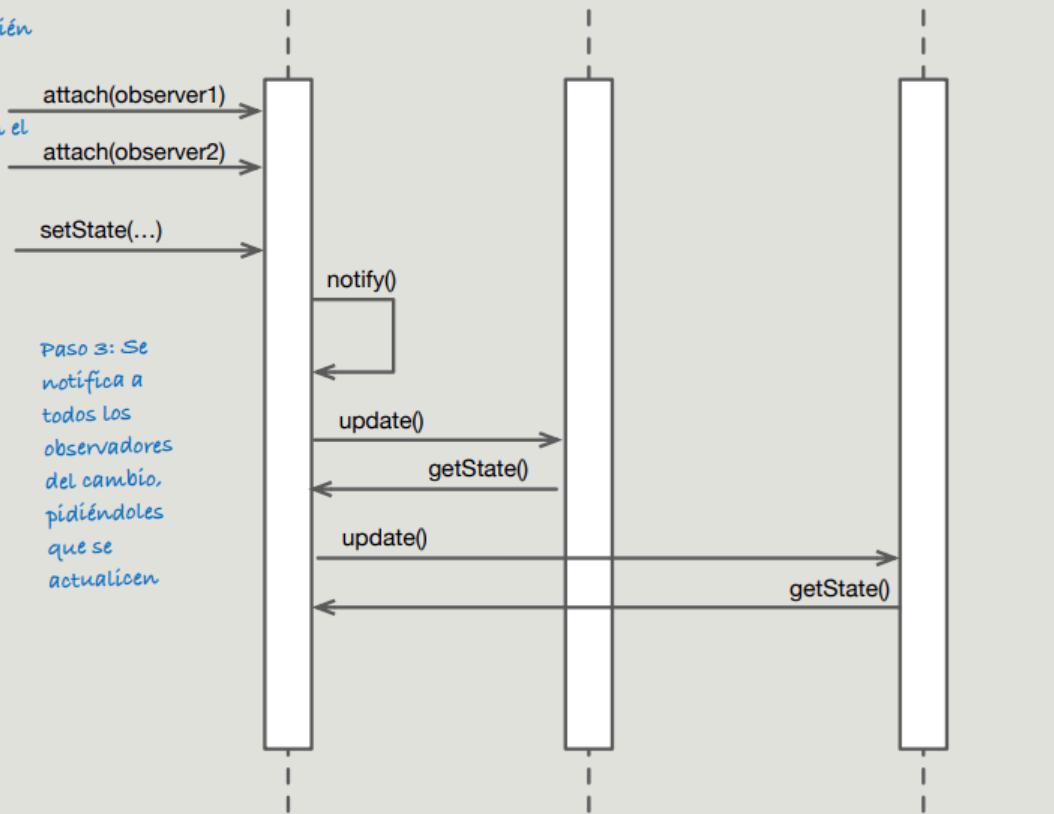
- **Sujeto (Subject)**
  - Conoce a sus observadores
  - Proporciona una interfaz para que se suscriban los objetos Observer (o que se borren)
- **Observador (Observer)**
  - Define una interfaz para actualizar los objetos que deben ser notificados de cambios en el objeto Subject
- **SujetoConcreto (ConcreteSubject)**
  - Guarda el estado de interés para los objetos ConcreteObserver
  - Envía una notificación a sus observadores cuando cambia su estado
- **ObservadorConcreto (ConcreteObserver)**
  - Mantiene una referencia a un objeto ConcreteSubject
  - Guarda el estado que debería permanecer sincronizado con el objeto observado
  - Implementa la interfaz Observer para mantener su estado consistente con el objeto observado

## Colaboraciones

- El objeto observado notifica a sus observadores cada vez que ocurre un cambio
- Después de ser informado de un cambio en el objeto observado, cada observador concreto puede pedirle información que necesita para reconciliar su estado con el de aquél

**PASO 1:** los observadores se suscriben al sujeto (esto se puede hacer desde fuera, no es necesario que sean los propios observadores; aunque también se podría hacer que fueran ellos mismos, al crearse, en los casos en que reciben al sujeto observando en el constructor)

**Paso 2:** cambia el estado del sujeto, por cualquier causa (algún cliente de la clase, algún observador...)



## Consecuencias

- Permite variar objetos observados y observadores independientemente
  - Se puede reutilizar los objetos observados sin sus observadores, y viceversa
  - Se pueden añadir nuevos observadores sin modificar ninguna de las clases existentes
- Acoplamiento abstracto entre Subject y Observer
  - Todo lo que un objeto sabe de sus observadores es que tiene una lista de objetos que satisfacen la interfaz Observer
- No se especifica el receptor de una actualización
  - Se envía a todos los objetos interesados
- Actualizaciones inesperadas
  - Se podrían producir actualizaciones en cascada muy inefficientes

## Implementación

- En vez de mantener una colección con referencias explícitas a los observadores en el objeto observado, sería posible hacerlo con una tabla hash que relacionase ambos
- Cuando un observador dependa de más de un objeto, es necesario ampliar la información de la operación `update`
- ¿Quién se encarga de llamar a la actualización (`notify`)?
  - El objeto observado, cada vez que cambia su estado
  - Los clientes
    - Más propenso a errores
- Protocolos de actualización
  - Modelo push

- El objeto observado envía información detallada a sus observadores sobre el cambio producido (la necesiten o no)
- Modelo pull
  - Tan sólo avisa de que cambió

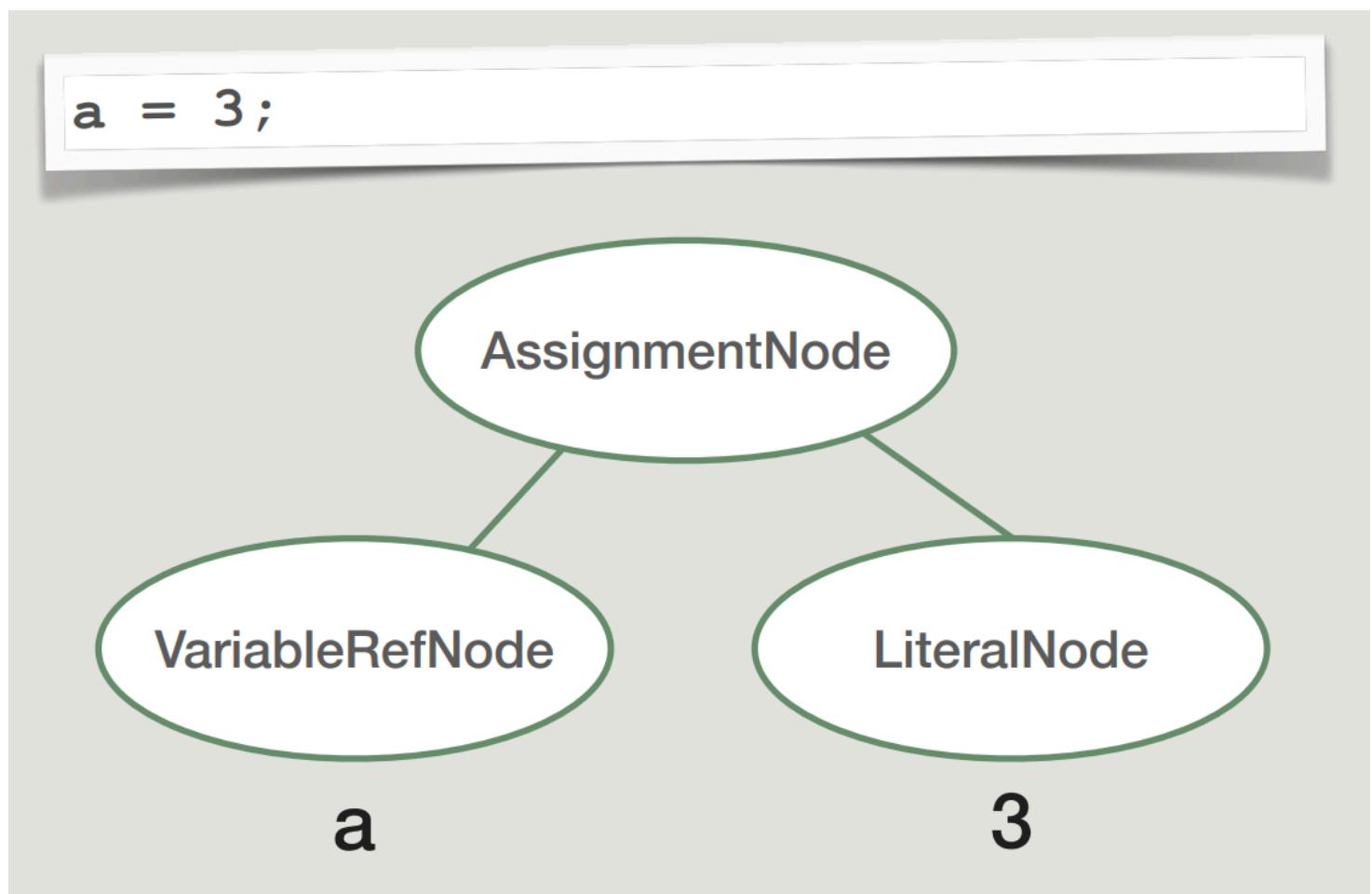
## Visitor

### Propósito

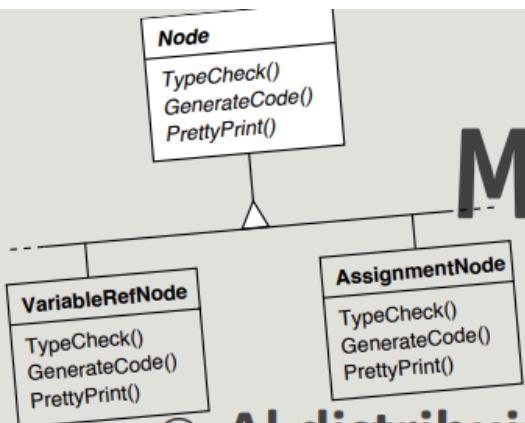
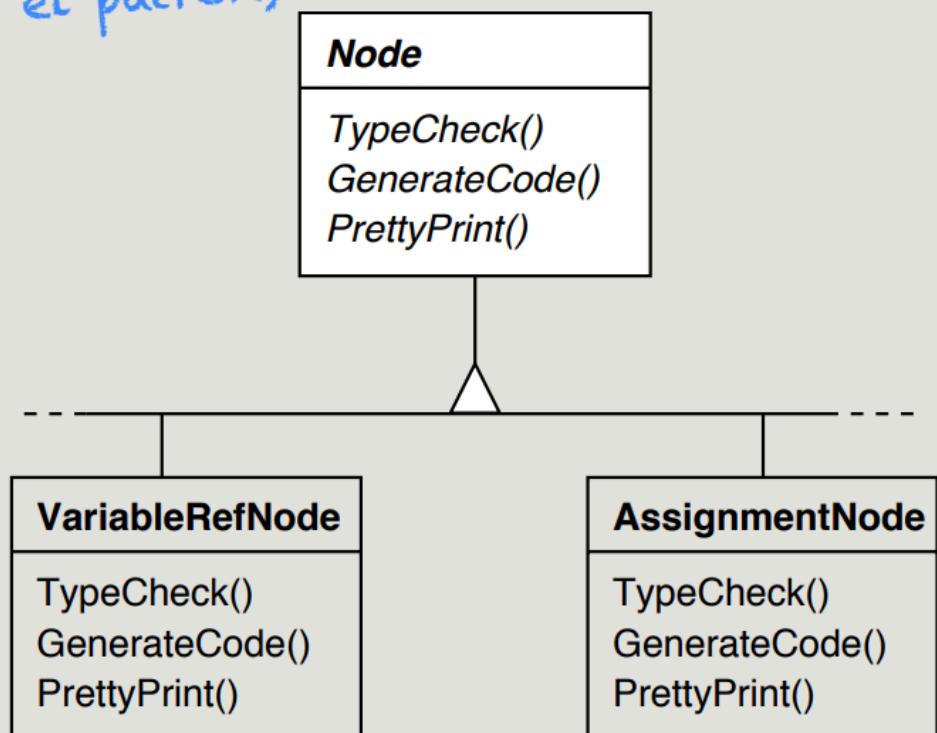
Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera

### Motivación

- Un compilador suele representar los programas mediante una estructura de árbol
- Necesitará realizar operaciones como:
  - Análisis sintáctico
  - Análisis semántico
  - Generación de código
  - ...
- Normalmente tendremos clases distintas para las distintas construcciones del lenguaje (referencias a variables, sentencias de asignación...)
- Serán los nodos del árbol



**Una posibilidad**  
(antes de aplicar el patrón)



## Motivación

Problema

- Al distribuir esas operaciones sobre todas las clases tenemos un sistema que es más difícil de comprender, mantener y cambiar

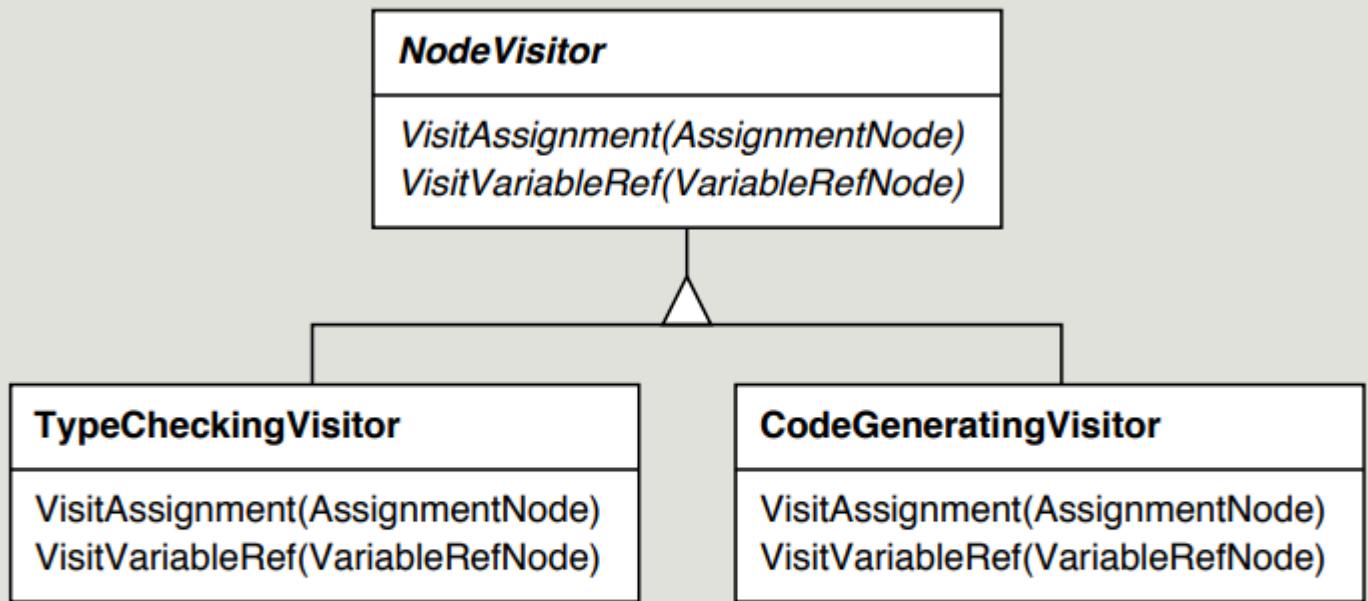
- Estamos mezclando responsabilidades que nada, o muy poco, tienen que ver entre sí  
*clases muy poco cohesivas*

*¿Qué habría que hacer para añadir una nueva operación (y eso sí es probable que ocurra)?*

- Encapsulamos cada tipo de operación en una clase "visitor" y se la pasamos al árbol

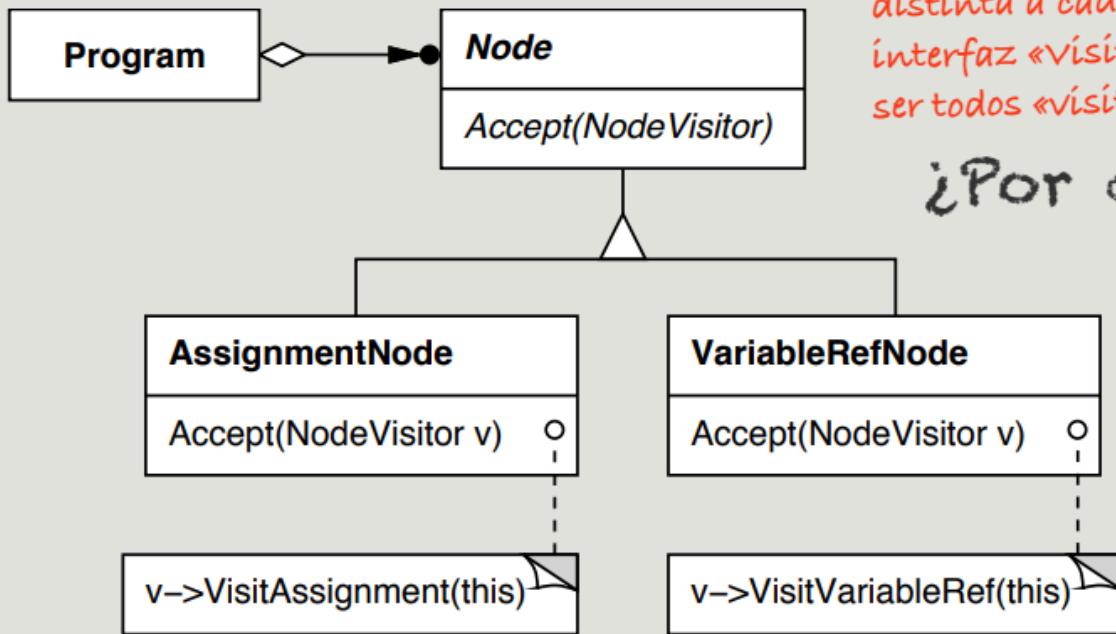
- Los nodos definirán una operación para aceptar visitantes
- Y llamarán a su vez a la operación apropiada del visitante pasándose a sí mismos como parámetros

## Los «visitantes»



cada clase «visitador» (una para cada tipo de operación distinta a realizar sobre el árbol: sintáctico, semántico, etc.) debe definir una operación para tratar cada tipo de nodo: sentencia de asignación, if, referencia a variable, valor literal, comentario...

# EL árbol



Realmente, al hacerlo así no haría falta llamar de forma distinta a cada método de la interfaz «visitor»: podrían ser todos «visit», a secas.

¿Por qué?

Con la operación «accept», que recibe un «visitor» como parámetro, definida en cada tipo de nodo

- Lo que se hace es simular el **despacho doble** (double dispatch)

## Aplicabilidad

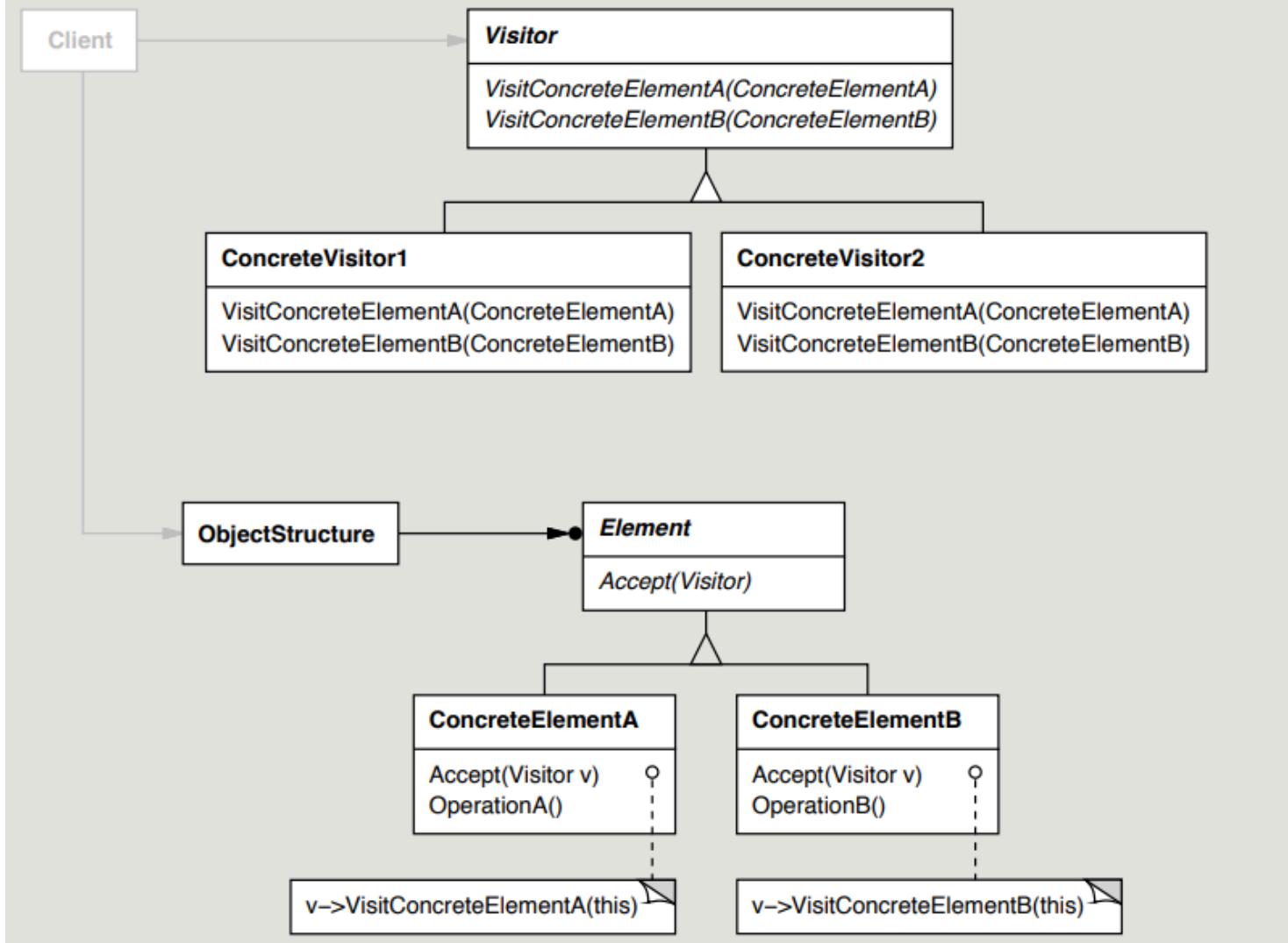
Úsese el patrón Visitor cuando:

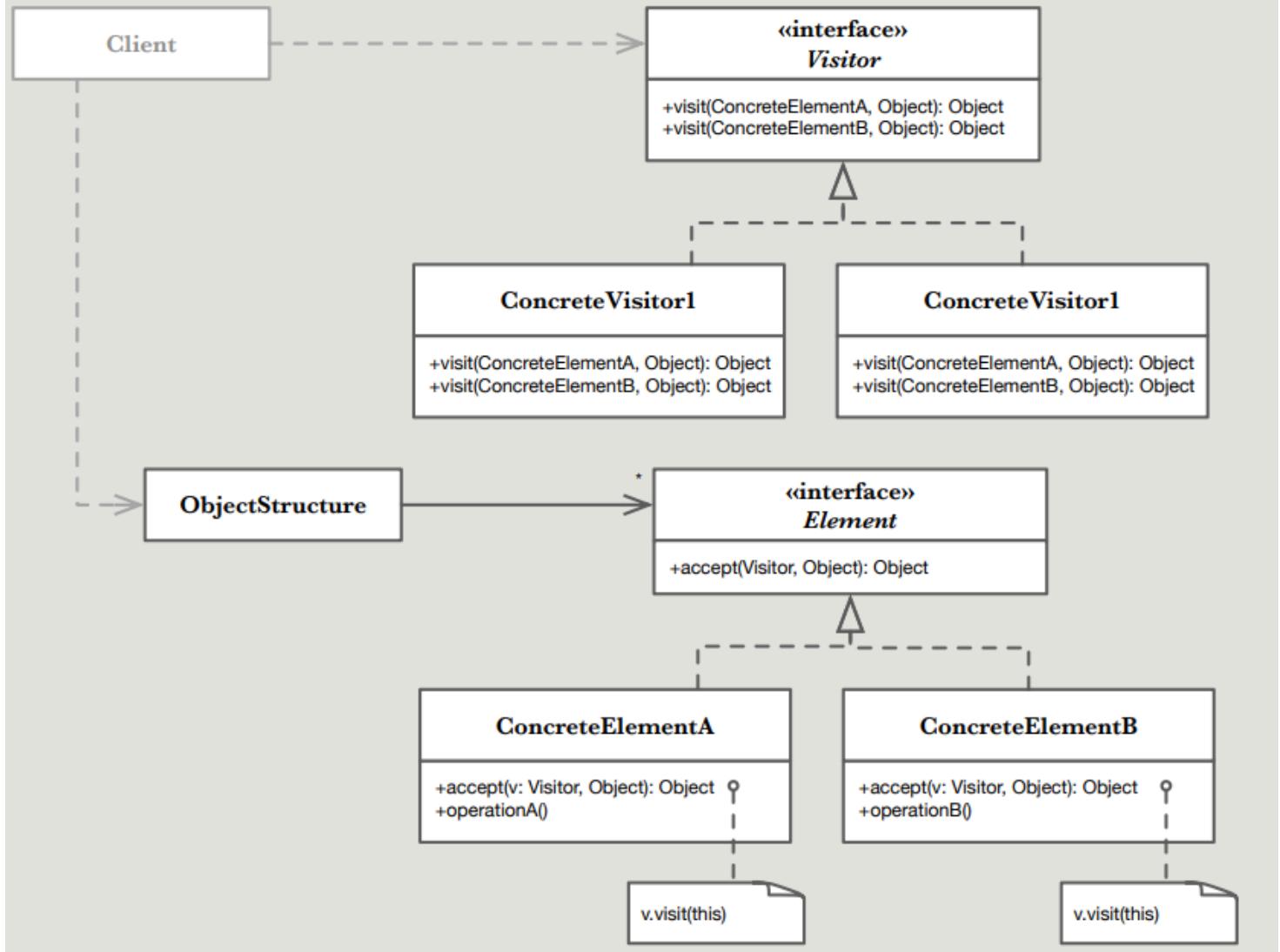
- Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y queremos realizar operaciones sobre esos elementos que dependen de su clase concreta
- Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos, y queremos evitar "contaminar" sus clases con dichas operaciones
  - Con el Visitor se pueden mantener juntas operaciones relacionadas en una clase

Debería aplicarse el patrón Visitor cuando:

- Las clases que definen la estructura de objetos rara vez cambian, pero muchas veces queremos definir nuevas operaciones sobre la estructura

## Estructura



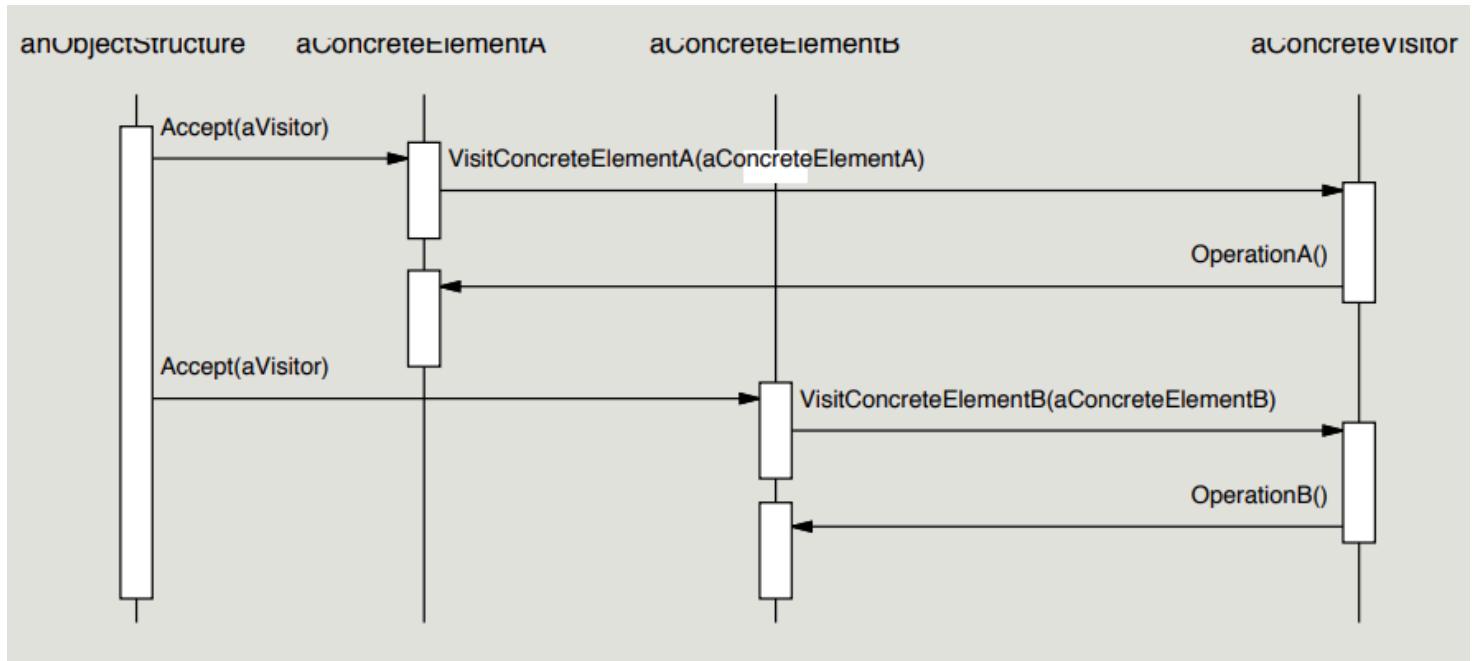


## Participantes

- **Visitante** (Visitor, NodeVisitor)
  - Declara una operación `visit` para cada clase de operación `ConcreteElement` de la estructura de objetos
  - El nombre (opcional) y firma de la operación identifican a la clase que envía la petición `visit` al visitante
- **VisitanteConcreto** (ConcreteVisitor, TypeCheckingVisitor)
  - Implementa cada operación declarada por `Visitor`
  - Cada operación implementa un fragmento del algoritmo definido para la clase correspondiente de la estructura
  - `ConcreteVisitor` proporciona el contexto para el algoritmo y guarda su estado local
- **Elemento** (Element, Node)
  - Define una operación `accept` que recibe un visitante como argumento
- **ElementoConcreto** (ConcreteElement, AssignmentNode, VariableRefNode)
  - Implementa una operación `accept` que recibe un visitante como argumento
- **EstructuraDeObjetos** (ObjectStructure, Program)
  - Permite enumerar sus elementos
  - Puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos
  - Puede ser un compuesto (patrón Composite) o una colección, como una lista o un conjunto

# Colaboraciones

- Un cliente que usa el patrón **Visitor** debe crear un objeto **ConcreteVisitor** y a continuación recorrer la estructura, visitando cada objeto con el visitante
- Cada vez que se visita a un elemento, éste llama a la operación del **Visitor** que se corresponde con su clase
  - El elemento se pasa a sí mismo como argumento de la operación



## Consecuencias

- El visitante facilita añadir nuevas operaciones
  - Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante
  - Si, por el contrario, extendiésemos la funcionalidad sobre muchas clases, habría que cambiar cada clase para definir una nueva operación
- Un visitante agrupa operaciones relacionadas y separa las que no lo están
- Es difícil añadir nuevas clases de elementos concretos

# Prototype

## Propósito

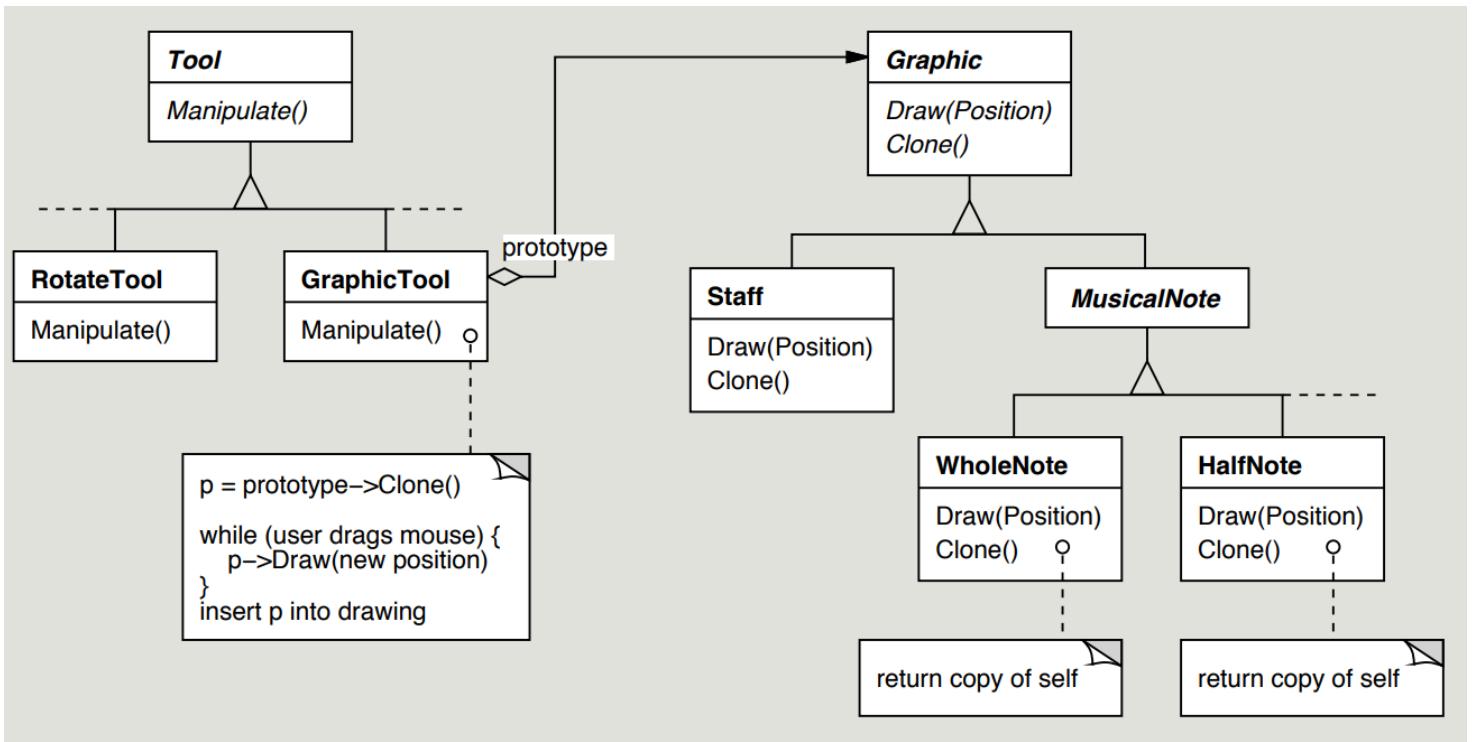
Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo

- Básicamente consiste en que los objetos sepan cómo clonarse a sí mismos

## Motivación

- Tenemos que crear un framework para editores gráficos
- Un usuario podría construir con él un editor de partituras musicales
- Supongamos que el framework provee una clase abstracta `Graphic` para los elementos gráficos

- El framework también proporciona una clase `GraphicTool` para los elementos de la paleta que permiten crear símbolos gráficos
- ¿Cómo podríamos parametrizar `GraphicTool` con el tipo de objeto a crear, aplicando la composición de objetos?
  - Haciendo que cada instancia de ella reciba en el constructor un objeto representando el tipo de figura a crear
  - Y que cada figura sepa cómo clonarse a sí misma

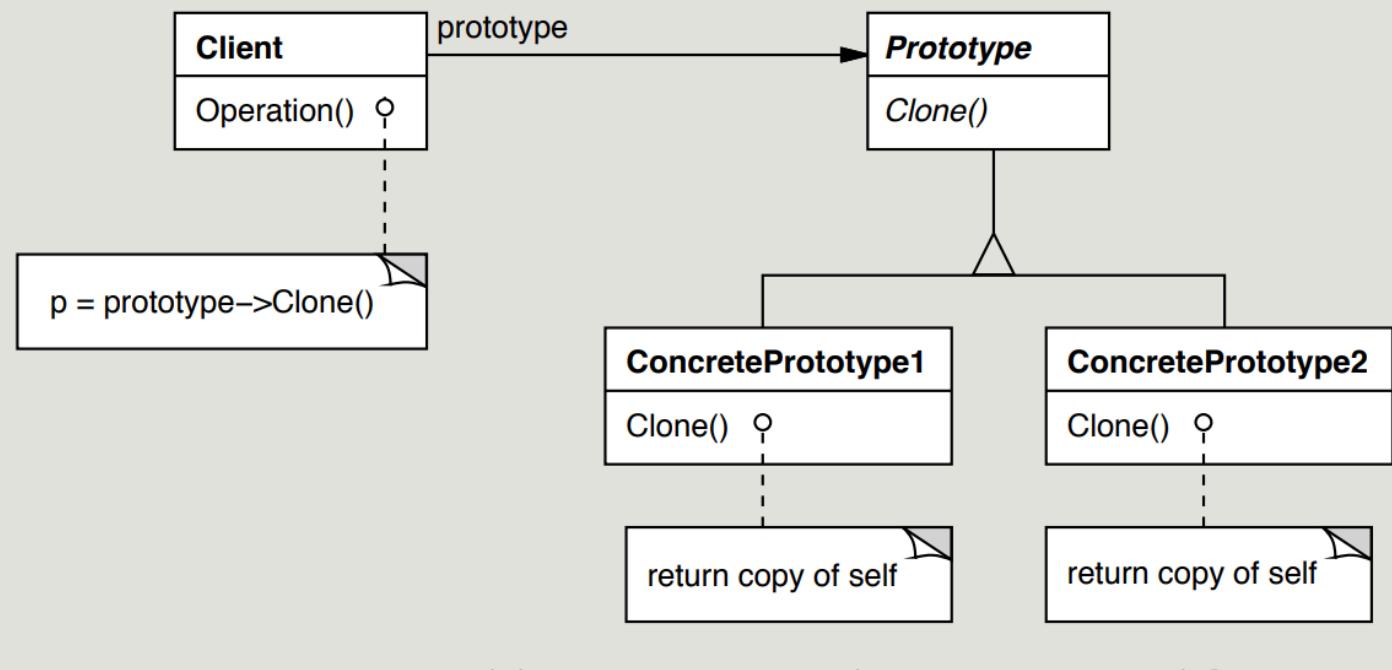


## Aplicabilidad

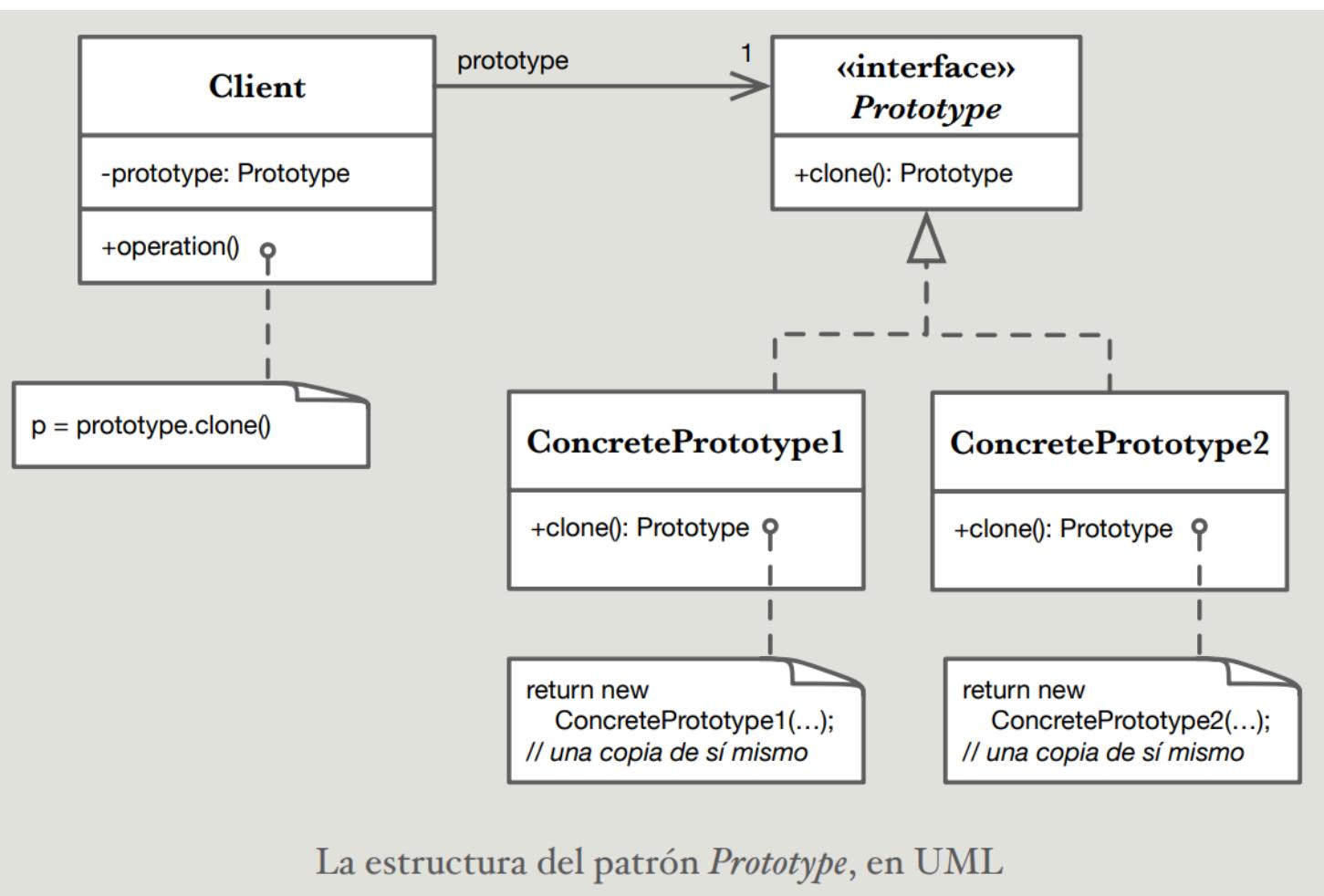
Úsese el patrón Prototype cuando un sistema deba ser independiente de cómo se crean, se componen y se representan sus productos, y además se da alguna de estas circunstancias:

- Las clases a instanciar son definidas en tiempo de ejecución
- Para evitar construir una jerarquía paralela de factorías de productos
- Cuando las instancias de una clase puedan tener sólo unos pocos posibles estados, y pueda resultar más conveniente crear los objetos correspondientes como prototipos y clonarlos, en vez de instanciar manualmente la clase, cada vez con el estado necesario

## Estructura



La estructura del patrón *Prototype*, tal como aparece en el GoF



La estructura del patrón *Prototype*, en UML

## Participantes

- **Prototipo** (Prototype, Graphic)
  - Declara la interfaz (normalmente una única operación para clonarse)
- **PrototipoConcreto** (ConcretePrototype, Staff, WholeNote, HalfNote)
  - Implementa la operación de clonación
- **Cliente** (Client, GraphicTool)
  - Crea un nuevo objeto diciéndole al prototipo que se clone

# Colaboraciones

- Un cliente le pide al prototipo que se clone

## Consecuencias

- Como el patrón *Abstract Factory*, oculta las clases concretas de producto al cliente
- Además permite:
  - Añadir y eliminar productos dinámicamente (en tiempo de ejecución)
  - Especificar nuevos objetos modificando valores de sus propiedades
    - Mediante composición de objetos
  - Especificar nuevos objetos variando su estructura
    - A partir de partes y subpartes
    - Podemos guardar esas estructuras complejas para crearlas una y otra vez
    - Entrará el juego el patrón *Composite*
  - Reduce las subclases
    - A diferencia del *Factory Method*
- **Inconvenientes:**
  - La implementación de la operación de clonación puede no ser fácil

## Implementación

- Es especialmente útil en lenguajes como Java y C++, donde las clases no son objetos
- Hay lenguajes que lo incorporan de serie
- Cuestiones a tener en cuenta:
  - Usar un gestor o registro de prototipos
  - Implementación de la operación de clonación
    - ¿Copia profunda o superficial?
  - Inicialización de los prototipos

## Posibles usos

- Plantillas de Word

## Prototype vs Abstract Factory vs Factory Method

- El objeto fábrica de *Abstract Factory* produce objetos de varias clases, mientras que el Prototype hace que el objeto fábrica construya un producto copiando un objeto prototípico. En este caso, el objeto fábrica y el prototipo son el mismo objeto, ya que el prototipo es el responsable de devolver el producto.
- El *Factory Method* puede requerir crear una nueva subclase simplemente para cambiar la clase del producto (dichos cambios pueden tener lugar en cascada). Mientras que el Prototype hace que el objeto fábrica construya un producto copiando un objeto prototípico. En este caso, el objeto fábrica y el prototipo son el mismo objeto, ya que el prototipo es el responsable de devolver el producto.

## Note

*Esto está sacado del GoF*