

# **Patrones de diseño en C++**

Antonio Espín Herranz

# Curiosidad

- **Gangof Four(GoF, que en español es la pandilla de los cuatro) formada por:**
  - –Erich Gamma.
  - –Richard Helm.
  - –Ralph Johnson.
  - –John Vlissides.

# ¿Qué es un patrón de diseño?

- “Un patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe el núcleo de la solución al problema, de forma que puedes utilizar esta solución millones de veces” [C. Alexander].
- “Descripciones de clases y objetos que se comunican y que son adaptadas para resolver un problema de diseño general en un contexto particular”.

# Elementos de un patrón

- Son 4 elementos:
  - Nombre del patrón: Se describe con una o dos palabras, un problema de diseño con sus soluciones y consecuencias. Nos permiten un mayor nivel de abstracción a la hora de diseñar.
  - Problema: Describe cuando aplicar el patrón. A veces el problema incluye una serie de condiciones para poder aplicar el patrón.
  - Solución: Describe los elementos que constituyen el diseño o una implementación en concreto.
  - Consecuencias: Los resultados, así como ventajas e inconvenientes que surgen al aplicar el patrón.

# Descripción de los patrones de diseño

- A la hora de describir los patrones podemos utilizar los siguientes conceptos:
  - Nombre del patrón y Clasificación:
    - Un buen nombre es vital y la clasificación nos ayuda a entender mejor su comportamiento.
  - Propósito:
    - ¿Qué hace ese patrón?
    - ¿En qué se basa ese patrón?
    - ¿Qué problema de diseño resuelve?
  - Otros nombres que se le asignan.

# Catálogo de Patrones

PROPOSITO				
		DE CREACIÓN	ESTRUCTURALES	COMPORTAMIENTO
Ámbito	Clase	Factory Method	Adapter (de Clases)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (de Objetos) Bridge Composite Decorator Facade FlyWeight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

# Catálogo de Patrones

- Hacemos una primera clasificación por el **propósito**:
  - **Creación**: Tienen que ver con el proceso de creación de objetos.
  - **Estructurales**: Tratan de la composición de clases y objetos.
  - **Comportamiento**: Especifica la forma en que las clases y objetos interactúan y se reparten las responsabilidades.
- En cuanto al segundo criterio **ámbito** especifica si el patrón se aplica a clases u objetos.

# Breve descripción I

- **Abstract Factory (Fábrica abstracta):**
  - Proporciona una interfaz de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.
- **Adapter (Adaptador):**
  - Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.
- **Bridge (Puentes):**
  - Desacopla una abstracción de su implementación, de manera que ambas pueden variar de forma independiente.
- **Builder (Constructor):**
  - Separa la construcción de un objeto de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.



# Breve descripción II

- **Chain of Responsibility (Cadena de responsabilidad):**
  - Evita el acoplar el emisor de una petición a su receptor, al dar a mas de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que esta sea tratada por algún objeto.
- **Command (Orden):**
  - Encapsula una petición de un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.
- **Composite (Compuesto):**
  - Combina objetos en estructura de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
- **Decorator (Decorador):**
  - Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

# Breve descripción III

- **Facade (Fachada):**
  - Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.
- **Factory Method (Método de fabricación):**
  - Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan que clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.
- **FlyWeight (Peso ligero):**
  - Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.
- **Interpreter (Intérprete):**
  - Dado un lenguaje, define una representación de su gramática junto con su intérprete que usa dicha representación para interpretar sentencias del lenguaje.

# Breve descripción IV

- **Iterator (Iterador):**
  - Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
- **Mediator (Mediador):**
  - Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
- **Memento (Recuerdo):**
  - Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que este puede volver a dicho estado mas tarde.
- **Observer (Observador):**
  - Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualiza automáticamente todos los objetos que dependen de él.

# Breve descripción V

- **Prototype (Prototipo):**
  - Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando de ese prototipo.
- **Proxy (Apoderado):**
  - Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.
- **Singleton (Único):**
  - Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.
- **State (Estado):**
  - Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

# Breve descripción VI

- **Strategy (Estrategia):**
  - 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.
- **Template Method (Método plantilla):**
  - Define en una operación el esqueleto de una plantilla, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.
- **Visitor (Visitante):**
  - 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.

# Aspectos de Diseño que los patrones permiten modificar

PROPÓSITO	Patrones de diseño	Aspectos que pueden variar
De creación	Abstract Factory	<i>La familia de los objetos producidos</i>
	Builder	<i>Cómo se crea un objeto compuesto</i>
	Factory Method	<i>La subclase del objeto que es instanciado</i>
	Prototype	<i>La clase del objeto que es instanciado</i>
	Singleton	<i>La única instancia de una clase</i>
Estructurales	Adapter	<i>La interfaz de un objeto</i>
	Bridge	<i>La implementación de un objeto</i>
	Composite	<i>La estructura y composición de un objeto</i>
	Decorator	<i>Las responsabilidades de un objeto sin usar la herencia</i>
	Facade	<i>La interfaz de un subsistema</i>
	Flyweight	<i>El coste de almacenamiento de los objetos</i>
	Proxy	<i>Como se accede a un objeto, su ubicación</i>
De comportamiento	Chain of responsibility	<i>El objeto que puede satisfacer una petición</i>
	Command	<i>Cuándo y cómo se satisface una petición</i>
	Interpreter	<i>La gramática e interpretación de un lenguaje</i>
	Iterator	<i>Cómo se recorren los elementos de un agregado</i>
	Mediator	<i>Qué objetos interactúan entre sí, y cómo</i>
	Memento	<i>Qué información privada se almacena fuera de un objeto, y cuando</i>
	Observer	<i>El número de objetos que dependen de otro, cómo se mantiene actualizado el objeto dependiente</i>
	State	<i>El estado de un objeto</i>
	Strategy	<i>Un algoritmo</i>
	Template Method	<i>Los pasos de un algoritmo</i>
	Visitor	<i>Las operaciones que pueden aplicarse a los objetos sin cambiar sus clases</i>

# Patrones de Creación

Singleton  
Abstract Factory  
Factory Method  
Builder  
Prototype

# Patrones de Creación

- Características principales:
  - Nos abstraen el proceso de creación de instancias.
  - Ayudan a hacer un sistema independiente de cómo se crean, se componen y se representan sus objetos.
  - Estos patrones se hacen mas importantes a medida que los sistemas pasan a  ***depender mas de la composición de objetos que de la herencia***  entre clases.
  - Los patrones de creación dan mucha flexibilidad a qué es lo que se crea, quién lo crea y cuándo.



# SINGLETON - Definición

- Singleton provee un mecanismo para limitar el número de instancias de una clase. Por lo tanto el mismo objeto es siempre compartido por distintas partes del código.
- Singleton puede ser visto como una solución más elegante para una variable global porque los datos son abstraídos por detrás de la interfaz de la clase singleton.
- Patrones relacionados : *Abstract Factory* – *Monostate*.

# SINGLETON –Motivacion & Aplicabilidad

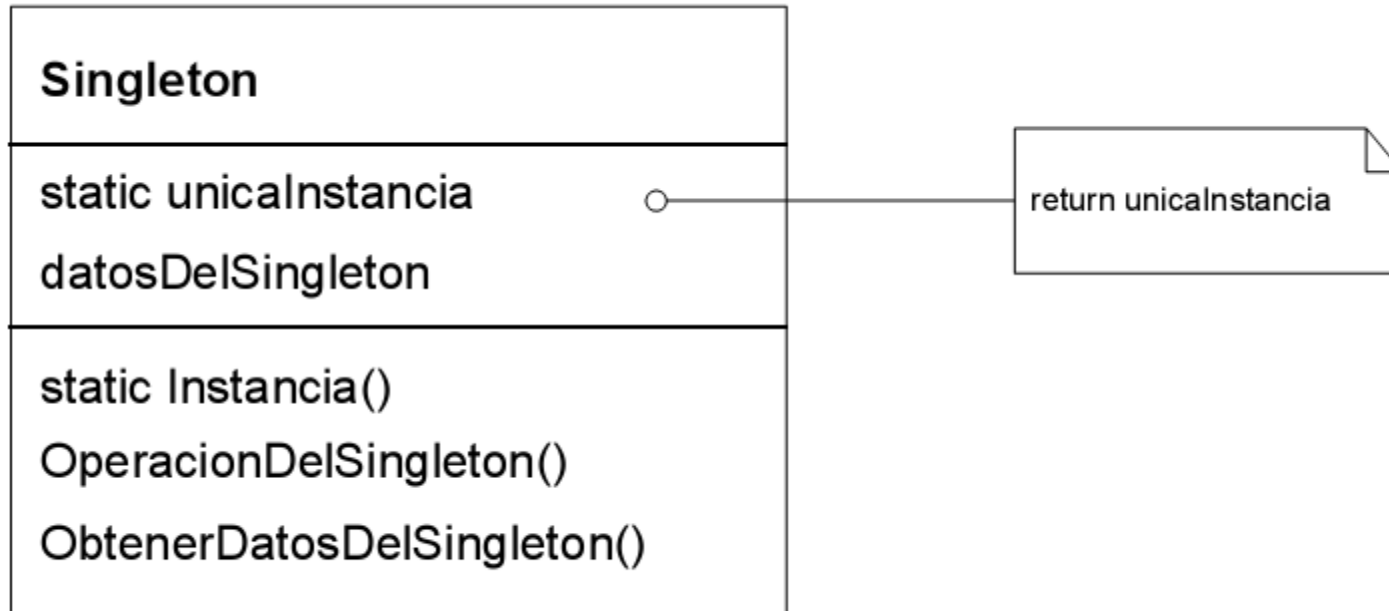
- Motivación

Algunas clases sólo pueden tener una instancia. Una variable global no garantiza que sólo se instancia una vez.

- Aplicabilidad: (utilizar cuando... )

Debe haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido.

# SINGLETON – Estructura



- **Participante:**  
Singleton- define un método instancia que permite que los clientes accedan a su única instancia. Instancia es un método de clase estático.
- **Colaboración:**  
Los clientes acceden a una instancia de un singleton exclusivamente a través de un método instancia de este.

# SINGLETON –Ventajas e inconvenientes

- Acceso controlado a la única instancia.
- Espacio de nombres reducido: no hay variables globales.
- Puede adaptarse para permitir más de una instancia.
- Puede hacerse genérica mediante *template* (*patrón de diseño*).

# SINGLETON – Implementación

- La clase se escribe de manera que solo se pueda crear una instancia.
- Se oculta la operación de creación de la instancia tras un método de clase de que garantice que solo se crea una única instancia.

# SINGLETON – Implementación

- Este método tiene acceso a la variable que contiene la instancia y asegura que la variable esta inicializada con dicha instancia antes de devolver su valor.
- Se utilizan punteros static que hacen referencia a la propia clase. Y a través de un método static capturamos la instancia.
- El constructor de la clase se protege para tener que utilizar el método getInstance().

# Abstract Factory - DEFINICIÓN

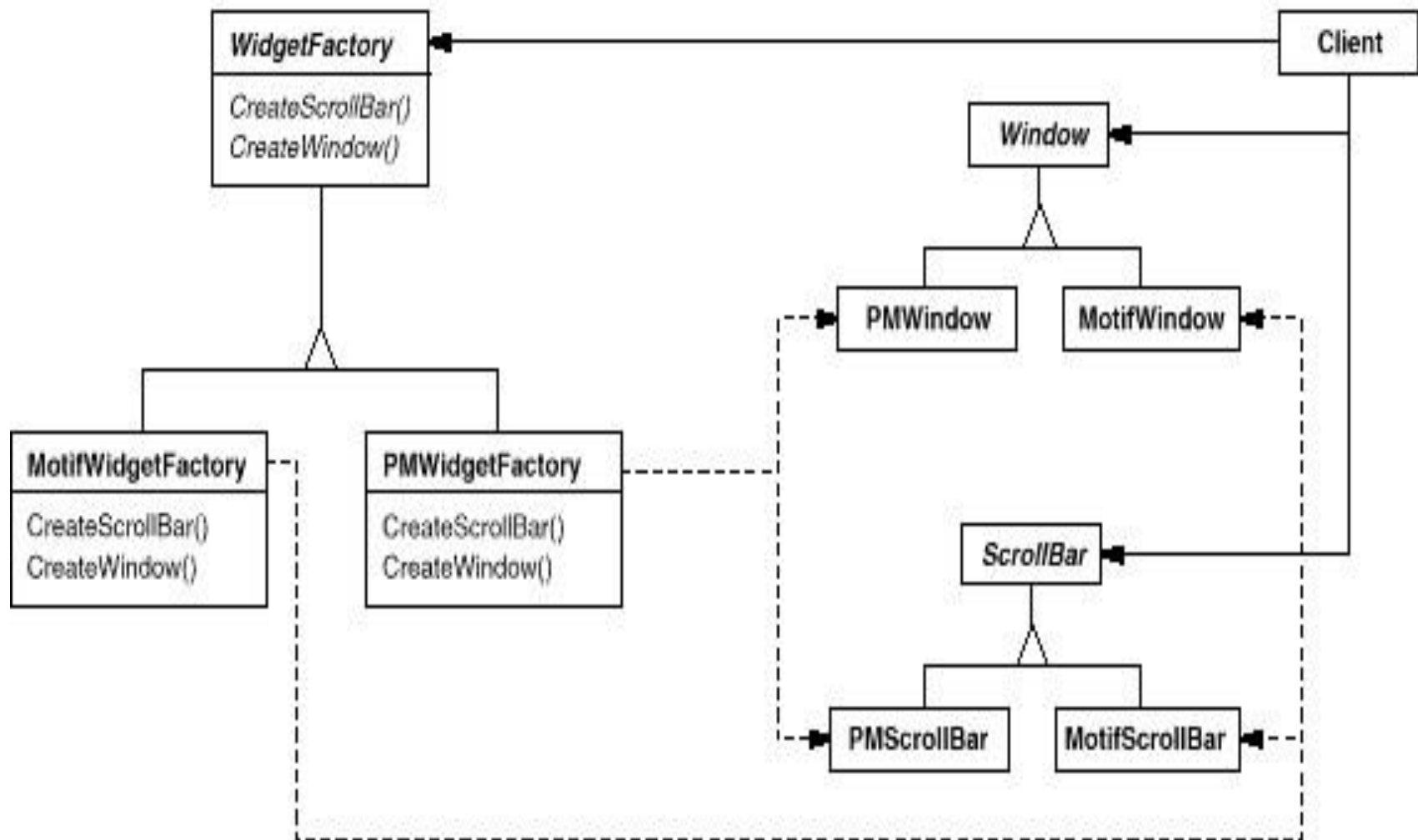
- El patrón Abstract Factory ofrece una interfaz para la creación de familias de productos relacionados o dependientes sin especificar las clases concretas a las que pertenecen.
- Look-And-Feel múltiple: diseño de GUI en entorno de ventanas.
- Su objetivo es soportar múltiples estándares (MS-Windows, Motif, Open Look,...).
- Extensible para futuros estándares.
- Restricciones: cambiar el Look-and-Feel sin recompilar y cambiar el Look-and-Feel en tiempo de ejecución.

# Ejemplo

- Ejemplo de partida: Creación de Widgets en aplicación cliente.
- Un armazón para interfaces gráficas que soporte distintos tipos de presentación (look-and-feel).
- Las aplicaciones cliente no deben cambiar porque cambie el aspecto de la interfaz de usuario.
- Los clientes no son conscientes de las clases concretas, sino sólo de las abstractas.
- Sin embargo los WidgetFactory imponen dependencias entre las clases concretas de los Widgets que contiene.



# Ejemplo



# Ejemplo

- **Problema:** Las aplicaciones clientes NO deben crear sus widgets para un Look-and-Feel concreto.

```
Scrollbar *sb = new MotifScrollbar();  
Menu *menu = new MotifMenu();
```

- **Solución:** Abstraer el proceso de creación de widgets. En lugar de:

```
Scrollbar *sb = new MotifScrollbar();
```

**Usar:**

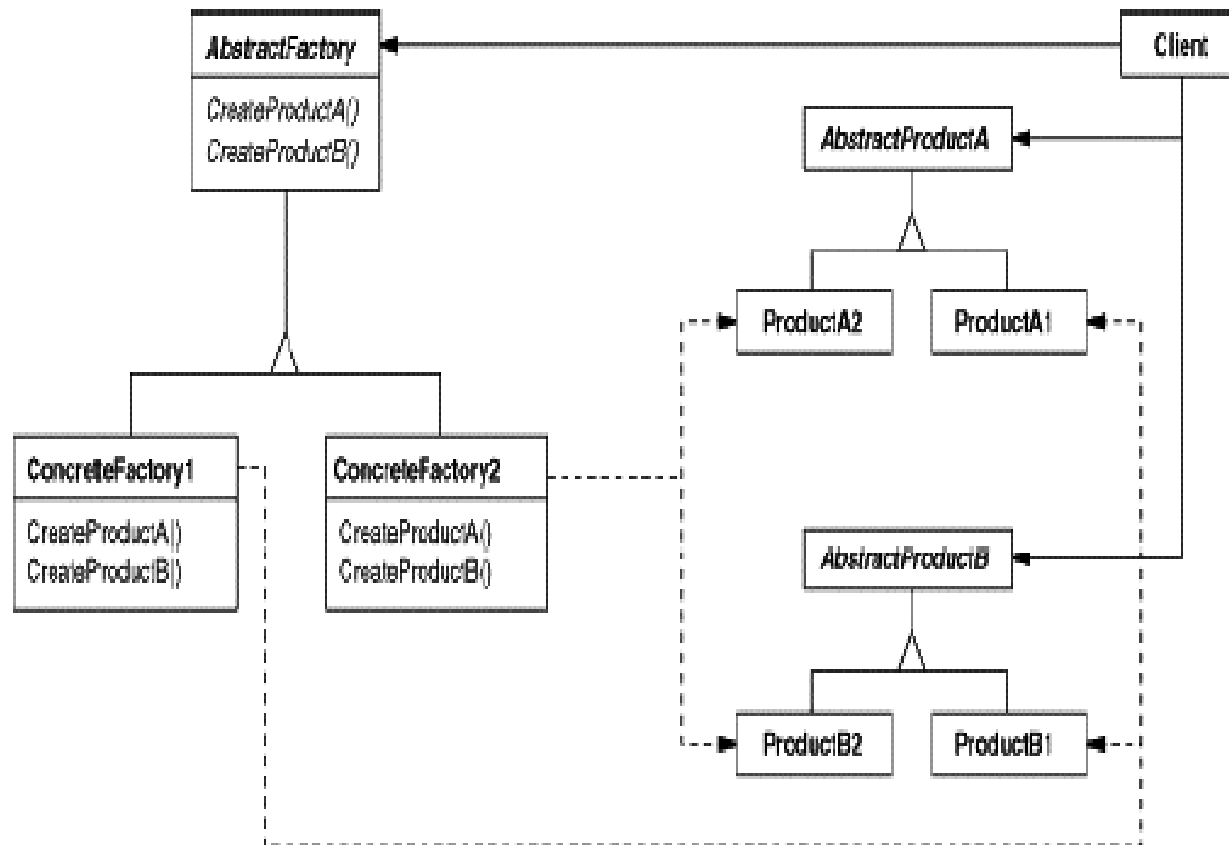
```
Scrollbar *sb = WidgetFactory->CreateScrollbar();
```

– WidgetFactory **será una instancia de**  
MotifWidgetFactory

# Aplicaciones

- Un sistema debe ser independiente de los procesos de creación, composición y representación de sus productos.
- Un sistema debe ser configurado con una familia múltiple de productos.
- Una familia de productos relacionados se ha diseñado para ser usados conjuntamente (y es necesario reforzar esta restricción).
- Se quiere proporcionar una librería de productos y no revelar su implementación (simplemente revelando sus interfaces).

# Estructura



# Participantes

- **AbstractFactory** (WidgetFactory): Declara un interfaz para las operaciones de creación de objetos de productos abstractos.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory): Implementa las operaciones para la creación de objetos de productos concretos.
- **AbstractProduct** (Window, ScrollBar): Declara una interfaz para los objetos de un tipo de productos.
- **ConcreteProduct** (MotifWindow, MotifScrollBar): Define un objeto de producto que creará la correspondiente *Concrete Factory*, a la vez que implementa la interfaz de *AbstractProduct*.
- **Client**: Usa solamente las interfaces declaradas por las clases *AbstractFactory* y *AbstractProduct*.

# Ejemplo

- Disponemos de los siguientes PRODUCTOS:
  - 1) La clase TV, que tiene dos hijas: Plasma y LCD.
  - 2) La clase Color, que tiene dos hijas: Amarillo y Azul.



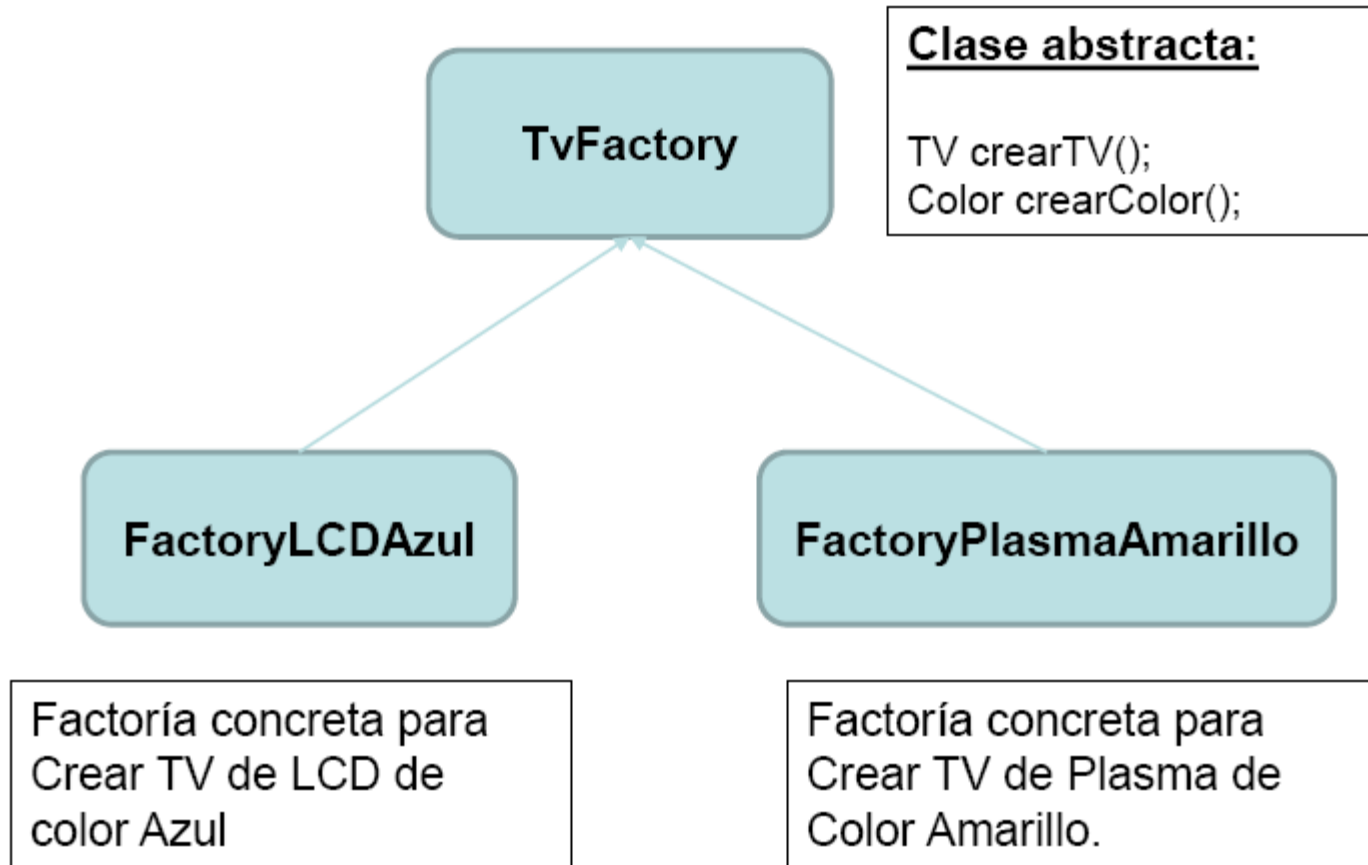
TV y Color pueden ser clases abstractas.

Color añade el método **abstracto**: `colorear(TV tv);`

# Situación

- La Empresa se dedica a darle un formato estético específico a los televisores LCD y Plasma.
- Se ha decidido que todos los **LCD que saldrán al mercado serán azules y los plasma serán amarillos.**
- Ahora bien, una solución simple sería en la clase Azul colocar el LCD y en la clase Amarillo colocar el Plasma y todo funcionaría.
- Pero esto puede cambiar.

# Las Fábricas





# Colaboraciones

- Una única instancia de cada **ConcreteFactory** es creada en tiempo de ejecución.
- **AbstractFactory** delega la creación de productos a sus subclases **ConcreteFactory**.

# Consecuencias

- Ventajas:
  - Aísla las clases de implementación: ayuda a controlar los objetos que se creen y encapsula la responsabilidad de creación.
  - Hace fácil el intercambio de familias de productos sin mezclarse, permitiendo configurar un sistema con una de entre varias familias de productos:  
cambio de factory  $\Rightarrow$  cambio de familia.
  - Fomenta la consistencia entre productos.

# Consecuencias

- Desventajas:
  - Puede ser difícil incorporar nuevos tipos de productos (cambiar **AbstractFactory** y sus factorias concretas).
  - Posible Solución:
    - Pasarle un parámetro a los métodos de creación de productos  $\Rightarrow$  clase abstracta común  $\Rightarrow$  necesidad de downcast  $\Rightarrow$  solución no segura.

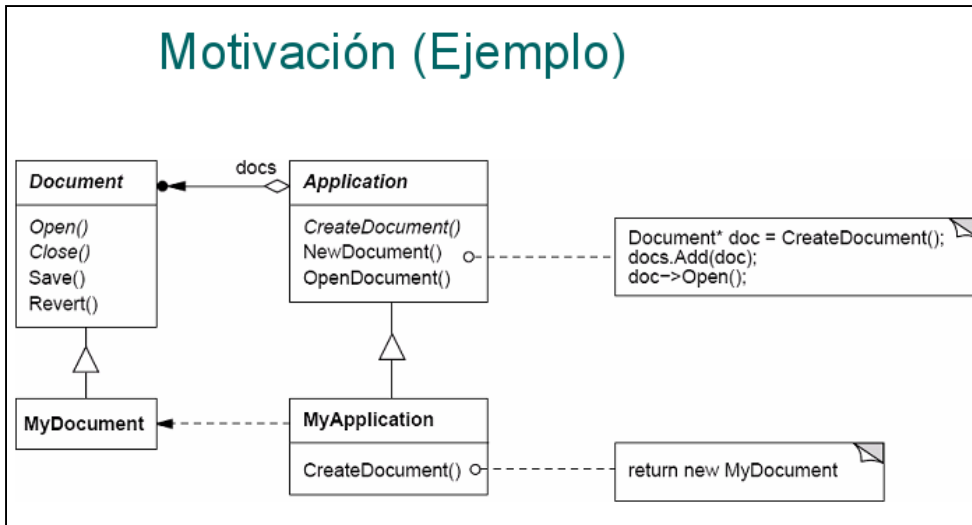
# Detalles de Implementación

- Factorías como singletons
  - Sólo una instancia de ConcreteFactory por familia de productos.
- Definir factorías extensibles
  - Añadiendo un parámetro en las operaciones de creación que indique el tipo de objeto a crear: mas flexible y menos seguro.
- ¿Cómo crear los productos? Utilizando un *Factory Method* para cada producto.

# FACTORY METHOD - Definición

- **FACTORY METHOD:** Define la interfaz de creación de un cierto tipo de objeto, permitiendo que las subclases decidan que clase concreta necesitan instancias.
- También conocido como: Método de fabricación.
- Es una simplificación del Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos; según usemos una u otra hija de esta clase abstracta, tendremos uno u otro comportamiento.

# FACTORY METHOD – Motivación



- Sea un framework para la construcción de editores de documentos de distintos tipos.

- ¿Cuál es el problema?

Que se nos presenta el dilema de que el framework es quien sabe, a través de su clase `Application`, cuándo se debe crear un nuevo documento, pero no sabe qué documento crear.

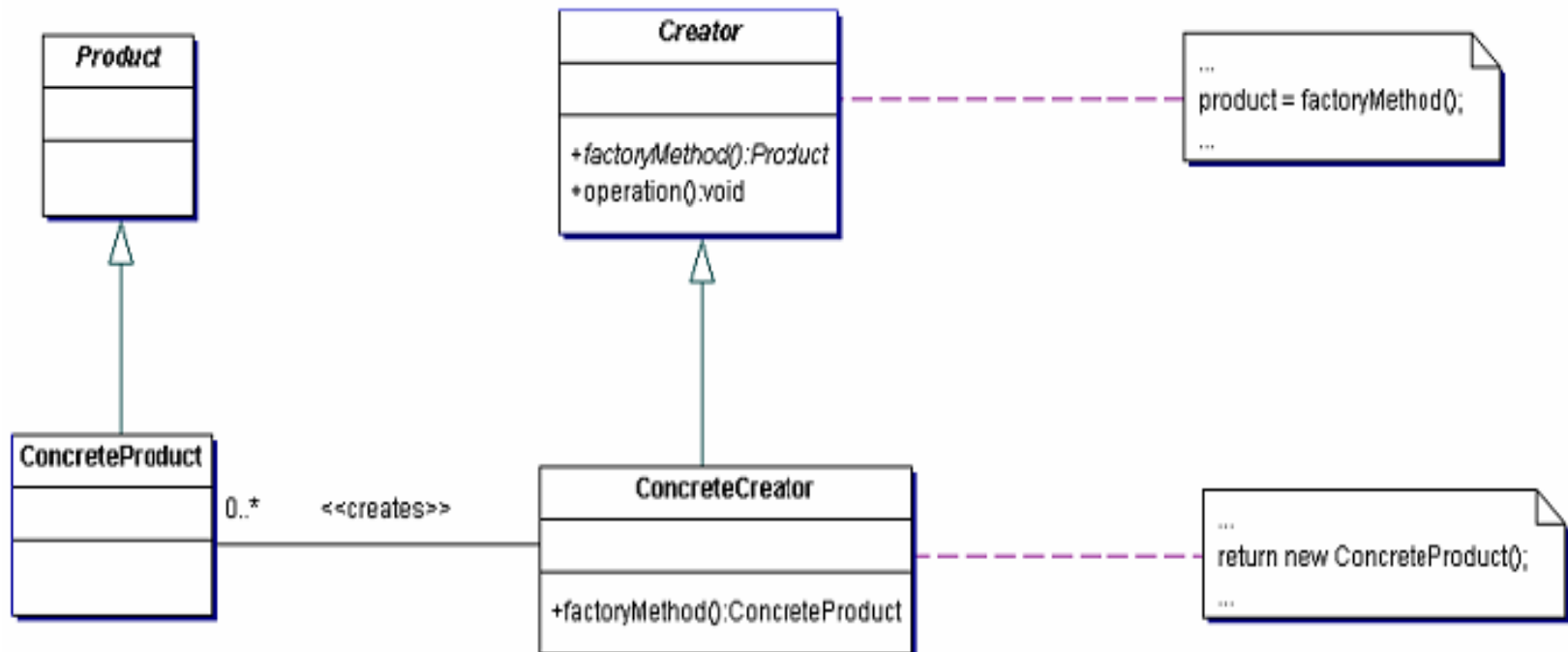
- Solución:

Encapsular el conocimiento de cuál es la subclase concreta del documento a crear y mover ese conocimiento fuera del framework

# FACTORY METHOD – Aplicabilidad

- Una clase no puede anticipar la clase de objeto que debe crear.
- Una clase quiere que sus subclases especifiquen los objetos a crear.
- Hay clases que delegan responsabilidades en una o varias subclases.

# FACTORY METHOD – Estructura





# FACTORY METHOD – Participantes

- **Product:** Define la interfaz de los objetos creados por el método de fabricación (FactoryMethod()).
- **Concret Product:** Implementa la interfaz Product.
- **Create:** Declara el método de fabricación, que devuelve un objeto de tipo product. Puede llamar a dicho método para crear un objeto producto.
- **ConcretCreator:** Redefine el metodo de fabricación para devolver un objeto concretProduct.

# Classes

```
class Product {  
    public:  
        Product();  
        virtual void operacion()=0;  
        ~Product();  
};
```

```
class ConcreteProduct: public Product {  
    public:  
        ConcreteProduct();  
        void operacion();  
        ~ConcreteProduct();  
};
```

```
class Creator {  
    public:  
        Creator();  
        virtual Product *factoryMethod()=0;  
        ~Creator();  
};
```

```
class ConcreteCreator: public Creator {  
    public:  
        ConcreteCreator();  
        Product *factoryMethod();  
        ~ConcreteCreator();  
};
```

```
Product *ConcreteCreator::factoryMethod(){  
    return new ConcreteProduct();  
}
```

# FACTORY METHOD – Colaboración

- Colaboración: El creador se apoya en sus subclases para definir el método de fabricación que devuelve el objeto apropiado.

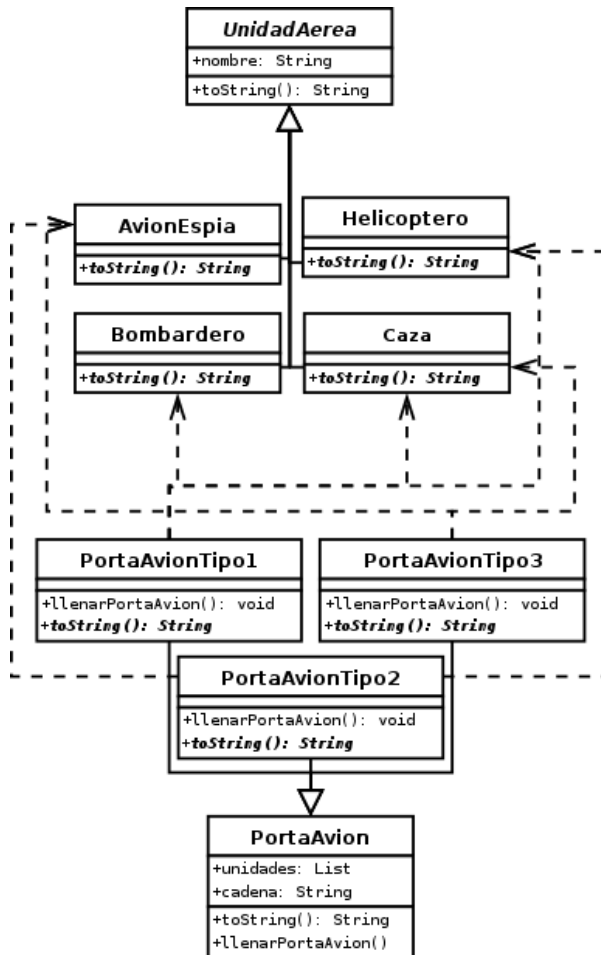
# FACTORY METHOD –Ventajas e inconvenientes

- Elimina la necesidad de introducir clases específicas en el código del creador. Solo maneja la interfaz Product, por lo que permite añadir cualquier clase ConcretProduct definida por el usuario.
- Tener que crear una subclase de Creator en los casos en los que esta no fuera necesaria de no aplicar el patrón.

# FACTORY METHOD – Implementación

- Convenios de nominación: Se suele usar el prefijo *create*.
- Diversas clases se suele implementar como un Singleton.
- El Factory Method parametrizado (Una variante del patrón que permite al método de fabricación crear varios tipos de productos) protege de la variación del tipo de producto concreto.

# FACTORY METHOD – Ejemplo

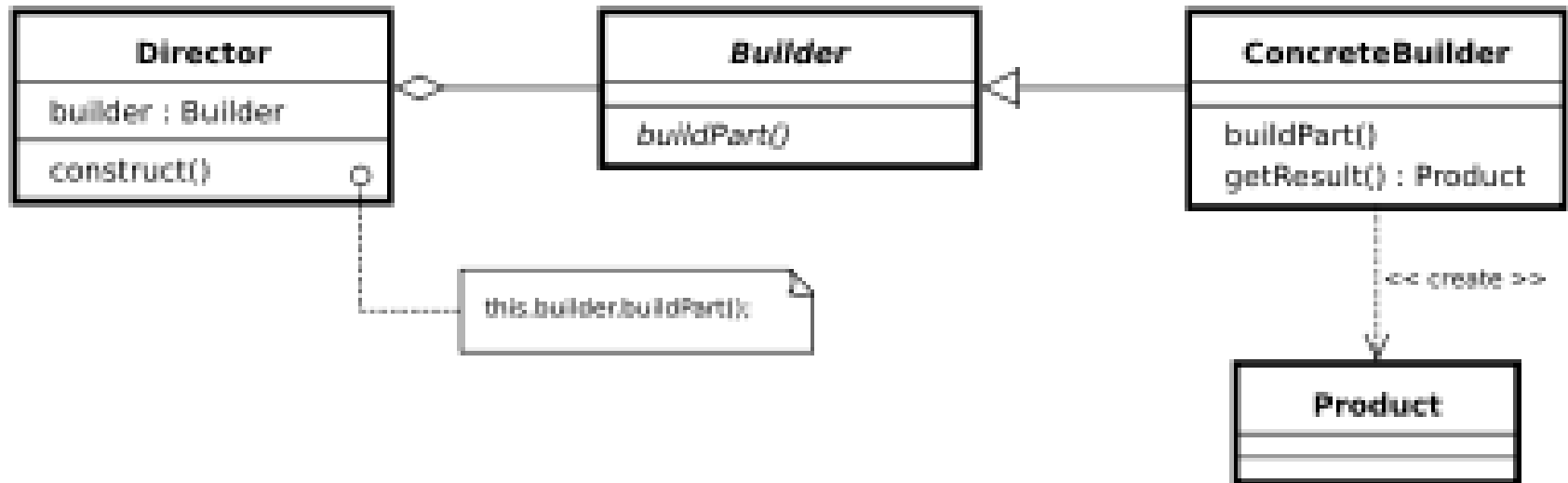


- En este ejemplo de porta-aviones simplemente creamos 3 porta-aviones de 3 tipos distintos, y nos olvidamos del tipo de aviones que tendrá cada uno de ellos dejando a las subclases barcoX decidir que tipo de aviones tendrán, ya no tenemos la responsabilidad de saber que aviones tiene cada barco .

# BUILDER

- Objetivo: Separa la construcción de un **objeto complejo** de su representación, de forma que puedan crearse distintas representaciones.
- También se le conoce con el nombre de constructor.

# Diagrama



**Builder:** Es la interfaz abstracta para construir un producto final. Podemos tener varios constructores (ConcreteBuilder)

**Product:** Es el producto final que se construye.

**Director:** Es la clase que se encarga de construir el producto final. utilizando un constructor.



# Aclaraciones Builder

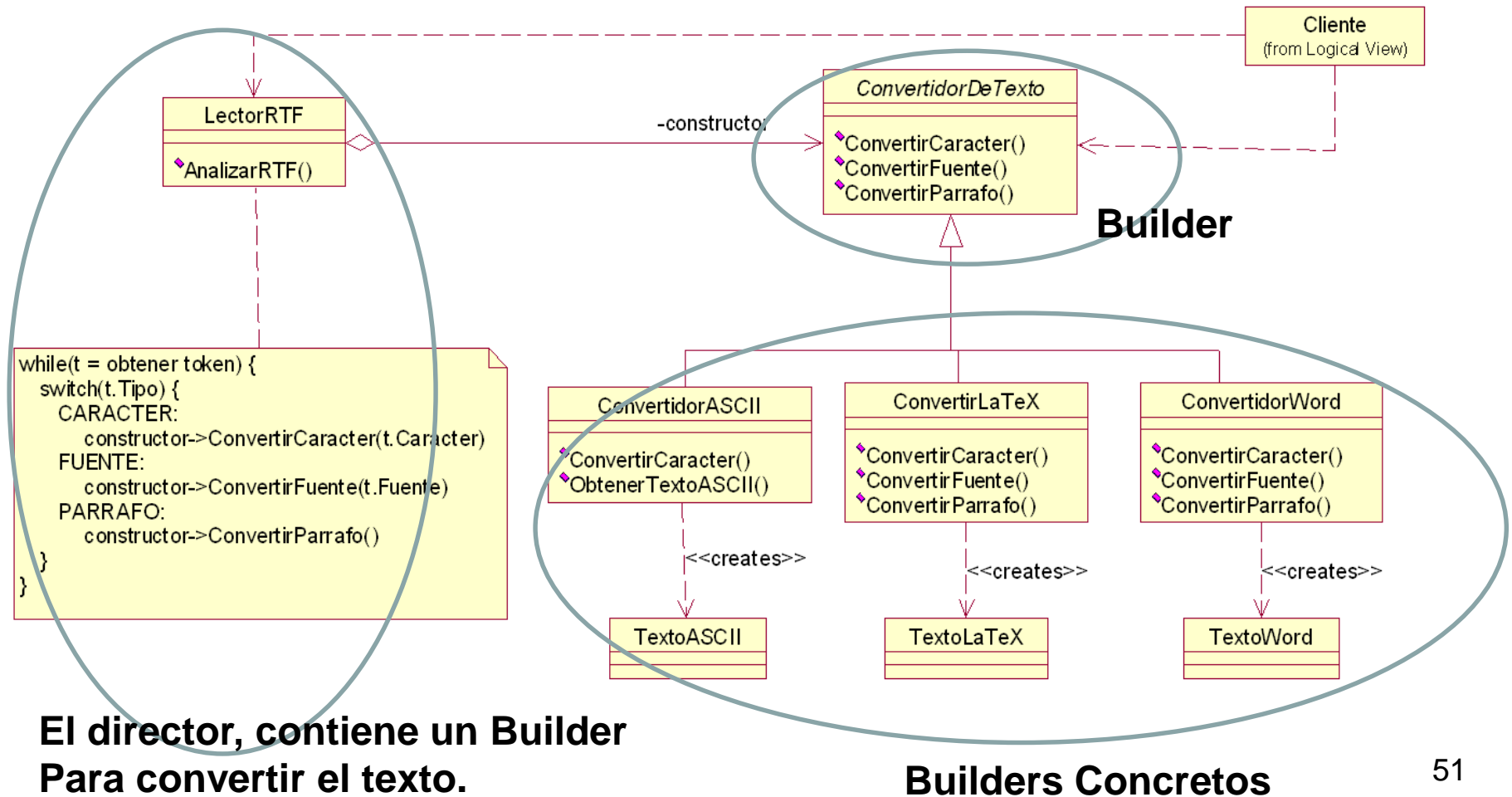
- El **producto final es complejo** y se divide en diversas parte (proporciona **métodos set**) y aplica **composición** con cada una de sus partes.
- El **builder**: (es **abstracta**) declara todas las partes que se pueden construir del producto final.
  - Proporciona métodos abstractos para crear cada parte o pieza del producto final
- El director

# Ejemplo 1

- Producto final: **Pizza**
  - Partes de la pizza: masa, salsa y relleno.
- **Builder:**
  - **Att. Protected: Pizza**
    - **Método: getPizza()**
  - void buildMasa()=0;
  - void buildSalsa()=0;
  - void buildRelleno()=0;
- A partir del Builder podemos tener distintos tipos de Builder (BuilderPizzaHawai, BuilderPizzaEspecial, ...) que generan distintos tipos de Pizza pero cada uno con un tipo de salsa, de relleno o de masa.
- El **Director**, sería el **cocinero que fabrica la Pizza**.
  - Para ello tiene un atributo un Builder (concreto). Que lo utiliza para fabricar la pizza.

# Ejemplo 2

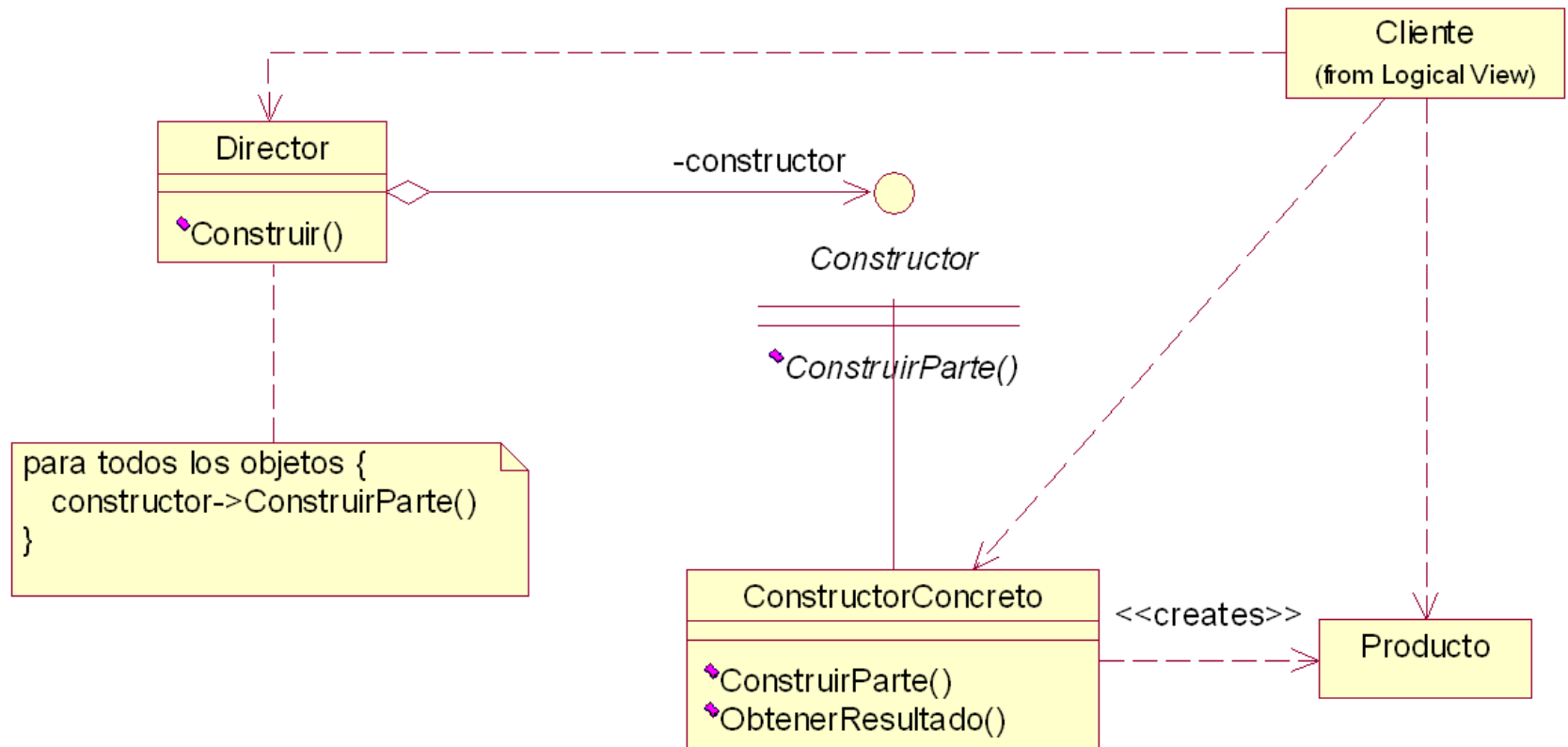
Convertir un fichero con formato RTF a otros formatos.



# Aplicabilidad

- El algoritmo para crear un objeto complejo debiera ser independiente de las partes que se compone dicho objeto y de cómo se ensamblan.
- El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.

# Estructura



# Participantes

- **Constructor** (ConvertidorDeTexto):
  - Especifica una interfaz abstracta para crear las partes de un objeto Producto.
- **ConstructorConcreto** (ConvertidorASCII, ConvertidorTeX, ConvertidorUtilDeTexto):
  - Implementa la interfaz Constructor para construir y ensamblar las partes del producto.
  - Define la representación a crear.
  - Proporciona una interfaz para devolver el producto (ObtenerTextoASCII, ObtenerUtilDeTexto).
- **Director**: (LectorRTF):
  - Construye un objeto usando una interfaz Constructor.
- **Producto**: (TextoASCII, TextoTeX, UtilDeTexto):
  - Representa el objeto complejo en construcción. El ConstructorConcreto construye la representación interna del producto y define el proceso de Ensamblaje.
  - Incluye las clases que definen sus partes constituyentes, incluyendo interfaces para ensamblar las partes del resultado final.

# Consecuencias

- Permite variar la representación interna de un producto.
- Aísla el código de construcción y de representación. Encapsula como se crean los objetos.
- Proporciona un control mas fino sobre el proceso de construcción.

# Consecuencias

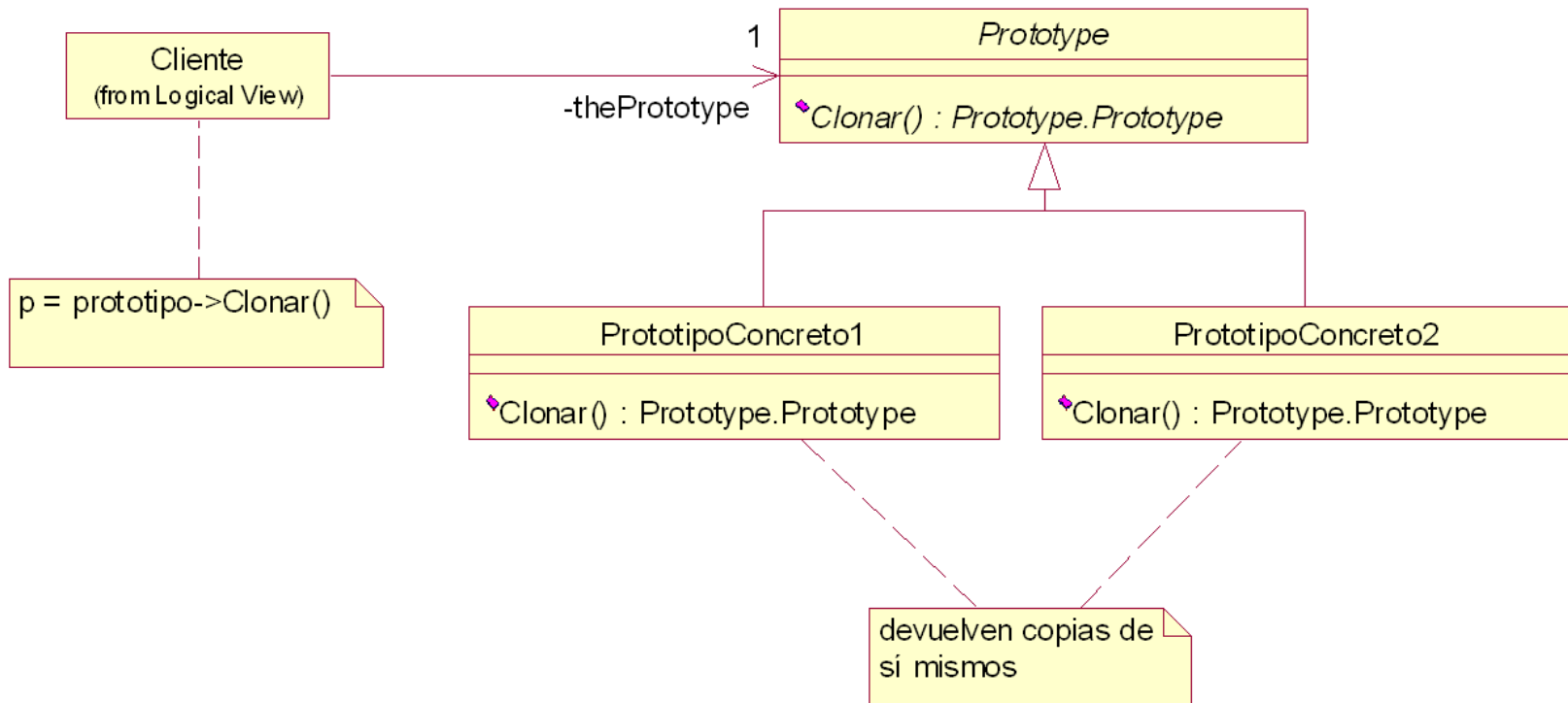
- El constructor principal no crea el objeto define las operaciones (que no se define virtuales puras para que las subclases implementen las que quieran) para crear un objeto pero son las subclases del constructor las que realmente hacen la construcción.



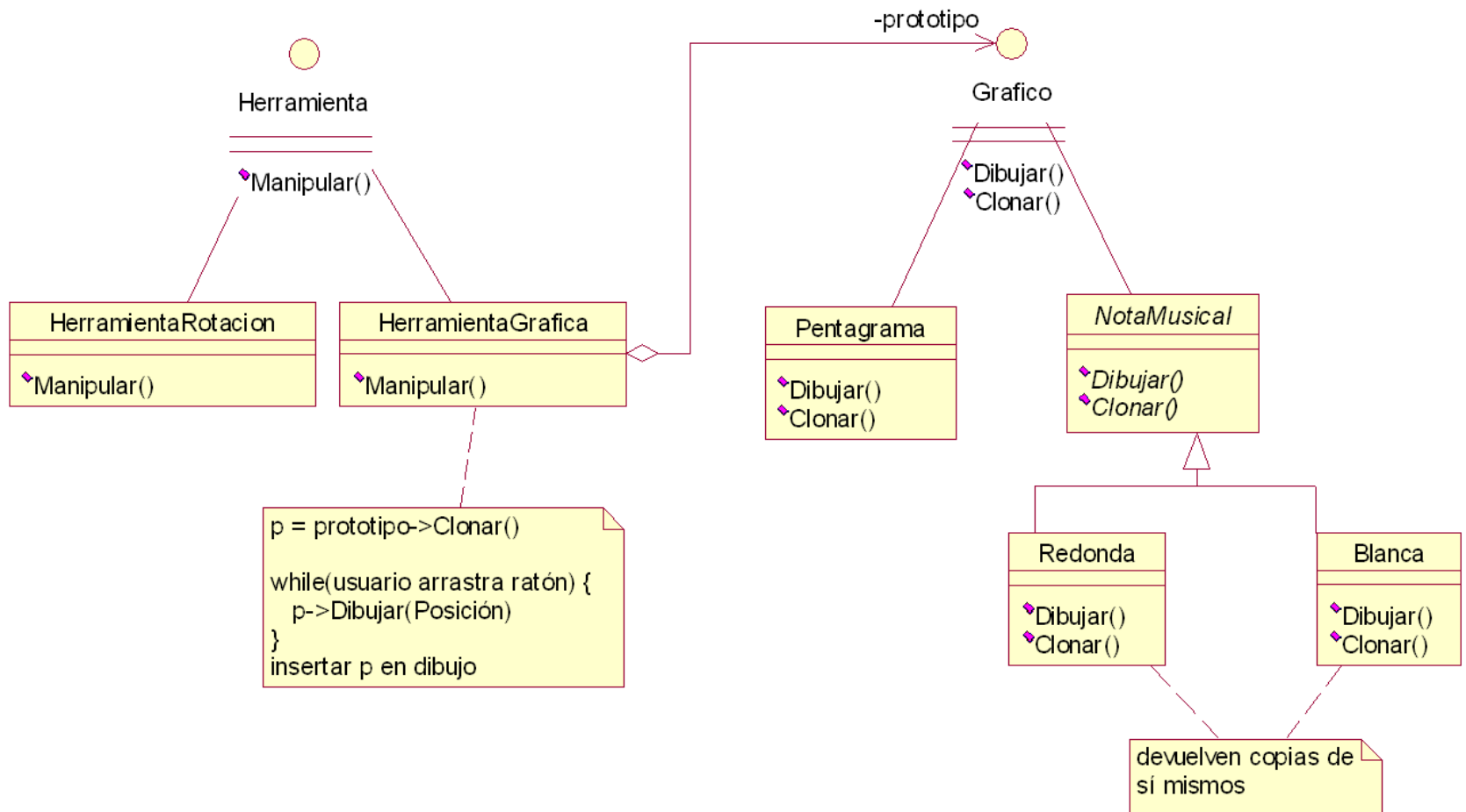
# PROTOTYPE

- Objetivo: Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.
- Nombres: Prototipo.
- Ejemplo: Editor de partituras musicales.

# Estructura



# Ejemplo



# Aplicabilidad

- Cuando las clases a instanciar sean especificadas en tiempo de ejecución.
- Para evitar jerarquías de clases de fábricas con jerarquía de clases de productos.
- Cuando las instancias de clases puedan tener un estado de entre un conjunto reducido.

# Participantes

- Prototipo (Gráfico):
  - Declara la interfaz para clonarse.
- Prototipo Concreto (Pentagrama, Redonda, Blanca):
  - Implementa una operación para clonarse.
- Cliente (Herramienta Gráfica):
  - Crea un nuevo objeto pidiéndole a un prototipo que se clone.
- ***Todos los prototipos se pueden almacenar en una clase Factoría y cuando hace falta un objeto se le solicita a la factoría (esta llamará al método clone del prototipo solicitado).***

# Consecuencias

- Añadir y eliminar productos en tiempo de ejecución. El cliente puede instalar y eliminar prototipos en tiempo de ejecución.
- Especificar nuevos objetos modificando valores. Para sistemas dinámicos permiten definir un comportamiento nuevo mediante la composición de objetos.
- Especificar nuevos objetos variando la estructura.
- Reduce la herencia: El patrón Factory Method suele producir una jerarquía de clases Creador que es paralela a la jerarquía de clases de productos.
  - Con el patrón prototype podemos clonar un prototipo en vez de decirle a un método de fabricación que cree un nuevo objeto.
- Configurar dinámicamente una aplicación con clases. Algunos entornos permiten cargar clases en tiempo de ejecución.

# Inconvenientes

- El principal inconveniente es que cada subclase del prototipo debe implementar la operación clonar, y la implementación puede ser compleja.

# Patrones Estructurales

Adapter

Facade

Decorator

Proxy

Composite

Bridge

FlyWeight



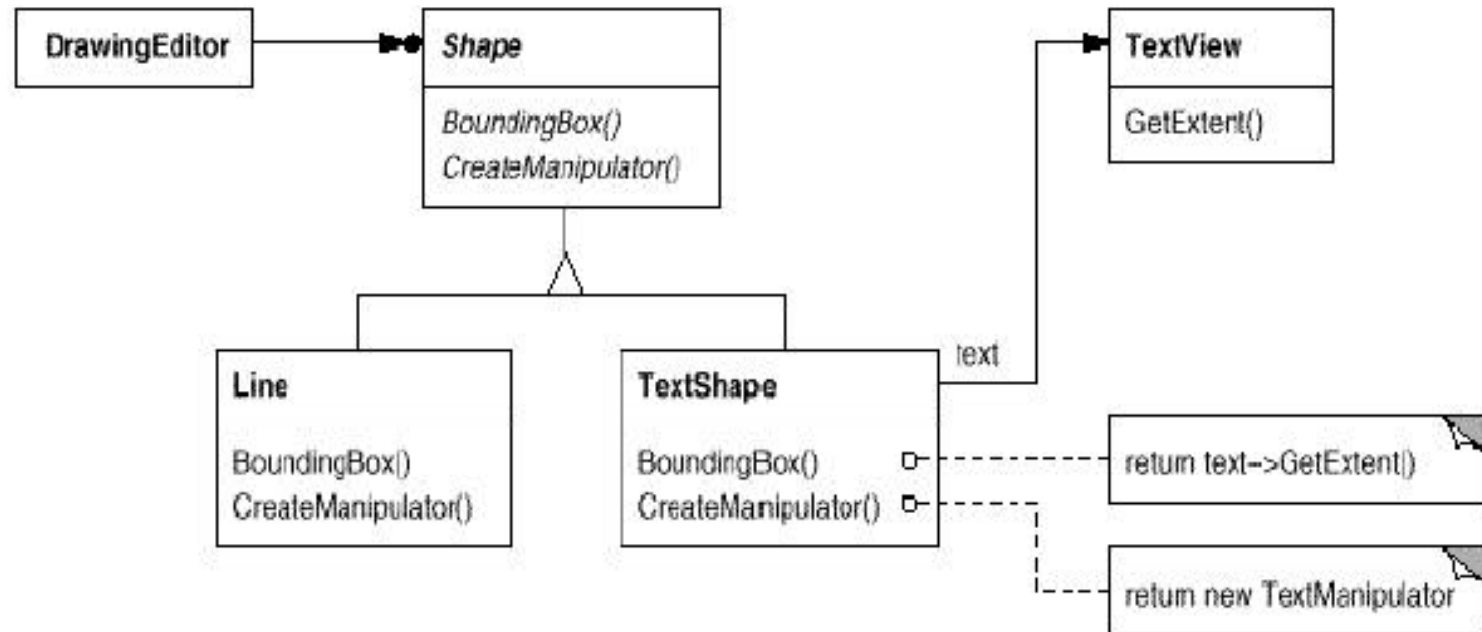
# Patrones Estructurales

- Los patrones Estructurales se ocupan de cómo se combinan las clases y los objetos para formar estructuras mas grandes.
  - Estructurales **de Clase**: hacen uso de la herencia para componer interfaces o implementaciones.
  - Estructurales **de Objetos**: describen formas de componer objetos para obtener una nueva funcionalidad. Añaden flexibilidad al poder cambiar la composición en tiempo de ejecución.

# Adapter

- Convierte la interfaz de una clase en la interfaz de otra que espera un cliente:
  - Muy útil si por ejemplo no se tiene acceso al código fuente de la interfaz inicial.
- Se puede implementar de dos formas:
  - Por Composición.
  - Por Herencia múltiple. (recibe el nombre de clase Adaptadora)
- Es uno de los patrones denominados “wrappers” → **Envoltorio**, (otro por ejemplo es el patrón **Decorator**)

# Ejemplo



“TextShape” es un adaptador para que un “TextView” se pueda considerar un “Shape”

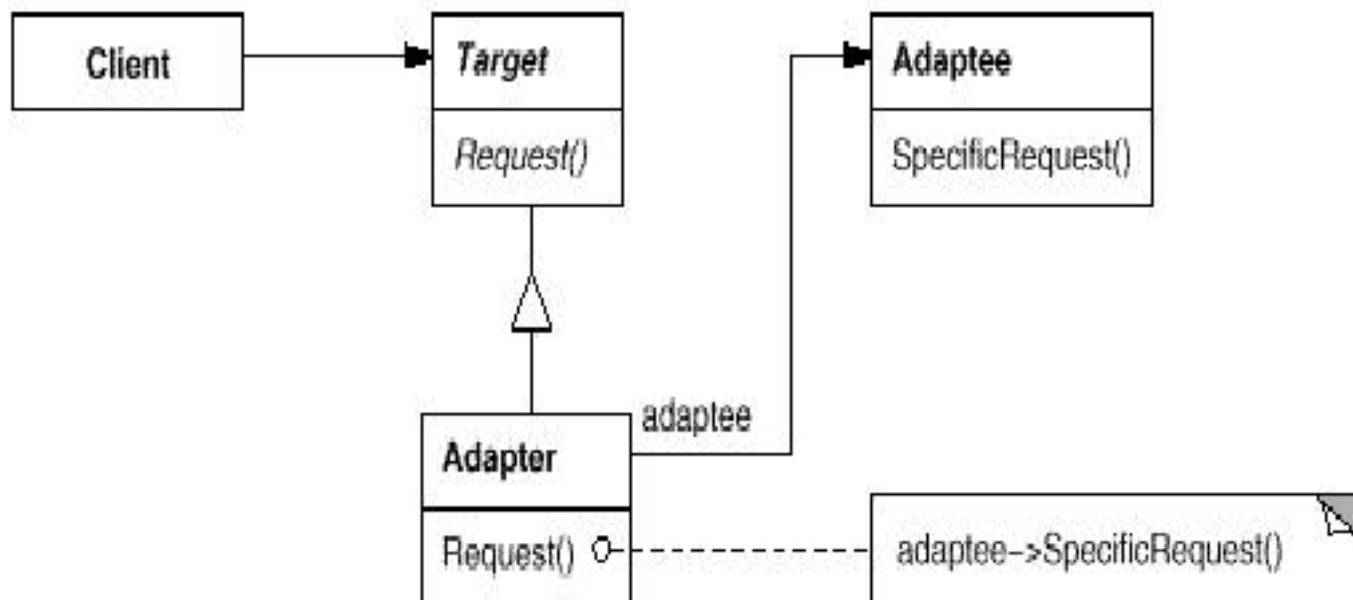
# Aplicabilidad

- Este patrón debería usarse 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 no relacionadas.

# Participantes

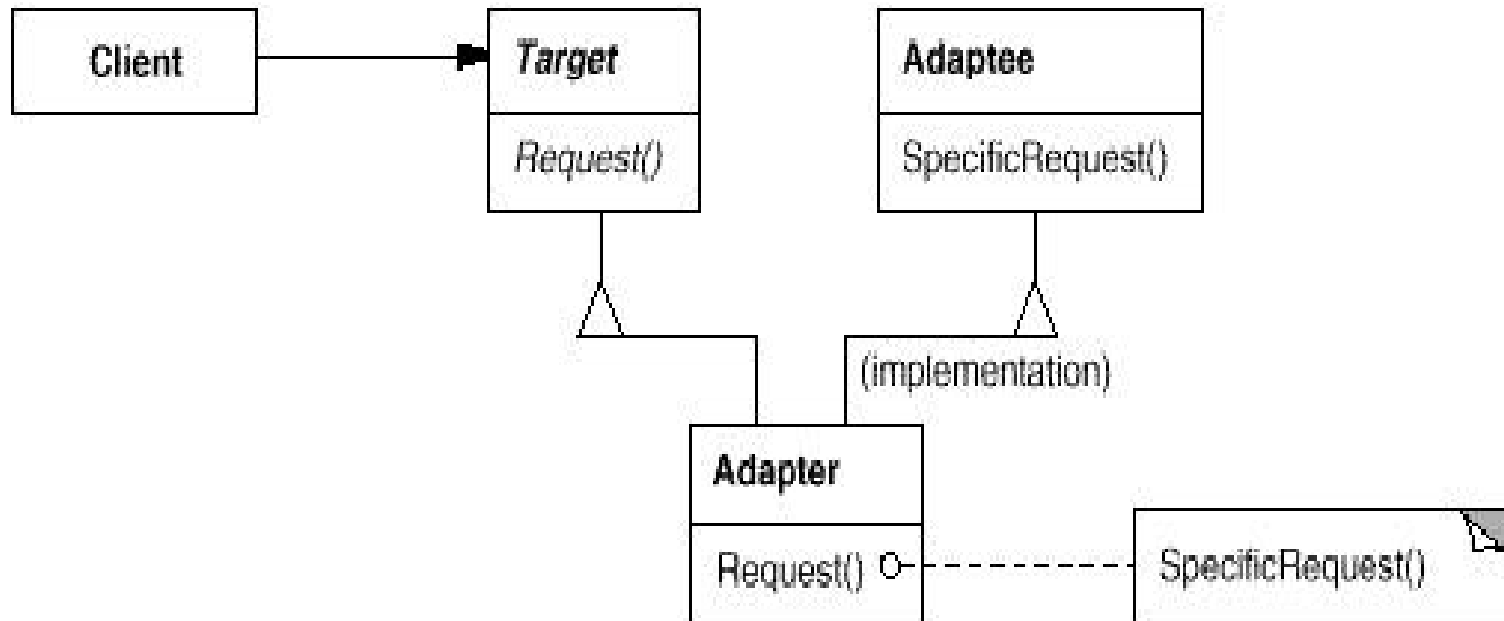
- Objetivo (Shape):
  - Define la interfaz específica del dominio que usa el cliente.
- Cliente (DrawingEditor):
  - Colabora con objetos que se ajustan a la interfaz Objetivo.
- Adaptable (TextView):
  - Define la interfaz existente que necesita ser adaptada.
- Adaptador (TextShape):
  - Adapta la interfaz de Adaptable a la interfaz Objetivo.

# Estructura I



Se puede utilizar “composicion”

# Estructura II



... O alternatively herencia múltiple

# Ejemplo

- Disponemos de una clase que representa un vector en 3D y necesitamos una clase VectorPlano que implemente una interface (clase con todos los métodos virtuales puros) para un Vector2D.
- Posibilidades:
  1. La clase VectorPlano hereda de Vector2D (public) y de Vector3D (private).
  2. La clase VectorPlano hereda de Vector2D y se compone de Vector3.



# Fachada

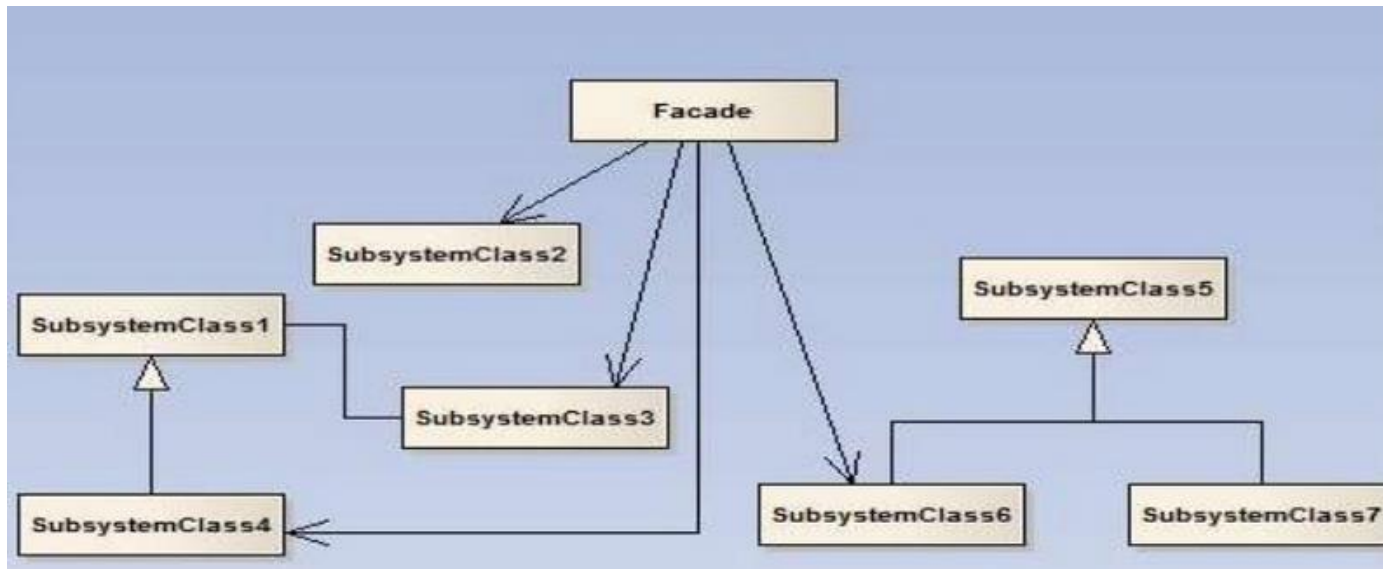
- Es otro patrón de diseño de tipo estructural.
- Sirve para proveer de una interfaz unificada sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces mas complejas.
- La fachada es un representante de un conjunto de objetos.
  - La fachada lleva a cabo sus responsabilidades reenviando mensajes a los objetos que representa.

# Para que se aplica

- Cuando un cliente necesita acceder a una parte de una funcionalidad de un sistema mas complejo.
- El problema se puede solucionar definiendo una interfaz que permita acceder a solamente esa funcionalidad.
- Se crea un código sencillo que interactúe entre el cliente y la librería interna. Hace las veces de intermediario.

# Estructura

- Simplifica el acceso a un conjunto de clases proporcionando una única clase que todos utilizan para comunicarse con dicho conjunto de clases.



# Ejemplo

- Para otorgar una Hipoteca a un cliente hay que realizar una serie de comprobaciones:
  - Comprobar en el banco si el cliente tiene ahorros suficientes.
  - Comprobar en el sistema de créditos si hay crédito positivo.
  - Comprobar en el sistema de préstamos si el cliente tiene impagos.

# Aplicación

- El patrón Fachada se encarga de realizar todas las comprobaciones necesarias con todos los sistemas necesarios.
- El cliente interactúa con la Fachada a través de un simple método que le dice si se le concede o no la Hipoteca.

EJEMPLO FACHADA

# Utilizar Fachada cuando

1. Se quiera proporcionar una interfaz sencilla para un subsistema complejo.
2. Se quiera desacoplar un subsistema de sus clientes y de otros subsistemas, haciéndolo mas independiente y portable.
3. Se quiere dividir los sistemas en niveles: las fachadas serían el punto de entrada de cada nivel.

# Decorator

- Se conoce también como **Wrapper**.
- Motivación:
  - Añadir responsabilidades a un objeto, pero no a toda la clase.
  - La herencia no es flexible, añade nuevas responsabilidades pero de una forma estática.
  - La solución consiste en rodear a un objeto con otro objeto que es al que se le añade la responsabilidad.
  - El nuevo objeto es el decorador.

# Se aplica cuando

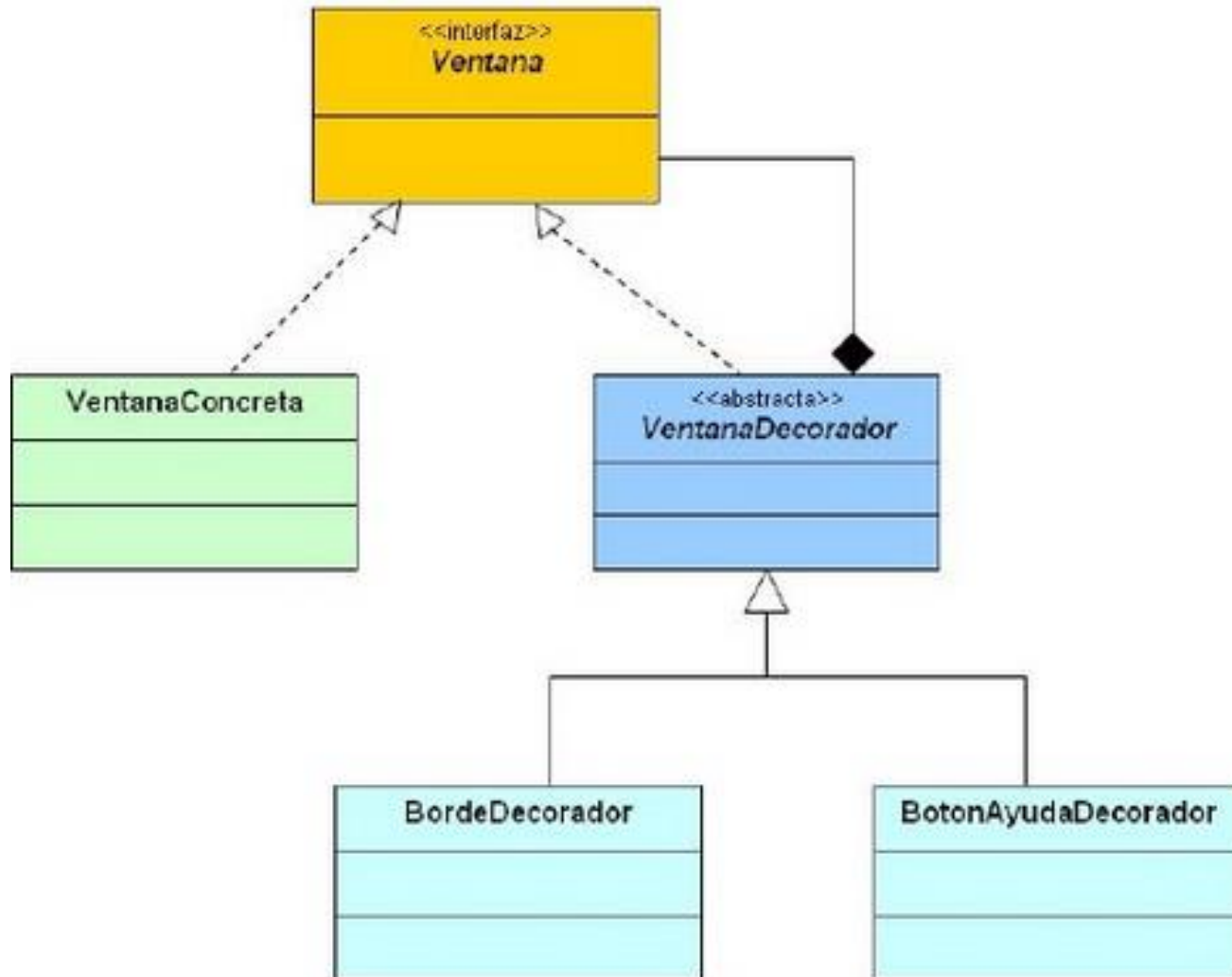
- Adicionar responsabilidades a objetos individuales dinámicamente sin afectar a otros objetos.
- Agregar responsabilidades que pueden ser retiradas.
- Cuando no es práctico adicionar responsabilidades por medio de la herencia.



# Ejemplo

- Partimos de una clase que representa una **Ventana gráfica**.
- Queremos añadir una funcionalidad que sea un borde alrededor.
  - Aplicando herencia podríamos obtener **VentanaConBorde**.
- Si después queremos tener la posibilidad de añadir un botón de ayuda podríamos crear la clase **VentanaConBotonDeAyuda**.
  - No cubre la posibilidad de tener ventanas con borde y con botón, podríamos seguir heredando, pero tampoco cubrimos todas las posibilidades: (multiplica las clases).
    - **Ventana, VentanaConBorde, VentanaConBotonDeAyuda y VentanaConBordeYBotonDeAyuda.**
    - **Para  $n$  funcionalidades se necesitan  $2^n$  clases.**

# Solución



# Solución

- Se crea a partir de una ventana la subclase abstracta `VentanaDecorator` y, heredando de ella `BordeDecorator`, y `BotonDeAyudaDecorator`.
- `VentanaDecorator` encapsula el comportamiento de `Ventana` y utiliza composición recursiva para añadir todos los decoradores que necesitemos.

# Solución

- En el main:
  - Podemos crear ventana de muchos tipos.
  - Aplicando composición recursiva.

```
BordeDecorator* ventanaConDobleBorde2 = new BordeDecorator(new  
BordeDecorator(new Ventana()));  
ventanaConDobleBorde2->Dibujar();
```

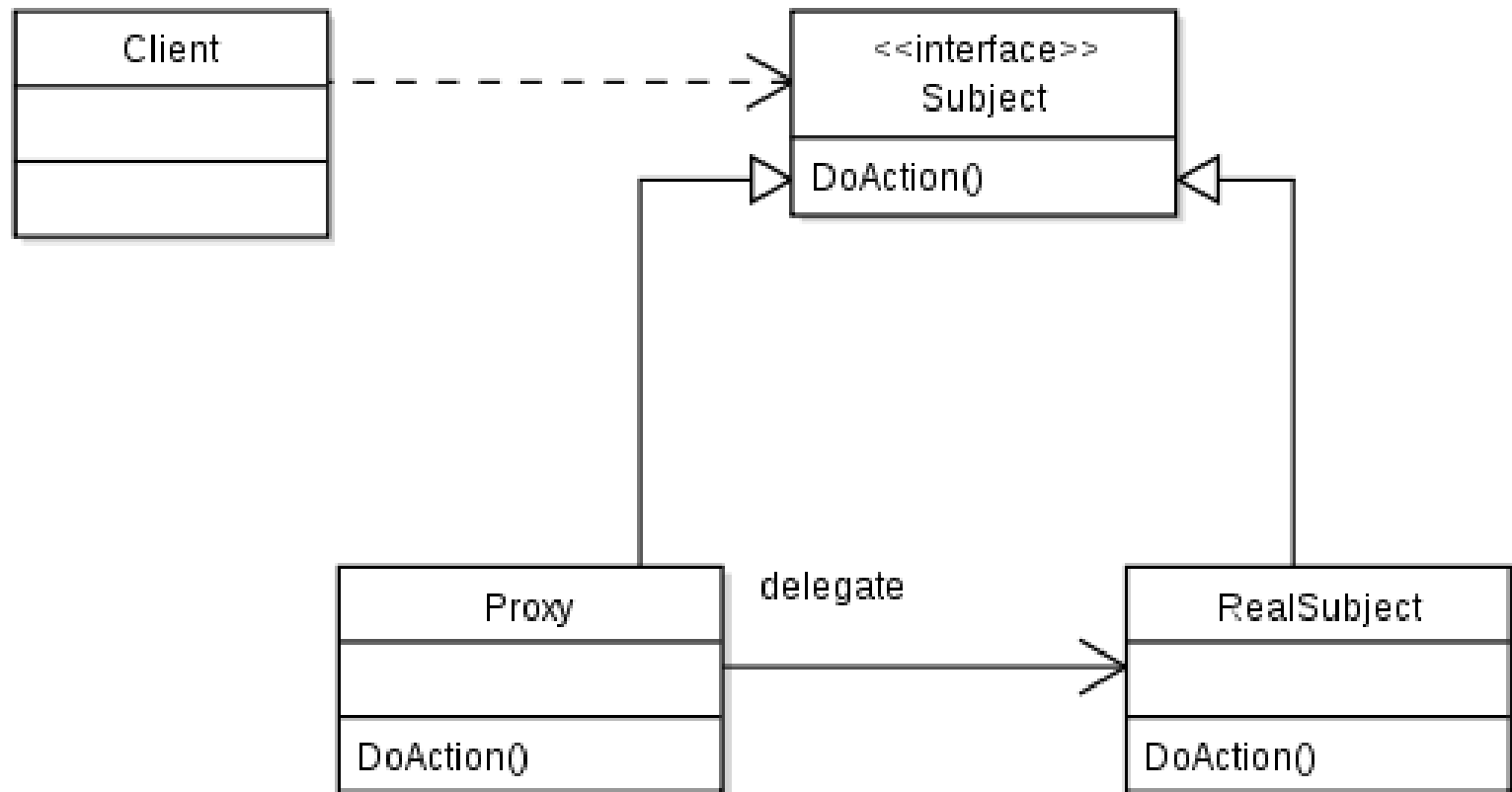
# Proxy (Apoderado)

- Proporciona un representante o sustituto de otro objeto para controlar el acceso a este.
- Controlar el acceso a un objeto puede estar motivado por:
  - El coste de crear el objeto real. El proxy lo que hace es **demorar** la **creación** de este.
  - Por motivos de **seguridad**.
  - Para **interceptarlo** para lanzar alguna operación adicional como ocurre con la **AOP** (Programación Orientada a Aspectos).

# Tipos de Proxies

- **Remotos:**
  - Son responsables de codificar una petición y sus argumentos para enviar la petición codificada al sujeto real.
- **Virtuales:**
  - Pueden guardar información adicional sobre el sujeto real, por lo que puede retardar el acceso al mismo.
- **De protección:**
  - Comprueban que el llamador (el Cliente) tenga los permisos necesarios.

# Estructura



# Participantes

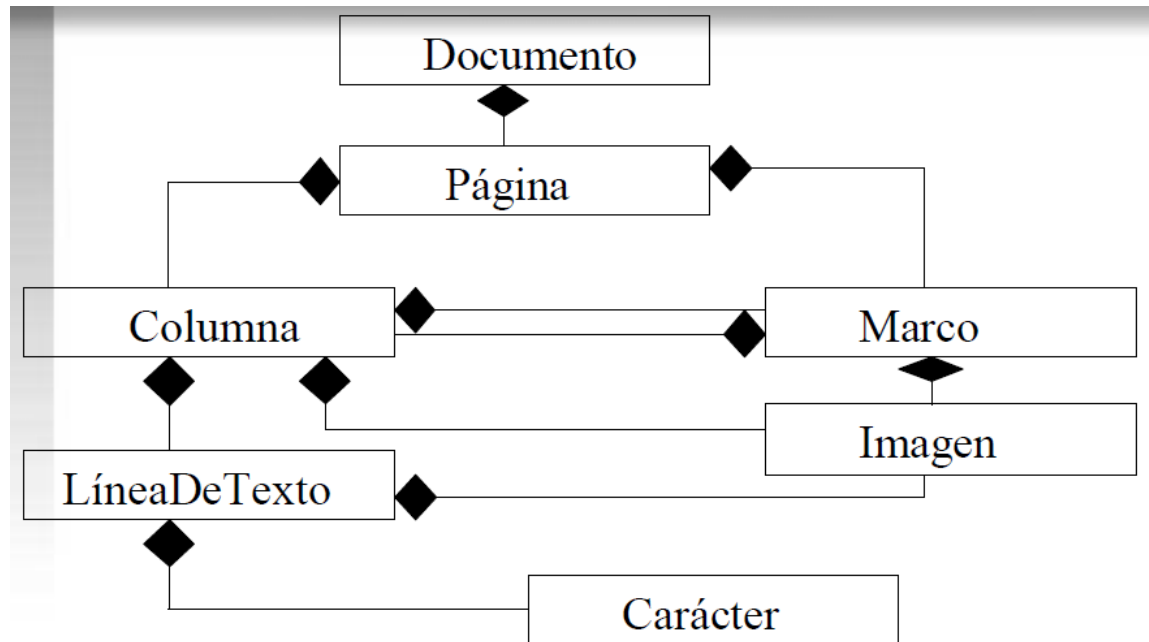
- **Subject:**
  - Representa una interface con las operaciones que implementarán el objeto Real y el Proxy.
- **Proxy:**
  - Mantiene una referencia al objeto Real así como los mismos atributos. De tal forma que controla la creación de este.
- **RealSubject:**
  - El objeto real, el que queremos demorar la creación, proteger, etc.



# Composite: Definición

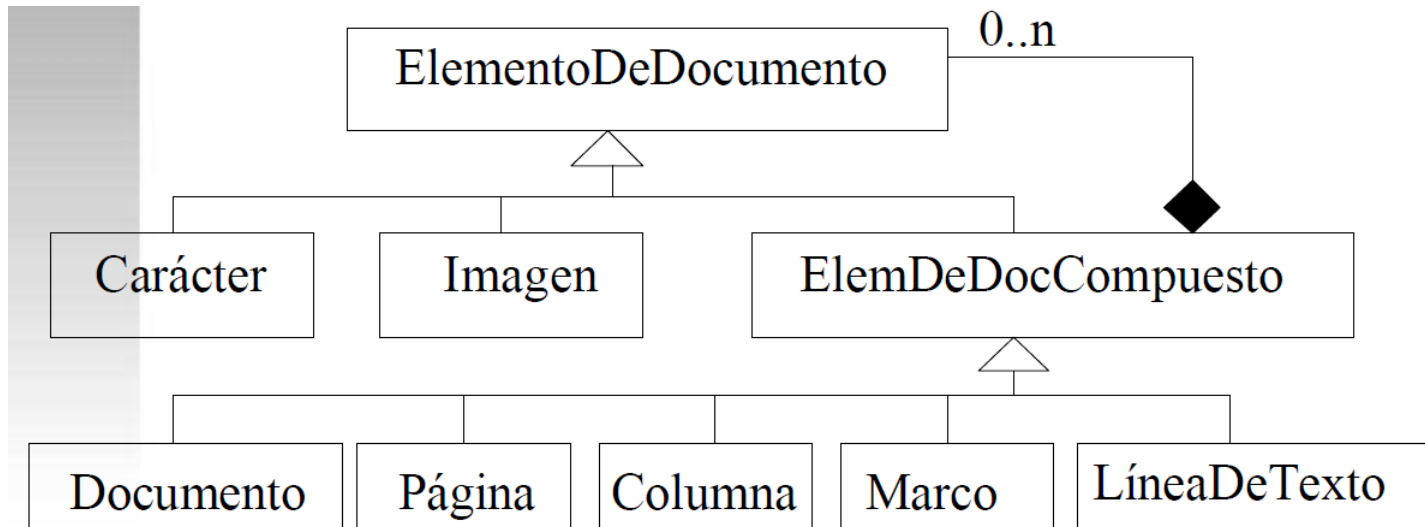
- Componer objetos en jerarquías todo-parte y permitir a los clientes tratar objetos simples y compuestos de manera uniforme
- Ventajas:
  - Permite tratamiento uniforme de objetos simples y complejos así como composiciones recursivas.
  - Simplifica el código de los clientes, que sólo usan una interfaz.
  - Facilita añadir nuevos componentes sin afectar a los clientes.
- Inconvenientes:
  - Es difícil restringir los tipos de los hijos.
  - Las operaciones de gestión de hijos en los objetos simples pueden presentar problemas: seguridad frente a flexibilidad.

# Problema: La Escalabilidad



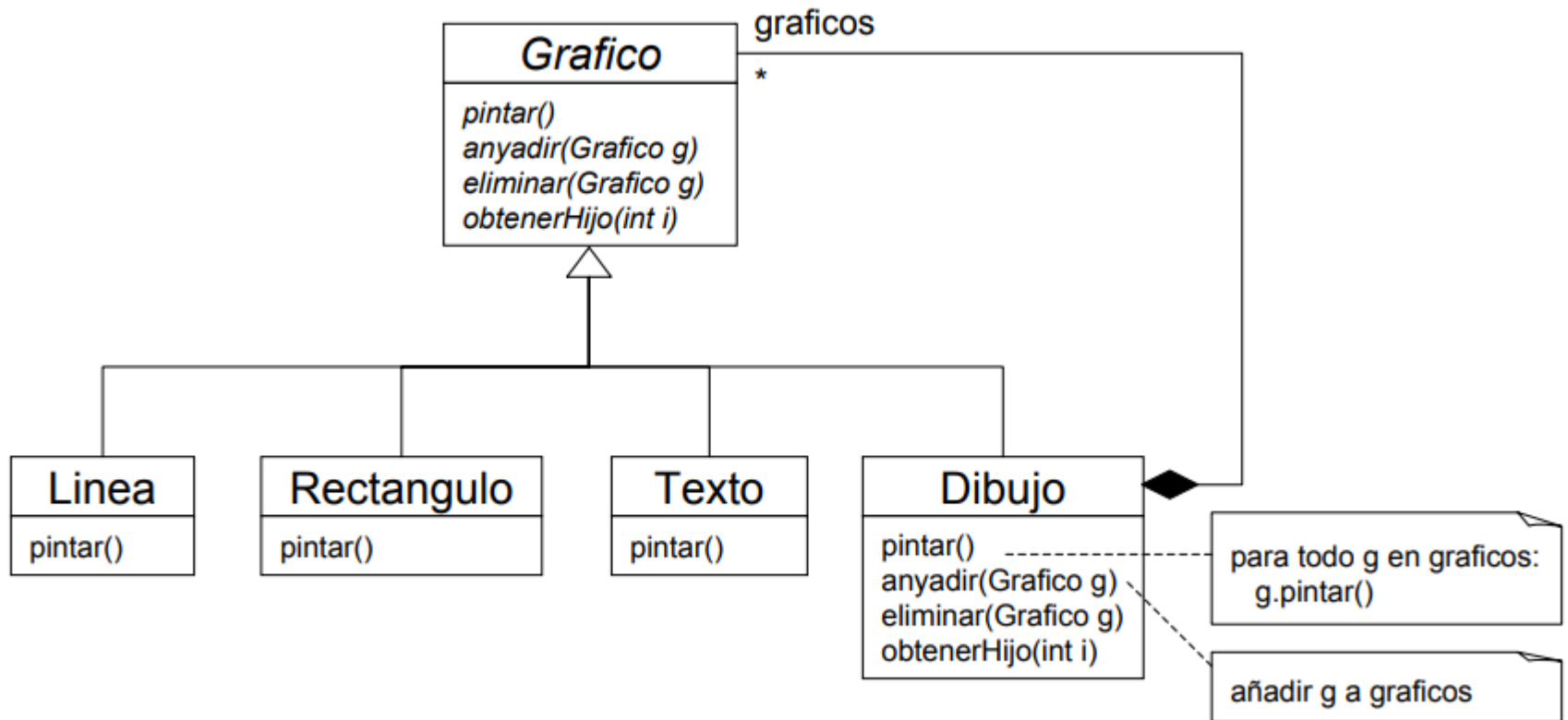
- Un **documento** está formado por varias **páginas**, las cuales están formadas por **columnas** que contienen **líneas de texto**, formadas por **caracteres**.
- Las **columnas** y **páginas** pueden contener **marcos**. Los **marcos** pueden contener **columnas**.
- Las **columnas**, **marcos** y **líneas de texto** pueden contener **imágenes**.

# Solución: Patrón Composite



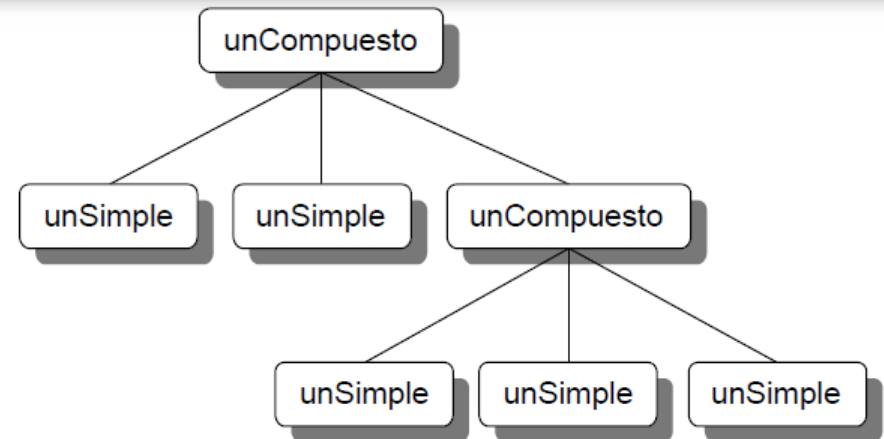
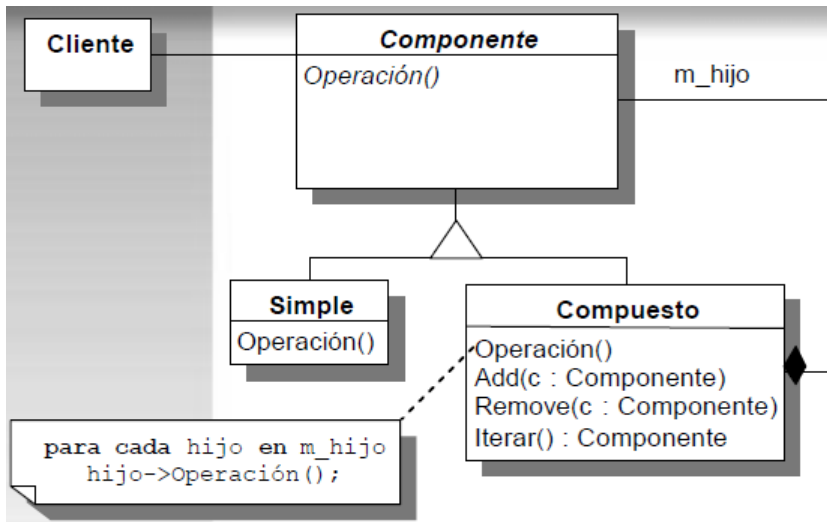
- Un documento está formado por varias páginas, las cuales están formadas por columnas que contienen líneas de texto, formadas por caracteres.
- Las columnas y páginas pueden contener marcos. Los marcos pueden contener columnas.
- Las columnas, marcos y líneas de texto pueden contener imágenes.

# Otro ejemplo (con gráficos)



Podemos tener gráficos sencillos, y gráficos que se componen de otros gráficos (los elementos mas complejos).

# Estructura



# Participantes

- **Componente:** declara una clase abstracta para la composición de objetos.
- **Simple:** representa los objetos de la composición que no tienen hijos e implementa sus operaciones.
- **Compuesto:** implementa las operaciones para los componentes con hijos y almacena a los hijos.
- **Cliente:** utiliza objetos de la composición mediante la interfaz de Componente.

# Ejemplos

- Sirve para diseñar clases que agrupen a objetos complejos, los cuales a su vez están formados por objetos complejos y/o simples.
  - Por ejemplo, las entradas de un directorio. Contienen elementos simples (los ficheros) y los elementos Compuestos (los directorios). Esta estructura no tiene límite.
- La jerarquía de clases **AWT** (de Java) se ha diseñado según el patrón **COMPOSITE**.

# Bridge

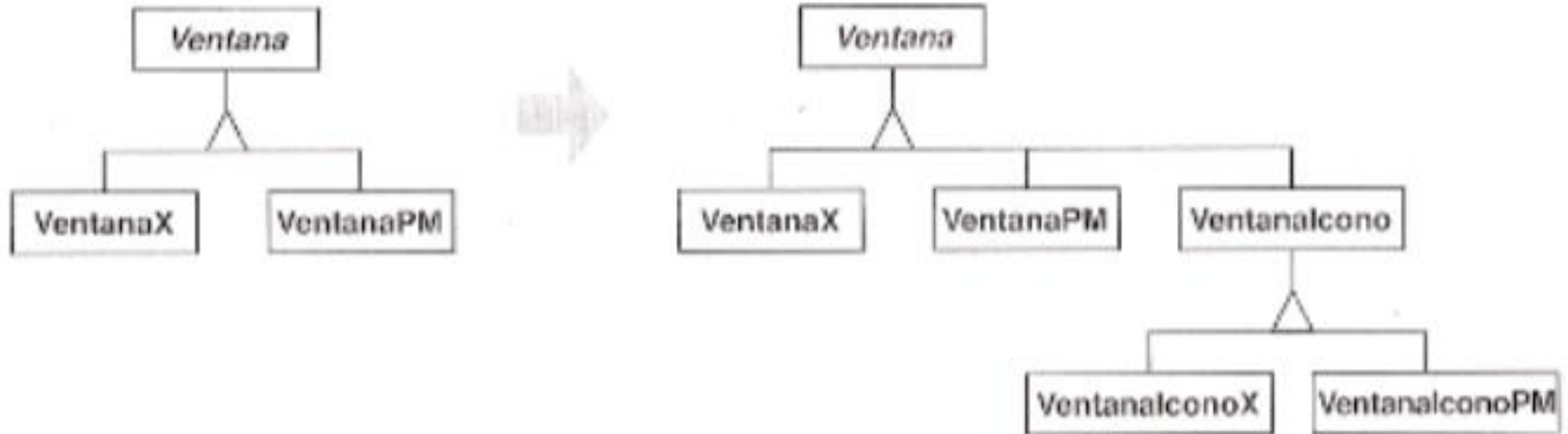
- Desacopla una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.
- *La herencia liga una implementación a una abstracción de forma permanente.*
- Con este patrón la configuración de una abstracción con implementación se puede hacer de forma dinámica.



# Motivación

- Partimos de una abstracción Ventana en un toolkit de interface de usuario y queremos tener Ventana para distintas plataformas: a) Ventanas X y Ventanas Presentation Manager de IBM.
- Podemos tener una clase abstracta Ventana y dos implementaciones: Ventana X y VentanaPM.
- Si queremos una Ventana como tendríamos que implementarla con las dos plataformas.

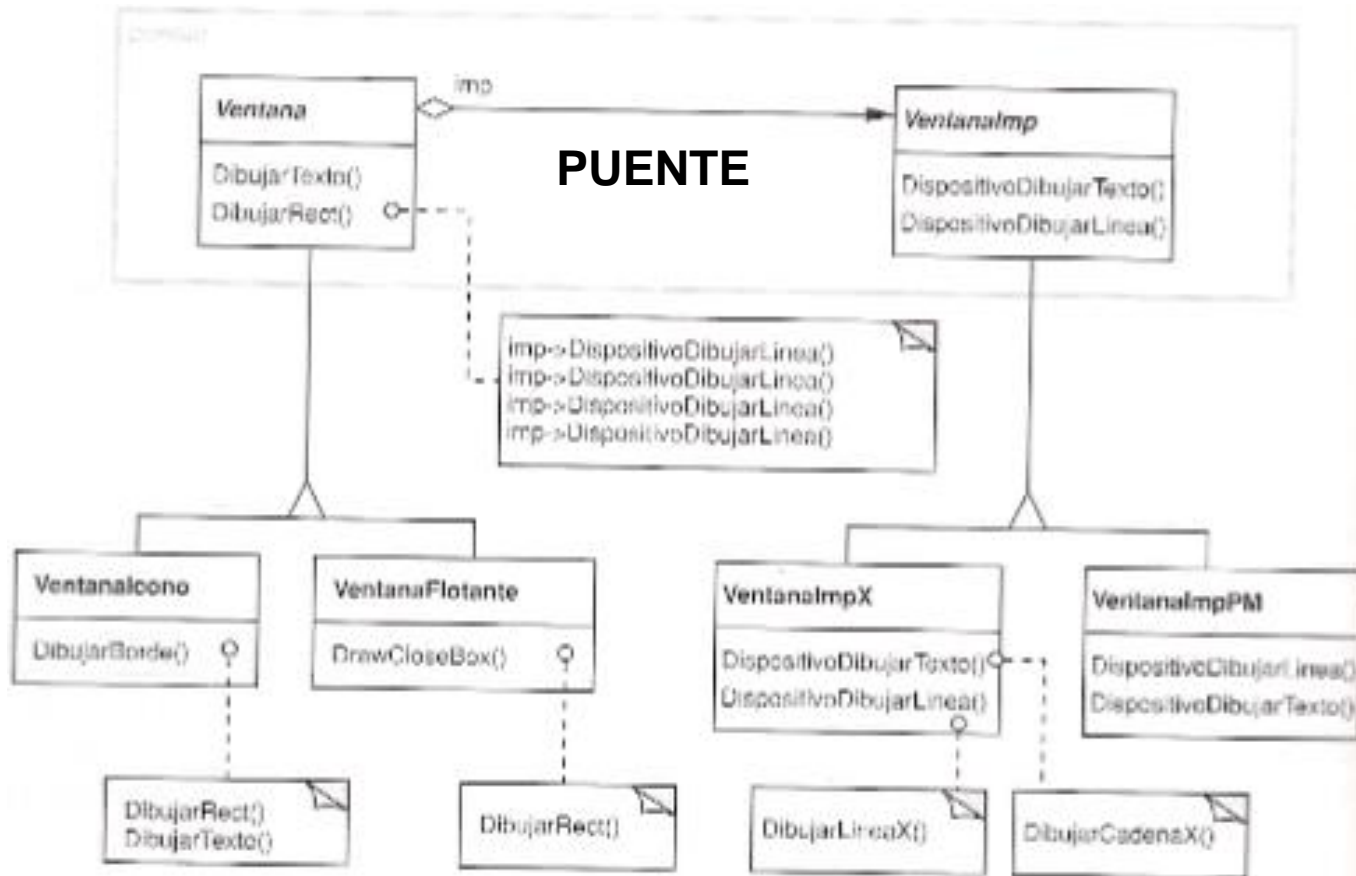
# Motivación



# Solución

- Separar las clases en dos jerarquías, por un lado los distintos tipos de ventanas y por otro lado las implementaciones.
- Los clientes deberían ser capaces de crear una ventana sin someterse a una implementación concreta.

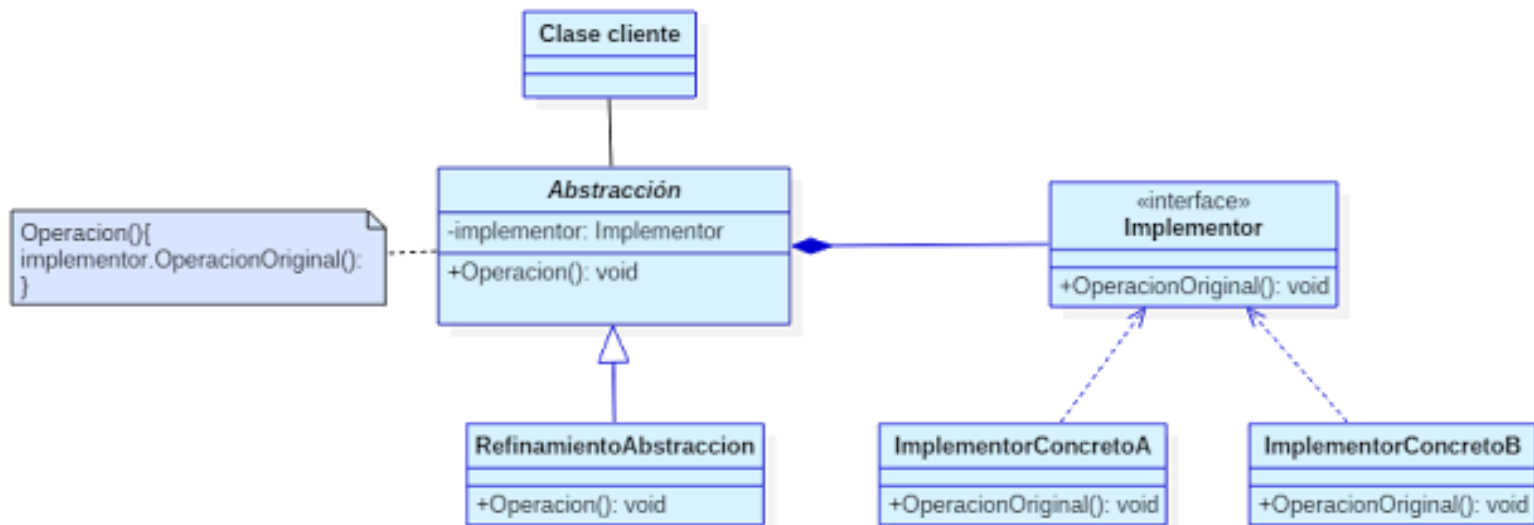
# Solución



# Aplicabilidad

- Se desea evitar un enlace permanente entre la abstracción y su implementación. Esto puede ser debido a que la implementación debe ser seleccionada o cambiada en tiempo de ejecución.
- Tanto las abstracciones como sus implementaciones deben ser extensibles por medio de subclases. En este caso, el patrón Bridge permite combinar abstracciones e implementaciones diferentes y extenderlas independientemente.
- Cambios en la implementación de una abstracción no deben impactar en los clientes, es decir, su código no debe tener que ser recompilado.
- Se desea compartir una implementación entre múltiples objetos (quizá usando contadores), y este hecho debe ser escondido a los clientes.

# Estructura



# Participantes

- **Abstraction** define una interface abstracta. Mantiene una referencia a un objeto de tipo Implementor.
- **RefinedAbstraction** extiende la interface definida por Abstraction
- **Implementor** define la interface para la implementación de clases. Esta interface no se tiene que corresponder exactamente con la interface de Abstraction; de hecho, las dos interfaces pueden ser bastante diferente. Típicamente la interface Implementor provee sólo operaciones primitivas, y Abstraction define operaciones de alto nivel basadas en estas primitivas.
- **ConcreteImplementor** implementa la interface de Implementor y define su implementación concreta.

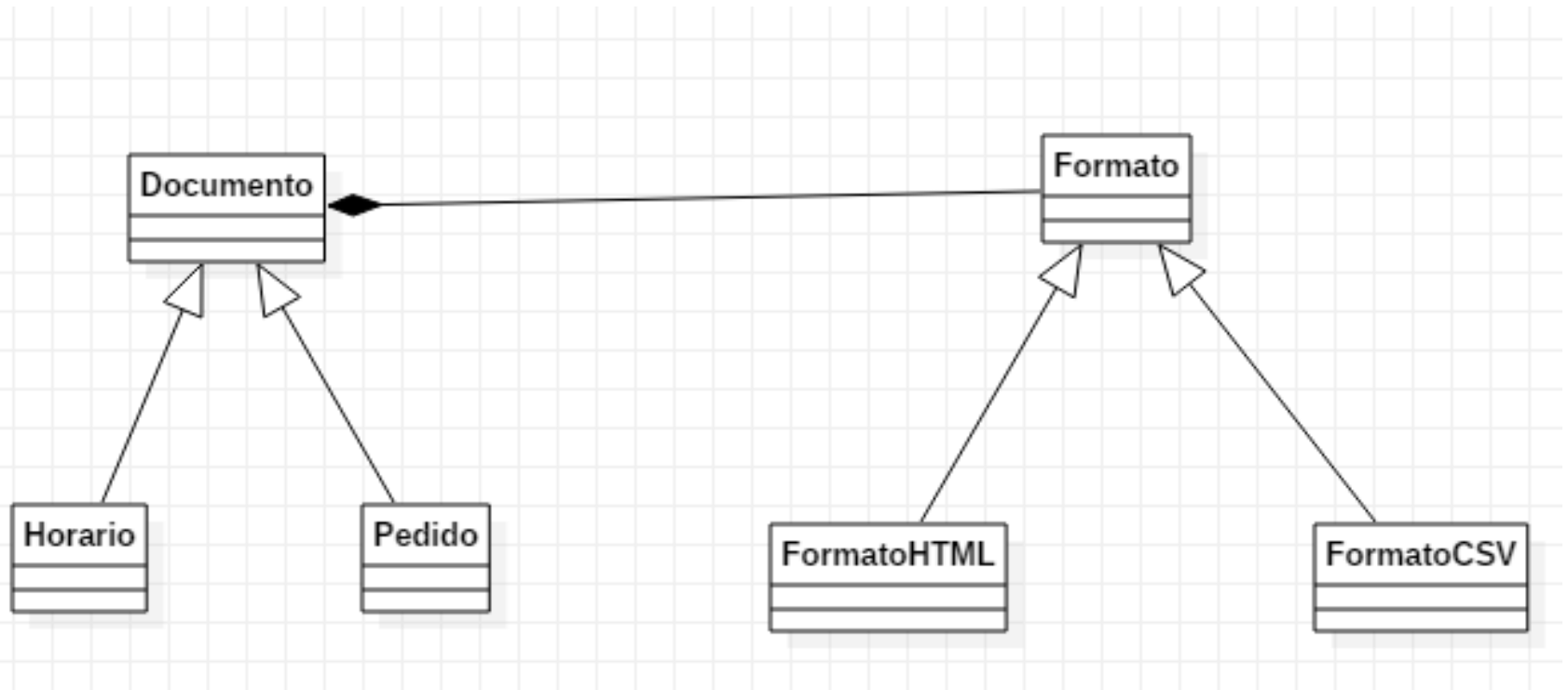
# Otro Ejemplo

- Jerarquía de documentos frente a una jerarquía de formatos:
  - Documentos:
    - Pedido
    - Horario
    - ...
  - Formatos:
    - HTML
    - CSV
    - ...

**El patrón Bridge** permite realizar las Combinaciones para disponer de Pedidos en formato HTML y CSV  
Y por otro lado:  
Horarios en formato HTML y CSV.



# UML



# FlyWeight

- El patrón **Flyweight** (u objeto ligero) sirve para eliminar o reducir la redundancia cuando tenemos gran cantidad de objetos que contienen información idéntica, además de lograr un equilibrio entre flexibilidad y rendimiento (uso de recursos).
- También se puede utilizar para objetos que consumen mucho tiempo en la inicialización.

# FlyWeight

- Un peso ligero es un **objeto compartido** que puede usarse a la vez en varios contextos.
- Hay que distinguir entre estado intrínseco y extrínseco.
  - El **estado intrínseco** se guarda con el propio objeto (**información** independiente que puede ser **compartida**).
  - El **estado extrínseco** depende del contexto y cambia con él. **No se puede compartir.** Los objetos cliente son responsables de pasar al peso ligero su estado extrínseco cuando lo necesite.

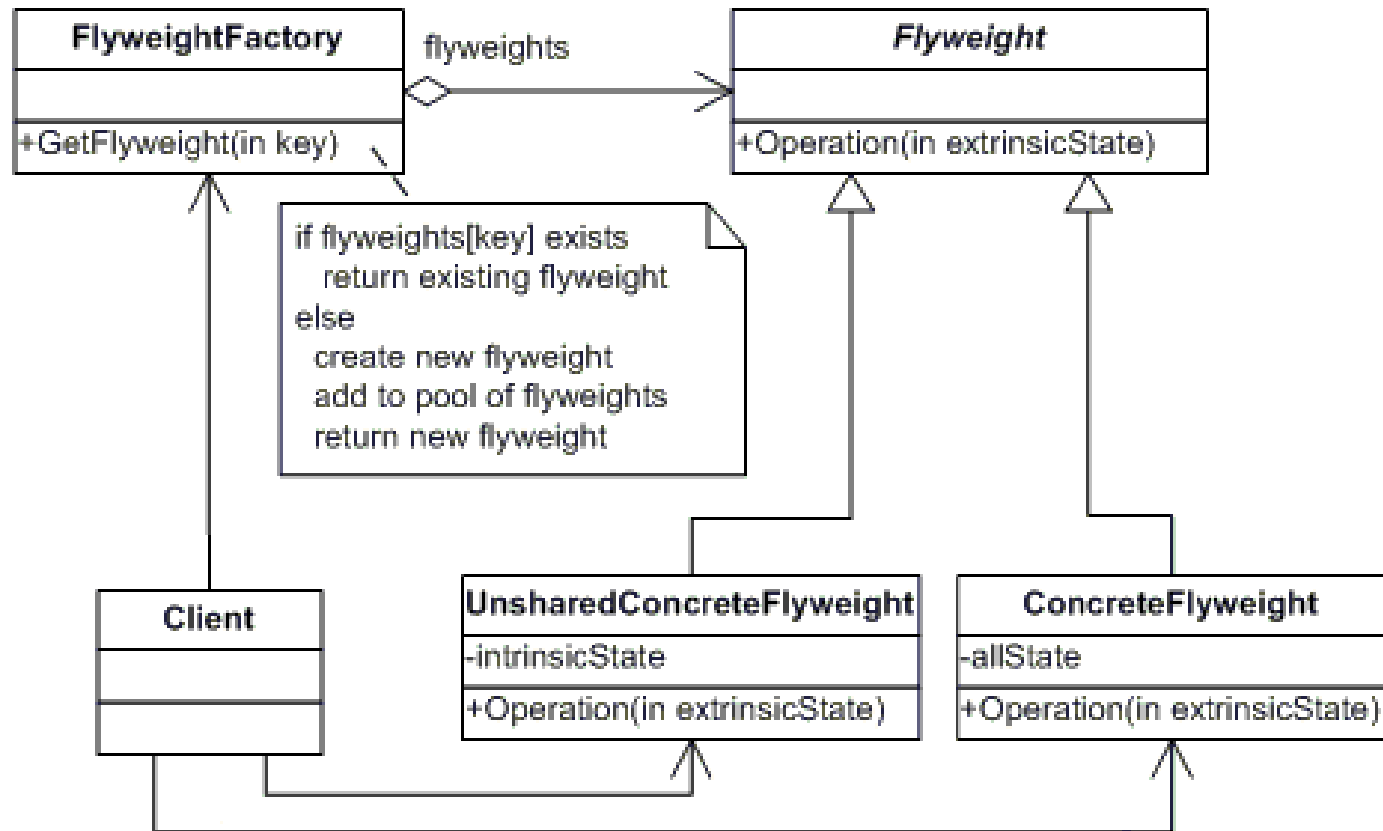
# Consideraciones

- Comprobar que el rendimiento en los objetos es un tema primordial, y si el cliente está dispuesto a asumir el reajuste.
- Dividir el objetivo principal en estados: estado intrínseco (elementos que se puedan compartir o son comunes) y estado extrínseco (elementos particulares a cada tipo).
- Retirar los elementos con estado extrínseco de los atributos de la clase, y añádale más bien una llamada a métodos.
- Crear una fábrica que pueda almacenar y reutilizar las instancias existentes de clases.
- El cliente debe usar la fábrica en vez de utilizar el operador new si requiere de creación de objetos.
- El cliente (o un tercero) debe revisar los estados extrínsecos, y reemplazar esos estados a métodos de la clase.

# Aplicabilidad

- Este patrón se debería de aplicar cuando cumpla lo siguiente:
  - Una aplicación utiliza un gran número de objetos.
  - Los costes de almacenamiento son elevados debido al gran número de objetos.
  - La mayor parte del objeto puede hacerse extrínseco.
  - Muchos grupos de objetos pueden reemplazarse por relativamente pocos objetos compartidos, una vez se ha eliminado el estado extrínseco.
  - La aplicación no depende de la identidad de un objeto. Puesto que los objetos de peso ligero pueden ser compartidos, las comprobaciones de identidad devolverán verdadero para objetos conceptualmente distintos.

# Estructura



# Participantes

- **FlyWeight**
  - Declara una interfaz a través de la cual los pesos ligeros pueden recibir un estado extrínseco y actuar sobre él.
- **ConcreteFlyWeight:**
  - Implementa la interfaz PesoLigero y permite almacenar el estado intrínseco, en caso de lo haya. Un objeto PesoLigeroConcreto debe poder ser compartido, por lo que cualquier estado que almacene debe ser intrínseco.
- **UnsharedConcreteFlyWeight:**
  - No todas las subclases de PesoLigero necesitan ser compartidas.
- **FlyWeightFactory:**
  - Crea y controla objetos de peso ligero.
  - Garantiza que los pesos ligeros se comparta de manera adecuada. Cuando un cliente solicita un peso ligero, el objeto FabricaDePesosLigeros proporciona una instancia concreta o crea uno nuevo, en caso de que no exista ninguno.
- **Cliente:**
  - Mantiene una referencia a los pesos ligeros.
  - Calcula o guarda el estado extrínseco de los pesos ligeros.

# Ventajas / Desventajas

- **Ventajas:** Reduce en gran cantidad el peso de los datos en un servidor.
- **Desventajas:** Consume un poco más de tiempo para realizar las búsquedas.



# Patrones de Comportamiento

Strategy

Chain of Responsibility

Memento

Template Method

Command

Interpreter

Iterator

Mediator

Observer

State

Visitor

# Patrones de Comportamiento

- Los patrones de Comportamiento tienen que ver con Algoritmos y con la asignación de responsabilidades a objetos.
- Describen no solo patrones de clases y objetos, sino también patrones de comunicación entre ellos.

# Estrategia

- El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.
- Permite seleccionar el algoritmo a aplicar.

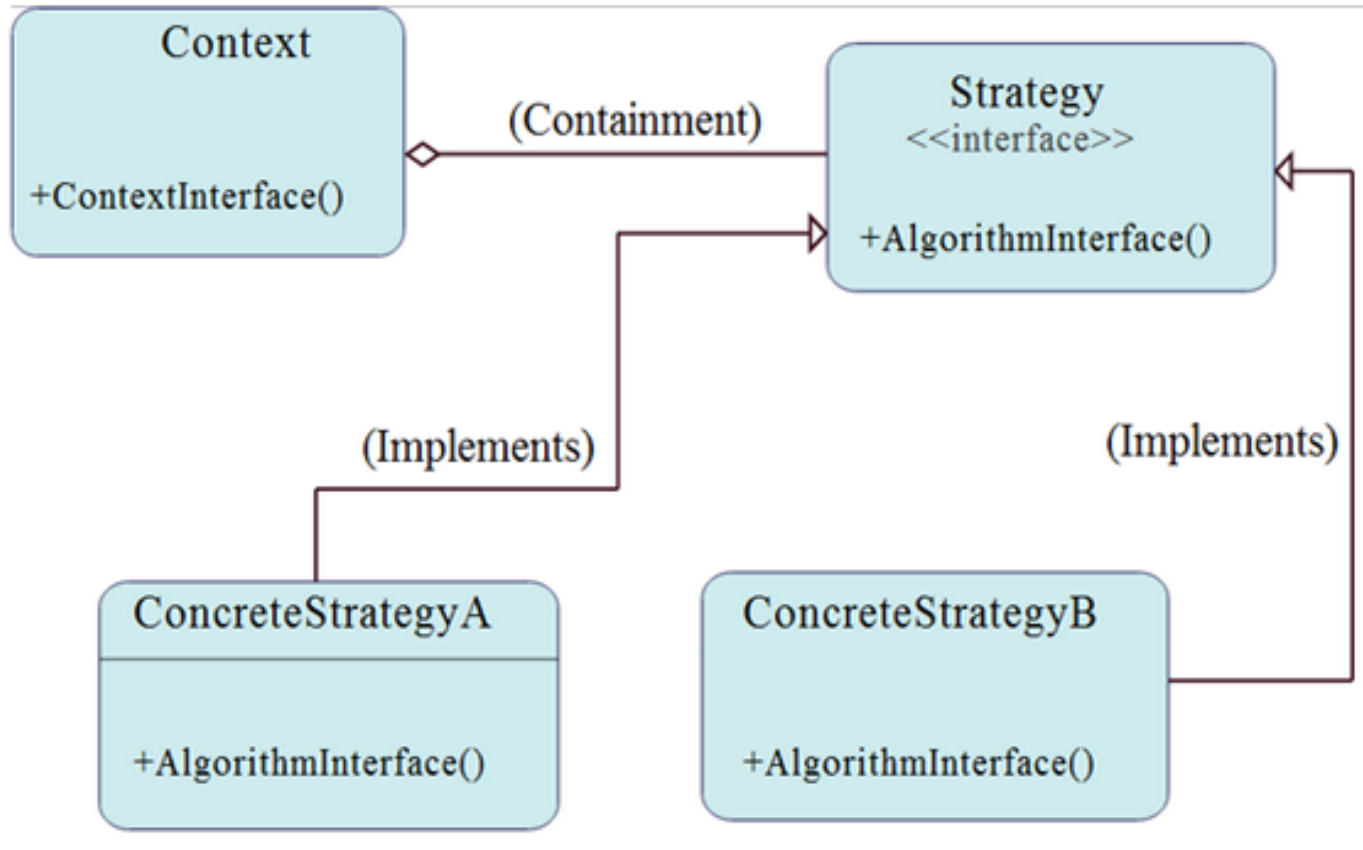
# Estrategia

- Suponiendo un **editor de textos** con diferentes algoritmos para particionar un texto en líneas (**justificado, alineado a la izquierda**, etc.), se desea separar las clases clientes de los diferentes algoritmos de partición, por diversos motivos:
  - **Incluir el código de los algoritmos en los clientes** hace que éstos sean **demasiado grandes y complicados** de mantener y/o extender.
  - **El cliente no va a necesitar todos los algoritmos** en todos los casos, de modo que no queremos que dicho cliente los almacene si no los va a usar.
  - Si existiesen **clientes distintos** que usasen los mismos algoritmos, habría que **duplicar el código**, por tanto, esta situación no favorece la reutilización.
- La solución que el patrón estrategia supone para este escenario pasa por encapsular los distintos algoritmos en una jerarquía y que el cliente trabaje contra un objeto intermediario contexto. El cliente puede elegir el algoritmo que prefiera de entre los disponibles, o el mismo contexto puede ser el que elija el más apropiado para cada situación.

# Cuando se aplica

- Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato a utilizar el patrón estrategia.

# Esquema UML



# Participantes

- **Contexto** (*Context*) : Es el elemento que usa los algoritmos, por tanto, delega en la jerarquía de estrategias. Configura una estrategia concreta mediante una referencia a la estrategia necesaria.
- **Estrategia** (*Strategy*): Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.
- **EstrategiaConcreta** (*ConcreteStrategy*): Implementa el algoritmo utilizando la interfaz definida por la estrategia.

# Pasar información

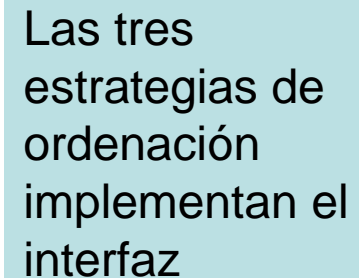
- Cuando es necesario el intercambio de información entre estrategia y contexto.
- Este **intercambio** puede realizarse de dos maneras:
  - Mediante **parámetros en los métodos** de la estrategia.
  - **Mandándose, el contexto**, a sí mismo a la estrategia.



# Ejemplo

- Disponemos de un array y lo podemos ordenar por distintos métodos de ordenación: burbuja, inserción directa y quicksort.

```
class InterfaceOrdenacion {  
    public:  
        virtual void ordenar(int *, int)=0;  
}
```



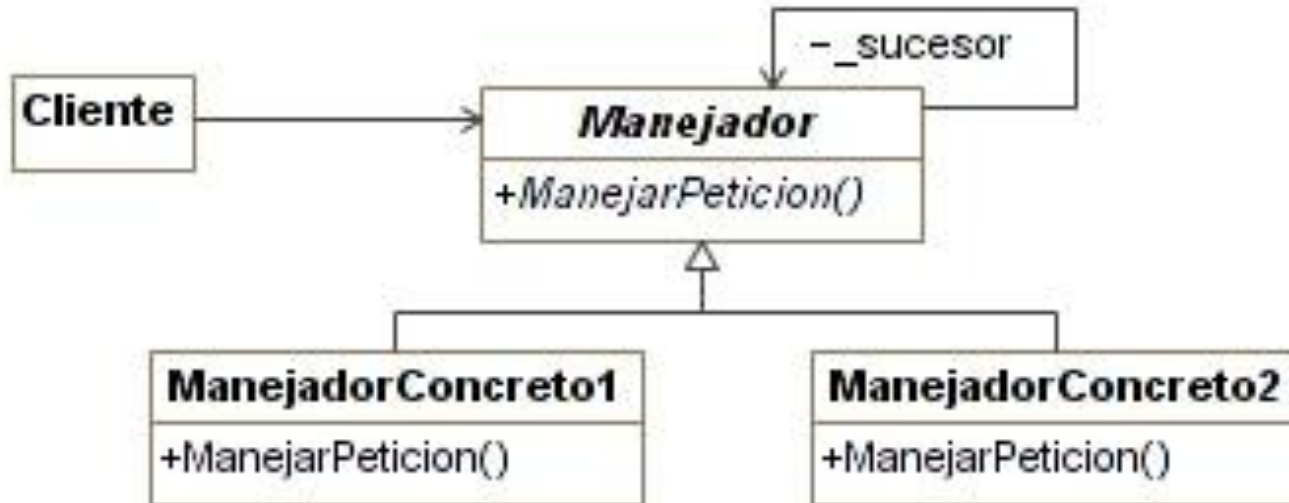
Las tres  
estrategias de  
ordenación  
implementan el  
interfaz

El contexto es que selecciona la estrategia, método de ordenación a aplicar  
**Se puede combinar con Template para ampliar los tipos de algoritmos.**

# Chain of Responsibility

- Es un patrón de comportamiento que evita acoplar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición.
- Se encadenan los receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.
- Este **patrón es utilizado a menudo en el contexto de las interfaces gráficas** de usuario donde un objeto puede contener varios objetos. Según si el ambiente de **ventanas genera eventos, los objetos los manejan o los pasan.**

# Esquema UML



Un objeto elige si procesa la petición o la pasa a su sucesor. La cadena de sucesores se configura desde fuera.

Nos podría valer para clasificar información, cada manejador podría realizar una serie de comprobaciones si no lo puede Clasificar lo manda a su sucesor.

# Se usa cuando

- Hay más de un objeto que puede manejar una petición, y el manejador no se conoce a priori, sino que debería determinarse automáticamente.
- Se quiere enviar una petición a un objeto entre varios sin especificar explícitamente el receptor.
- El conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

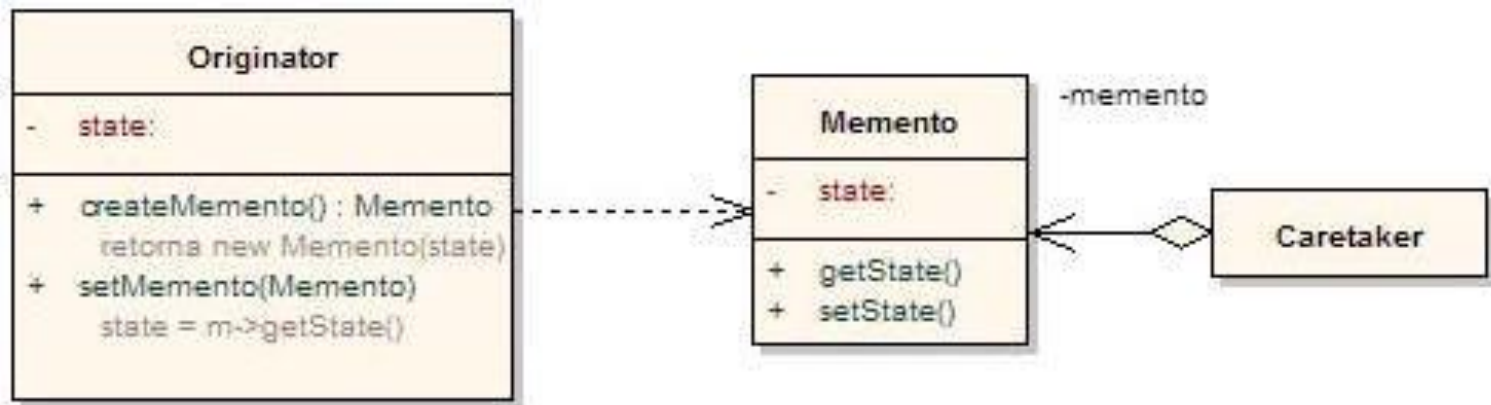
# MEMENTO (Recuerdo)

- Este patrón permite capturar el estado actual de un objeto para luego poder recuperarlo (*sin violar la encapsulación del objeto*).
- Con este patrón se pueden implementar las típicas operaciones de **deshacer** / **rehacer** de los editores.

# Aplicabilidad

- Cuando se necesita restaurar un sistema o parte de un sistema a un punto anterior.
- Facilitar en el sistema operaciones de deshacer y rehacer para lo cual hay que guardar estado anteriores.
- Llevar un histórico de los estados por los que va pasando un objeto.
- Es muy intuitivo de cuando hay que aplicarlo.

# Estructura



# Participantes

- **Memento:**
  - Guarda el estado interno del objeto Creador. El memento puede guardar tanta información del estado interno del creador como sea necesario.
  - Mantiene las mismas propiedades que el Objeto a guardar.
- **Creador:**
  - Crea un memento que contiene una instantánea de su estado interno actual.
  - Utiliza el memento para volver a su estado anterior.
  - Mantiene los dos métodos uno para crear el memento a partir del estado actual y otro para restaurar el estado a partir del memento.



# Participantes

- **Conserje:**
  - Es responsable de guardar en lugar seguro el memento. Almacena todos los mementos para luego poder restaurarlos.
  - Nunca examina los contenidos del memento, ni opera con ellos.

# Inconvenientes

- El almacenamiento de los mementos es **costoso**, el cliente que desea guardar su estado, no conoce el tamaño real del Memento.

# Ejemplo

- Se diseña una clase persona que almacena el nombre que puede ir cambiando a lo largo del tiempo.
- Se implementa una patrón Memento que permita almacenar los posibles cambios y que se pueda restaurar.

# Ejemplo

```
class Persona
{
    string nombre;

    public:
        Persona();
        inline string getNombre(){ return this->nombre; }
        inline void setNombre(string nombre){ this->nombre = nombre;}
        inline Memento saveToMemento(){ return Memento(this->nombre); }
        inline void restoreFromMemento(Memento m){
            this->nombre = m.getSavedState();
        }
        ~Persona();
    protected:
};
```

# Ejemplo

```
class Memento
{
    string estado;

    public:
        Memento(string);
        inline string getSavedState(){ return this->estado; }
        ~Memento();

    protected:
};
```

# Ejemplo

```
class Conserje
{
    vector<Memento> estados;

    public:
        Conserje();
        void addMemento(Memento);
        void historial();
        Memento getMemento(int);
        ~Conserje();

    protected:

};
```

# Template Method

## (Método plantilla)

- Es un patrón de diseño que define una serie de pasos de un algoritmo de forma genérica en una super clase.
- De tal forma que sean las subclases las que rellenen el comportamiento (redefinan) los métodos.

# Propósito

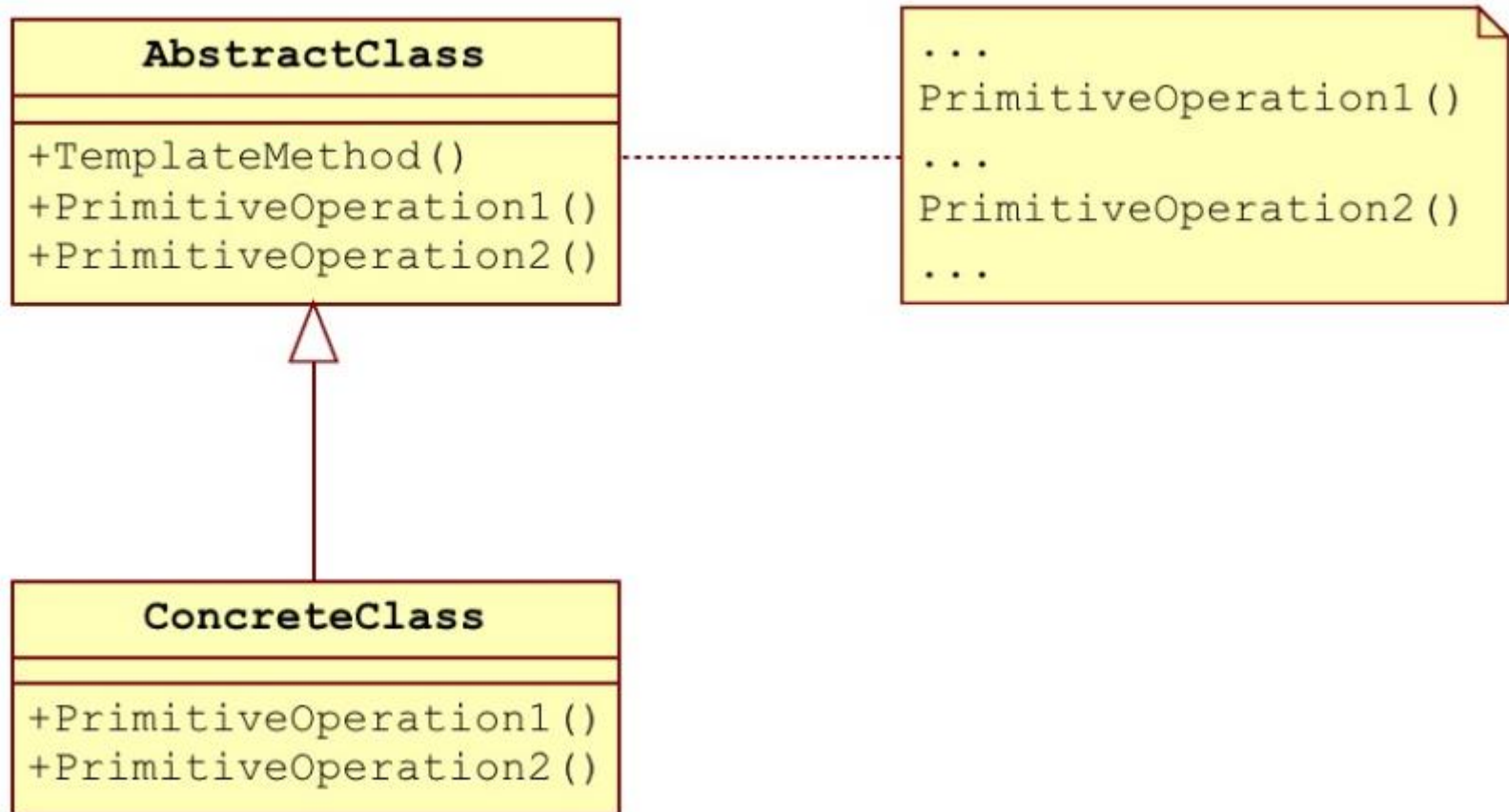
- Proporcionar un método que permite que las subclases redefinan partes del método sin rescribirlo.



# Aplicabilidad

- Es útil cuando tenemos muchas clases donde es necesario aplicar un algoritmo muy similar (donde hay pequeñas variaciones).
- Proporciona un esqueleto para un método permitiendo que las subclases redefinan partes específicas del método.

# Estructura



# Participantes

- **AbstractClass** (DataObject)
  - Define las operaciones primitivas abstractas que las subclases concretas definen para implementar los pasos de un algoritmo.
- **ConcreteClass** (CustomerDataObject)
  - Implementa las operaciones primitivas para realizar los pasos del algoritmo específicos.

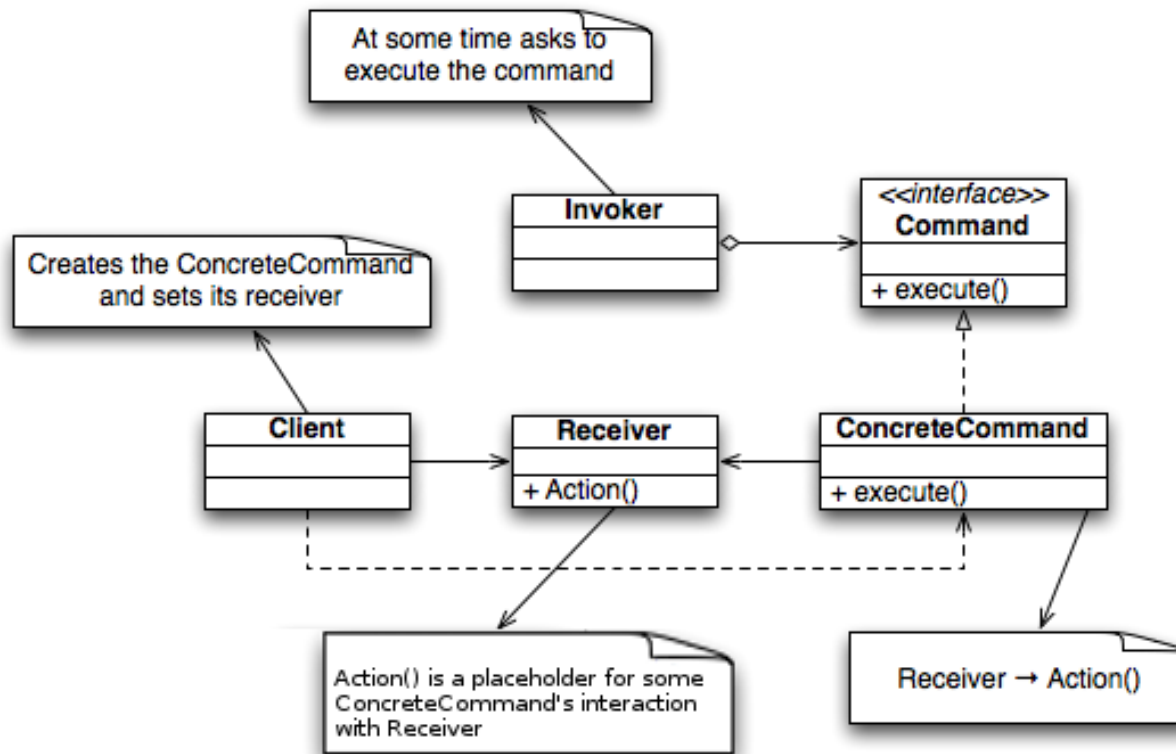
# Command

- Encapsula la 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.
- Este patrón permite **solicitar una operación** a un objeto **sin conocer realmente el contenido** de esta operación, ni el receptor real de la misma.
  - Encapsula la petición como un objeto, con lo que además facilita la parametrización de los métodos.

# Command

- Un ejemplo de patrón Command lo tenemos en los framework de desarrollo de ventanas.
  - Los eventos click de los botones o cuando pulsamos sobre un menú que se ejecuta el evento asociado pero no sabemos ni esta ligado a nada.

# Estructura



# Participantes

- **Command:**
  - Declara una interfaz para ejecutar una operación.
- **ConcreteCommand**
  - Cualquiera de las órdenes concretas.
- **Cliente.**
  - Crea un objeto CommandConcrete y establece su receptor.
- **Invoker**
  - Le pide a la orden (Command) que ejecute la petición.
- **Receiver:**
  - Sabe como llevar a cabo las operaciones asociadas a una petición.

# Colaboraciones

- El cliente crea un objeto OrdenConcreta y especifica su receptor.
  - Cada orden tiene una referencia al receptor.
- Un objeto Invocador almacena el objeto OrdenConcreta.
- El invocador envía una petición llamando a Ejecutar sobre la orden.
- El objeto OrdenConcreta invoca operaciones de su receptor para llevar a cabo la petición.



# Consecuencias

- Command / Orden: desacopla el objeto que invoca la operación de aquel que sabe como realizarla.
- Las órdenes, se pueden extender y se pueden agrupar.
- Es fácil añadir nuevas órdenes no hay que cambiar nada en las clases existentes.

# Ejemplo

- Supongamos un Interruptor que enciende y apaga una Luz.
  - Interruptor es el objeto invocador.
  - Enciende y apaga serán objetos Command concretos (que contiene el Interruptor) y que implementan el método execute que lanzan el método adecuado al receptor (en este caso la Luz).
- La Luz tiene dos estados: encendida o apagada.
  - La Luz sería la clase receptora.

# Interpreter

- Partimos de un lenguaje, define una representación de su gramática junto a un interprete que utiliza dicha representación para interpretar sentencias del lenguaje.
- Enfocado para PEQUEÑAS GRAMÁTICAS o lenguajes SIMPLES que hay que evaluar.

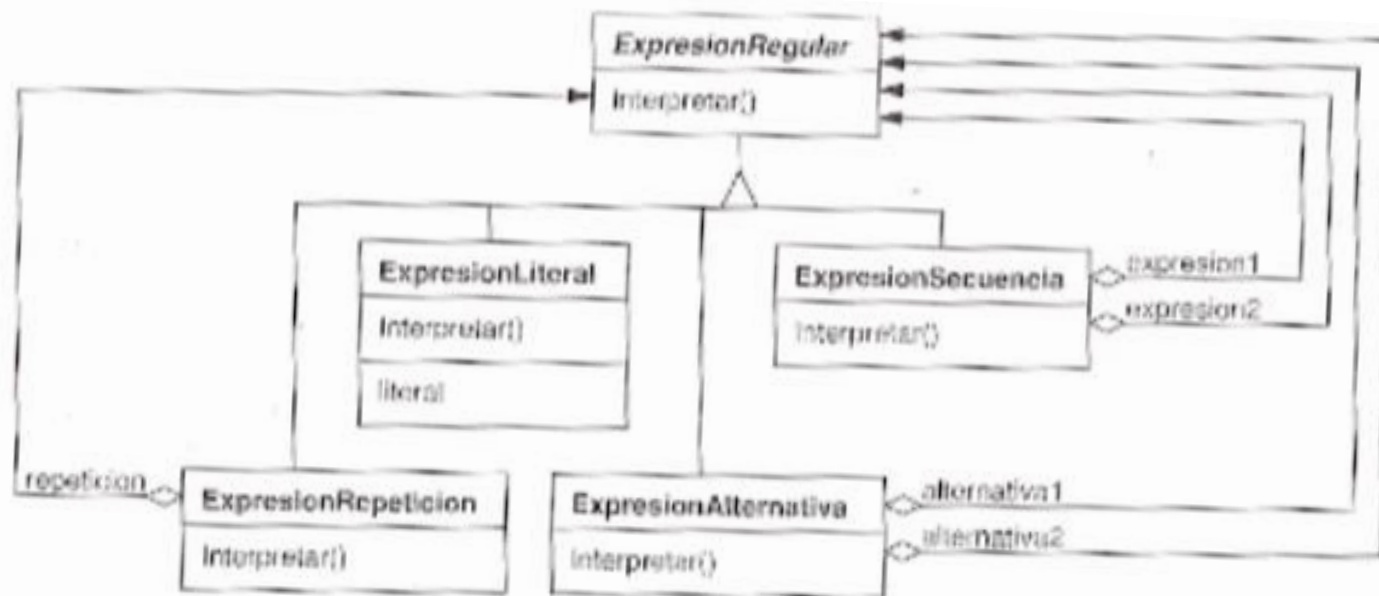
# Interpreter

- Definir gramáticas simples que concuerden con un patrón.
- Este patrón describe:
  - Cómo definir una gramática para lenguajes simples.
  - Cómo representar las instrucciones y como interpretarlas.

# Ejemplo

- Podemos definir expresiones regulares mediante la siguiente gramática:
  - Expresión ::= Literal | Alternativa | Secuencia | Repetición | '(' Expresión ')'
  - Alternativa ::= Expresión | Expresión
  - Secuencia ::= Expresión '&' Expresión
  - Repetición ::= Expresión '\*'
  - Literal ::= 'a' | 'b' | 'c' | ...{ 'a' | 'b' | 'c' ... }\*
- El símbolo inicial es Expresión y el Literal es un símbolo terminal que define palabras.

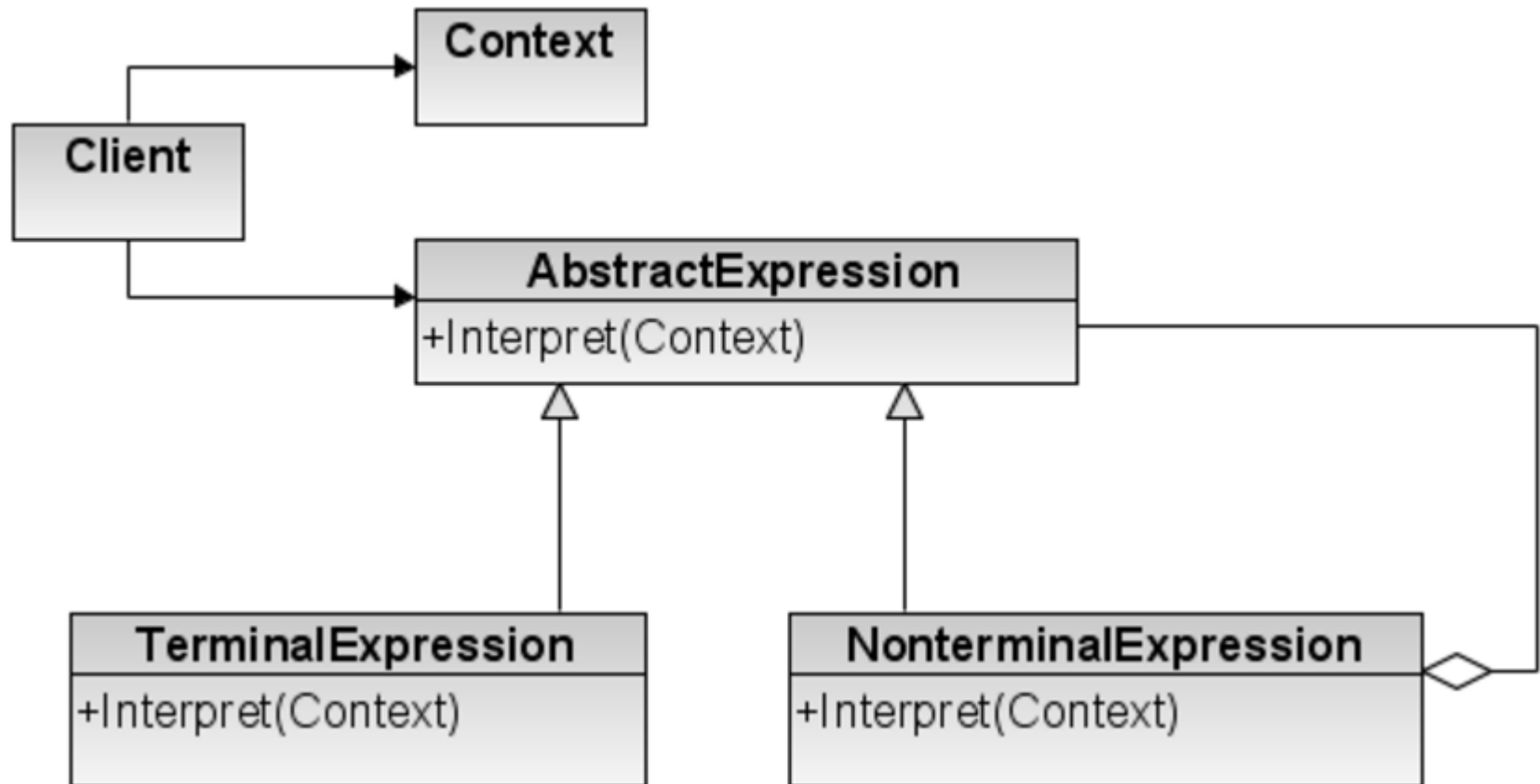
# Ejemplo - Representación



# Aplicabilidad

- Aplicar este patrón cuando hay un lenguaje que interpretar y se pueden representar las sentencias del lenguaje como árboles sintácticos abstractos.
- La gramática es SIMPLE. Para gramáticas grandes se puede volver inmanejable.

# Estructura

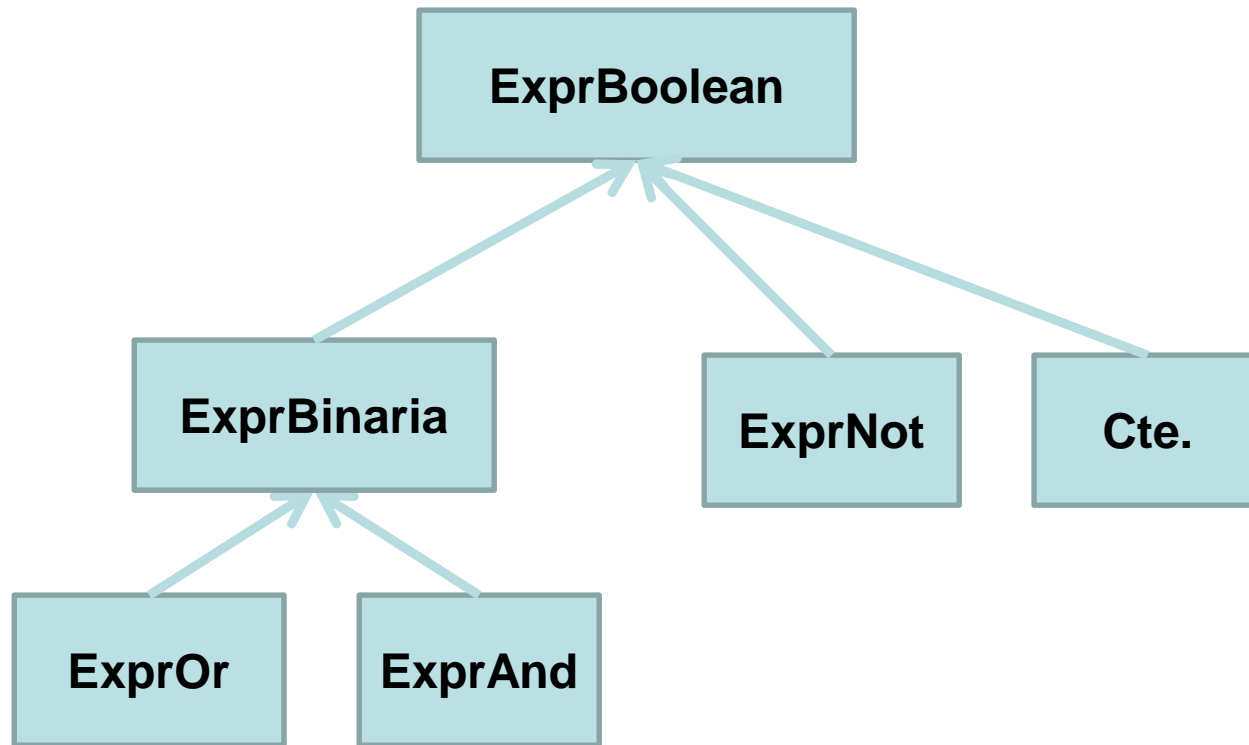




# Participantes

- **ExpresionAbstract**
  - Define una operación para todas las expresiones como puede ser interpretar.
- **ExpresionTerminal**
  - Representa los símbolos terminales de la gramática.
- **ExpresionNoTerminal**, todos los tipos de expresiones. Normalmente tendremos una clase por cada tipo de expresión.
- **Contexto**: información global, por ejemplo, los valores de las variables.
- **Cliente**: construye la expresión, asigna valores a las variables y evalúa la expresión.

# Interprete (para expresiones Boolean)



El interprete sería capaz de evaluar **expresiones booleanas**:  
**F || ((T && (F || T)) && NOT (T))**

# Iterator

- Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

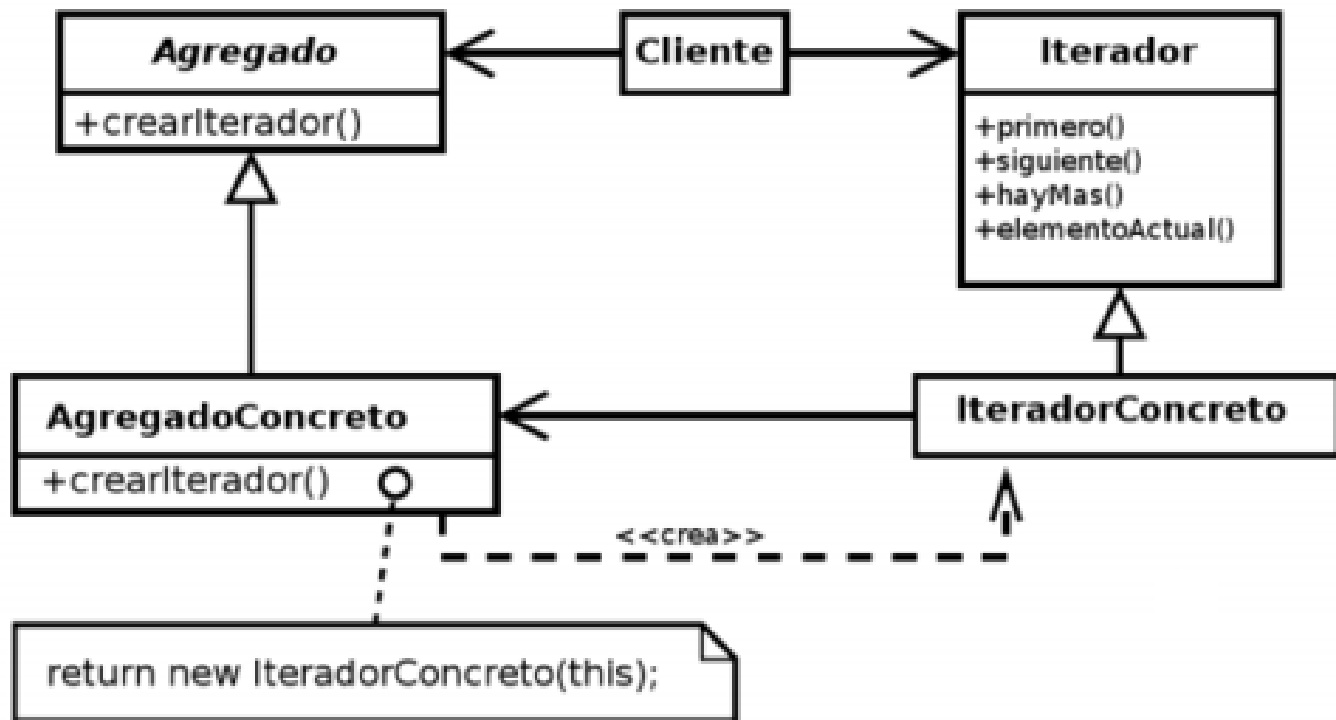
# Iterator

- Un objeto agregado (como p.j. una lista) debería darnos una forma de acceder a sus elementos sin exponer a su estructura interna.
- Podemos necesitar recorrer la lista de diferentes formas y no queremos plagar la lista de diferentes métodos.

# Aplicabilidad

- Este patrón se puede utilizar cuando:
  - Se quiere acceder al contenido de un objeto agregado sin exponer su representación interna.
  - Para permitir varios recorridos sobre objetos agregados.
  - Para proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (permitir la iteración polimórfica).

# Estructura

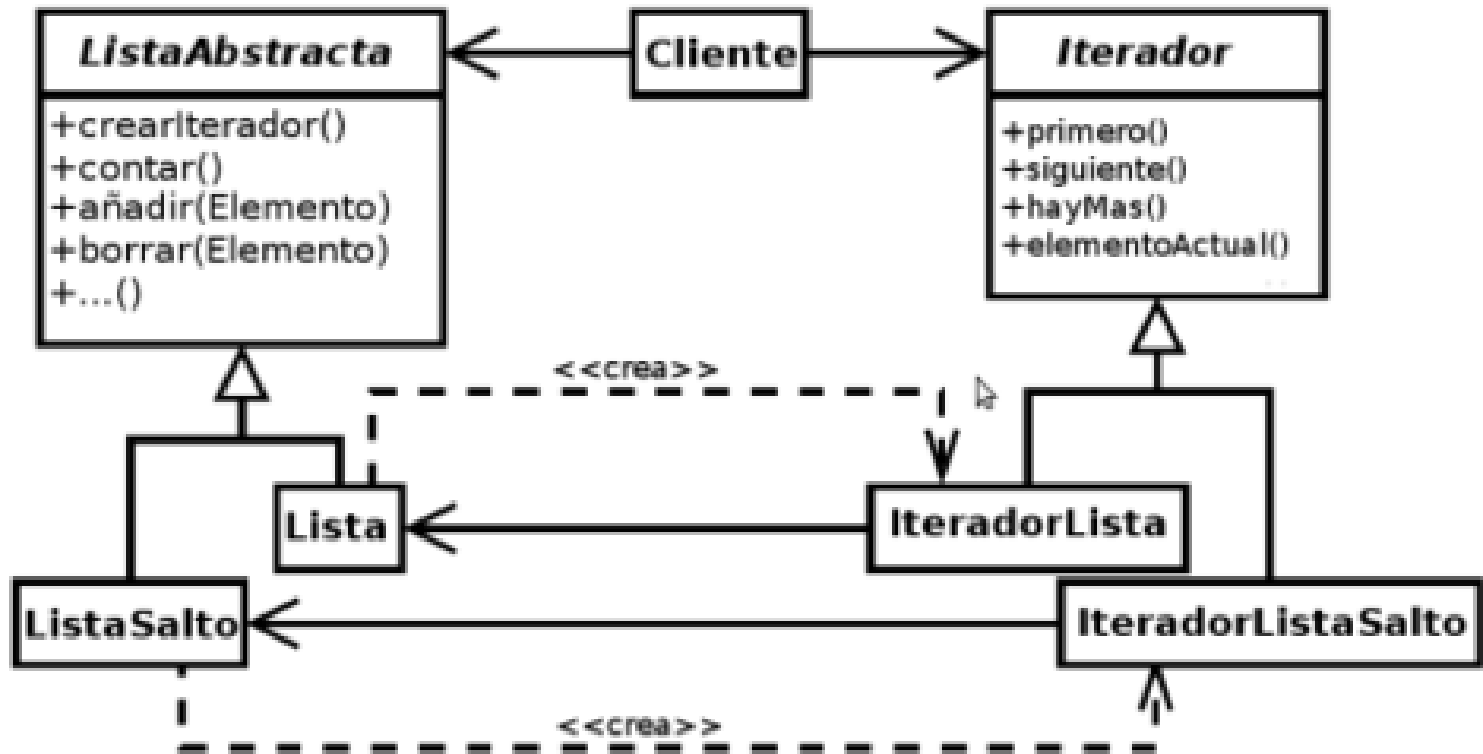


# Participantes

- **Iterador:**
  - Define una interfaz para recorrer los elementos y acceder a ellos.
- **IteradorConcreto:**
  - Implementa la interfaz Iterador, mantiene la posición actual en el recorrido del agregado.
- **Agregado:**
  - Define una interfaz para crear un objeto Iterador.
- **AgregadoConcreto:**
  - Implementa la interfaz de creación de Iterator para devolver una instancia del IteradorConcreto.

# Estructura

## Varios tipos de Listas e Iteradores





# Consecuencias

- El patrón **Iterador** permite por tanto diferentes tipos de recorrido de un agregado y varios recorridos simultáneos, simplificando la interfaz del agregado.

# Java

- **Java** ya dispone de un interface Iterator:
  - interface Iterator<E>
  - Con los métodos:
    - boolean hasNext()
    - E next();
    - void remove()

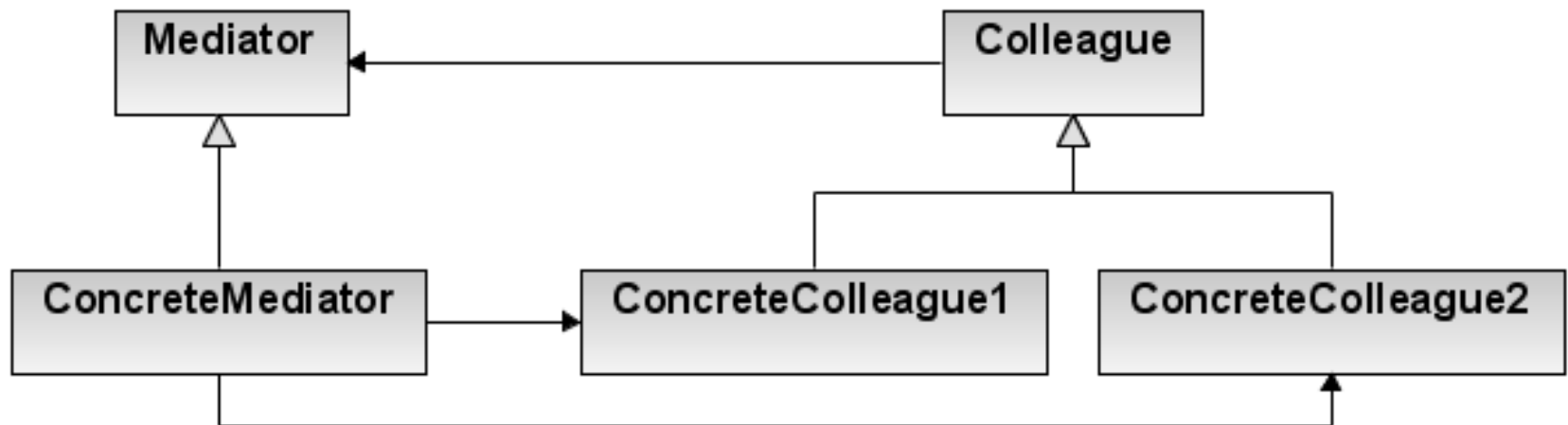
# Mediator

- Define un objeto que encapsula como interactúan una serie de objetos.
- Promueve el bajo acoplamiento.
- Permite variar la interacción entre ellos de forma independiente.

# Aplicabilidad

- Un conjunto de objetos se comunican de forma bien definida pero compleja.
- Las dependencias entre ellos son complejas y difíciles de comprender.
- Es difícil reutilizar un objeto ya que este se refiere a muchos objetos.
- Un comportamiento esta distribuido entre varias clases debería poder ser adaptado sin necesidad de gran cantidad de subclases.

# Estructura

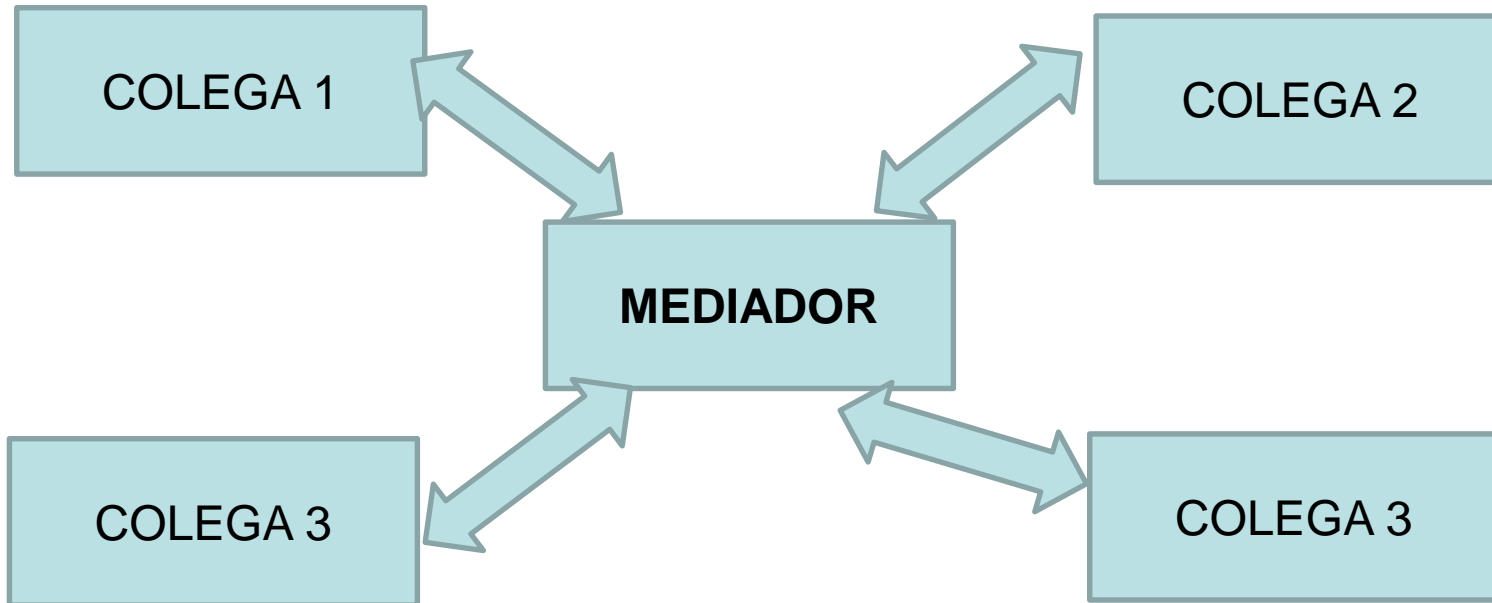


# Participantes

- **Mediador:**
  - Define una interfaz para comunicarse con sus objetos Colega.
- **MediadorConcreto:**
  - Implementa el comportamiento cooperativo coordinando objeto Colega.
  - Conoce a todos sus objetos colega.
- **Colega:**
  - Cada clase Colega conoce a su objeto Mediador.
  - No se comunica con otros colegas sino siempre a través de su Mediador.

# Consecuencias

- **Desacopla** a los colegas:
  - Los objetos colega siempre se comunican a través de su mediador.
- Control **centralizado** en el Mediador
- **Abstrae** de cómo se comunican los objetos.
- Simplifica los protocolos de los objetos, una comunicación de **muchos a muchos** se convierte en una comunicación **1 a muchos**.



# Implementación

- Se puede **omitir la clase abstracta del Mediador** si los colegas solo interactúan con un tipo de Mediador.
  - Con la clase Abstracta los colegas podrían utilizar distintos tipos de mediador.
- **Para la comunicación entre el Mediador y los colegas se puede utilizar un patrón Observer.**
  - Las clases Colega hacen de Sujeto, enviando notificaciones al Mediador cuando cambia su estado.



# Observer

- Define una dependencia de 1 a N (entre objetos).
- Un objeto cambia de estado y notifica a todos los objetos dependientes su cambio para que se actualicen.

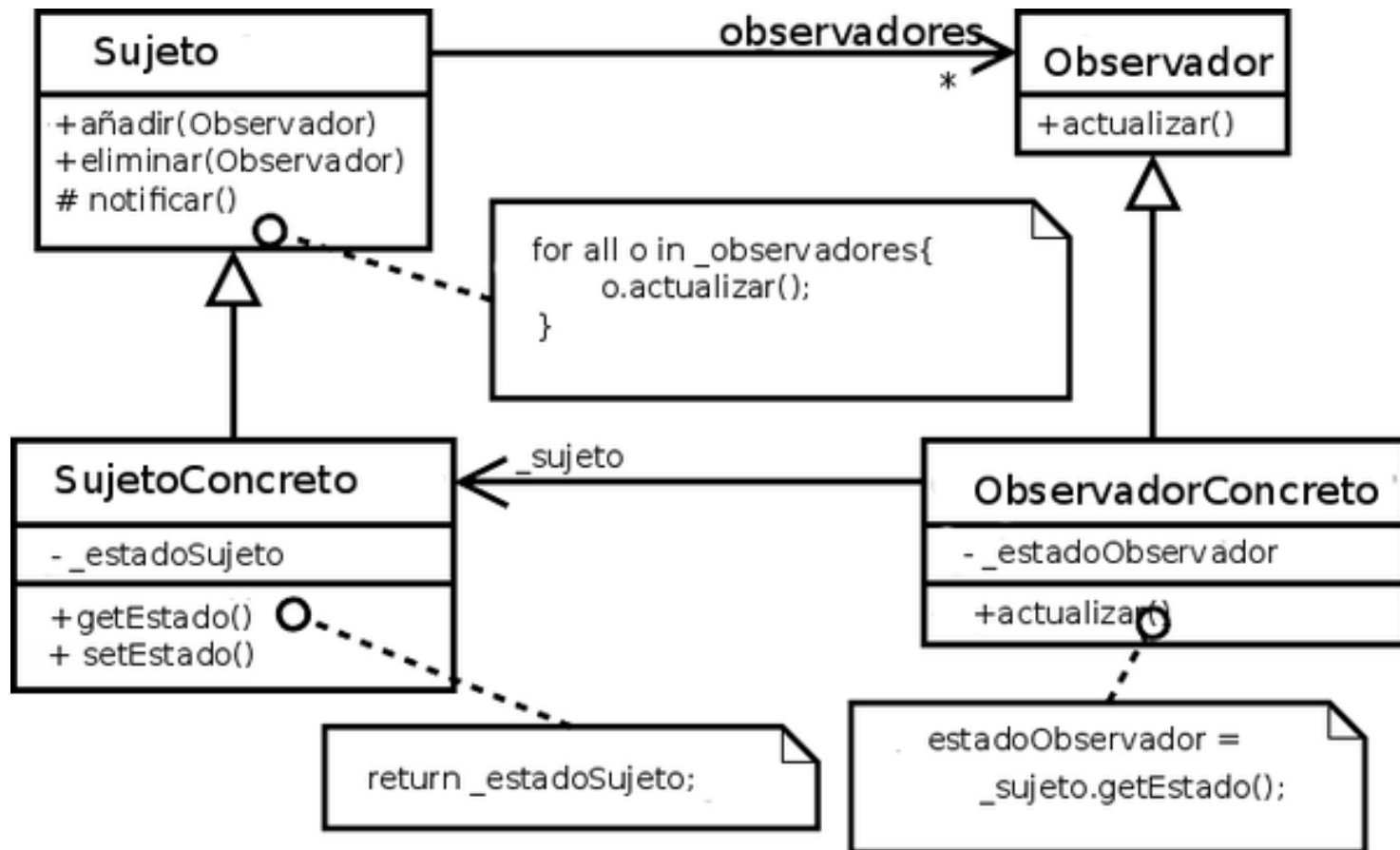
# Observer

- Este patrón es el que nos permite disponer de una serie de datos y queremos mostrarlos en gráfico de barras, en gráfico de tarta, en una tabla.
- Los objetos dependientes se llaman Observadores.

# Aplicabilidad

- El patrón se aplica cuando:
  - Cuando una abstracción tiene dos aspectos y uno depende de otro.
  - Encapsular estos aspectos en objeto separados permite modificarlos y reutilizarlos de forma independiente.
  - Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuantos objetos necesitan cambiarse.
  - Cuando se necesita que un objeto notifique a otros.

# Estructura



# Participantes

- **Sujeto (el objeto a observar)**
  - Conoce a sus observadores. Un sujeto puede ser observado por cualquier número de objetos Observador.
- **Observador:**
  - Define una interfaz para actualizar los objetos que deben ser notificados ante cambios en un sujeto.
- **SujetoConcreto:**
  - Almacena el estado de interés para los objetos ObservadoresConcreto.
  - Envía una notificación a sus observadores cuando cambia de estado.
- **ObservadorConcreto:**
  - Mantiene una referencia a un objeto, SujetoConcreto.

# Problemas a tener en cuenta

- **Muchos sujetos sin observador:**
  - La estructura de los observadores está desaprovechada.
  - Se puede tener un intermediario que centralice el almacenamiento de la asociación de cada sujeto con sus observadores → GestorObservadores.
  - En este caso puede ser un Singleton.
  - En Java Map<Sujeto, List<Observable>>
- **El sujeto es el que notifica los cambios, pero si este se actualiza mucho tendría que realizar muchas notificaciones en poco tiempo.**
  - Se podría suspender temporalmente el aviso de notificaciones para ello se podría utilizar un patrón State para que notifique cuando el objeto se encuentre en un determinado estado.

# Java

- Java ya proporciona la implementación de este patrón:
  - **Clase `java.util.Observable`**
    - `addObserver(o Observer)`
    - `deleteObserver(o Observer)`
    - `notifyObserver()`
    - `notifyObservers(Object data)`
  - **Interface `java.util.Observer`**
    - `void update (Observable o, Object data)`

# State

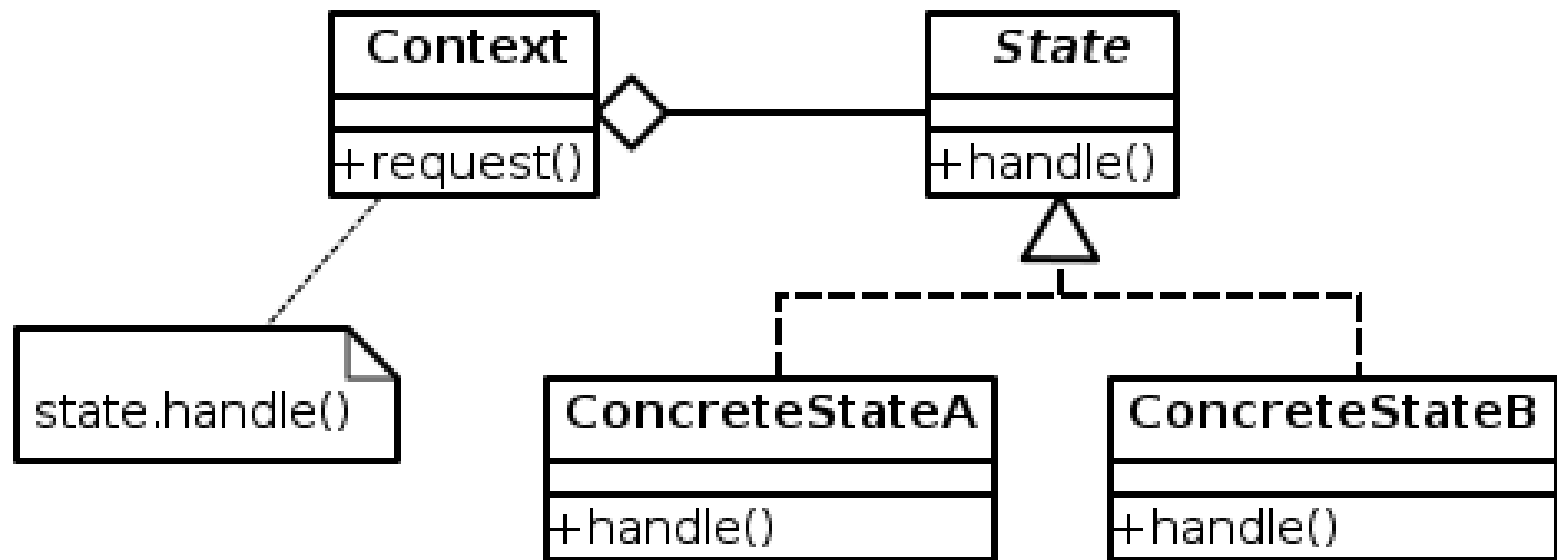
- Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.
- Parecerá que cambia la clase del objeto.
- Nos permite representar mediante objetos Máquinas de estado.
  - Cada estado se representa por una clase.
  - El objeto de interés (el que va cambiando de estado) delega sus cambios en las subclases de State.



# Aplicabilidad

- Este patrón viene bien cuando:
  - El comportamiento de un objeto depende de su estado.
    - Y debe de cambiar en tiempo de ejecución.
  - Cuando las operaciones de una clase presentan sentencias condicionales con múltiples ramas que dependen del estado del objeto.
    - Con este patrón cada una de las ramas se representa por un objeto.

# Estructura



# Participantes

- **Context(Contexto):** Este integrante define la interfaz con el cliente. Mantiene una instancia de ConcreteState (Estado Concreto) que define su estado actual.
- **State (Estado):** Define una interfaz para el encapsulamiento de la responsabilidades asociadas con un estado particular de Context.
- **Subclase ConcreteState:** Cada una de estas subclases implementa el comportamiento o responsabilidad de Context.

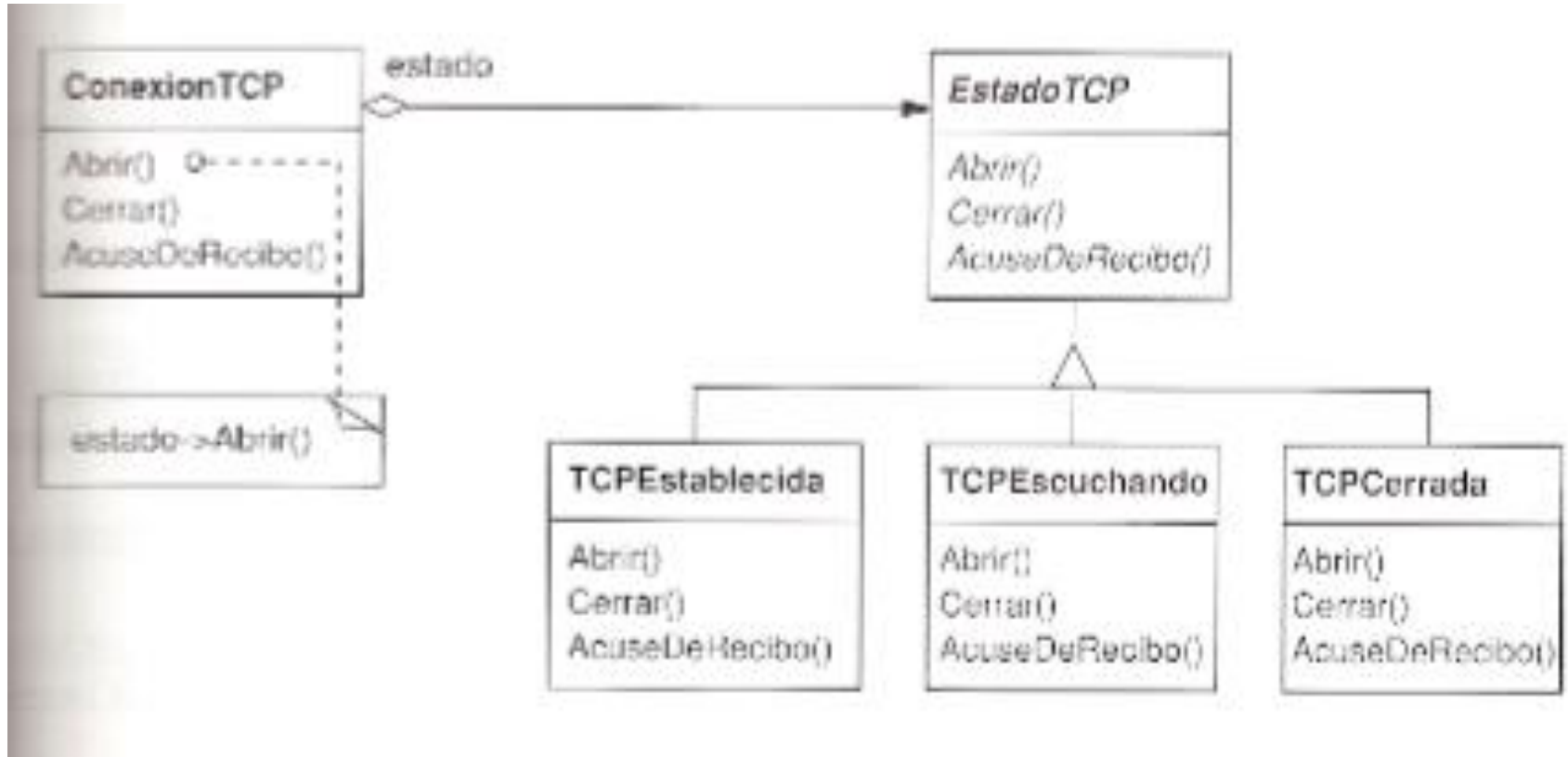
# Consecuencias

- El contexto (el objeto de interés con quien interactúa el cliente) delega las peticiones que dependen del estado en el objeto EstadoConcreto actual.
- El cliente no trata con los estados directamente si no con el contexto.
- El contexto o las subclases de State pueden decidir cual es el siguiente estado.

# Ejemplo

- Una conexión TCP puede estar representada por 3 posibles estados.
  - Establecida, Cerrada, Escuchando.
    - Estas heredan de un State genérico (EstadoTCP).
- Dentro de la conexión TCP se pueden realizar 3 operaciones:
  - Abrir(), Cerrar(), AcuseDeRecibo().
- Conexión TCP mantiene una referencia a EstadoTCP.
  - Y las llamadas a los métodos de conexión TCP se delegan a su atributo de tipo EstadoTCP

# Estructura



# Visitor

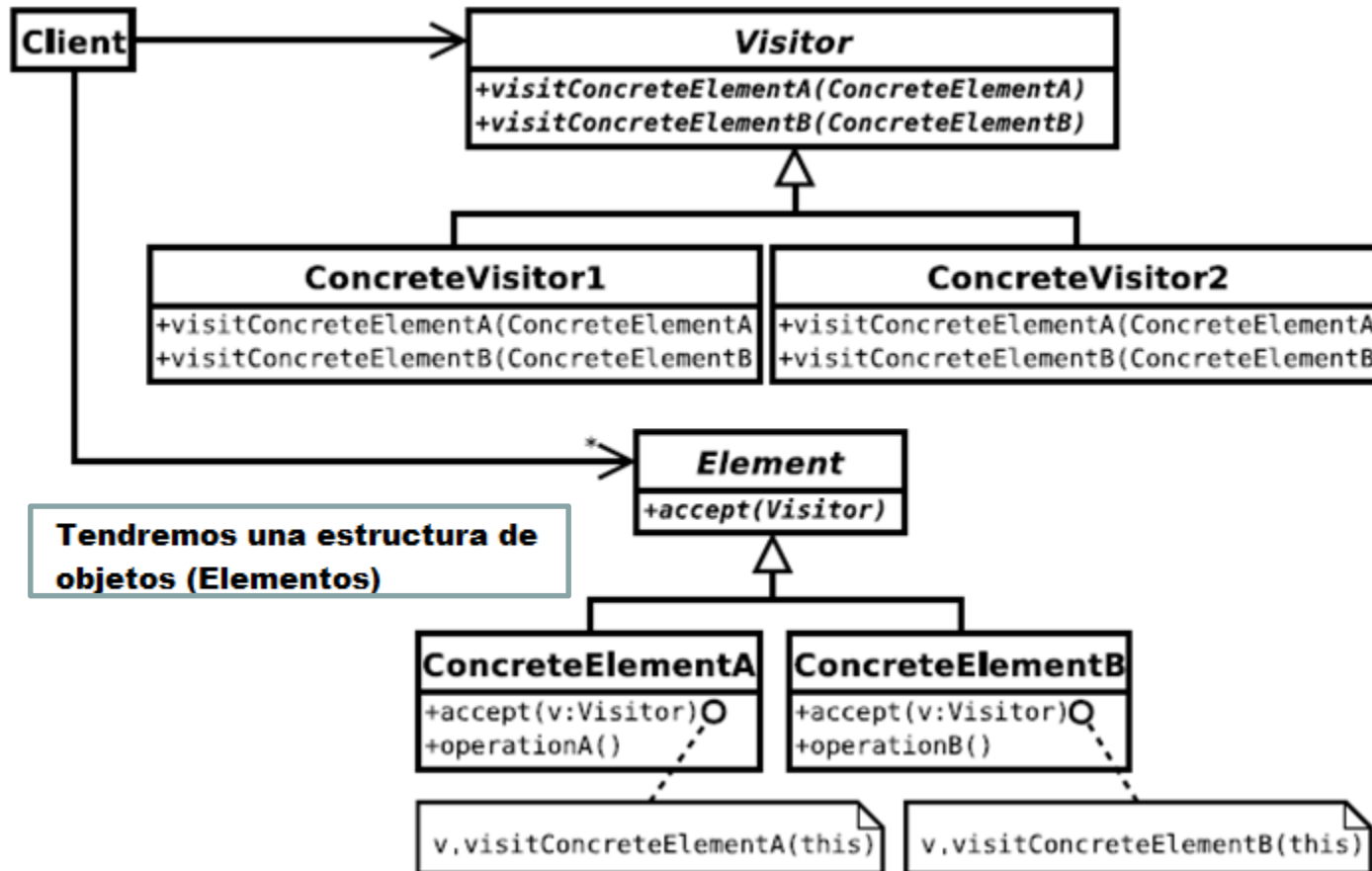
- Representa una operación sobre los elementos de una estructura de objetos.
- Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.
- ***“Es una forma de separar el algoritmo de la estructura de un objeto”***

# Aplicabilidad

- Utilizar cuando:
  - Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces y queremos realizar operaciones sobre esos elementos que dependen de una clase concreta.
  - Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos y no queremos contaminar las clases con esos métodos.
  - La estructura de objetos es bastante estable y no cambia pero queremos añadir nuevas operaciones, con este patrón esas nuevas operaciones se pueden externalizar dentro de una clase (Visitor).



# Estructura



# Participantes

- **Visitante (Visitor):**
  - Declara una operación de visita para cada elemento concreto en la estructura de objetos, que incluye el propio objeto visitado
- **Visitante Concreto (ConcreteVisitor1/2):**
  - Implementa las operaciones del visitante y acumula resultados como estado local
- **Elemento (Element):**
  - Define una operación “Accept” que toma un visitante como argumento
- **Elemento Concreto (ConcreteElementA/B):**
  - Implementa la operación “Accept”
- **EstructuraDeObjetos**
  - Un conjunto de elementos.

# Consecuencias

- **El Visitante facilita el añadir nuevas operaciones** que dependen de los componentes de objetos complejos.
  - Se puede definir una nueva operación simplemente añadiendo un nuevo visitante.
- **Se complica añadir nuevas clases**  
**ElementoConcreto** ya implica el añadir un nuevo método abstracto y su correspondiente implementación en cada Clase VisitanteConcreto.

# Ejemplo

En el ejemplo, hay una jerarquía de expresiones aritméticas simples sobre las que se desea definir visitantes. Uno de los visitantes será la operación PrettyPrint que convierte a cadena de caracteres la expresión aritmética.

