



# Tema 7. Patrones de diseño

## Modelado del Software

Raquel Martínez España

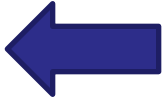
Escuela Politécnica



# Contenidos

- ¿Qué son?
- Introducción
- Tipos de patrones
- Patrones de creación
- Patrones estructurales
- Patrones de comportamiento

# Contenidos

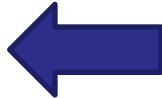
- **¿Qué son?** 
- Introducción
- Tipos de patrones
- Patrones de creación
- Patrones estructurales
- Patrones de comportamiento

# ¿Qué son?

- Un patrón es una solución probada que se puede aplicar con éxito a un determinado **tipo de problemas** que aparecen **repetidamente**.
- Descripción de un problema que se repite a menudo y la solución general del mismo. De forma que puede emplearse esta solución una y otra vez.
  - ➔ **Reutilización** del diseño
  - ➔ **Esqueleto básico** que cada diseñador adaptar a las peculiaridades de su aplicación.
- Propuestos por el *Gang of Four* (**GoF**): Gamma, Helm, Johnson y Vlissides.

**Design Patterns. Elements of Reusable Object-Oriented Software** - *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides* - Addison Wesley. 1994

# Contenidos

- ¿Qué son?
- **Introducción** 
- Tipos de patrones
- Patrones de creación
- Patrones estructurales
- Patrones de comportamiento

# Elementos principales (i)

- **Nombre**

- Descripción del problema en una o dos palabras a lo sumo
- Debe ser representativo de su aplicabilidad
- Un buen nombre facilita la comunicación entre los desarrolladores

- **Problema**

- Especifica cuándo aplicar el patrón: **problema + contexto**
- Lista de condiciones para que se pueda aplicar el patrón.

# Elementos principales (ii)

- **Solución**

- Elementos que forman parte del patrón, relaciones entre ellos, responsabilidades y colaboraciones.
- No se describe una implementación en particular, ofrece una plantilla que se puede aplicar en situaciones distintas.

- **Consecuencias**

- Resultado de aplicar el patrón
- Son críticas para evaluar las alternativas de uso de un patrón concreto.
- Ayudan a entender los costos y beneficios de aplicar el patrón.

# Descripción del patrón

- Nombre y tipo
- **Propósito**
- Sinónimos
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- Participantes
- Colaboraciones
- Consecuencias
- Implementación
- Código de ejemplo
- Usos conocidos
- Patrones relacionados



# Beneficios

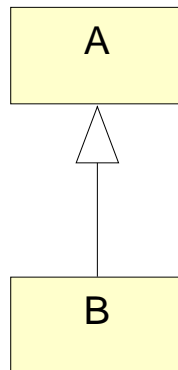
- Ayudan a determinar los **objetos** de una aplicación
- Especifican las **interfaces** de las clases
- Especifican la **implementación** de las clases
- Facilitan realmente la **reutilización**
- Favorecen la reutilización a través de la **composición** en vez de la herencia

# Conceptos importantes

- Herencia vs. Clientela
- Delegación

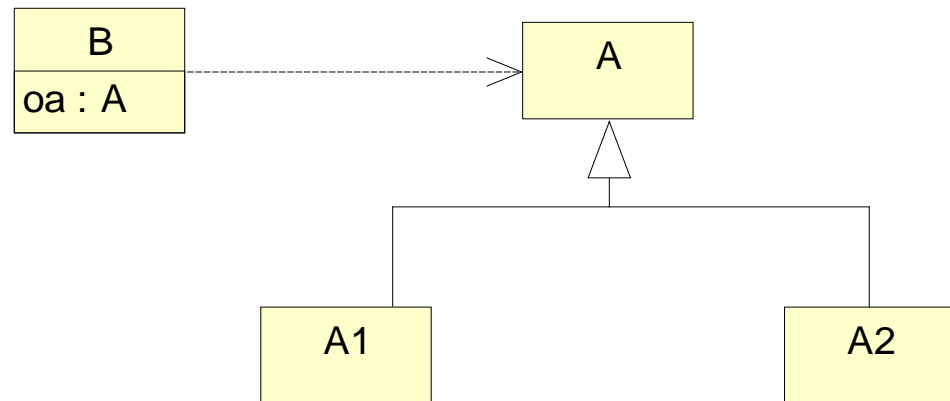
# Herencia vs. Clientela

## Herencia



Relación fija  
Reuso “caja blanca”

## Clientela

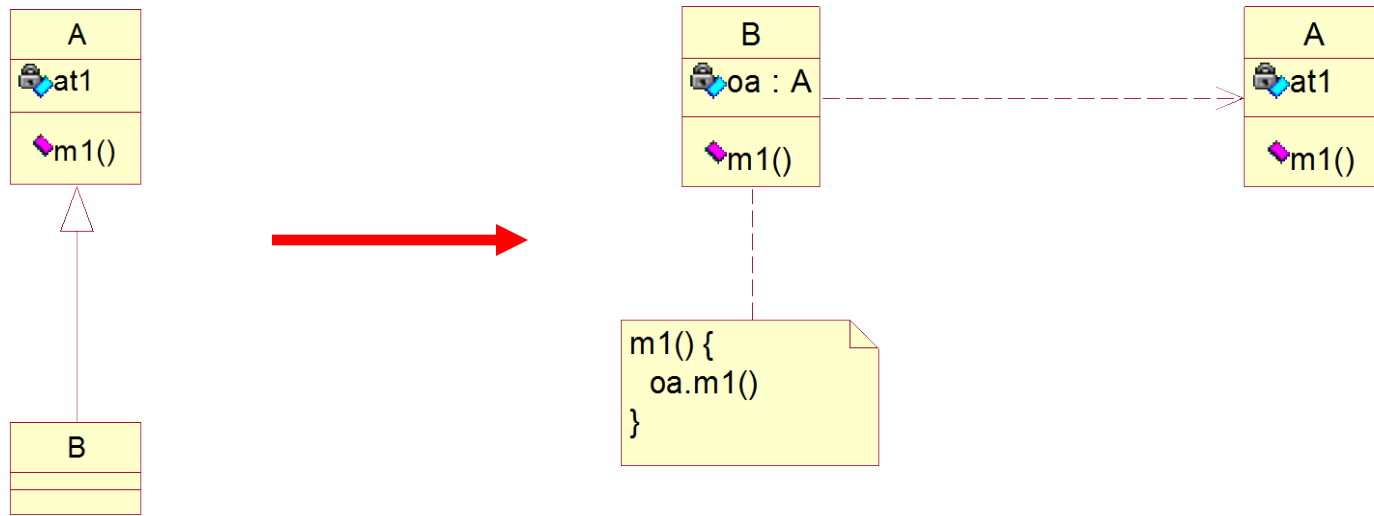


Relación variable  
Reuso “caja negra”

# Delegación

- Forma de hacer que la composición sea tan potente como la herencia.
- Un objeto receptor delega operaciones en su delegado
- Presente en muchos patrones: *State*, *Strategy*, *Visitor*, ...
- Caso extremo de composición, muestra que siempre puede sustituirse la herencia por composición.

# Delegación

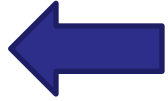


- Si una clase B quiere utilizar los métodos de otra clase A puede ser interesante declarar un atributo de A en B e invocar el método deseado.
- Las clases que hereden de A sobrescribirán **m1** para ofrecer distinta funcionalidad. Pero no habrá que modificar B

# ¿Cómo seleccionar un patrón?

- Considera de que **forma los patrones** resuelven problemas de diseño
- Lee la sección que describe el **propósito** de cada patrón
- Estudia las **interrelaciones** entre patrones
- Analiza **patrones con el mismo propósito**
- Examina las causas de **rediseñar**
- Considera que debería ser **variable** en tu diseño

# Contenidos

- ¿Qué son?
- Introducción
- **Tipos de patrones** 
- Patrones de creación
- Patrones estructurales
- Patrones de comportamiento

# Clasificación

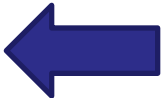
- Según el propósito (¿Qué hace el patrón?)
  - Creacional
  - Estructural
  - De Comportamiento
- Según el ámbito:
  - De Clases
  - De Objetos



# Tipos de patrones

		PROPÓSITO		
		De creación	Estructural	De comportamiento
ÁMBITO	Clase (Herencia)	Factory Method	Adapter(class)	Template Method
	Objeto (Composición)	<b>Abstract Factory</b> <b>Builder</b> Prototype Singleton	Adapter(object) <b>Bridge</b> <b>Composite</b> <b>Decorator</b> Facade <b>Flyweight</b> Proxy	<b>Chain of responsibility</b> Command Iterator Mediator Memento Observer <b>State</b> <b>Strategy</b> <b>Visitor</b>

# Contenidos

- ¿Qué son?
- Introducción
- Tipos de patrones
- **Patrones de creación** 
- Patrones estructurales
- Patrones de comportamiento

# Patrones de creación

- Abstraen el proceso de creación de objetos.
- Ayudan a crear sistemas independientes de cómo los objetos son creados, compuestos y representados.
- El sistema conoce las clases abstractas
- Flexibilidad en **qué** se crea, **quién** lo crea, **cuándo** se crea y **cómo** se crea.

# Patrones de creación

- Creación objetos: mecanismo de instanciación
- Típicamente, llamada el operador *new()*  
*objeto = new Clase();*
- Cuando hay que distinguir el tipo de objeto a crear:

```
public Documento construyeDoc(String tipoDo) {  
    Documento resultado;  
    if (tipoDoc.equals("PDF"))  
        resultado = new DocumentoPDF();  
    else if (tipoDoc.equals("RTF"))  
        resultado = new DocumentoRTF();  
    else if (tipoDoc.equals("HTML"))  
        resultado = new DocumentoHTML();  
    // continuación del método }  
}
```

# Patrones de creación

- Es difícil configurar el mecanismo de creación de objetos.
- ¿Qué sucede si cambia la jerarquía de clases a instanciar? ➔ El uso de sentencias condicionales tiene no es la forma idónea.
- **Objetivo de los patrones de creación:** ofrecer un método más flexible para la creación de clases donde el código del cliente no se vea afectado ante un cambio en la jerarquía de clases.

# Patrones de creación

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

# Abstract Factory

- **Propósito**

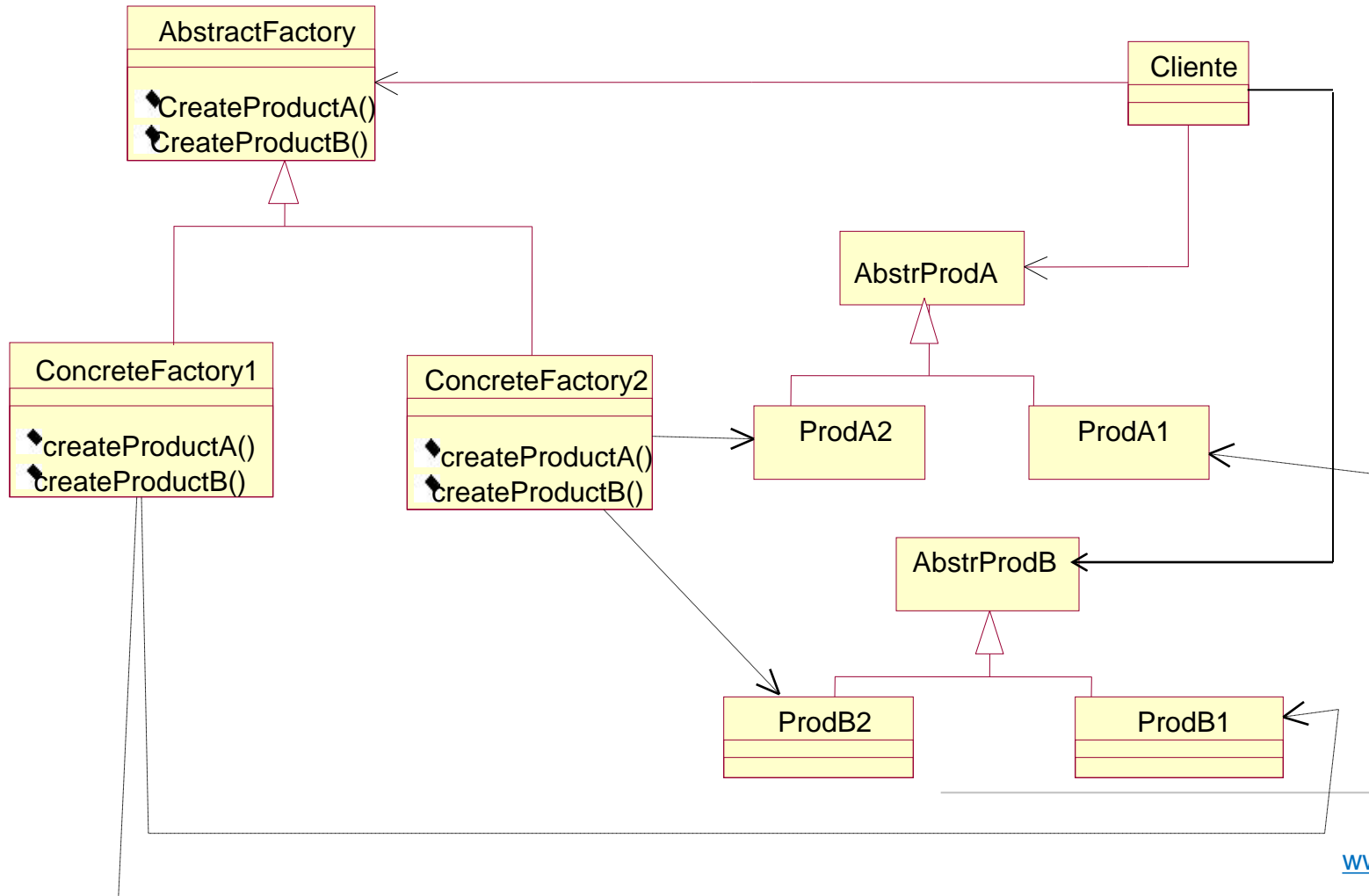
- Proporcionar una interfaz para crear **familias de objetos** relacionados o dependientes sin especificar la clase concreta

- **Motivación**

- Un *toolkit* interfaz de usuario que soporta diferentes formatos: Windows, Motif, X-Windows,...

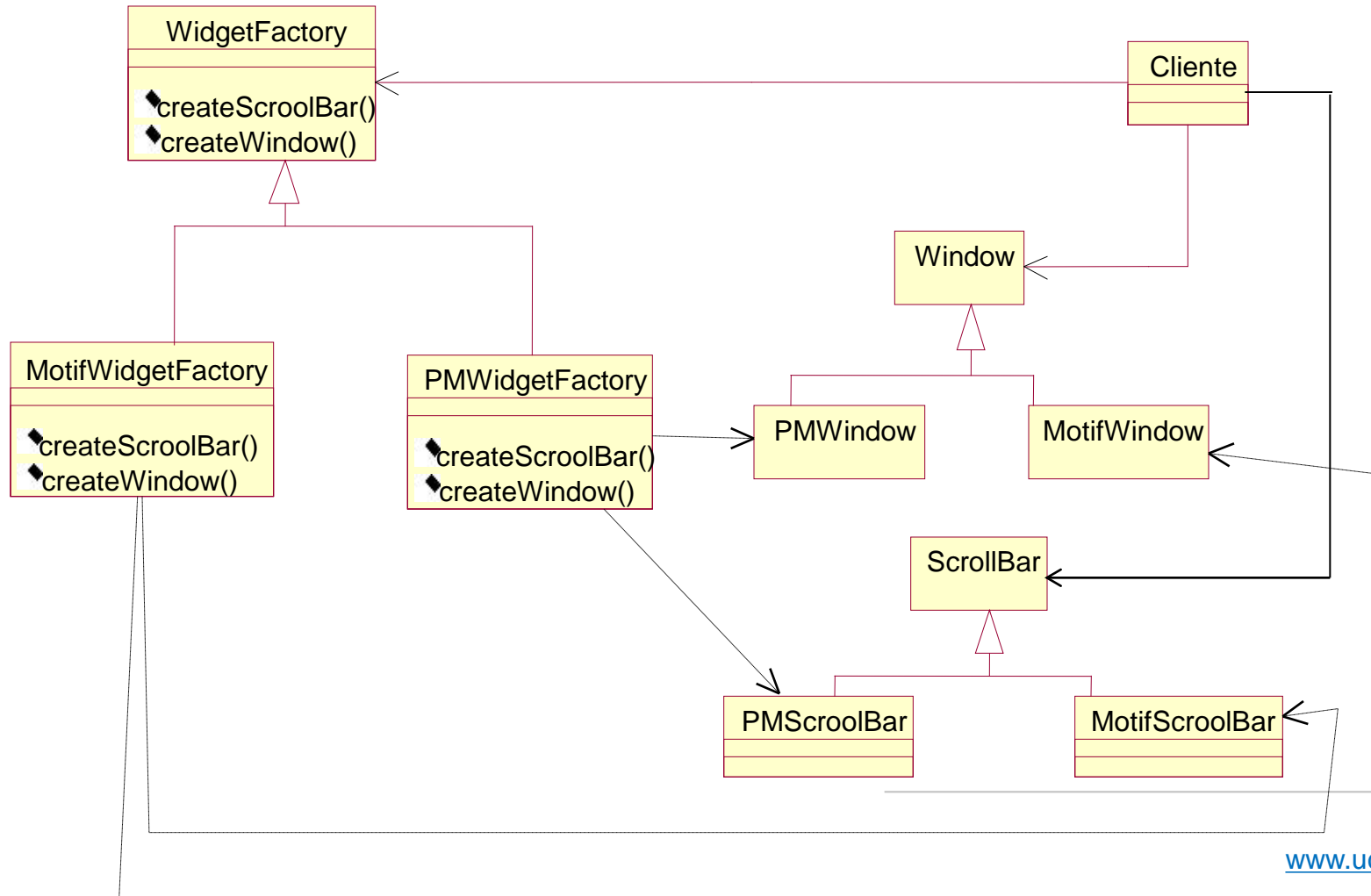
## Estructura

# Abstract Factory





# Abstract Factory



# Abstract Factory

- **Aplicabilidad**

- Un sistema debería ser independiente de cómo sus productos son creados, compuestos y representados
- Un sistema debería ser configurado para una familia de productos.
- Una familia de objetos “productos” relacionados es diseñada para ser utilizada juntos y se necesita forzar la restricción.

# Abstract Factory

- **Consecuencias**

- Facilita el cambio de productos ya que la instanciación de la **ConcreteFactory** aparece en un único lugar. Incluso podría ser un parámetro.
- Dificulta la inclusión de nuevos productos. Un nuevo producto implica cambiar **AbstractFactory** y sus subclases

- **Implementación**

- Definir las fábricas con el patrón *Singleton*
- Incluir un método **createFactory** para cada producto

# Builder

- **Propósito**

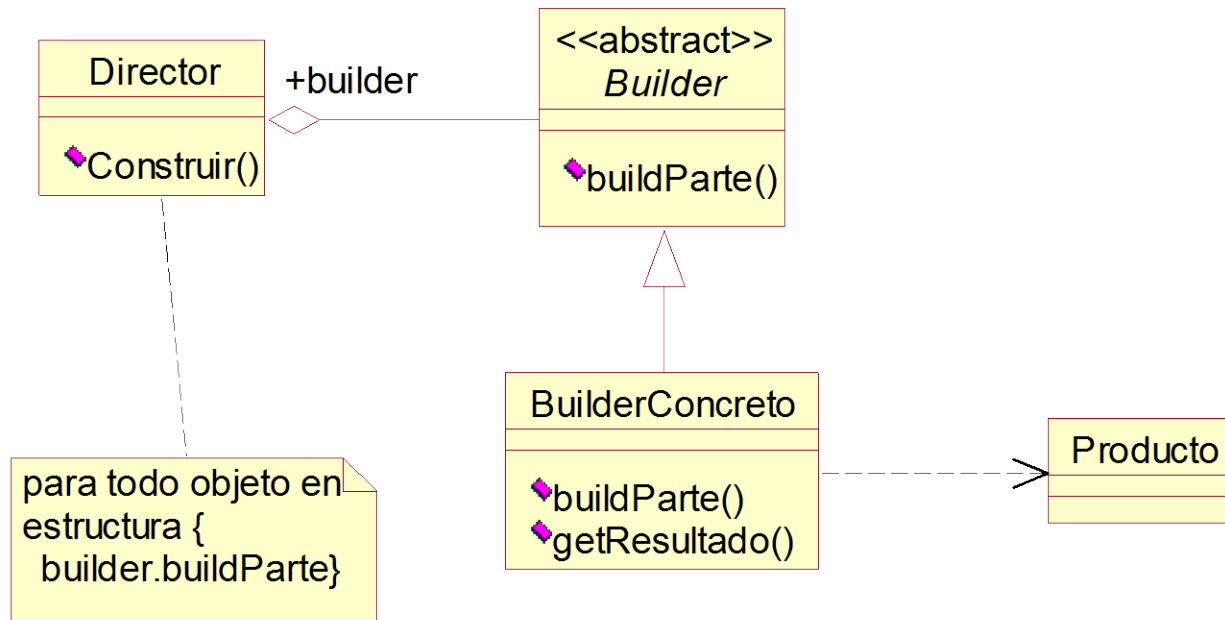
- Separa la **construcción de un objeto complejo** de su representación, así que el mismo proceso de construcción puede crear diferentes representaciones.

- **Motivación**

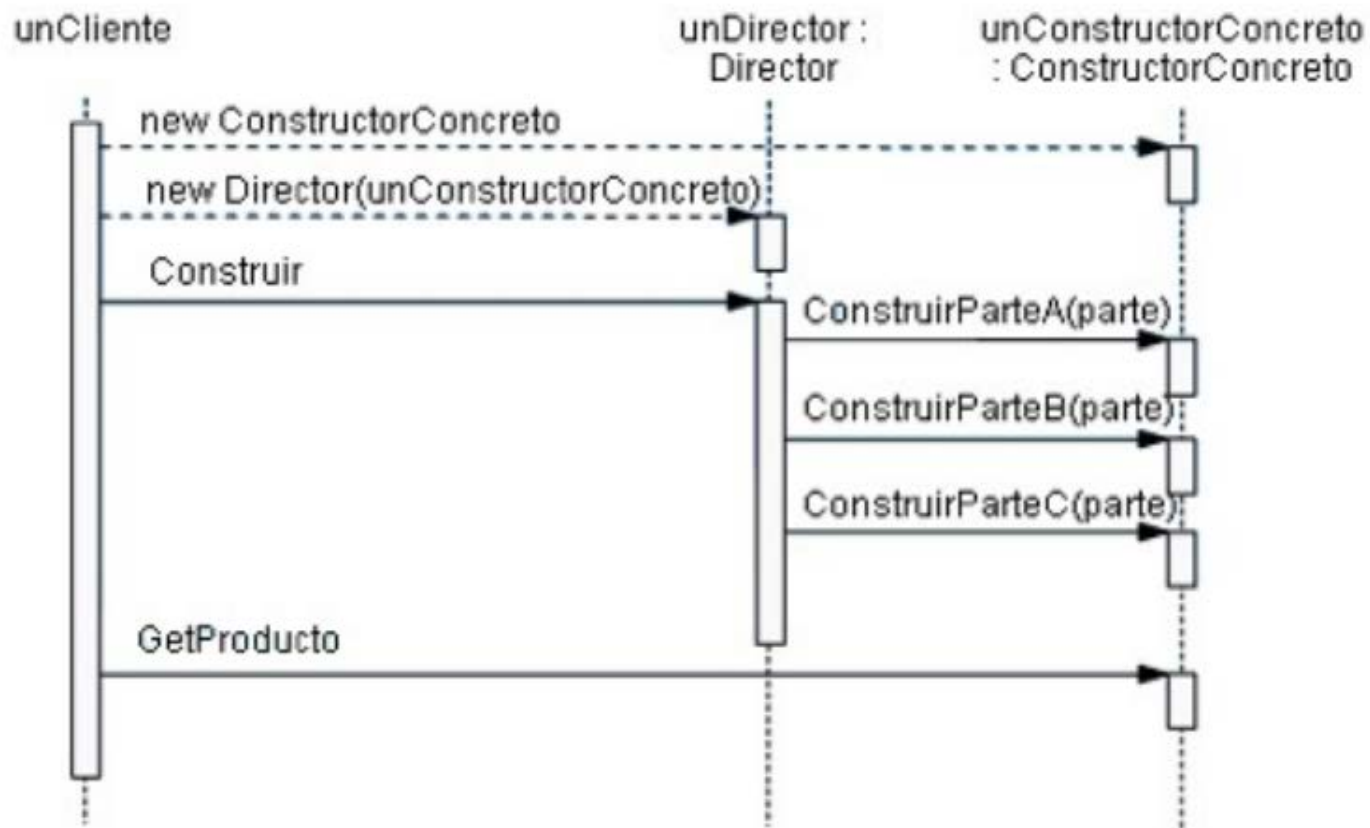
- Un traductor de documentos RTF a otros formatos. ¿Es posible añadir una nueva conversión sin modificar el traductor?

# Builder

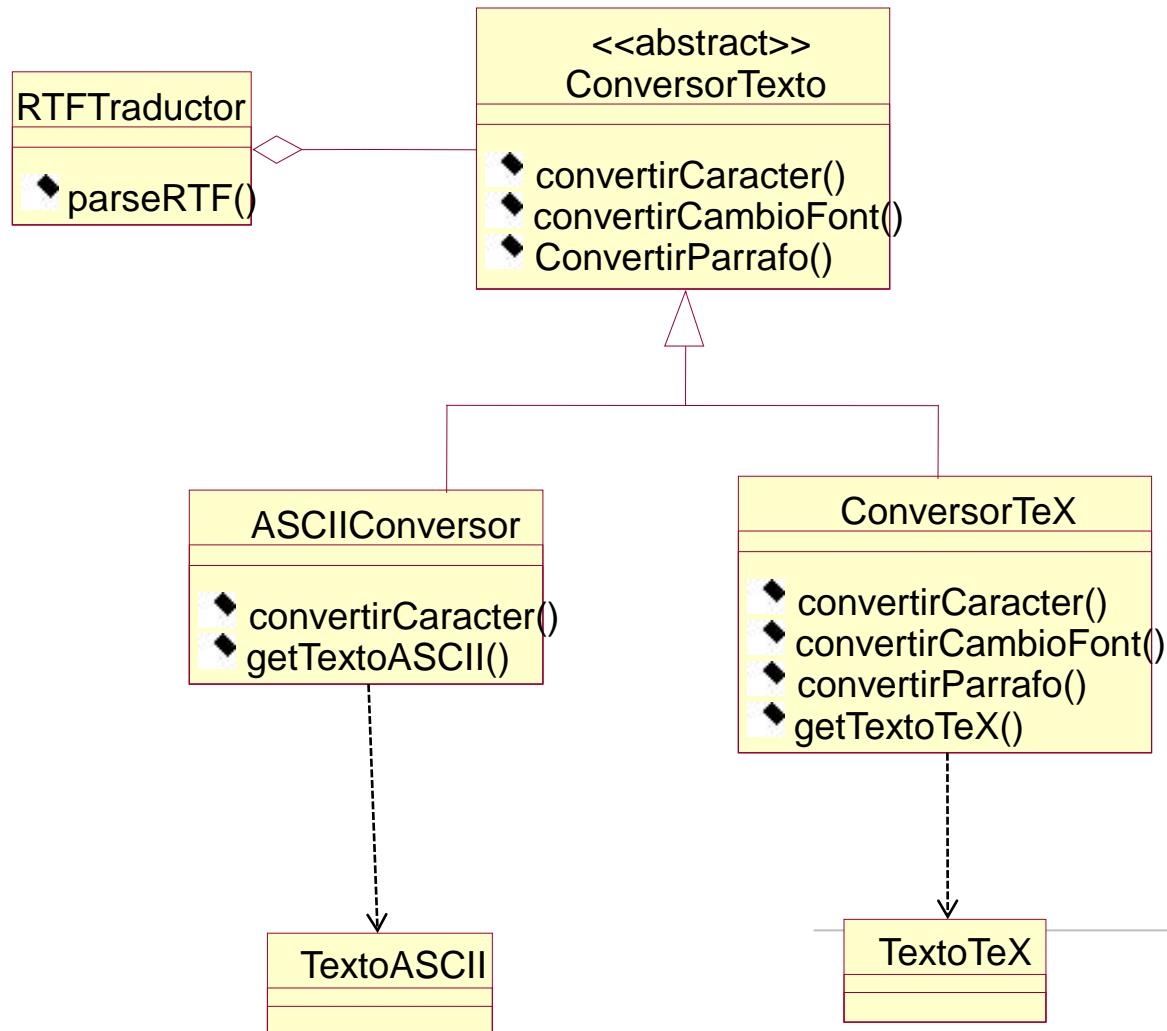
## Estructura



# Builder



# Builder



# Builder

- **Aplicabilidad**
  - Cuando **el algoritmo para crear un objeto complejo debe ser independiente de las piezas que conforman el objeto** y de cómo se ensamblan.
  - El proceso de construcción debe permitir diferentes representaciones para el objeto que se construye.



# Builder

- **Consecuencias**

- Permite cambiar la representación interna del producto.
- Separa el código para la representación y para la construcción.
- Los clientes no necesitan conocer nada sobre la estructura interna.
- Diferentes “directores” pueden reutilizar un mismo “builder”
- Proporciona un control fino del proceso de construcción.

# Builder

- **Implementación**

- La interfaz de *Builder* debe ser lo suficientemente general para permitir la construcción de productos para cualquier *Builder* concreto.
- La construcción puede ser más complicada de añadir el nuevo *token* al producto en construcción.
- Los métodos de *Builder* no son abstractos sino vacíos.
- Las clases de los productos no siempre tienen una clase abstracta común.

# Factory Method

- **Propósito**

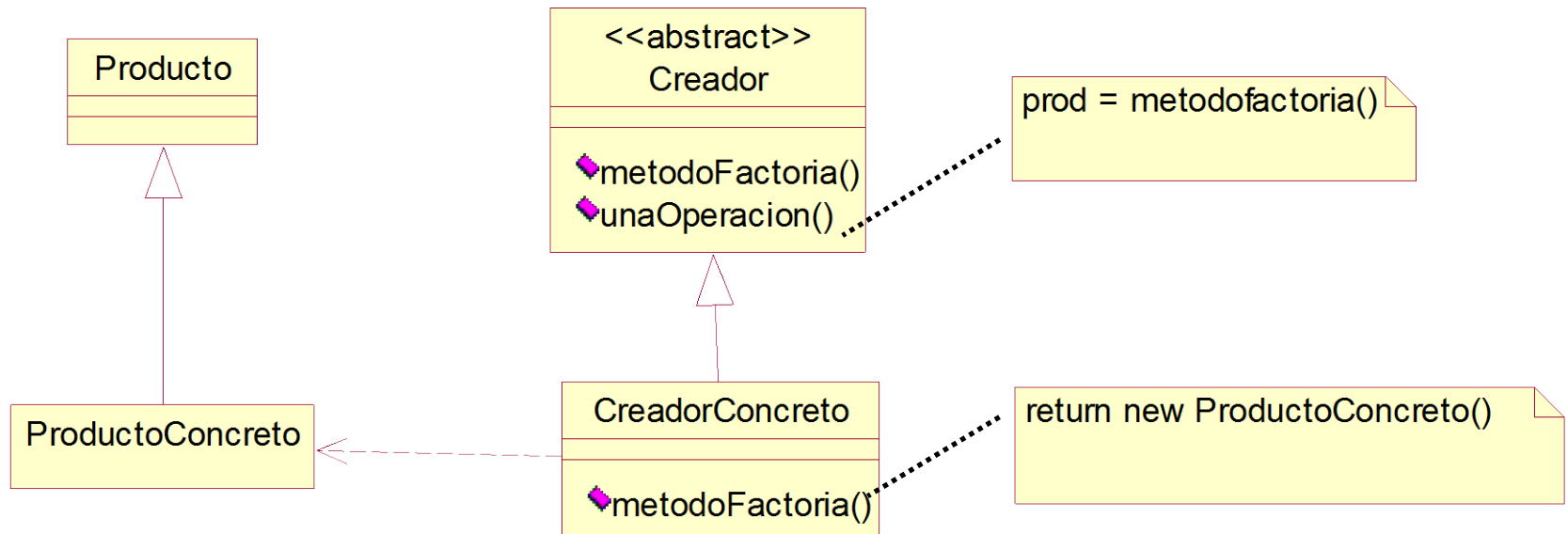
- Define un interfaz para **crear un objeto**, pero permite a las subclases decidir la clase a instanciar: ***instanciación diferida a las subclases.***

- **Motivación**

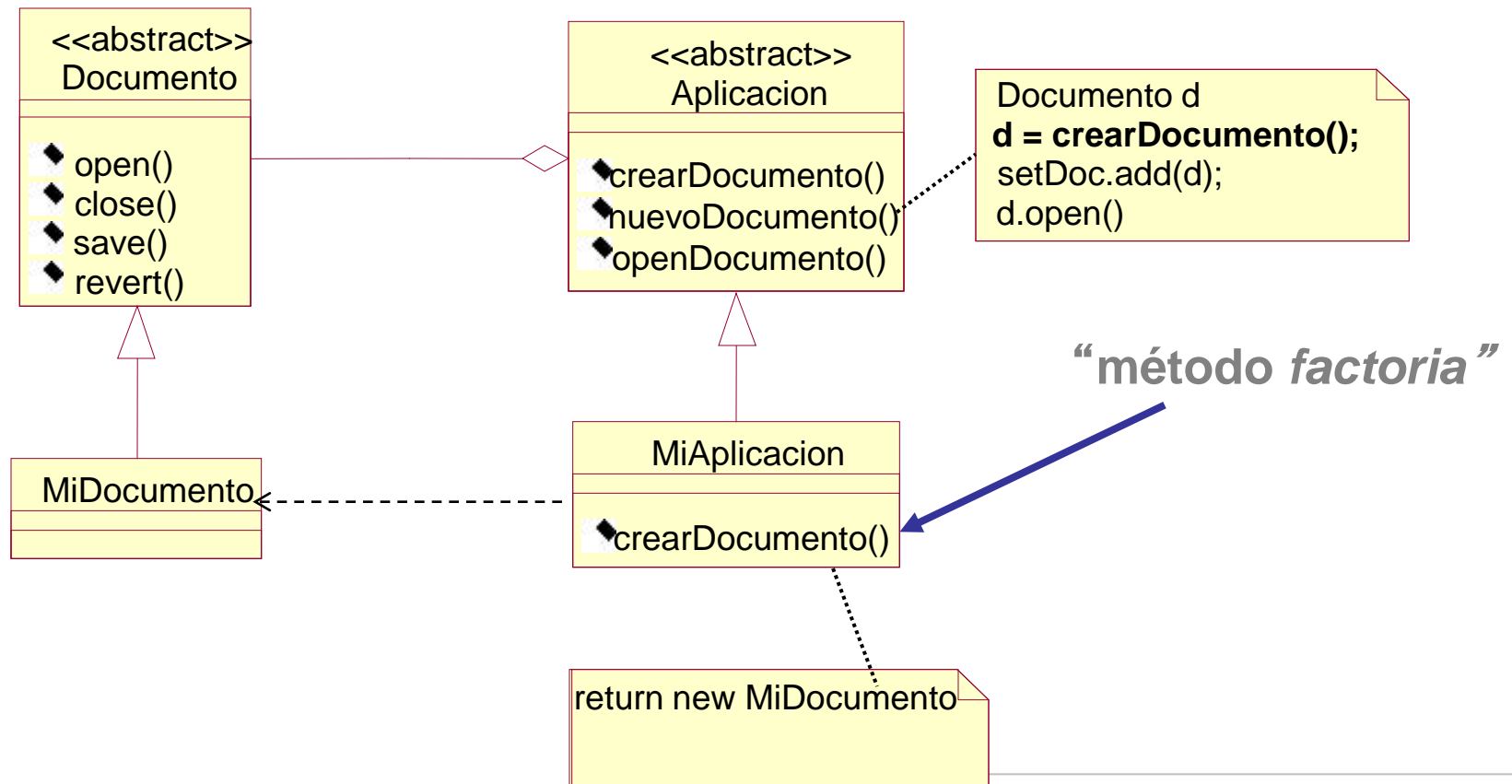
- Una clase C cliente de una clase abstracta A necesita crear instancias de subclases de A que no conoce.
- En un *framework* para aplicaciones que pueden presentar al usuario documentos de distinto tipo: clases Aplicación y Documento.
- Se conoce **cuándo** crear un documento, no se conoce de **qué** tipo.

# Factory Method

## Estructura



# Factory Method



# Factory Method

- **Aplicabilidad**

- Una clase no puede anticipar la clase de objetos que debe crear.
- Una clase desea que sus subclases especifiquen los objetos que debe crear.

- **Consecuencias**

- Evita ligar un código a clases específicas de la aplicación.
- Puede suceder que las subclases de *Creador* sólo se crean con el fin de la creación de objetos.
- Mayor flexibilidad en la creación: subclases ofreciendo versiones extendidas de un objeto.

# Prototipo

- **Propósito**

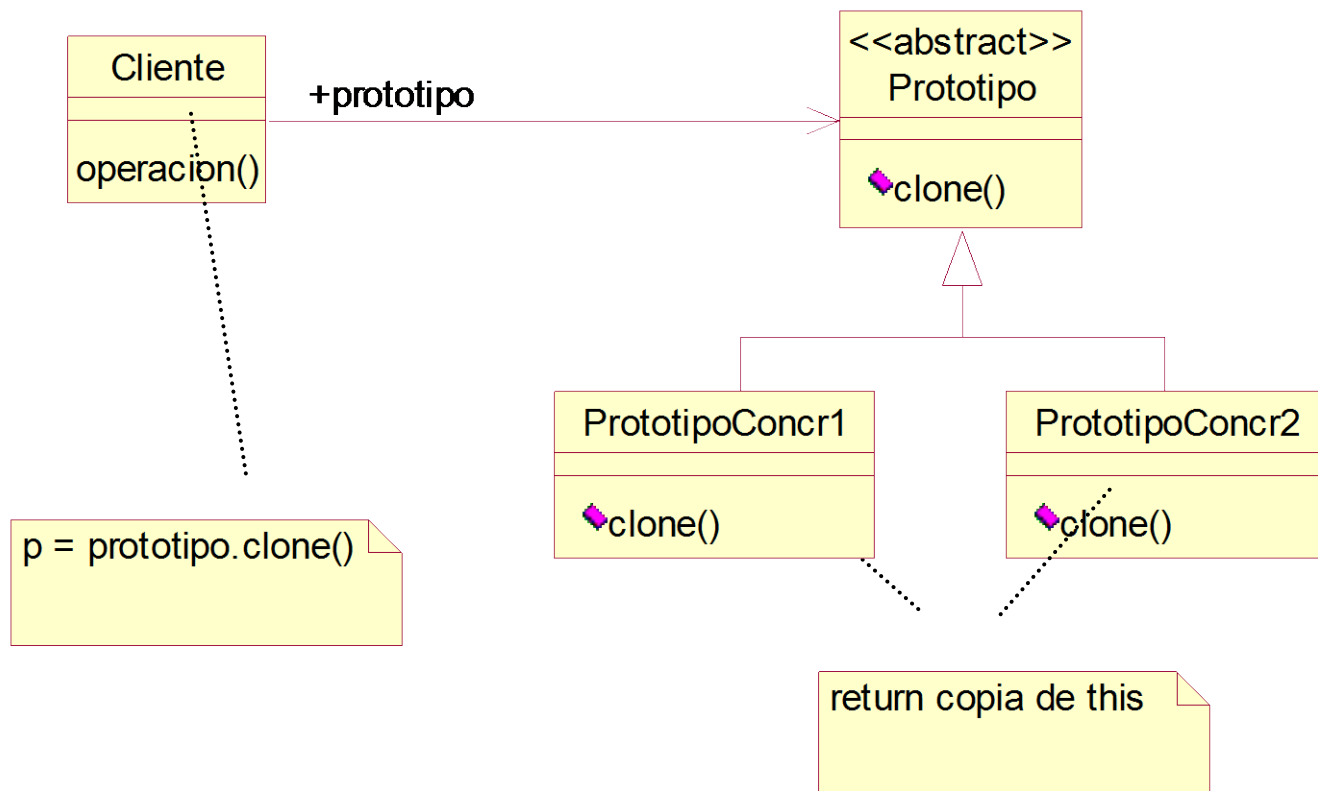
- Especificar los tipos de objetos a crear utilizando una **instancia prototípica**, y crear nuevas instancias mediante **copias del prototipo**.

- **Motivación**

- GraphicTool es una clase dentro de un *framework* de creación de editores gráficos, que crea objetos gráficos, (instancias de subclases de Graphic) y los inserta en un documento, **¿cómo puede hacerlo si no sabe qué objetos gráficos debe crear?**

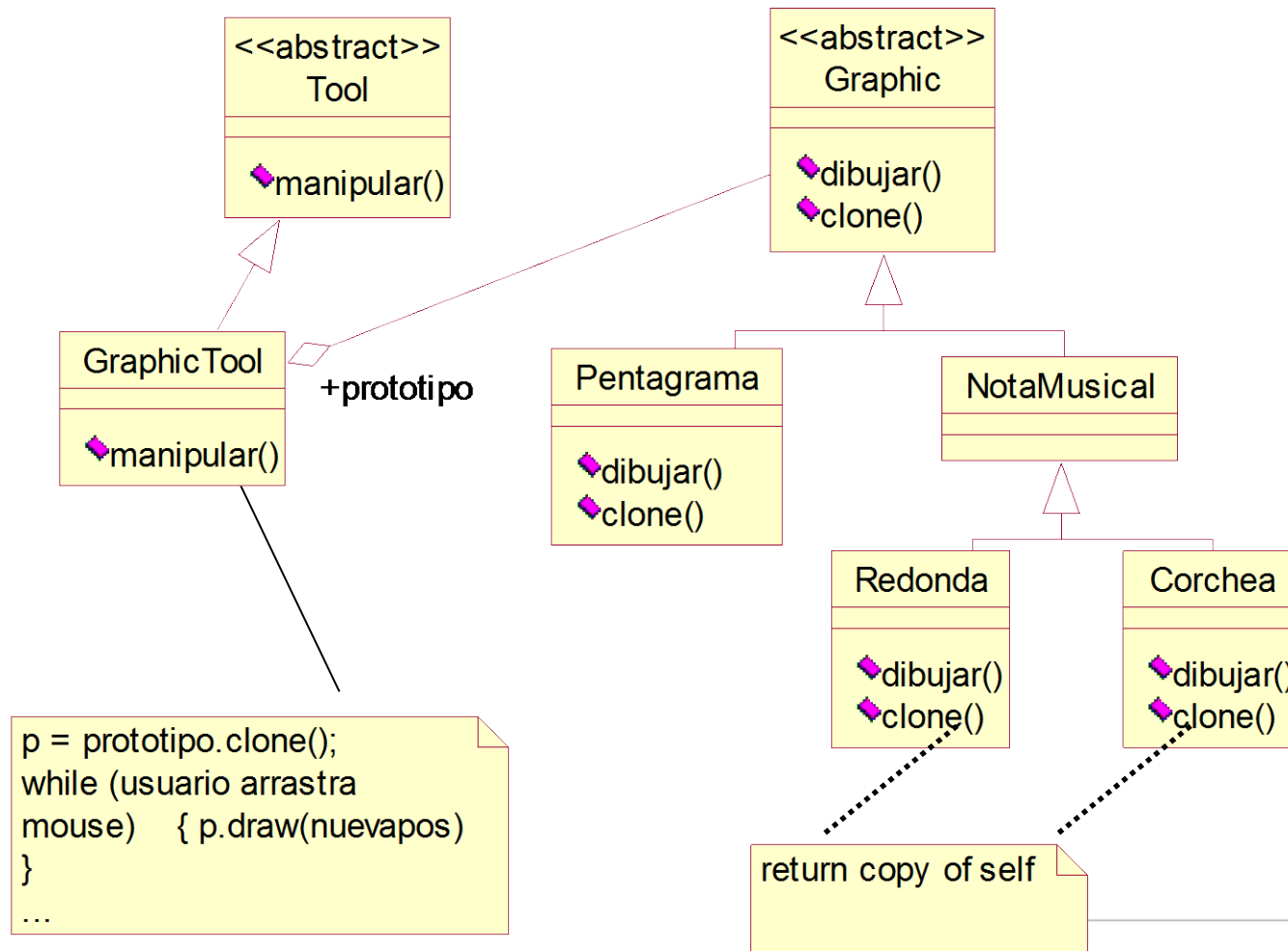
# Prototipo

## Estructura





# Prototipo



# Prototipo

- **Aplicabilidad**

- Cuando un sistema deba ser independiente de cómo sus productos son creados, compuestos y representados y:
  - Las clases a instanciar son especificadas **en tiempo de ejecución**.
  - Para evitar crear una jerarquía de factorías paralela a la de productos.
  - Cuando las instancias de una clase sólo pueden encontrarse en uno de un pequeño grupo de estados.
  - Inicializar un objeto es costoso

# Prototipo

- **Consecuencias**

- Al igual que con *Builder* y *Abstract Factory* oculta al cliente las clases producto concretas.
- Es posible añadir y eliminar productos en tiempo de ejecución: mayor flexibilidad que los anteriores.
- Reduce la necesidad de crear subclases.

# Prototipo

- **Implementación**

- Importante en lenguajes que no soportan el **concepto de metacalse**, así en Smalltalk o Java es menos importante.
- Usar un manejador de prototipos que se encarga del mantenimiento de una tabla de prototipos.
- Implementar la operación *clone* es lo más complicado.
- A veces se requiere inicializar el objeto creado mediante copia: prototipo con operaciones “*set*”

# Singleton

- **Propósito**

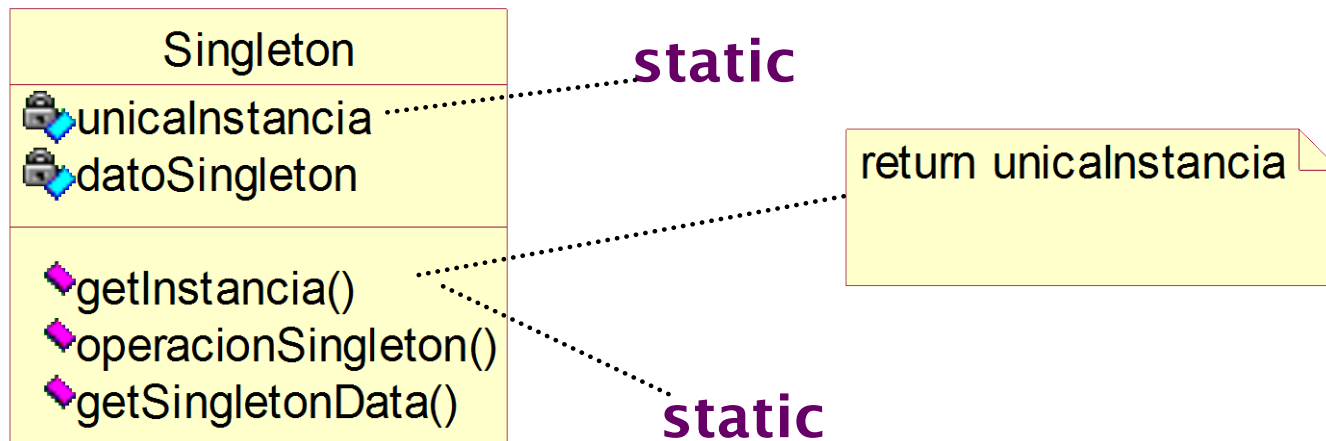
- Asegurar que una clase tiene una única instancia y asegurar un punto de acceso global.

- **Motivación**

- Un manejador de ventanas, o un *spooler* de impresoras o de un sistema de ficheros o una factoría abstracta.
- La clase se encarga de asegurar que exista una única instancia y de su acceso.

# Singleton

## Estructura



`Singleton.getInstance().operacionSingleton()`

# Codigo clase Singleton

```
public class Singleton {  
    public static Singleton getInstance() {  
        if (unicaInstancia == null)  
            unicaInstancia = new Singleton();  
        return unicaInstancia;  
    }  
    private Singleton() { }  
    private static Singleton unicaInstancia = null;  
  
    private int dato = 0;  
    public int getDato() {return dato;}  
    public void setDato(int i) {dato = i;}  
}
```

# Singleton


- **Aplicabilidad**
  - Debe existir una única instancia de una clase, accesible globalmente.
- **Consecuencias**
  - Acceso controlado a la única instancia
  - Evita usar variables globales
  - Generalizar a un número variable de instancias
  - La clase *Singleton* puede tener subclases.



# Singleton

```
public class TestSingleton {  
    public static void main(String args[]) {  
        Singleton s = Singleton.getInstance();  
        s.setDato(34);  
        System.out.println("Primera referencia: " + s);  
        System.out.println("Valor de dato es: " + s.getDato());  
        s = null;  
        s = Singleton.getInstance();  
        System.out.println("Segunda referencia: " + s);  
        System.out.println("Valor de dato es " + s.getDato());  
    }  
}
```

# Contenidos

- ¿Qué son?
- Introducción
- Tipos de patrones
- Patrones de creación
- **Patrones estructurales** 
- Patrones de comportamiento

# Patrones estructurales

- Cómo clases y objetos se combinan para formar estructuras más complejas.
- **Patrones basados en herencia** para componer interfaces o implementaciones.
- **Patrones basados en composición**
  - Describen formas de combinar objetos para obtener nueva funcionalidad
  - Posibilidad de cambiar la composición en tiempo de ejecución.

# Patrones estructurales

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

# Adapter

- **Propósito**

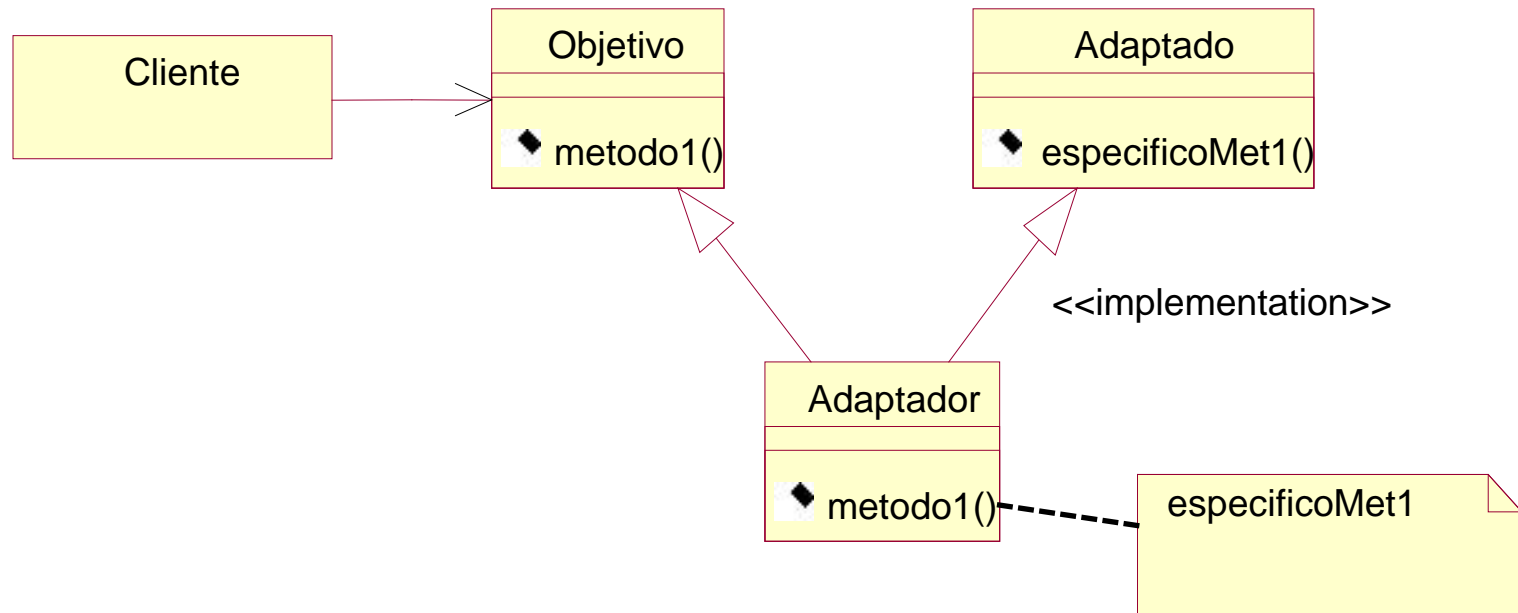
- **Convertir la interfaz de una clase en otra que el cliente espera.** Permite la colaboración de ciertas clases a pesar de tener interfaces incompatibles

- **Motivación**

- Un editor gráfico incluye una jerarquía que modela elementos gráficos (líneas, polígonos, texto,..) y se desea reutilizar una clase existente *TextView* para implementar la clase que representa elementos de texto, pero que tiene una interfaz incompatible.
- A menudo un *toolkit* o librería de clases no se puede utilizar debido a que su interfaz es incompatible con la interfaz requerida por la aplicación.
- No debemos o podemos cambiar la interfaz de la librería de clases

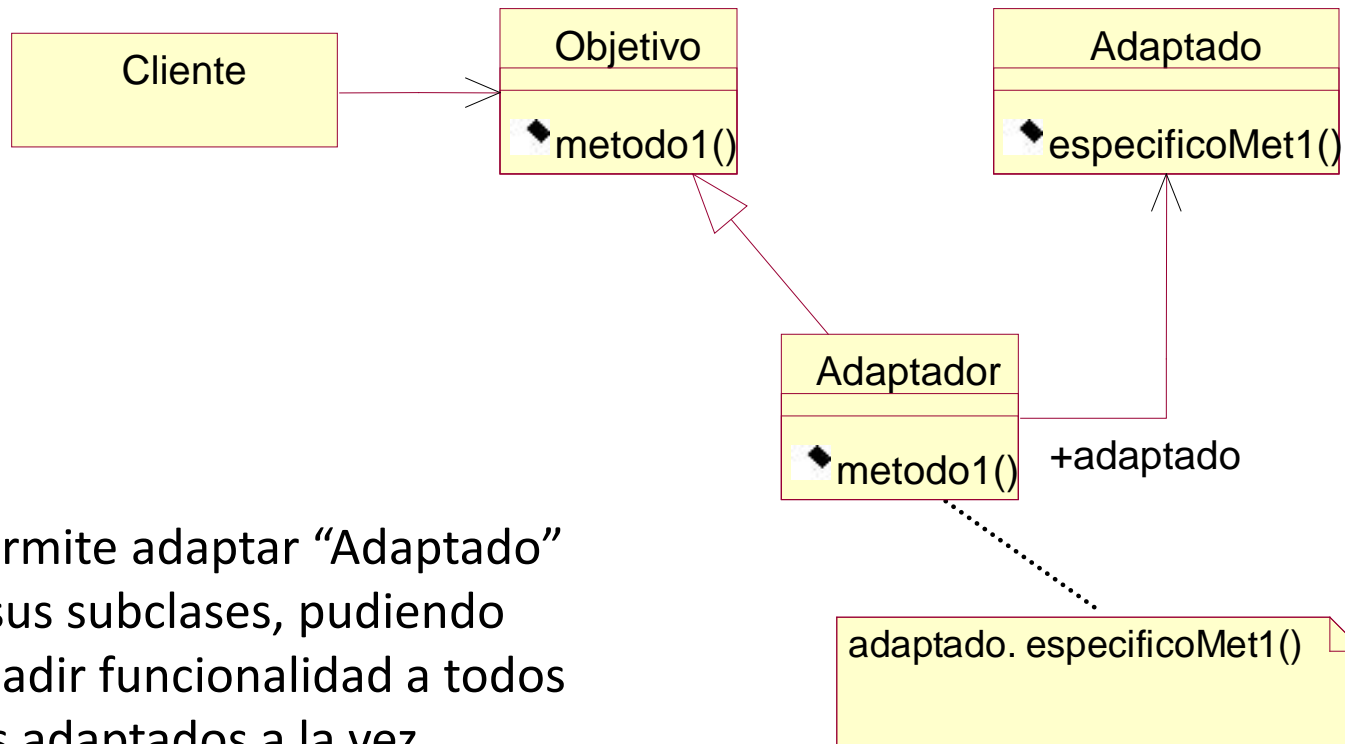
# Adapter (Herencia)

## Estructura



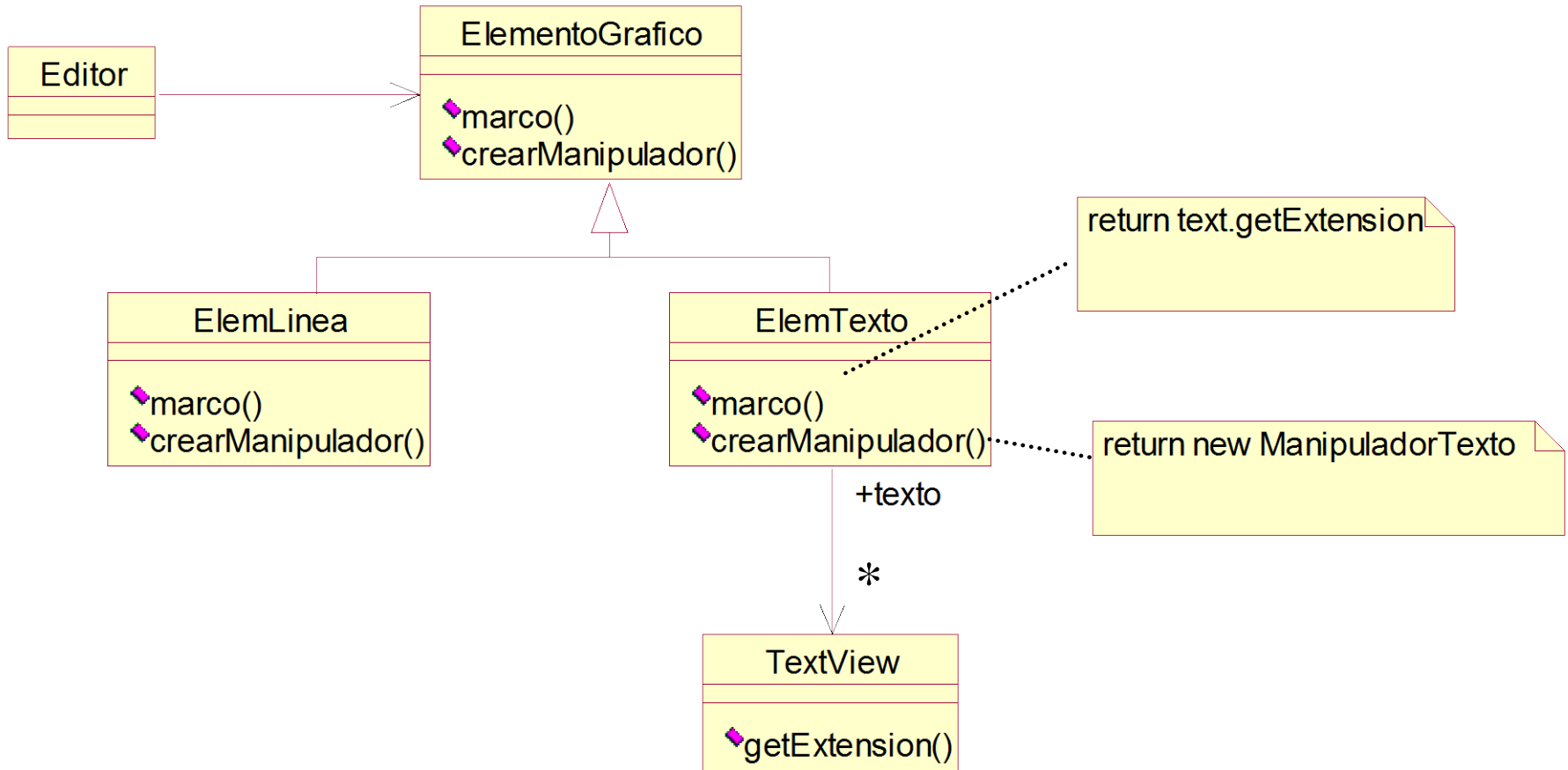
# Adapter (Composición)

## Estructura



Permite adaptar “Adaptado” y sus subclases, pudiendo añadir funcionalidad a todos los adaptados a la vez.

# Adapter





# Adapter

- **Aplicabilidad**
  - Se desea usar una clase existente y su interfaz no coincide con la que se necesita.
  - Se desea crear una clase reutilizable que debe colaborar con clases no relacionadas o imprevistas.

# Bridge

- **Propósito**

- **Desacoplar una abstracción de su implementación**, de modo que los dos puedan cambiar independientemente.

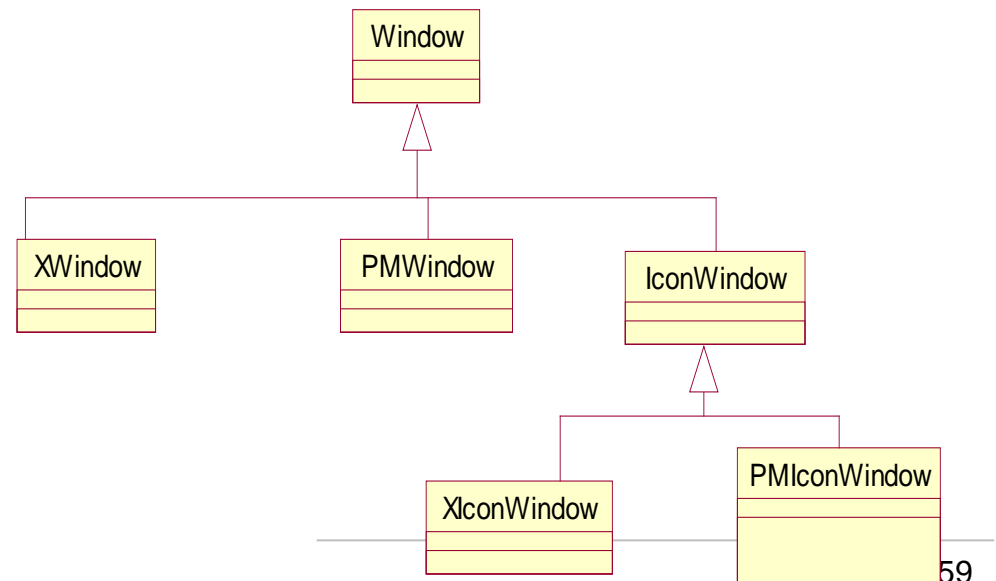
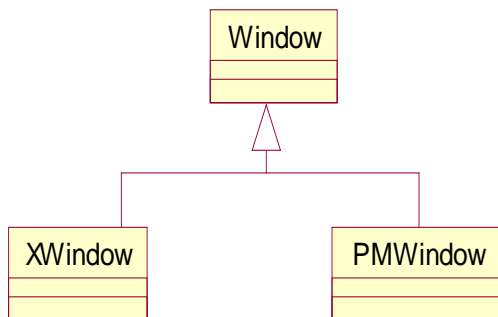
- **Motivación**

- Clase que modela la abstracción con subclases que la implementan de distintos modos.
- **Herencia hace difícil reutilizar abstracciones** e implementaciones de forma independiente
  - Si refinamos la abstracción en una nueva subclase, esta tendrá tantas subclases como tenía su superclase (ver siguiente transp.).
  - El código cliente es dependiente de la implementación.

# Bridge

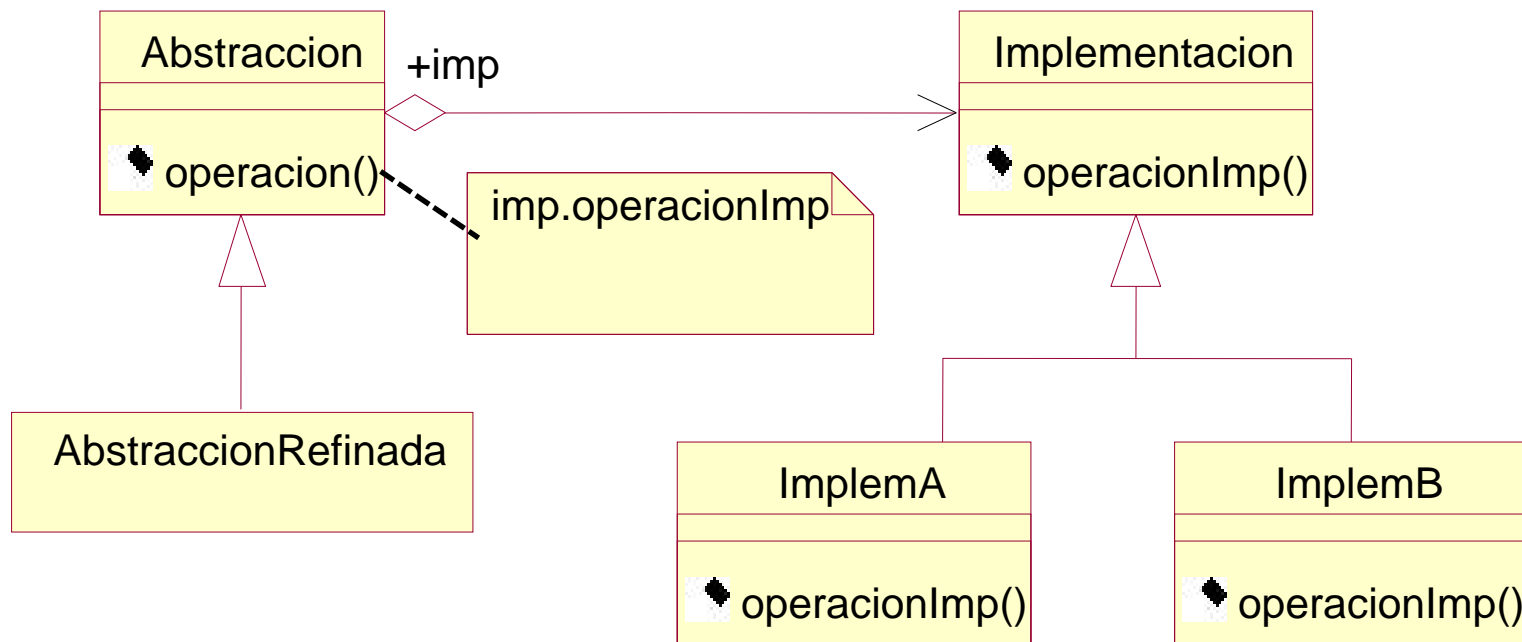
- **Motivación**

- Implementación de una abstracción “*window*” portable en una librería GUI.

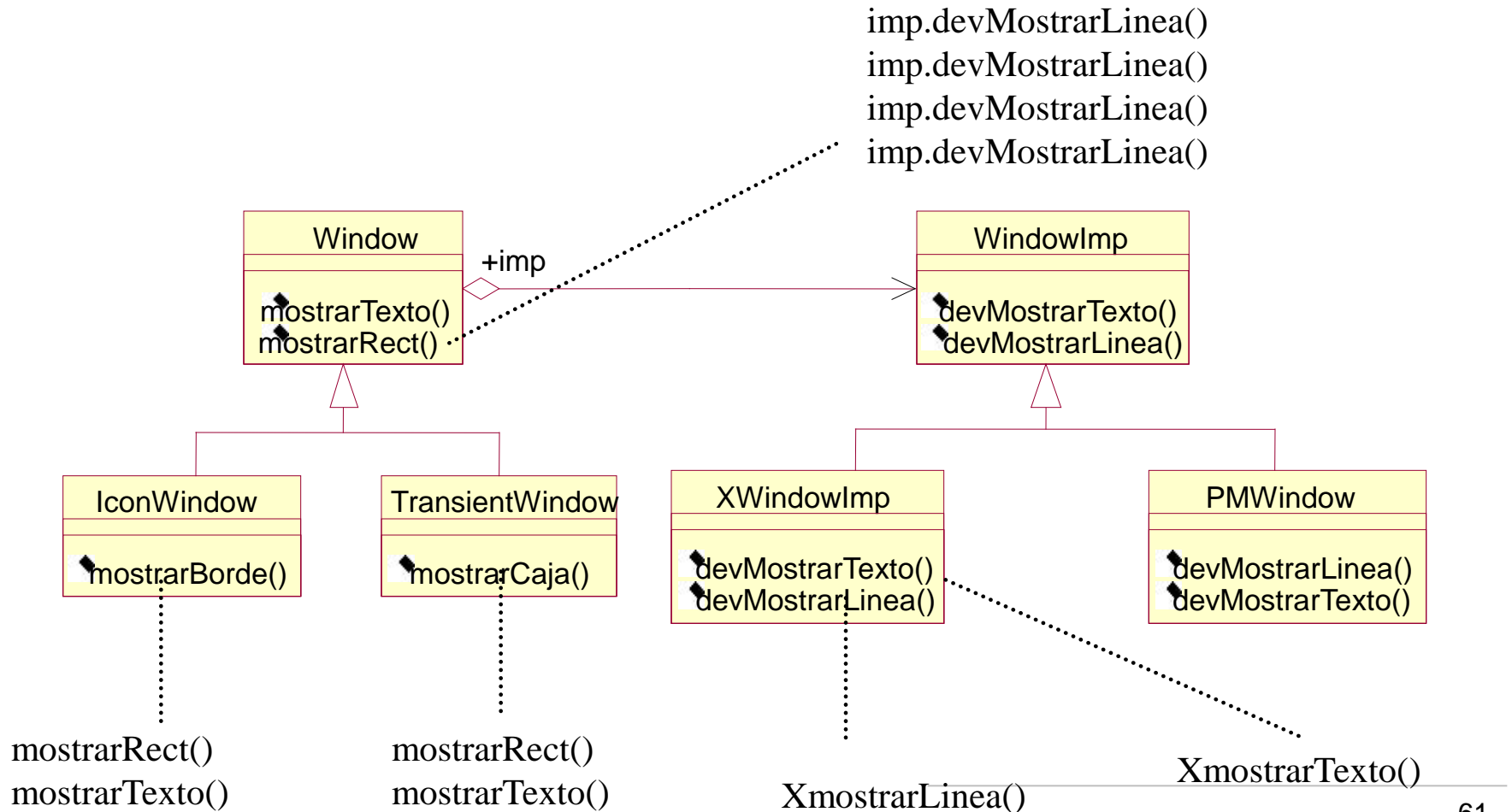


# Bridge

## Estructura



# Bridge



# Bridge

- **Aplicabilidad**

- Se quiere **evitar una ligadura permanente entre una abstracción y su implementación**, p.ej. porque se quiere elegir en tiempo de ejecución.
- Abstracciones e implementaciones son extensibles.
- Cambios en la implementación de una abstracción no deben afectar a los clientes (no recompilación).
- Ocultar a los clientes la implementación de la interfaz.
- Se tiene una proliferación de clases, como sucedía en la *Motivación*.

# Bridge

- **Consecuencias**

- Un objeto puede cambiar su implementación en tiempo de ejecución.
- Cambios en la implementación no requerirán compilar de nuevo la clase Abstracción y sus clientes.
- Se mejora la extensibilidad.
- Se ocultan detalles de implementación a los clientes.

# Bridge

- **Implementación**

- Aunque exista una única implementación puede usarse el patrón para evitar que un cliente se vea afectado si cambia.
- ¿Cómo, cuándo y dónde se decide que implementación usar?
  - Constructor de la clase `Abstraccion`
  - Elegir una implementación por defecto
  - Delegar a otro objeto, por ejemplo un objeto factoría



# Composite

- **Propósito**

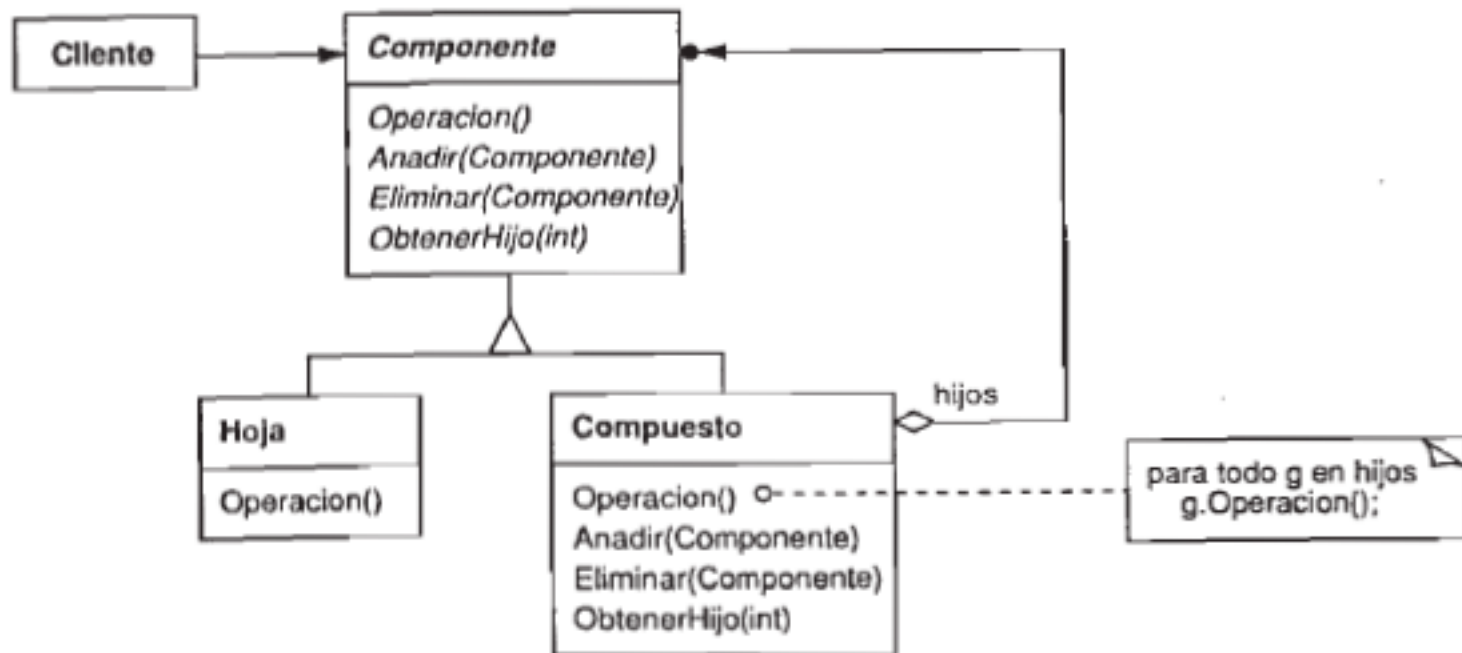
- Componer objetos en estructuras jerárquicas para representar jerarquías parte/todo. **Permite a los clientes manejar a los objetos primitivos y compuestos de forma uniforme.**

- **Motivación**

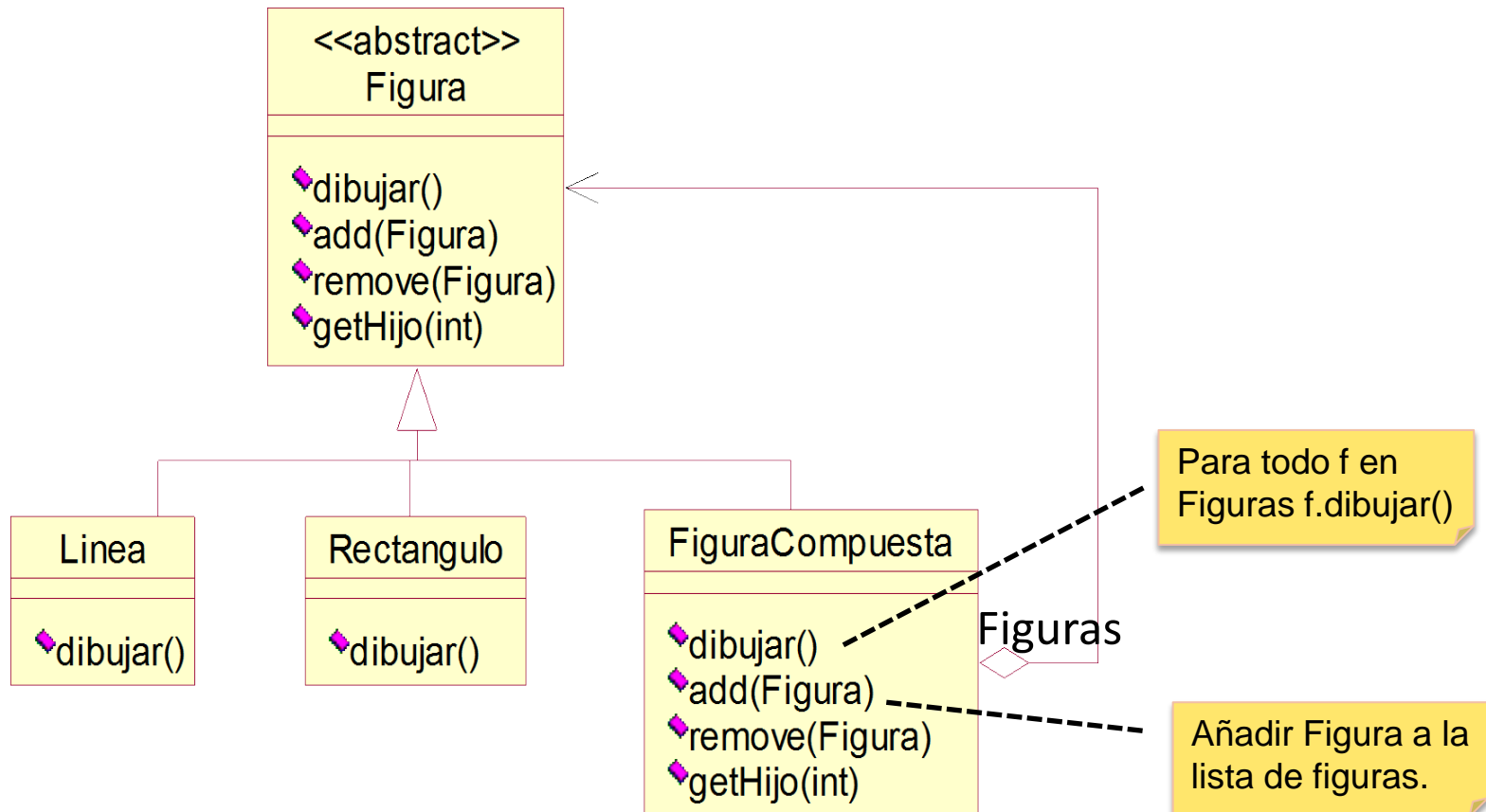
- Modelar figuras compuestas
- Modelar documentos
- Modelar paquetes de valores (acciones, bonos,...)

# Composite

## Estructura



# Composite



# Composite

- **Aplicabilidad**
  - Se quiere representar jerarquías parte/todo
  - Se quiere que los **clientes ignoren la diferencia entre objetos compuestos y los objetos individuales** que los forman.
- **Consecuencias**
  - Jerarquía con clases que modelan objetos primitivos y objetos compuestos, que permite composición recursiva.
  - Clientes pueden tratar objetos primitivos y compuestos de modo uniforme.
  - Es fácil añadir nuevos tipos de componentes.
  - No se puede confiar al sistema de tipos que asegure que un objeto compuesto sólo contendrá objetos de ciertas clases, necesidad de comprobaciones en tiempo de ejecución.

# Decorator

- **Propósito**

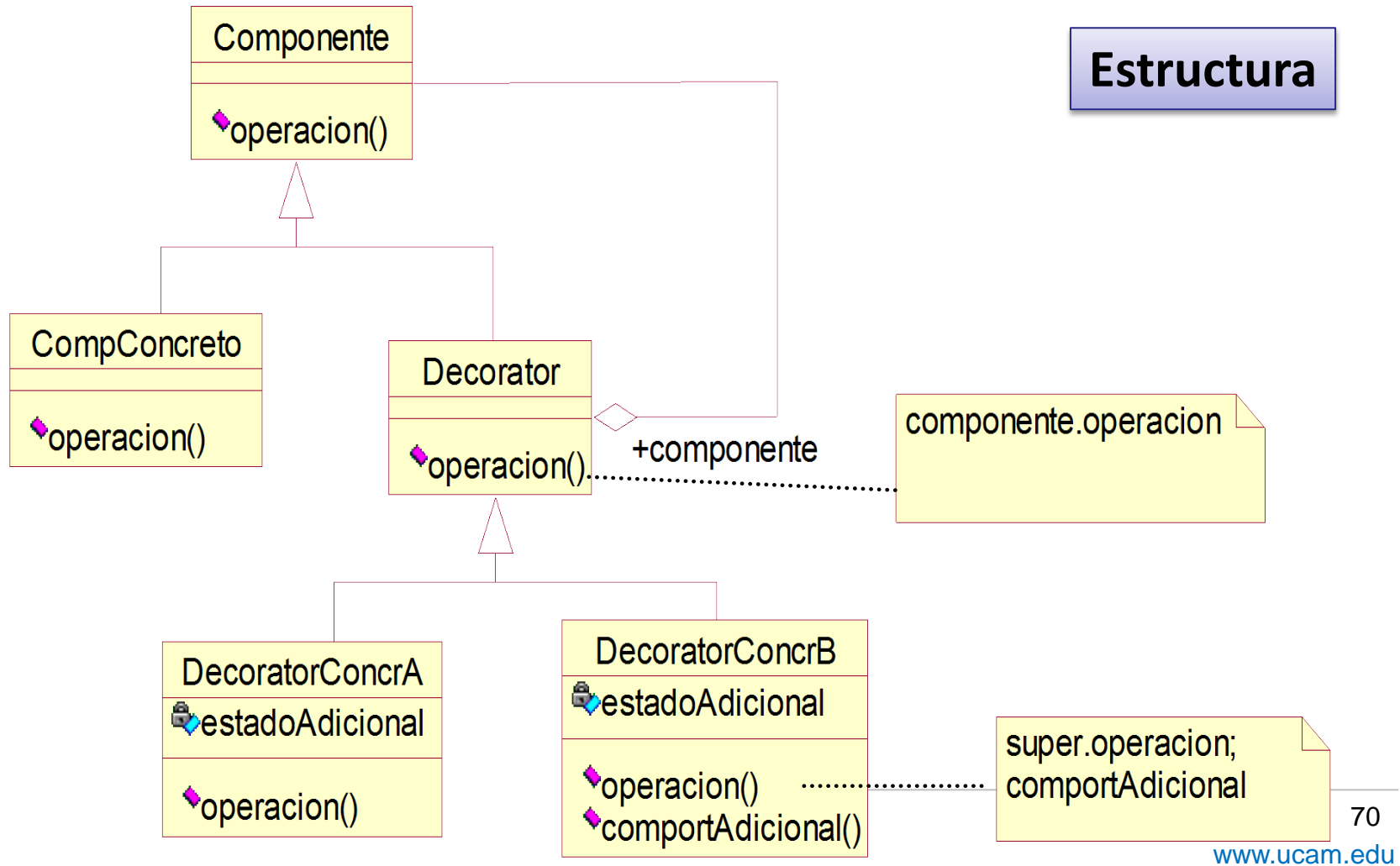
- **Asignar dinámicamente nuevas responsabilidades a un objeto.** Alternativa más flexible a crear subclases para extender la funcionalidad de una clase.

- **Motivación**

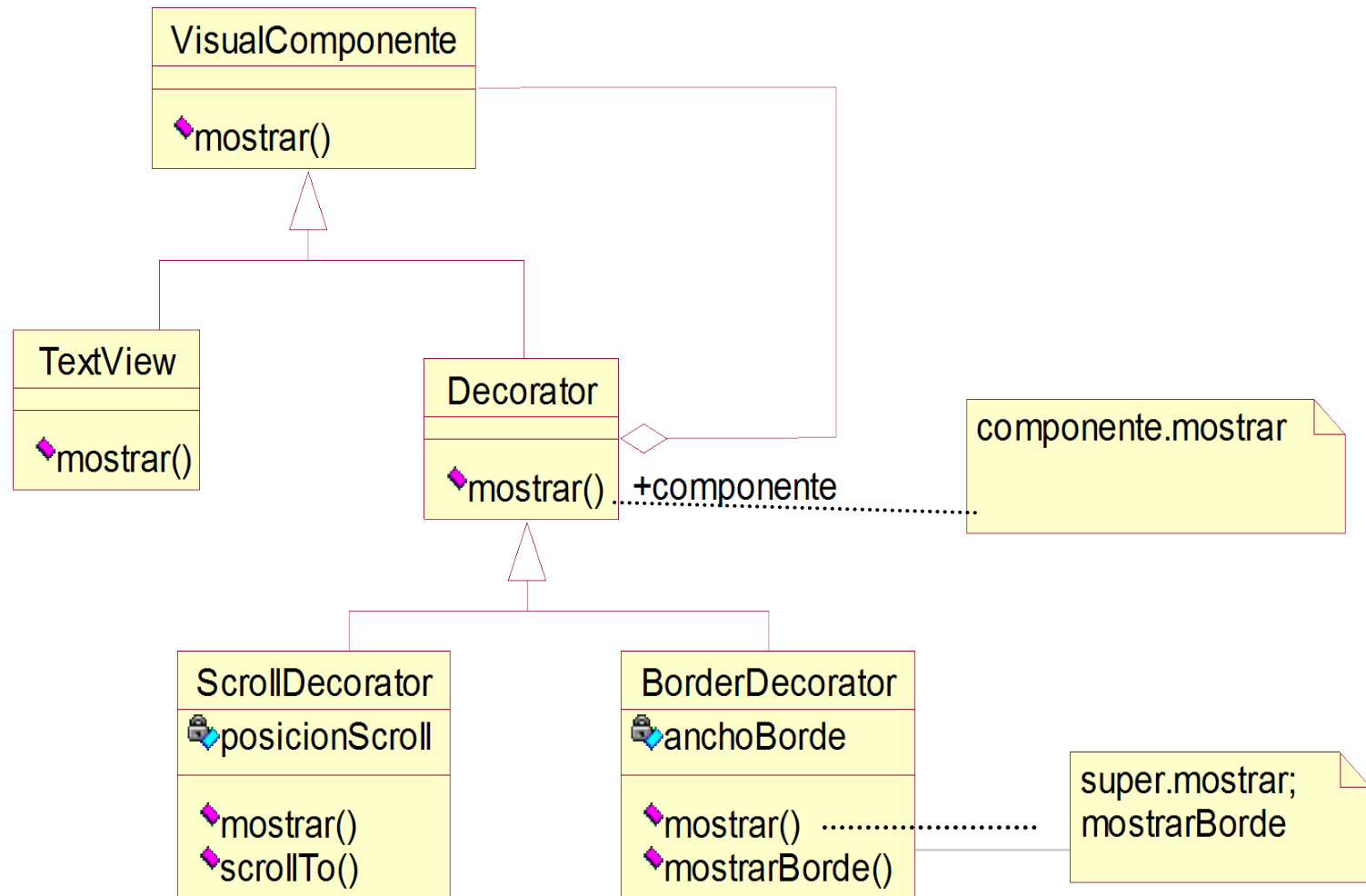
- Algunas veces se desea añadir atributos o comportamiento adicional a un objeto concreto no a una clase.
- Ejemplo: bordes o *scrolling* a una ventana.
- Herencia no lo permite.

# Decorator

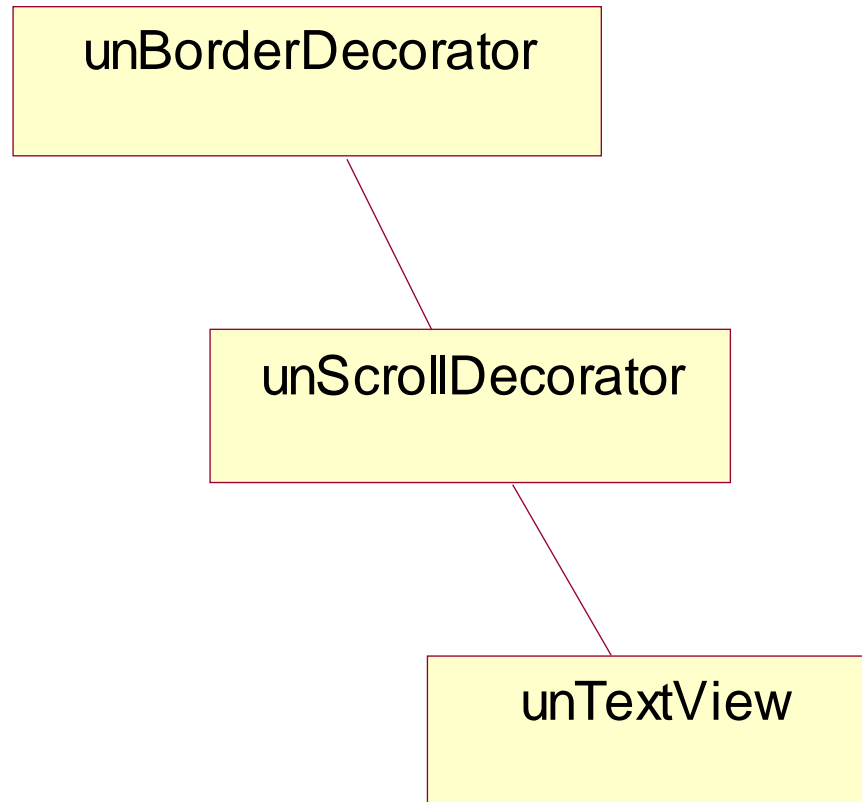
## Estructura



# Decorator



# Decorator





# Decorator

- **Aplicabilidad**

- Añadir dinámicamente responsabilidades a objetos individuales de forma transparente, sin afectar a otros objetos.
- Responsabilidades de un objeto pueden ser eliminadas.
- Para evitar una explosión de clases que produce una jerarquía inmanejable.

# Decorator

- **Consecuencias**

- Más flexible que la herencia: responsabilidades pueden añadirse y eliminarse en tiempo de ejecución.
- Diferentes decoradores pueden ser conectados a un mismo objeto.
- Reduce el número de propiedades en las clases de la parte alta de la jerarquía.
- Es simple añadir nuevos decoradores de forma independiente a las clases que extienden.
- Un objeto decorador tiene diferente OID al del objeto que decora.
- Sistemas con muchos y pequeños objetos.

# Decorator

- **Implementación**

- La interfaz de un objeto *decorador* debe conformar con la interfaz del objeto que decora. Clases *decorador* deben heredar de una clase común.
- *Componentes* y *decoradores* deben heredar de una clase común que debe ser “ligera” en funcionalidad.
- Si la clase *Componente* no es ligera, es mejor usar el patrón *Estrategia* que permite alterar o extender el comportamiento de un objeto.
- Un componente no sabe nada acerca de sus decoradores, con *Estrategia* sucede lo contrario.

# Facade

- **Propósito**

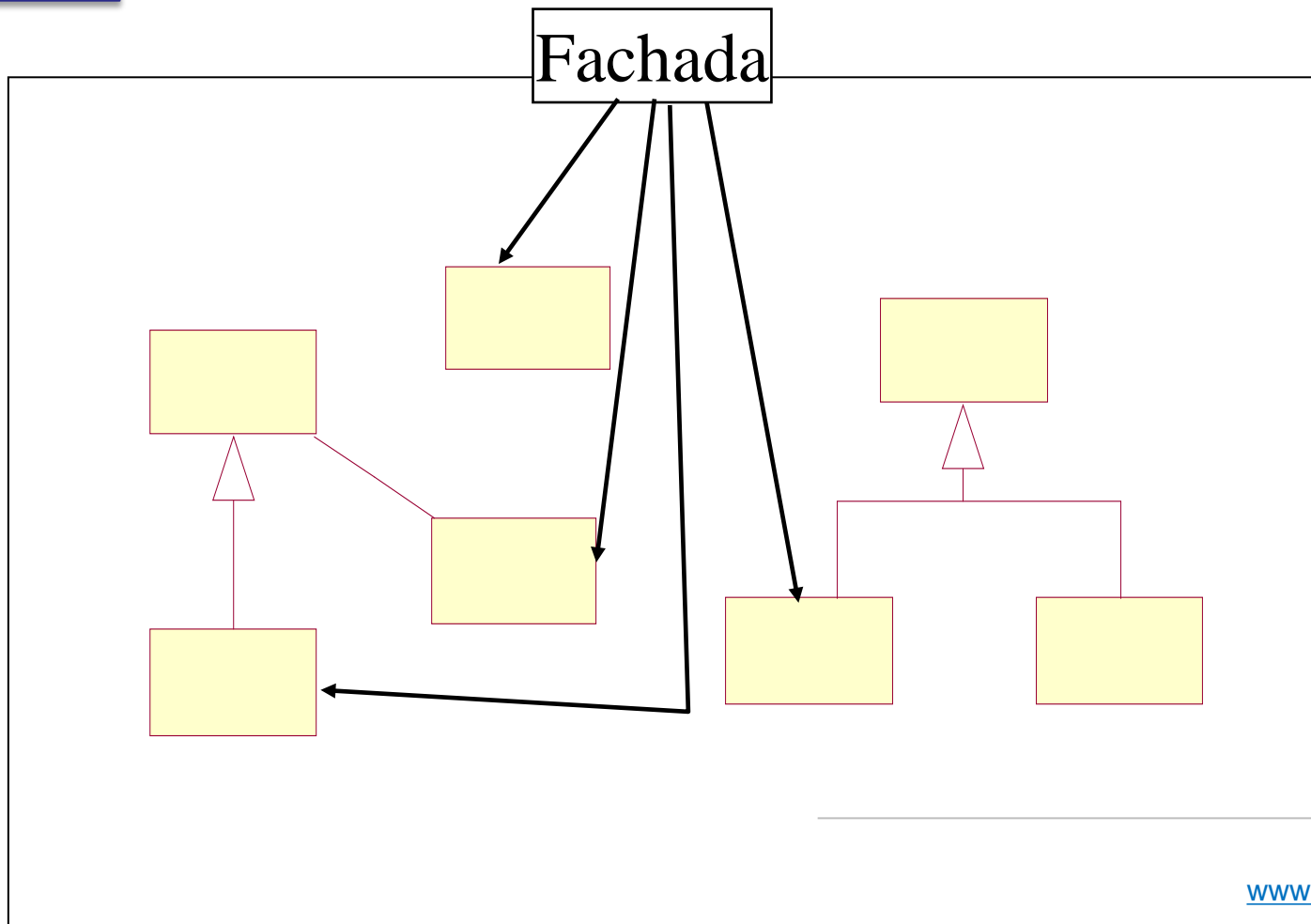
- **Proporciona una única interfaz a un conjunto de clases de un subsistema.** Define una interfaz de más alto nivel que facilita el uso de un subsistema.

- **Motivación**

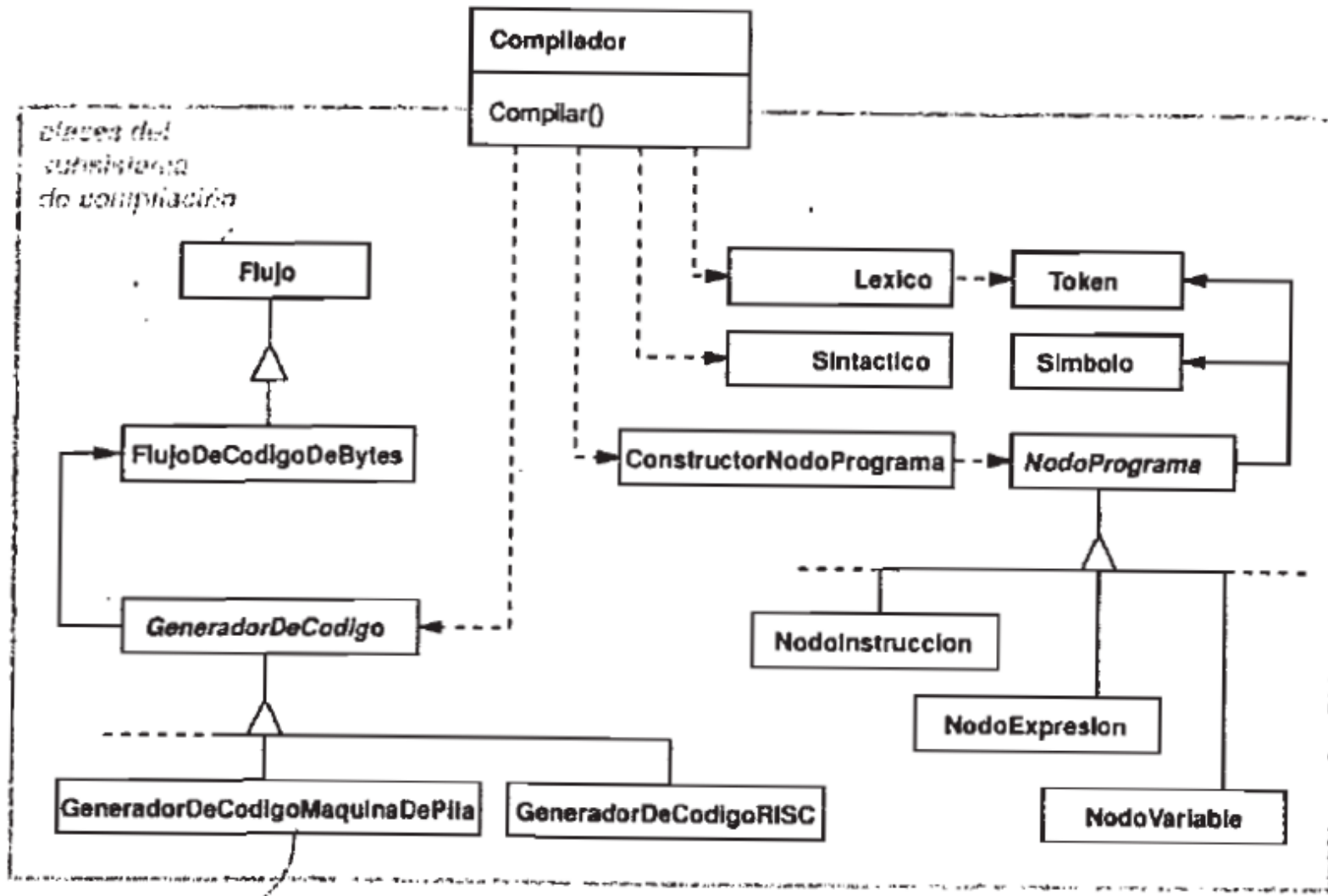
- Reducir las dependencias entre subsistemas.
- Un entorno de programación que ofrece una librería de clases que proporcionan acceso a su subsistema compilador: *Scanner*, *Parser*, *ProgramNode*, *ByteCodeStream* y *ProgramNodeBuilder*. Clase *Compiler* actúa como fachada.

# Facade

## Estructura



# Facade



# Facade

- **Aplicabilidad**
  - Proporcionar una interfaz simple a un subsistema.
  - Hay muchas dependencias entre clientes y las clases que implementan una abstracción.
  - Se desea una arquitectura de varios niveles: una fachada define el punto de entrada para cada nivel-subsistema.

# Facade

- Una fachada ofrece los siguientes **beneficios**
  - Facilita a los clientes el uso de un subsistema, al ocultar sus componentes.
  - Proporciona un acoplamiento débil entre un subsistema y los clientes: cambios en los componentes no afectan a los clientes.
  - No se impide a los clientes el uso de las clases del subsistema si lo necesitan.
- **Implementación**
  - Es posible reducir el acoplamiento entre clientes y subsistema, definiendo la fachada como una clase abstracta con una subclase por cada implementación del subsistema.



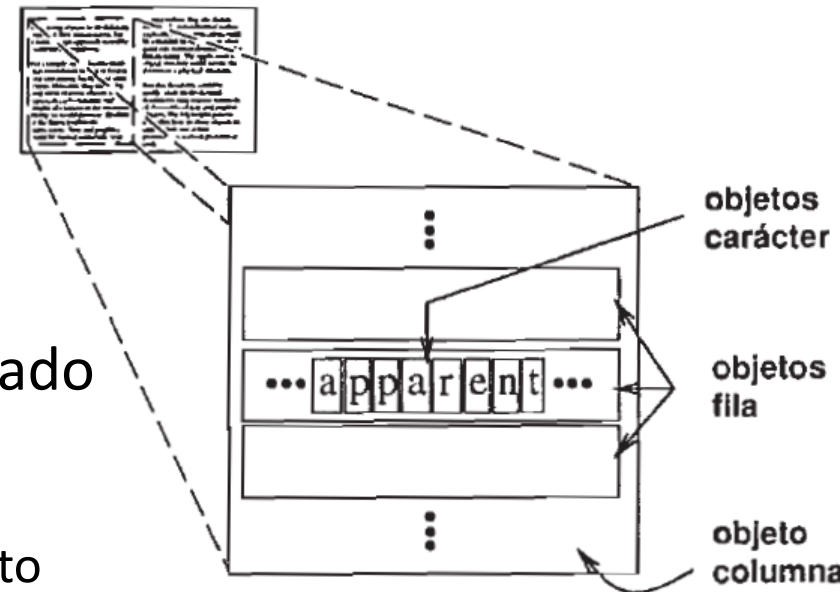
# Flyweight (Peso Ligero)

- **Propósito**

- Uso de **objetos compartidos** para soportar eficientemente un gran número de objetos de poco tamaño.

- **Motivación**

- En una aplicación editor de documentos, ¿modelamos los caracteres mediante una clase?
- Un *flyweight* es un objeto **compartido** que puede ser utilizado en **diferentes contextos simultáneamente**.
  - No hace asunciones sobre el contexto
  - Estado intrínseco vs. Estado extrínseco

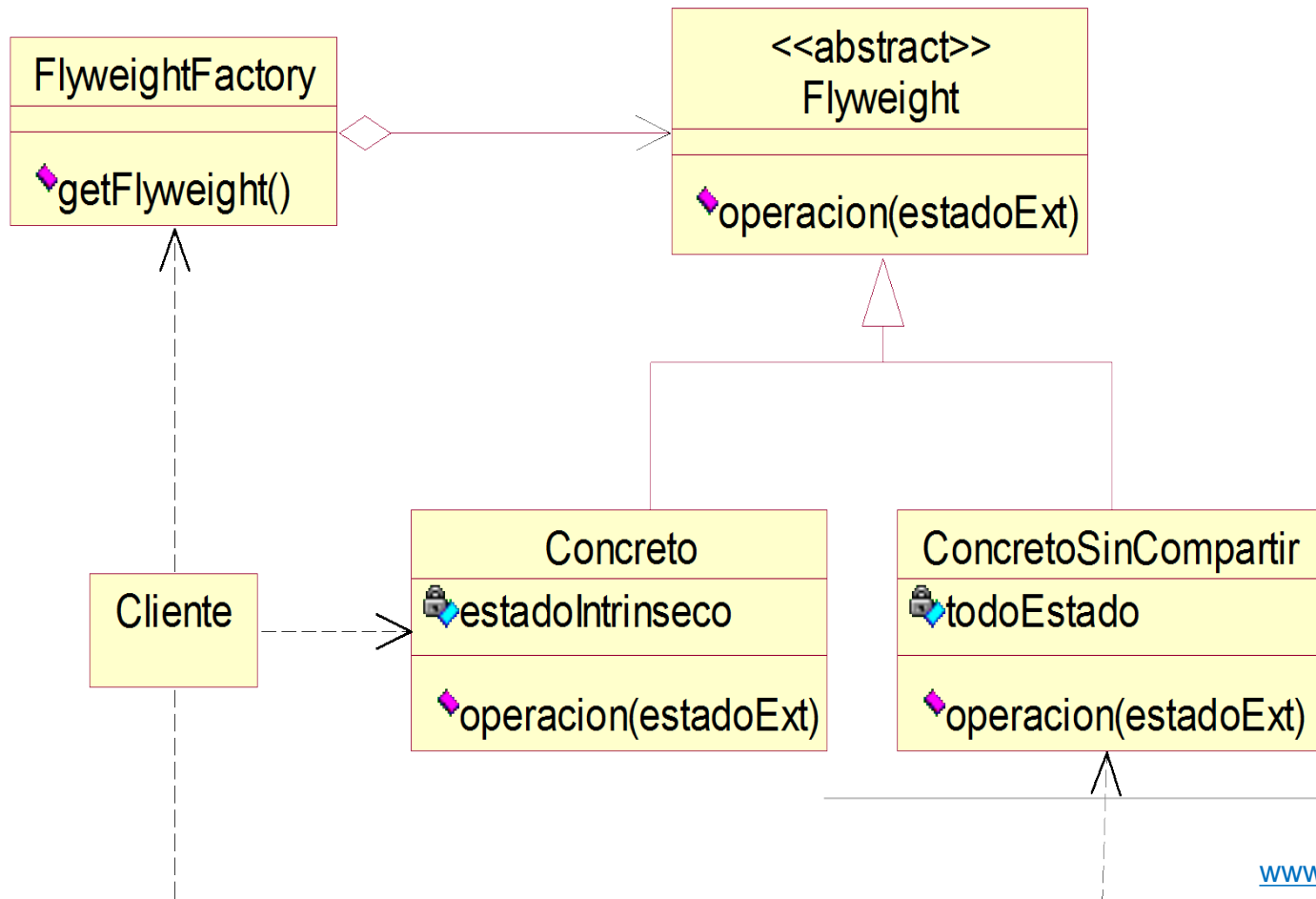


# Flyweight

- **Estado intrínseco** se almacena en el *flyweight* y consiste de información que es independiente del contexto y se puede compartir.
  - Objetos *flyweight* se usan para modelar conceptos o entidades de los que se necesita una gran cantidad en una aplicación, p.e. caracteres de un texto.
- **Estado extrínseco** depende del contexto y por tanto no puede ser compartido.
  - Objetos clientes son responsables de pasar el estado extrínseco al *flyweight* cuando lo necesita.

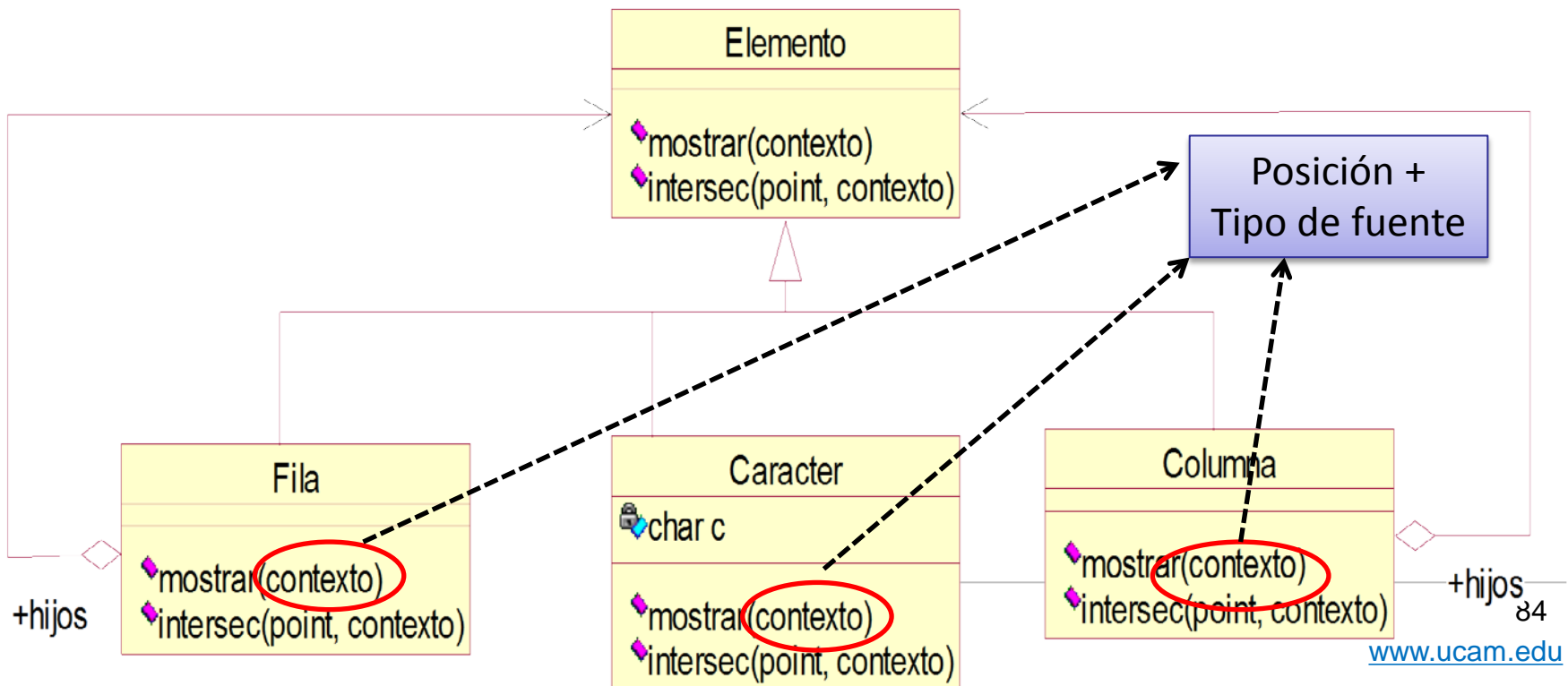
# Flyweight

## Estructura



# Flyweight

- *Flyweight* carácter de un texto:
  - estado intrínseco: código del carácter
  - estado extrínseco: información sobre posición y estilo



# Flyweight

- **Aplicabilidad**

- Aplicarlo siempre que se cumplan las siguientes condiciones:
  - Una aplicación utiliza un gran número de cierto tipo de objetos.
  - El coste de almacenamiento es alto debido al excesivo número de objetos.
  - La mayor parte del estado de esos objetos puede hacerse extrínseco.
  - Al separar el estado extrínseco, muchos grupos de objetos pueden reemplazarse por unos pocos objetos compartidos.
  - La aplicación no depende de la identidad de los objetos.

# Flyweight

- **Implementación**

- La aplicabilidad depende de la facilidad de obtener el estado extrínseco. Idealmente debe poder calcularse a partir de una estructura de objetos que necesite poco espacio de memoria.
- Debido a que los *flyweight* son compartidos, no deberían ser instanciados directamente por los clientes: clase *FlyweightFactory*.

# Proxy

- **Propósito**

- Proporcionar un sustituto (*surrogate*) de un objeto para controlar el acceso a dicho objeto.

- **Motivación**

- Diferir el coste de crear un objeto hasta que sea necesario usarlo: **creación bajo demanda**.
- Un editor de documentos que incluyen objetos gráficos.
- ¿Cómo ocultamos que una imagen se creará cuando se necesite?: manejar el documento requiere conocer información sobre la imagen.

# Proxy

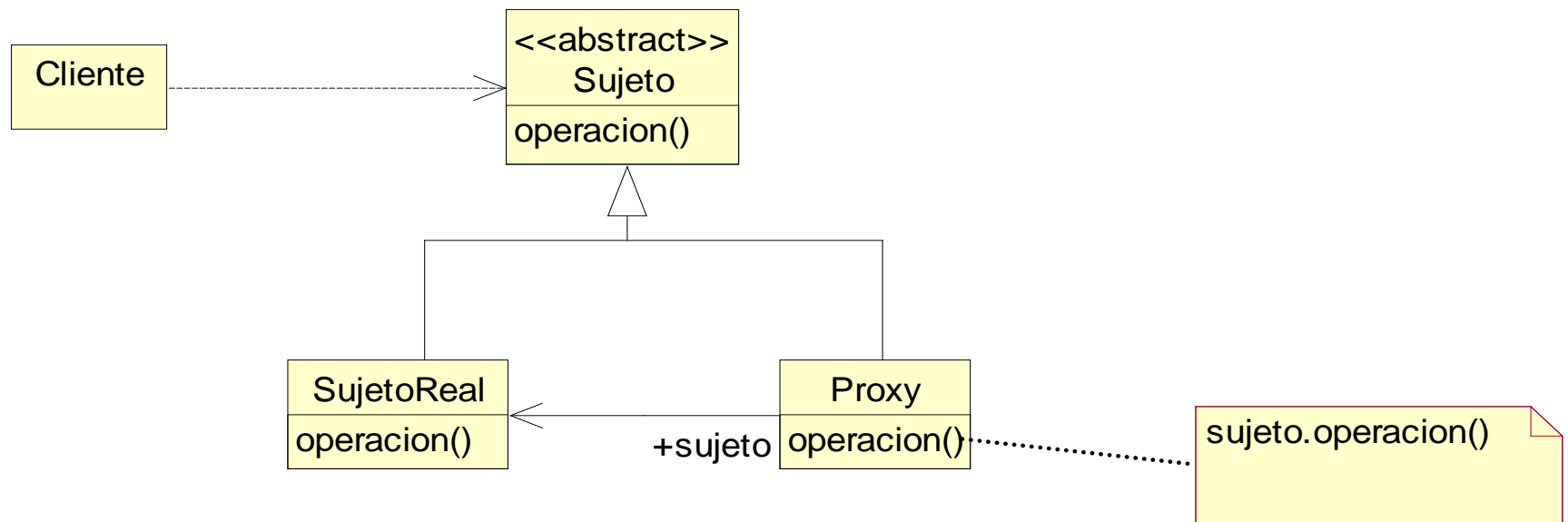
- **Motivación**

- Hay situaciones en las que un cliente no referencia o no puede referenciar a un objeto directamente, pero necesita interactuar con él.
- Un objeto *proxy* puede actuar como intermediario entre el objeto cliente y el objeto destino.
- El objeto *proxy* **tiene la misma interfaz** como el objeto destino.
- El objeto *proxy* **mantiene una referencia** al objeto destino y puede pasarle a él los mensajes recibidos (delegación).

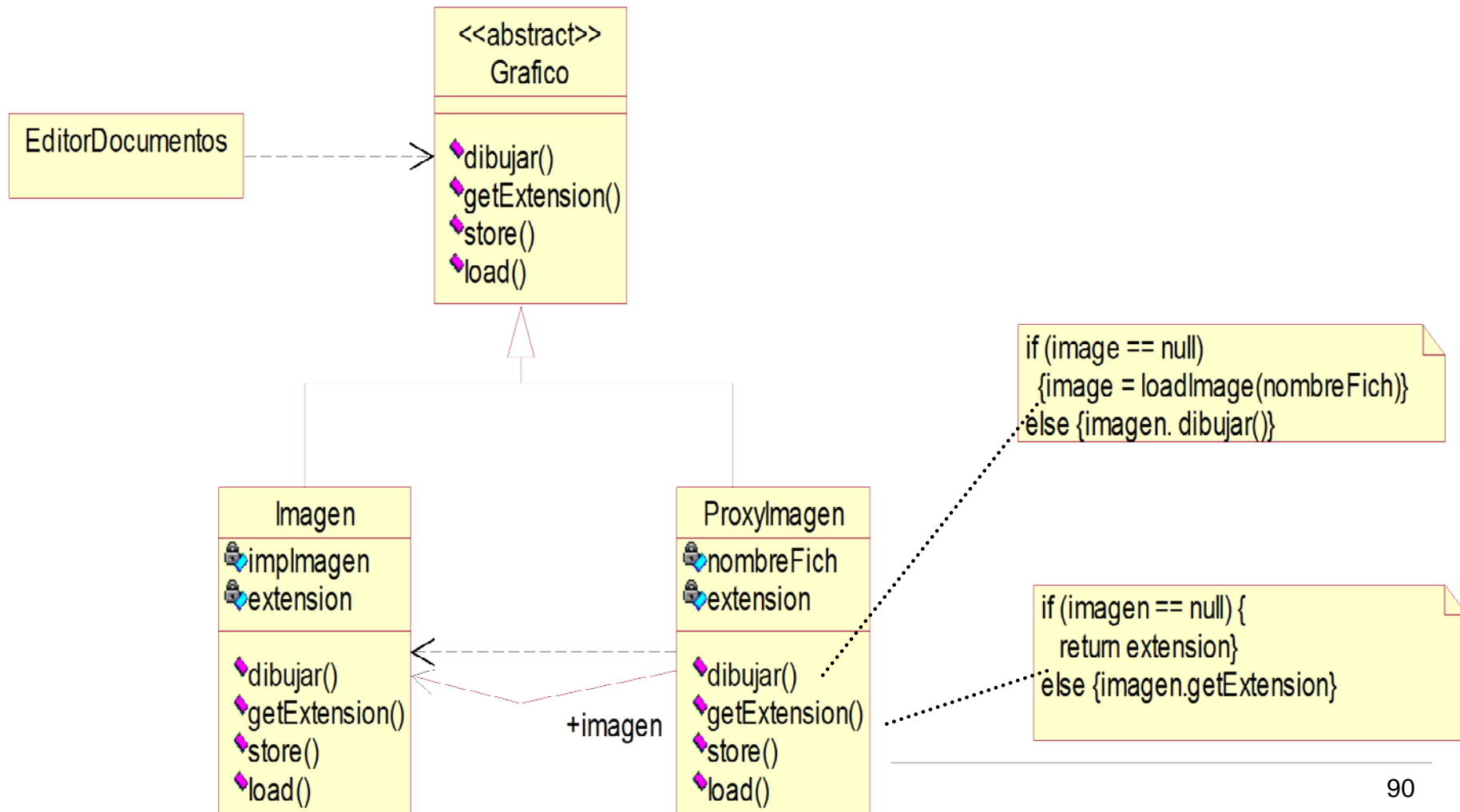


# Proxy

## Estructura



# Proxy




# Proxy

- **Consecuencias**

- Introduce un nivel de indirección para:
  1. Un *proxy* remoto oculta el hecho que objetos residen en diferentes espacios de direcciones.
  2. Un *proxy* virtual tales como crear o copiar un objeto bajo demanda.
  3. Un *proxy* para protección o las referencias inteligentes permiten realizar tareas de control sobre los objetos accedidos.

# Contenidos

- ¿Qué son?
- Introducción
- Tipos de patrones
- Patrones de creación
- Patrones estructurales
- **Patrones de comportamiento** 

# Patrones de comportamiento

- Relacionados con la asignación de responsabilidades entre clases.
- Enfatizan la colaboración entre objetos.
- Caracterizan un flujo de control más o menos complejo que será transparente al que utilice el patrón.
- Basados en **clases** usan la herencia: *Template Method e Interpreter*
- Basados en **objetos** usan la composición: *Mediator, Observer,..*

# Patrones de comportamiento

- Cadena de Responsabilidad
- Command
- Iterator
- **Intérprete (NO)**
- Memento
- Mediador
- Observer
- Estado
- Estrategia
- Método Plantilla
- Visitor

# Chain of Responsibility

## (Cadena de Responsabilidad)

- **Propósito**

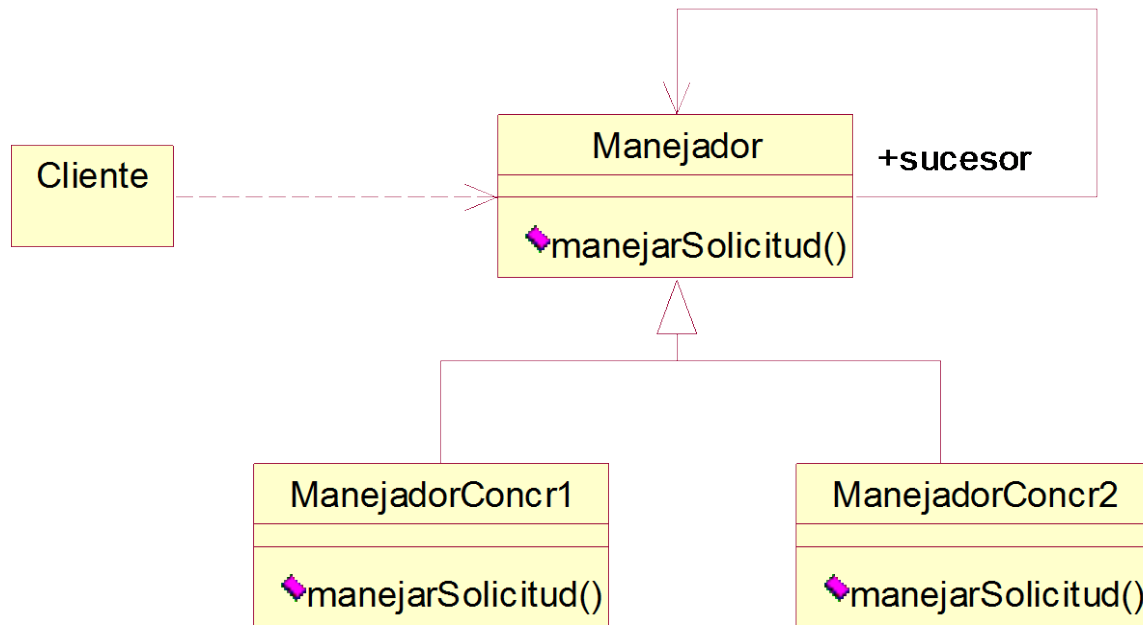
- Evita acoplar el emisor de un mensaje a su receptor dándole a más de un objeto la posibilidad de manejar la solicitud. Se define una cadena de objetos, de modo que un objeto pasa la solicitud al siguiente en la cadena hasta que uno la maneja.

- **Motivación**

- Facilidad de ayuda sensible al contexto.
- El objeto que proporciona la ayuda no es conocido al objeto (p.e. un Button) que inicia la solicitud de ayuda.

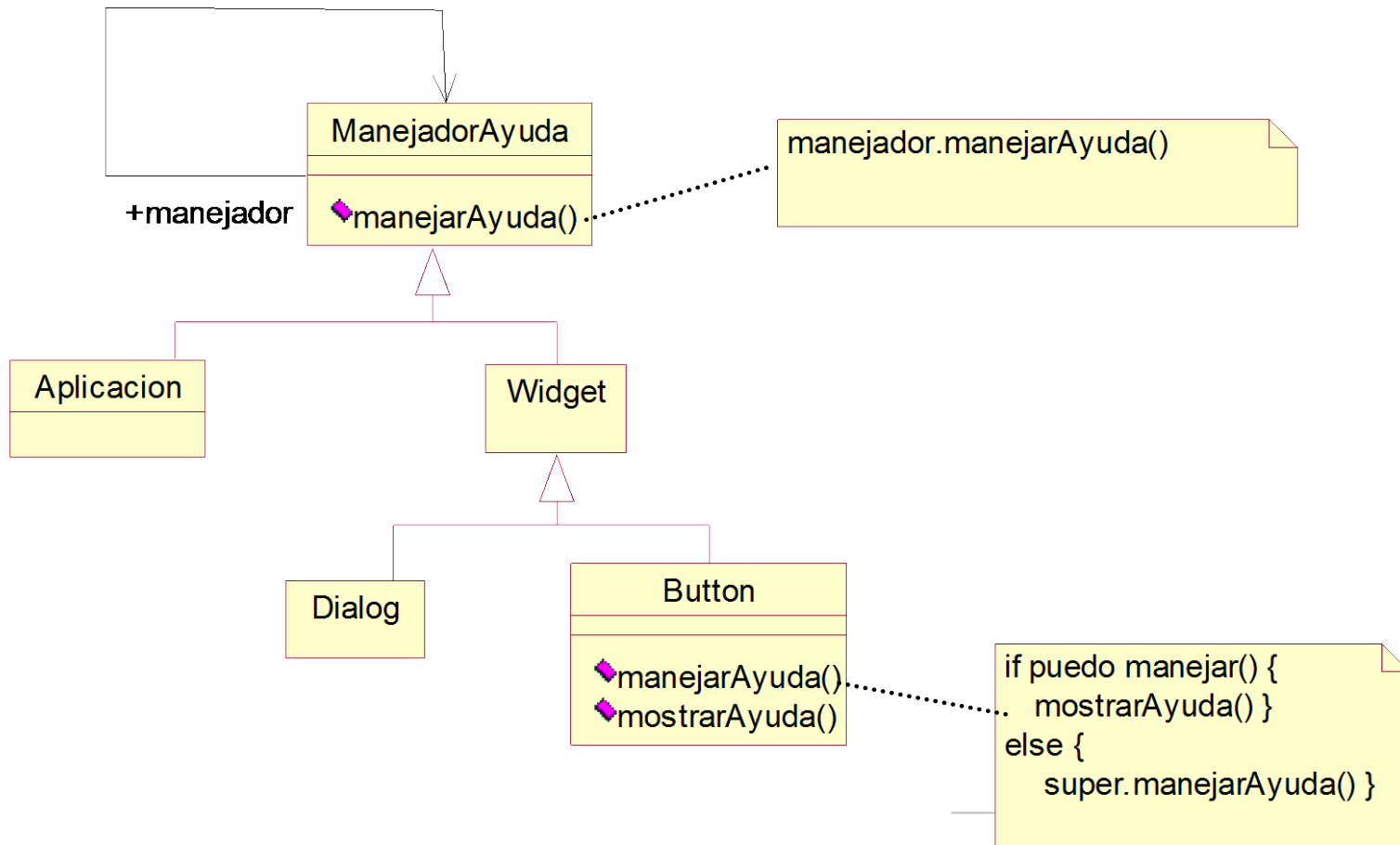
# Cadena de Responsabilidad

## Estructura

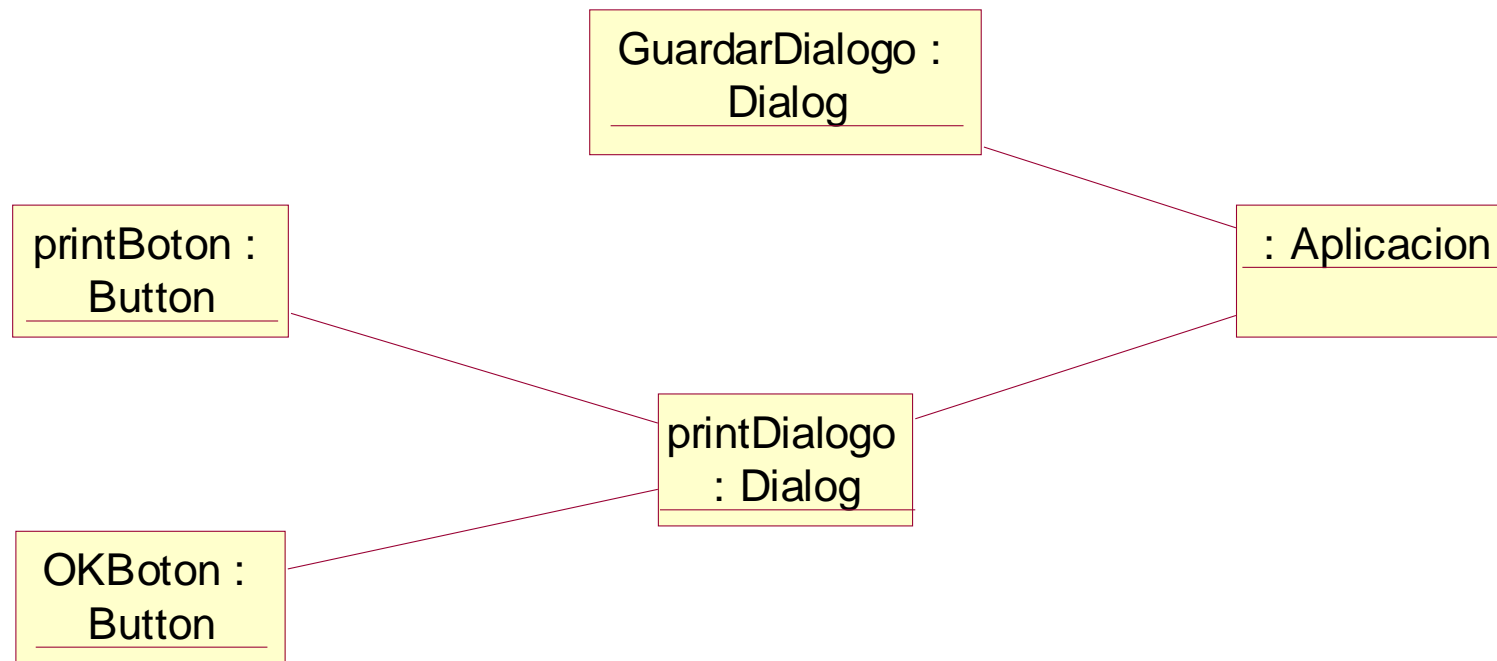




# Cadena de Responsabilidad



# Cadena de Responsabilidad



# Cadena de Responsabilidad

- **Aplicabilidad**

- Más de un objeto puede manejar una solicitud, y el manejador no se conoce a priori.
- Se desea enviar una solicitud a uno entre varios objetos sin especificar explícitamente el receptor.
- El conjunto de objetos que puede manejar una solicitud puede ser especificado dinámicamente.

# Command (Orden)

- **Propósito**

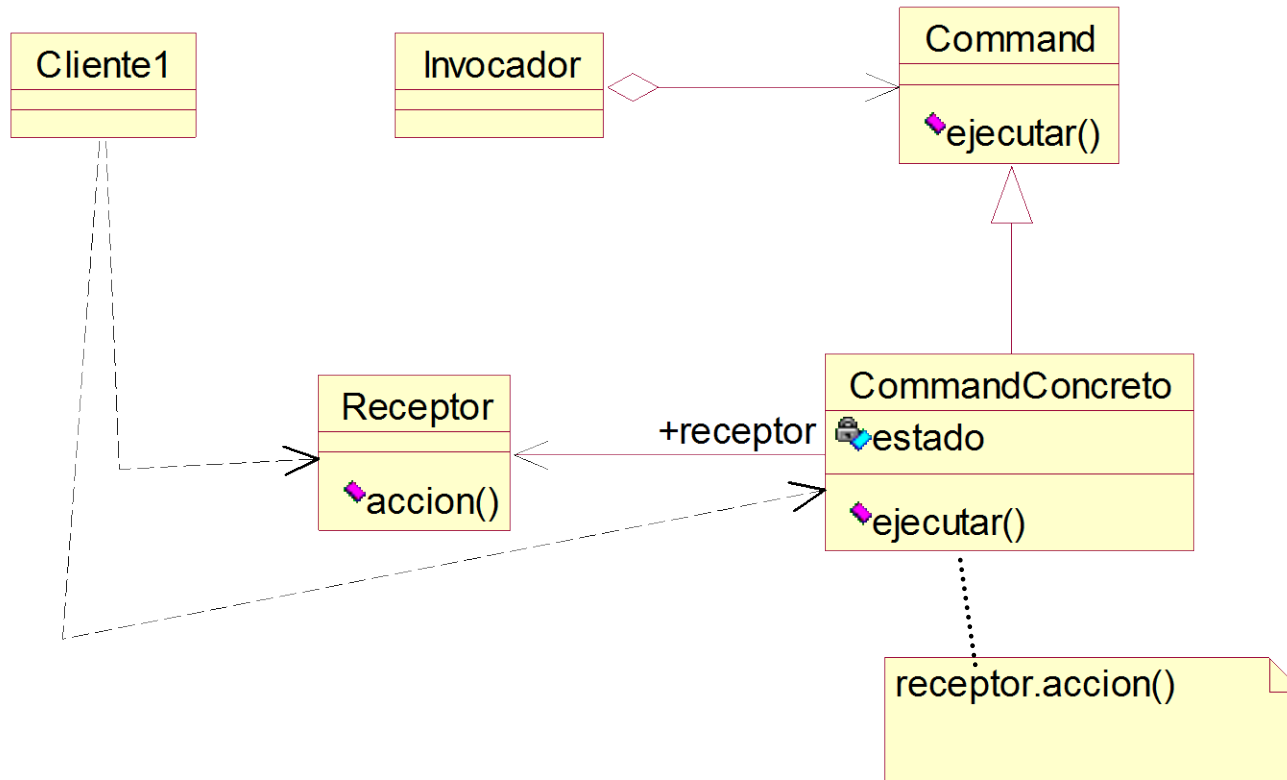
- Encapsula un mensaje como un objeto, permitiendo parametrizar los clientes con diferentes solicitudes, añadir a una cola las solicitudes y soportar funcionalidad deshacer/rehacer (undo/redo)

- **Motivación**

- Algunas veces es necesario enviar un mensaje a un objeto sin conocer el selector del mensaje ni el objeto receptor.
- Por ejemplo widgets (botones, menús,..) realizan una acción como respuesta a la interacción del usuario, pero no se puede explicitar en su implementación.

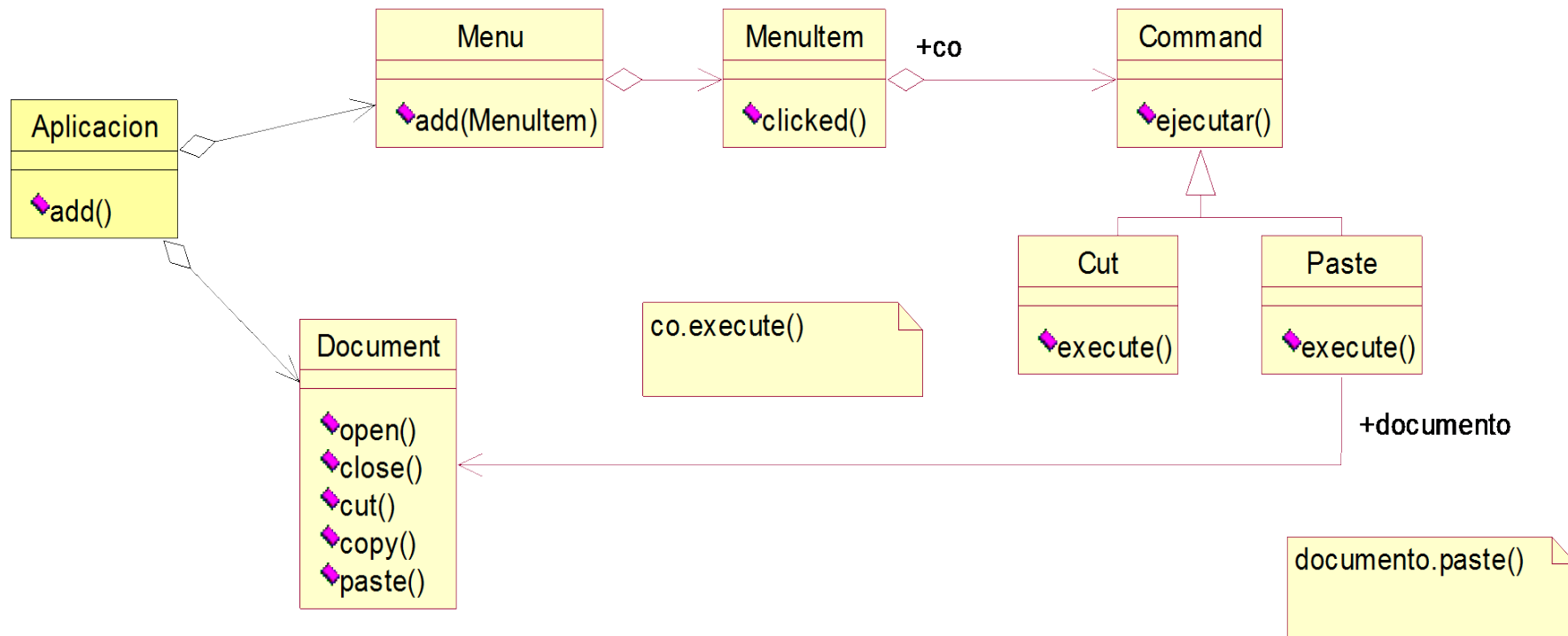
# Command

## Estructura



# Command

## Motivación



# Command

- **Aplicabilidad**

- Parametrizar objetos por la acción a realizar (alternativa a funciones **Callback**: función que es registrada en el sistema para ser llamada más tarde; en C++ se puede usar punteros a funciones)
- Especificar, añadir a una cola y ejecutar mensajes en diferentes instantes: un objeto Command tiene un tiempo de vida independiente de la solicitud original.
- Soportar facilidad undo/redo.
- Recuperación de fallos.

# Command

- **Consecuencias**

- Desacopla el objeto que invoca la operación del objeto que sabe cómo realizarla.
- Cada subclase **CommandConcreto** especifica un par **receptor/acción**, almacenando el receptor como un atributo e implementando el método ejecutar.
- Objetos command pueden ser manipulados como cualquier otro objeto.
- Se pueden crear **command compuestos** (aplicando el patrón Composite).
- Es fácil añadir nuevos commands.



# Interpreter

- **Propósito**

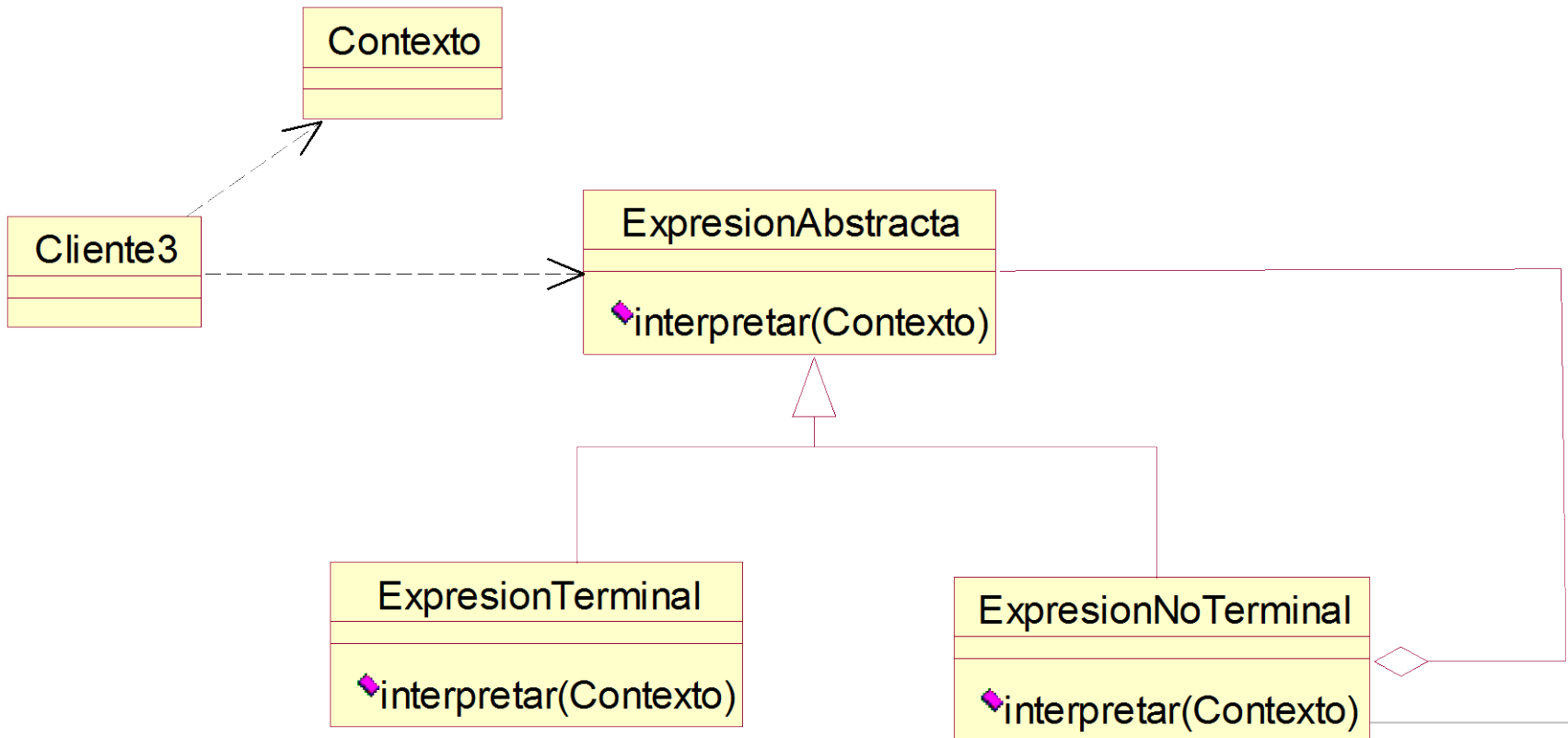
- Dado un lenguaje, definir una representación para su gramática junto con un intérprete que utiliza la representación para interpretar sentencias en dicho lenguaje.

- **Motivación**

- Interpretar una expresión regular.
- Usa una clase para representar cada regla, los símbolos en la parte derecha son atributos.
- Usar si la gramática es simple y la eficiencia no es importante.

# Intérprete

## Estructura



`expre := literal | cond | seq | rep`

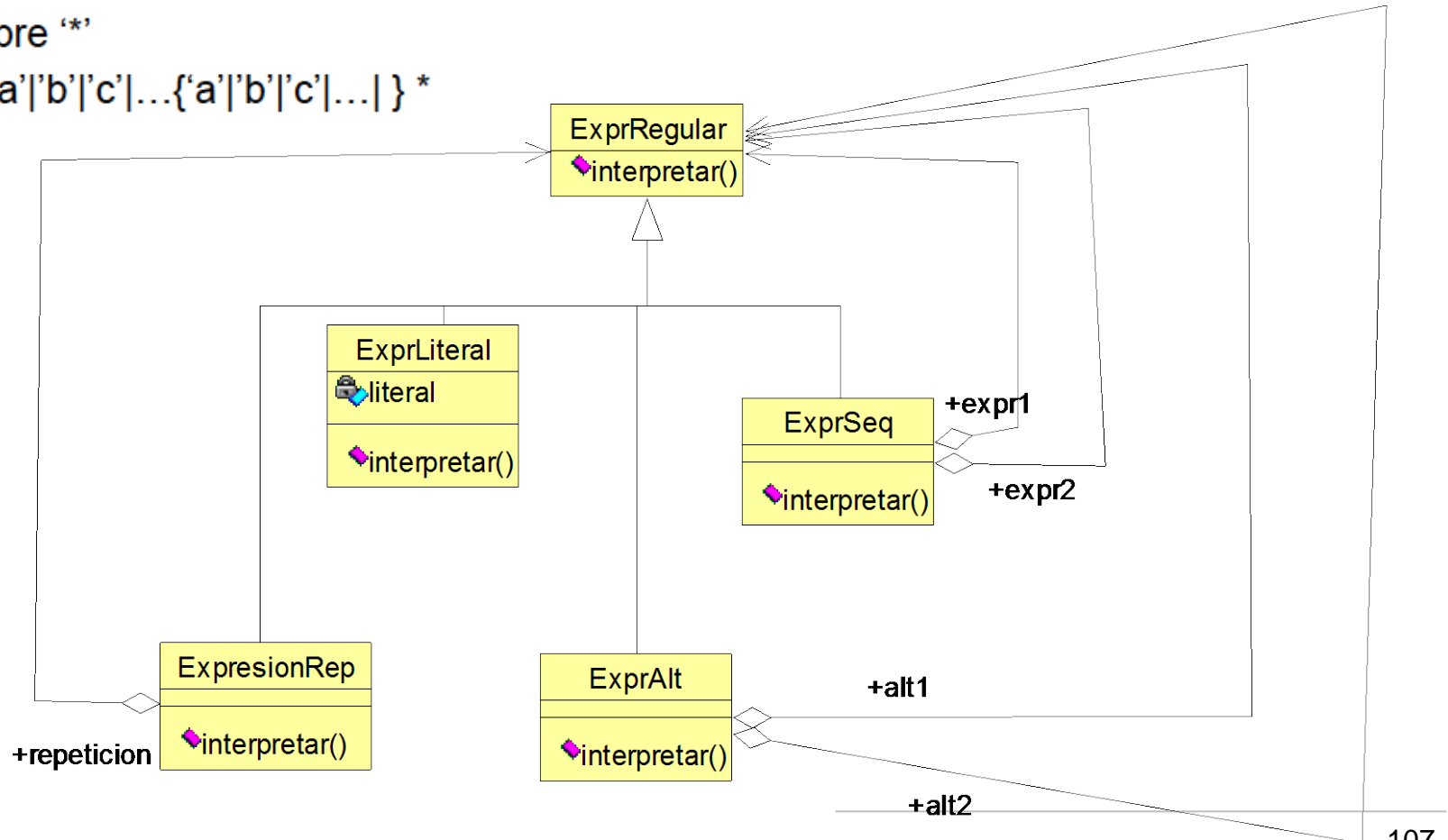
`alt := expre '[' expre`

`seq := expre '&' expre`

`rep := expre '*'`

`literal := 'a'|'b'|'c'|...{'a'|'b'|'c'|...|} *`

# Intérprete



# Iterator (Iterador)

- **Propósito**

- Proporciona una forma para acceder a los elementos de una estructura de datos sin exponer los detalles de la representación.

- **Motivación**

- Un objeto contenedor (agregado o colección) tal como una lista debe permitir una forma de recorrer sus elementos sin exponer su estructura interna.
- Debería permitir diferentes métodos de recorrido
- Debería permitir recorridos concurrentes
- No queremos añadir esa funcionalidad a la interfaz de la colección

# Iterador

- **Motivación**

- Una clase Iterator define la interfaz para acceder a una estructura de datos (p.ej. una lista).
- Iteradores externos vs. Iteradores internos.
  - *Iteradores externos*: recorrido controlado por el cliente
  - *Iteradores internos*: recorrido controlado por el iterador

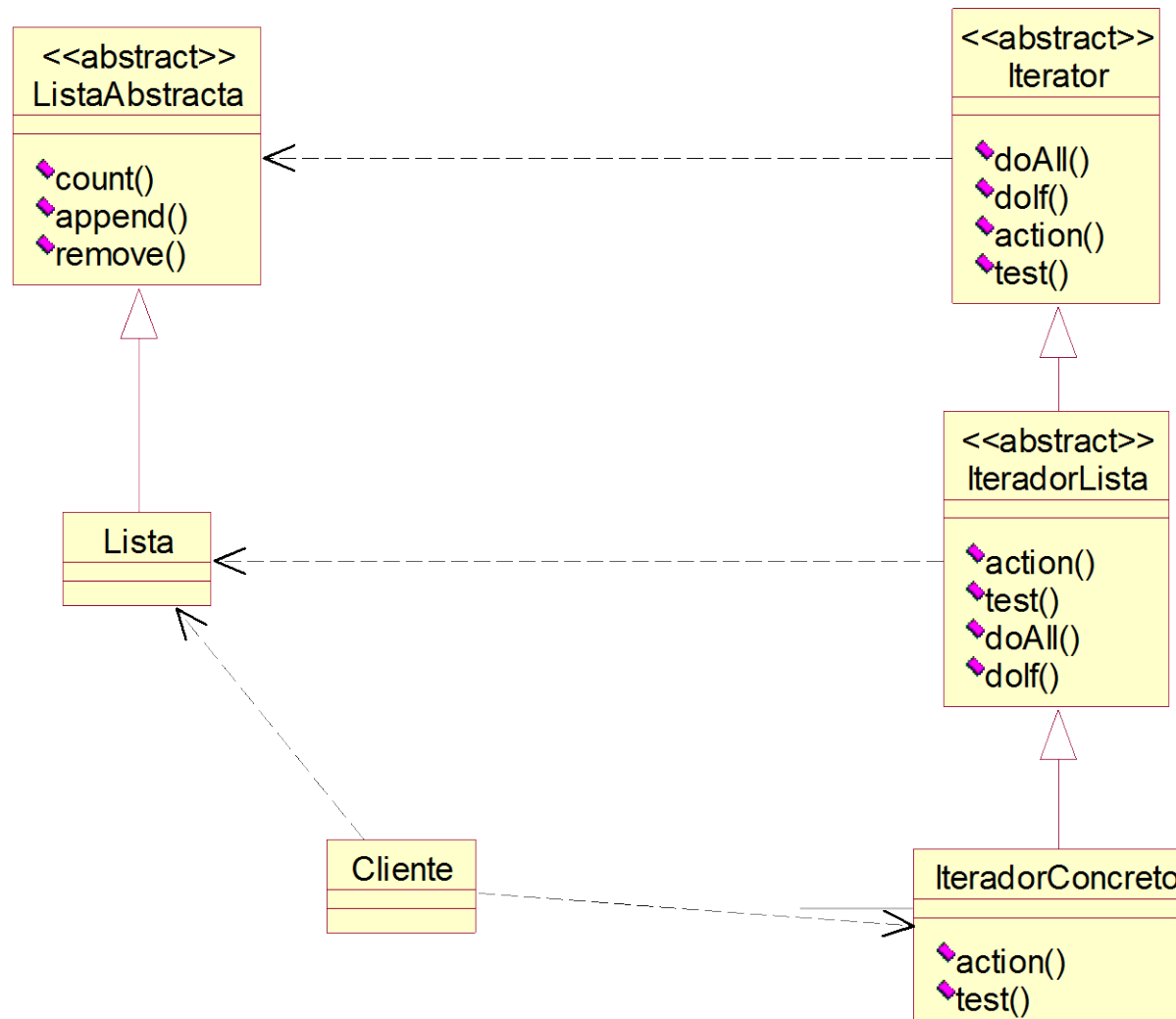
# Iterador externo



# Iterador externo

```
List lista = new List();  
...  
ListIterator iterator = new ListIterator(lista);  
  
iterator.first();  
while (!iterator.isDone()) {  
    Object item = iterator.item();  
    // código para procesar item  
    iterator.next();  
}  
...
```

# Iterador interno





# Iterador interno

```
public void doIf() {  
    Iterator it = col.iterator();  
    while (it.hasNext()) {  
        Object o = iterator.next();  
        if (test(o)) action(o)  
    }  
}
```

# Iterador

- **Consecuencias**

- Simplifica la interfaz de un contenedor al extraer los métodos de recorrido
- Permite varios recorridos concurrentes
- Soporta variantes en las técnicas de recorrido

# Mediator (Mediador)

- **Propósito**

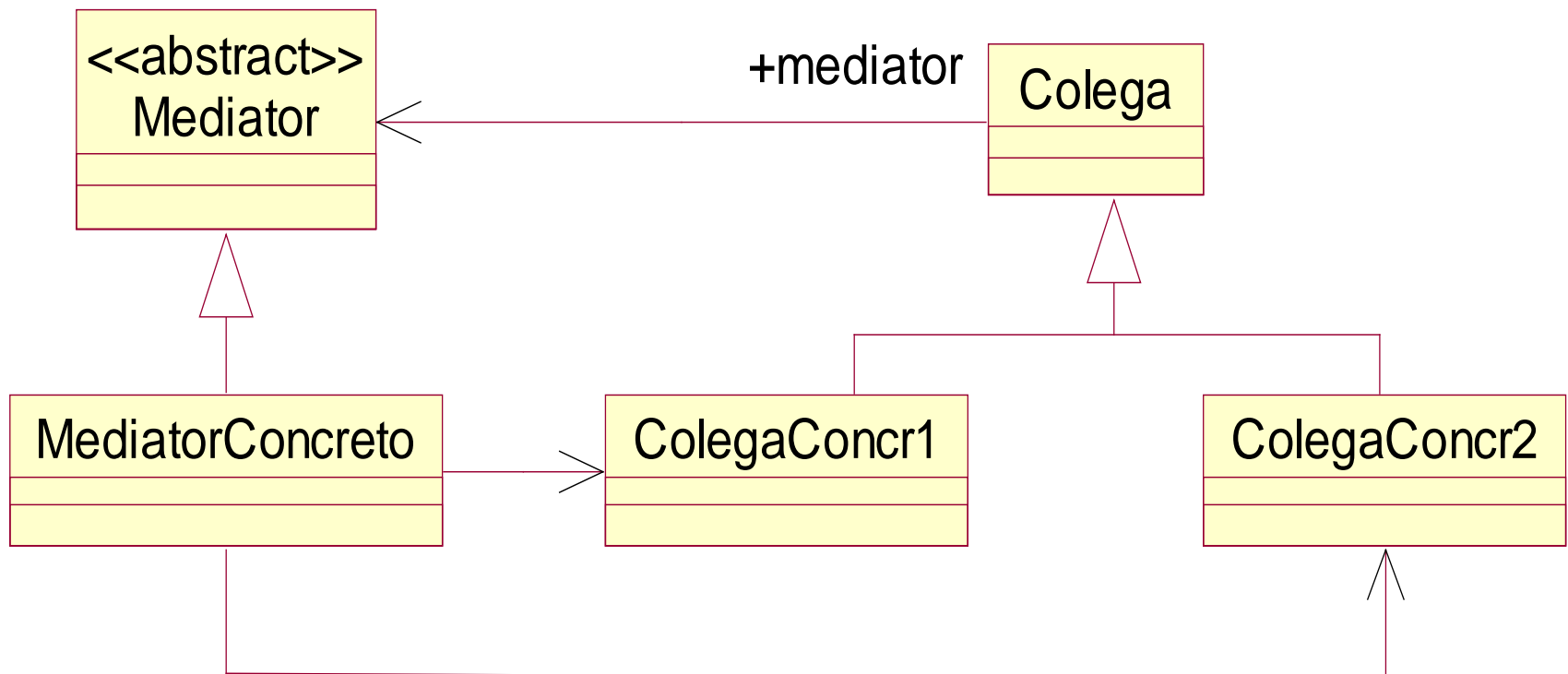
- Define un **objeto que encapsula cómo interaccionan un conjunto de objetos**. Favorece un bajo acoplamiento, liberando a los objetos de referenciarse unos a otros explícitamente, y permite variar la interacción de manera independiente.

- **Motivación**

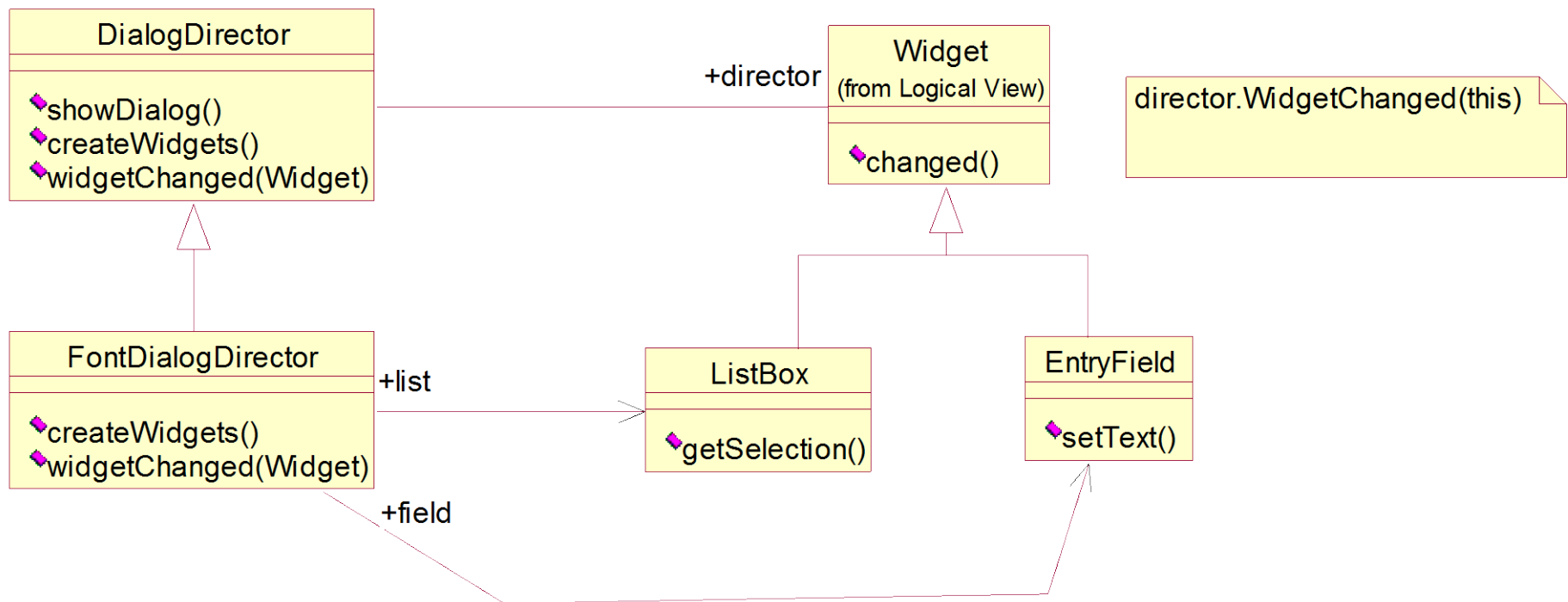
- Muchas interconexiones entre objetos dificulta la reutilización y la especialización del comportamiento.
- Ventana que incluye un conjunto de elementos gráficos con dependencias entre ellos.
  - Botón se desactiva cuando está vacío un campo de entrada concreto
  - Seleccionar una opción de una lista desplegable rellena ciertos campos de texto de un formulario.
  - Cuando haya cierto texto en un campo de texto se activan otros botones

# Mediador

## Estructura



# Mediador



# Mediador

- **Aplicabilidad**

- Un conjunto de objetos se comunica entre sí de una forma bien definida, pero compleja. Las interdependencias son poco estructuradas y difíciles de comprender.
- Reutilizar una clase es difícil porque tiene dependencias con muchas otras clases.
- Un comportamiento que es distribuido entre varias clases debería ser adaptable sin crear muchas subclases.

# Mediador

- **Consecuencias**

- Evita crear subclases de los colegas, sólo se crean subclases del *mediador*.
- Desacopla a los *colegas*.
- Simplifica los protocolos entre las clases
- Abstrae el cómo cooperan los objetos
- Centraliza el control en el mediador: clase difícil de mantener

# Memento

- **Propósito**

- Captura y **externaliza el estado interno de un objeto**, sin violar la encapsulación, de modo que el objeto puede ser restaurado a este estado más tarde.

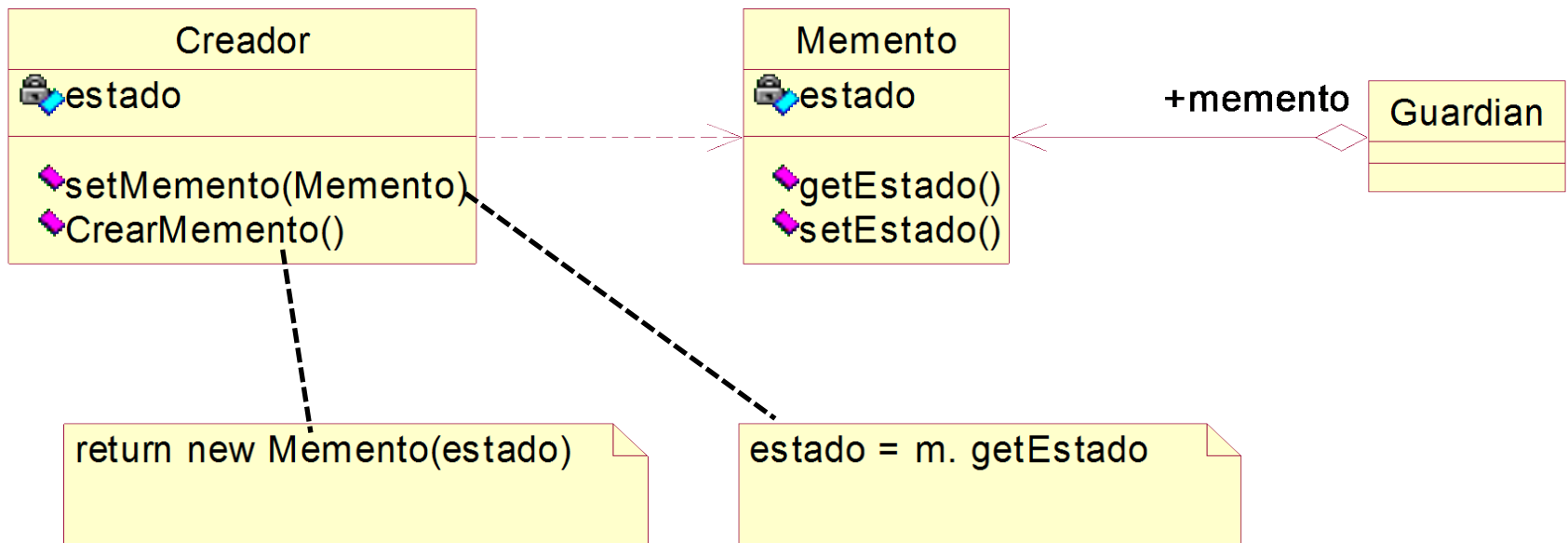
- **Motivación**

- Algunas veces es necesario registrar el estado interno de un objeto: mecanismos *checkpoints* y deshacer cambios que permiten probar operaciones o recuperación de errores.



# Memento

## Estructura



# Memento

- **Aplicabilidad**

- Una parte del estado de un objeto debe ser guardado para que pueda ser restaurado más tarde y una interfaz para obtener el estado de un objeto podría romper la encapsulación exponiendo detalles de implementación.

# Memento

- **Consecuencias**

- Mantiene la encapsulación
- Simplifica la clase *Creador* ya que no debe preocuparse de mantener las versiones del estado interno.
- Podría incurrir en un considerable gasto de memoria: encapsular y restaurar el estado no debe ser costoso.
- Puede ser difícil en algunos lenguajes **asegurar que** sólo el *Creador* tenga acceso al estado del *Memento*.

# Observer

- **Propósito**

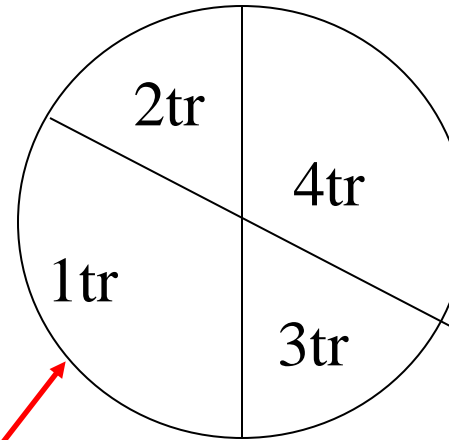
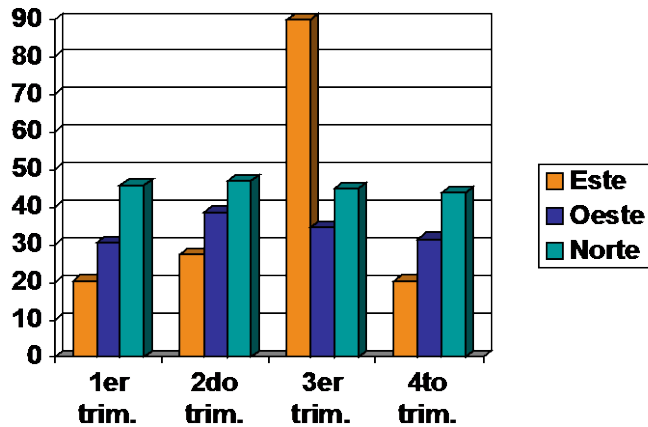
- Define una dependencia uno-a-muchos entre objetos, de modo que cuando cambia el estado de un objeto, todos sus dependientes automáticamente son notificados y actualizados.

- **Motivación**

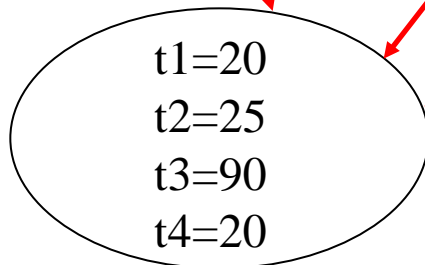
- ¿Cómo mantener la consistencia entre objetos relacionados, sin establecer un acoplamiento fuerte entre sus clases?
- Ejemplo: Separación *Modelo-Vista*

# Motivación

## Vistas



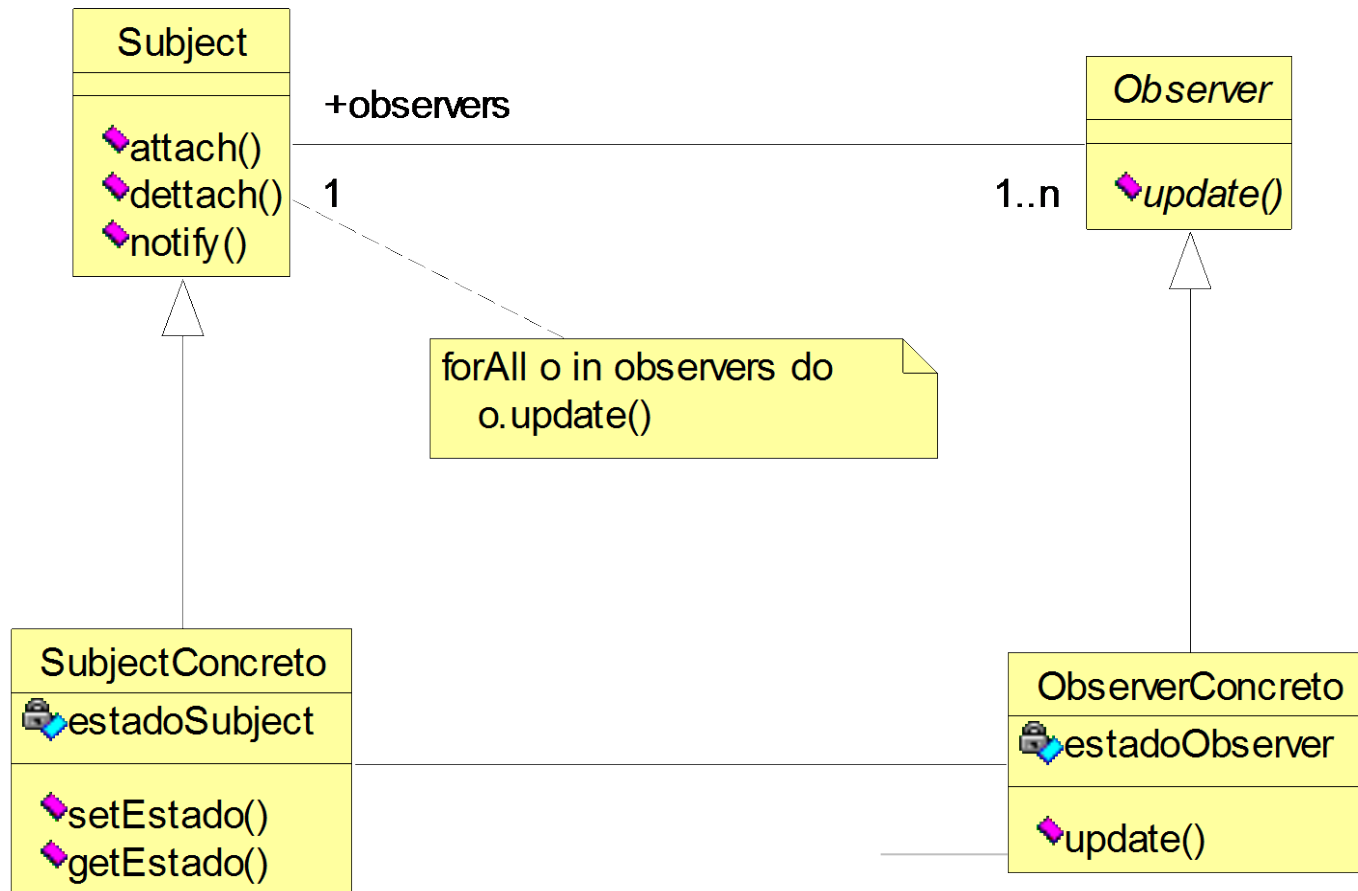
## Modelo



	<u>1tr.</u>	<u>2tr.</u>	<u>3tr.</u>	<u>4tr.</u>
Este	20	25	90	20
Oeste	30	38	32	32
Norte	47	47	45	45

# Observer

## Estructura



# Observer

- **Aplicabilidad**

- Cuando **un cambio de estado en un objeto requiere cambios en otros objetos**, y no sabe sobre qué objetos debe aplicarse el cambio.
- Cuando un objeto debe ser capaz de notificar algo a otros objetos, sin hacer asunciones sobre quiénes son estos objetos.

# Observer

## Modelo de eventos en Java...

```
public class CounterView extends Frame {  
    private TextField tf = new TextField(10);  
    private Counter counter;           // referencia al modelo  
    public CounterView(String title, Counter c) {  
        super(title); counter = c;  
        Panel tfPanel = new Panel();  
        ...  
        Button incButton = new Button("Increment");  
        incButton.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                counter.incCount();  
                tf.setText(counter.getCount() + ""); } } );  
        buttonPanel.add(incButton);  
    }  
}
```



# State

- **Propósito**

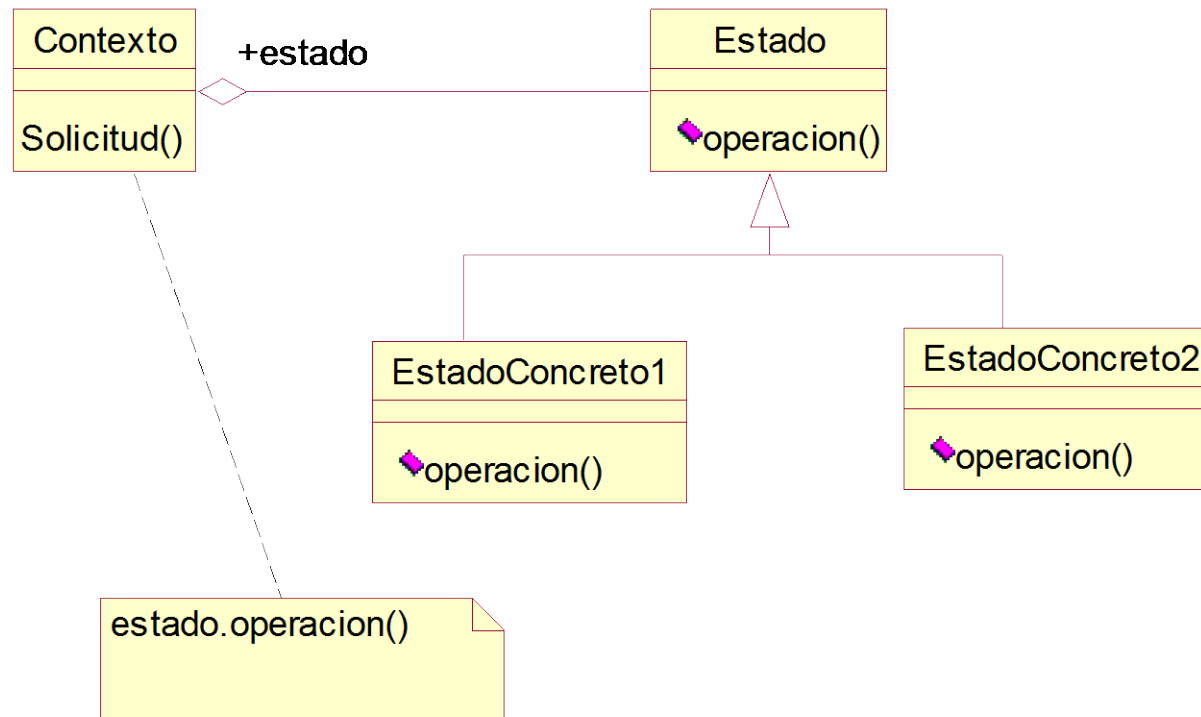
- Permite a un objeto cambiar su comportamiento cuando cambia su estado. El objeto parece cambiar de clase.

- **Motivación**

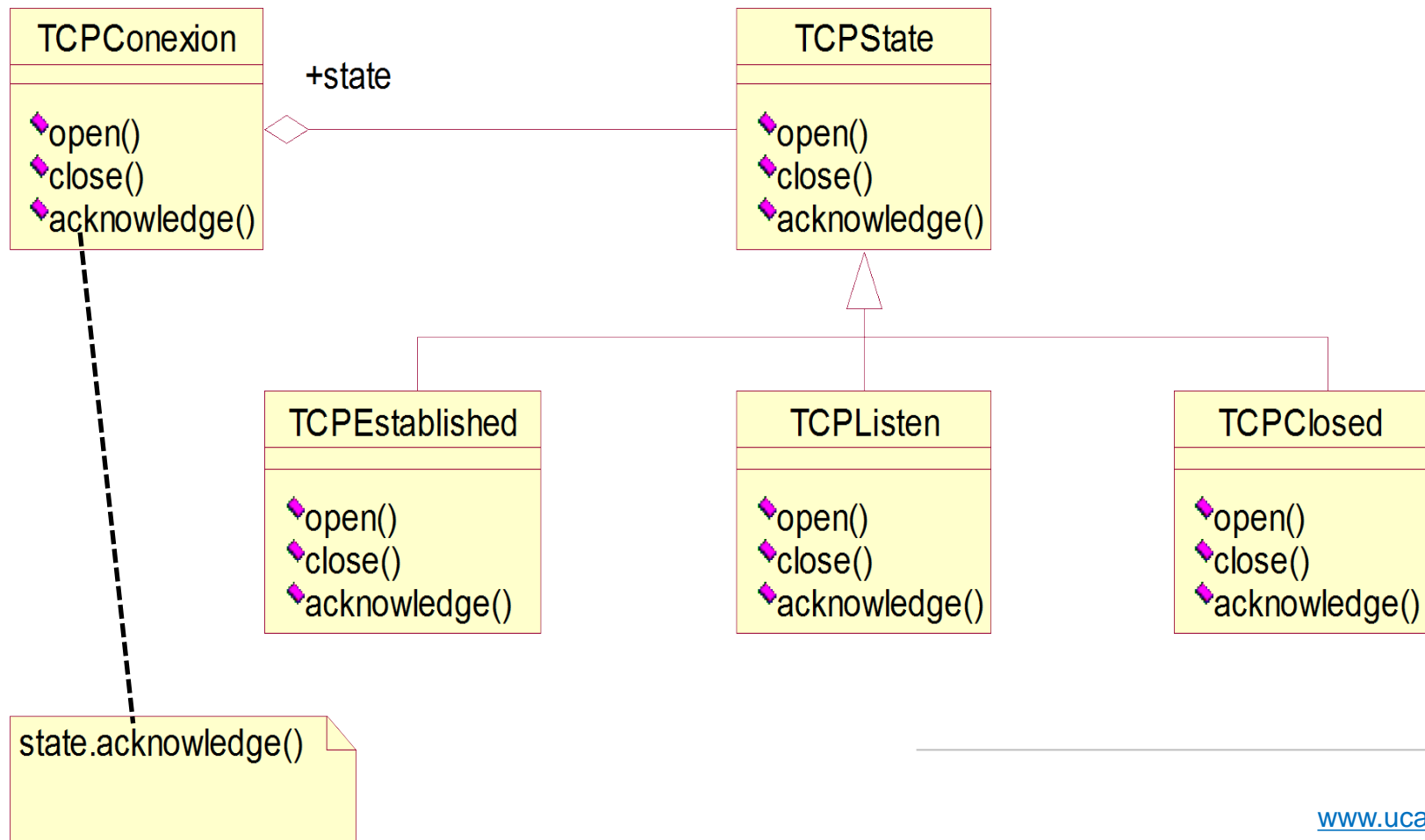
- Una conexión TCP puede encontrarse en uno de varios estados, y dependiendo del estado responderá de un modo diferente a los mensajes de otros objetos para solicitudes tales como abrir, cerrar o establecer conexión.

# State

## Estructura



# State



# State

- **Aplicabilidad**

- El comportamiento del objeto **depende de su estado**, y debe cambiar su comportamiento en tiempo de ejecución dependiendo de su estado.
- Las operaciones tienen grandes **estructuras CASE que dependen del estado del objeto**, que es representado por uno o más constantes de tipo enumerado.

# Strategy

- **Propósito**

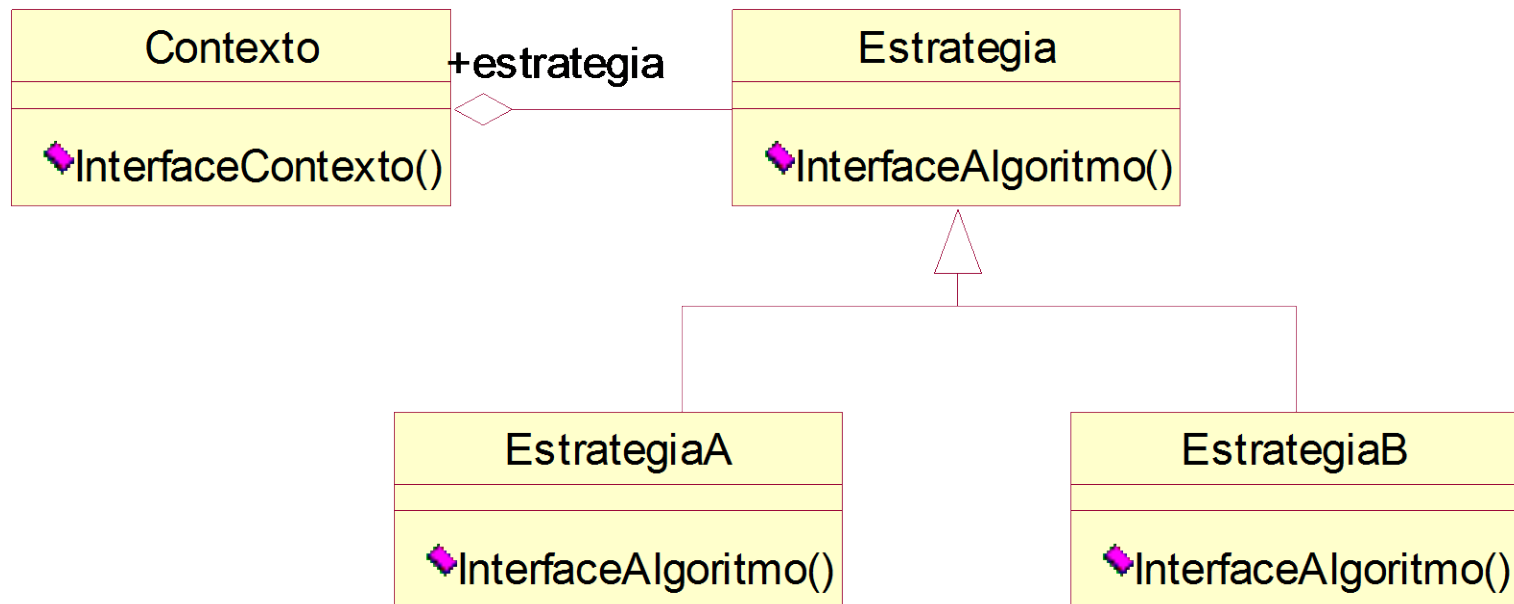
- Define una familia de algoritmos, encapsula cada uno, y permite intercambiarlos. Permite variar los algoritmos de forma independiente a los clientes que los usan.

- **Motivación**

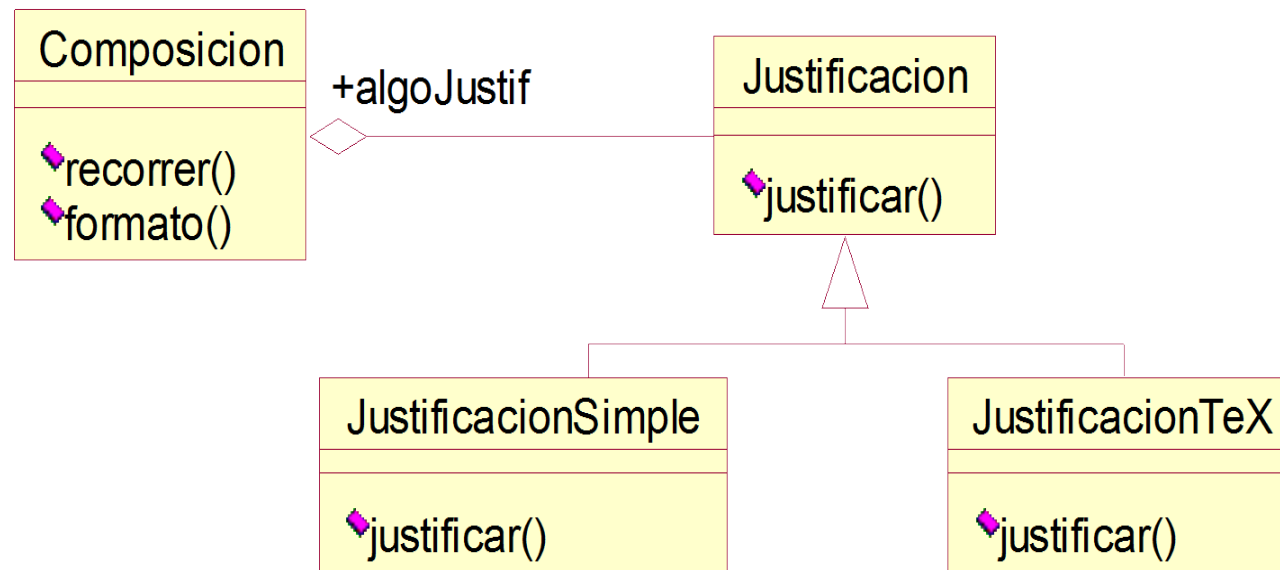
- Existen muchos algoritmos para justificación de texto, ¿debe implementarlo el cliente que lo necesita?

# Strategy

## Estructura



# Strategy



algoJustif.justificar

# Strategy

- **Aplicabilidad**

- Configurar una clase con uno de varios comportamientos posibles.
- Se necesitan diferentes **variantes de un algoritmo**.
- Una clase define muchos comportamientos que aparecen como sentencias CASE en sus métodos.



# Template Method

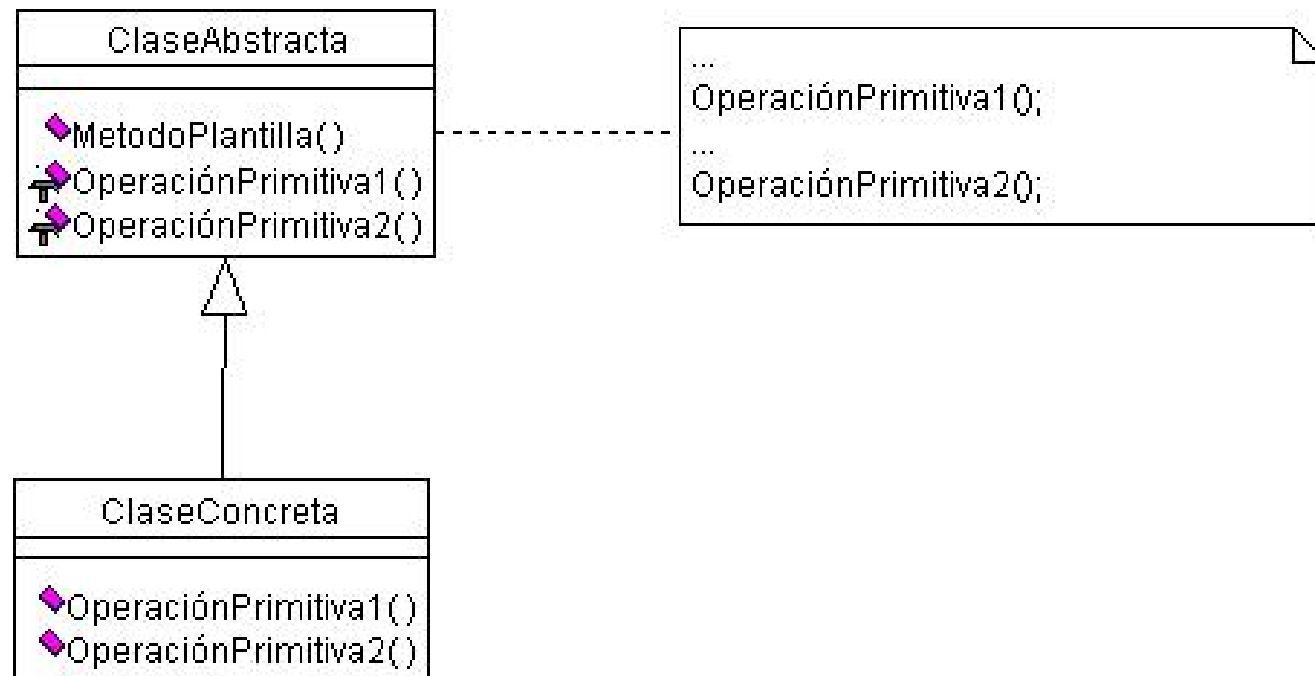
- **Propósito**

- Define el esqueleto (esquema, patrón) de un algoritmo en una operación, difiriendo algunos pasos a las subclases. Permite a las subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

- **Motivación**

- Fundamental para escribir código en un framework.
- Clase Aplicación que maneja objetos de la clase Documento: método OpenDocument

# Template Method



# Template Method

- **Aplicabilidad**
  - Factorizar el comportamiento común entre varias subclases.
  - Implementar las partes fijas de un algoritmo y dejar que las subclases implementen el comportamiento que puede variar.
  - Controlar extensiones de las subclases: algoritmos con puntos de extensión

# Visitor

- **Propósito**

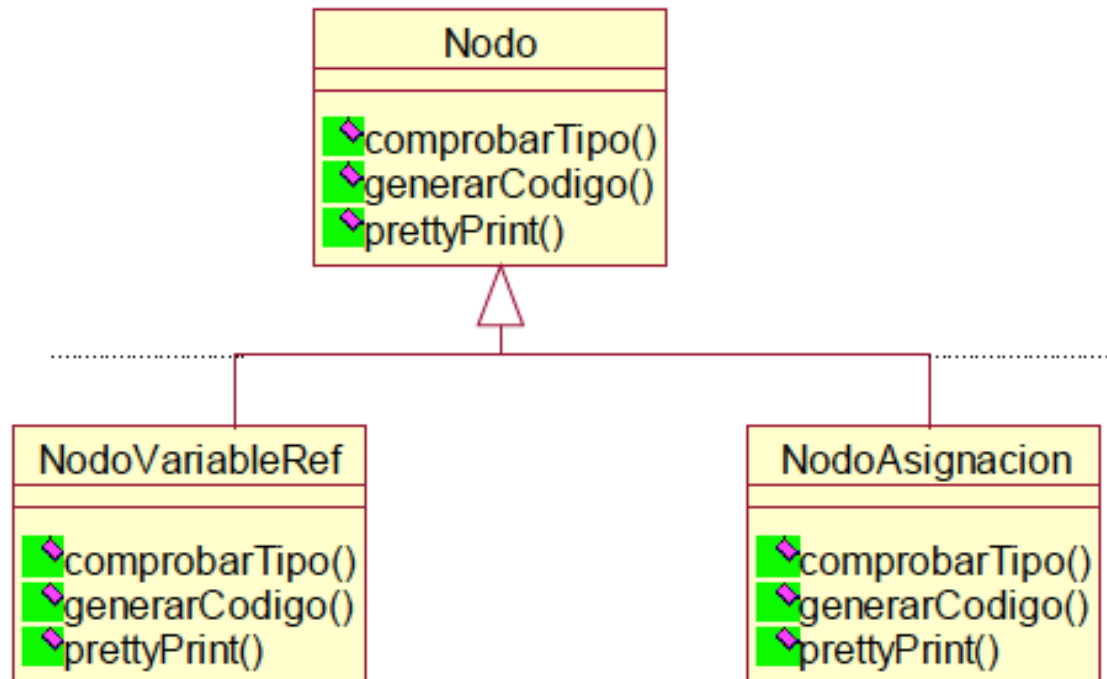
- Representar una operación que debe ser aplicada sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

- **Motivación**

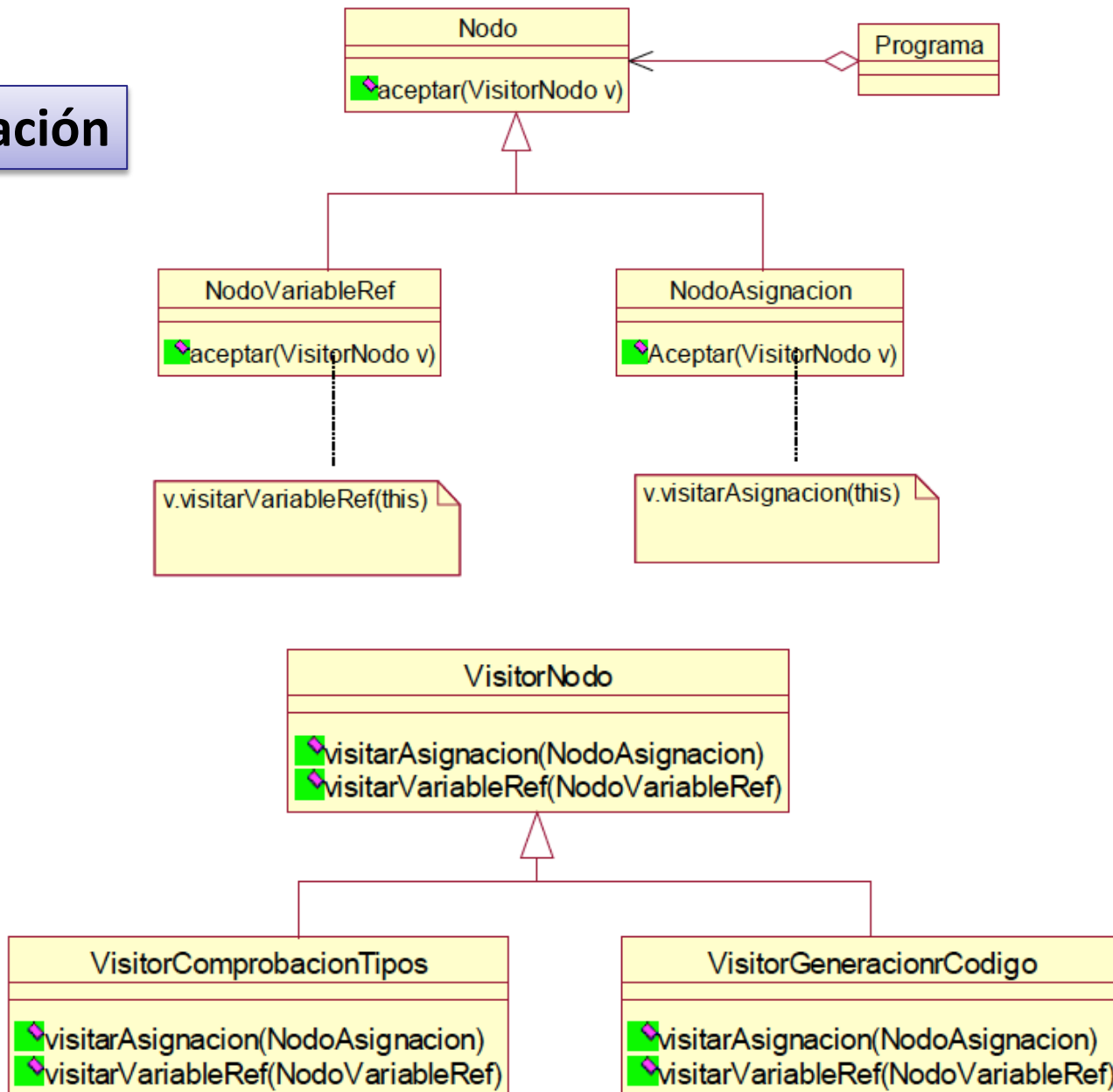
- Un compilador que representa los programas como árboles de derivación de la sintaxis abstracta necesita aplicar diferentes operaciones sobre ellos: comprobación de tipos, generación de código, .. y además listados de código fuente, reestructuración de programas,...

# Visitor

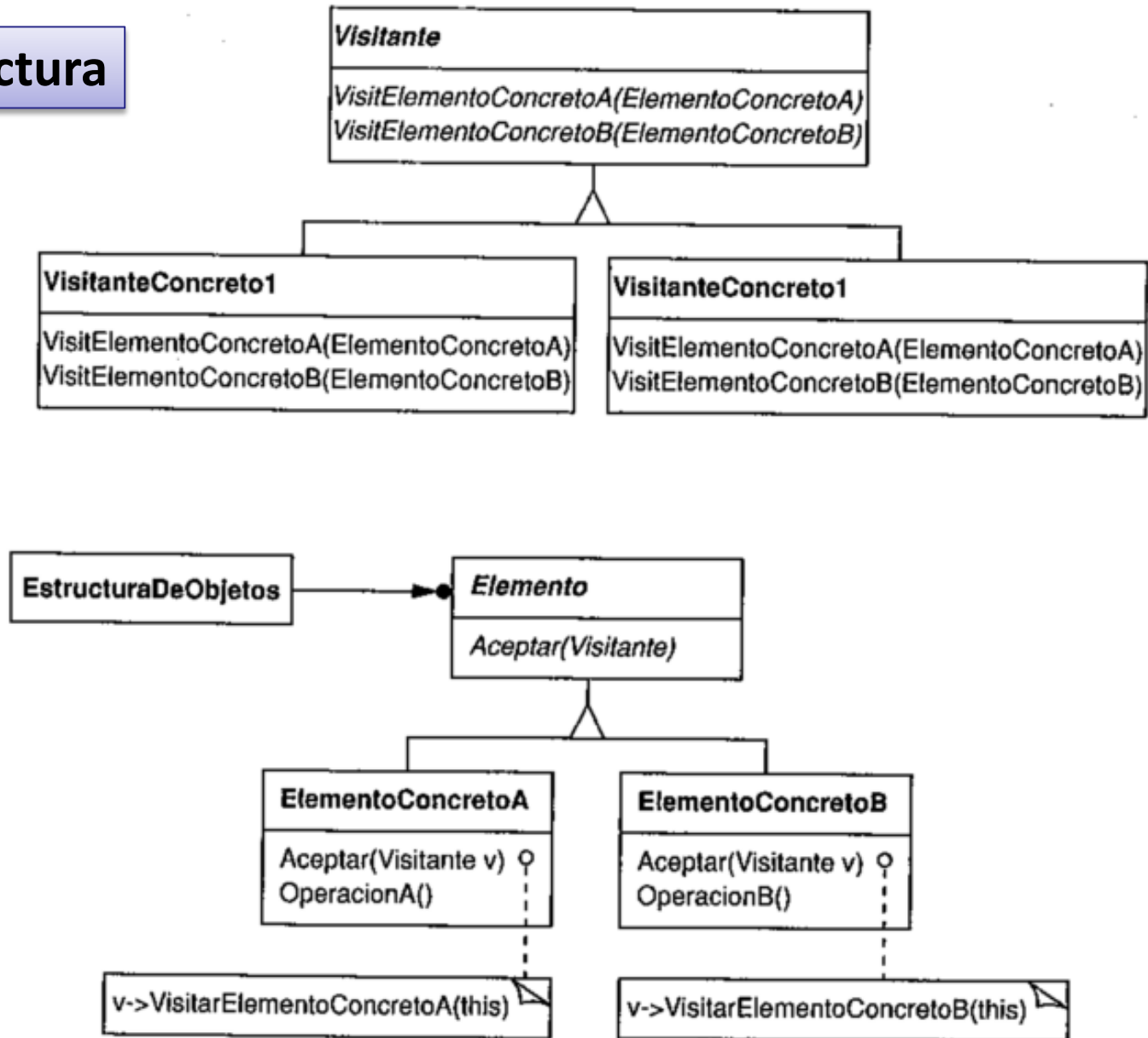
## Motivación



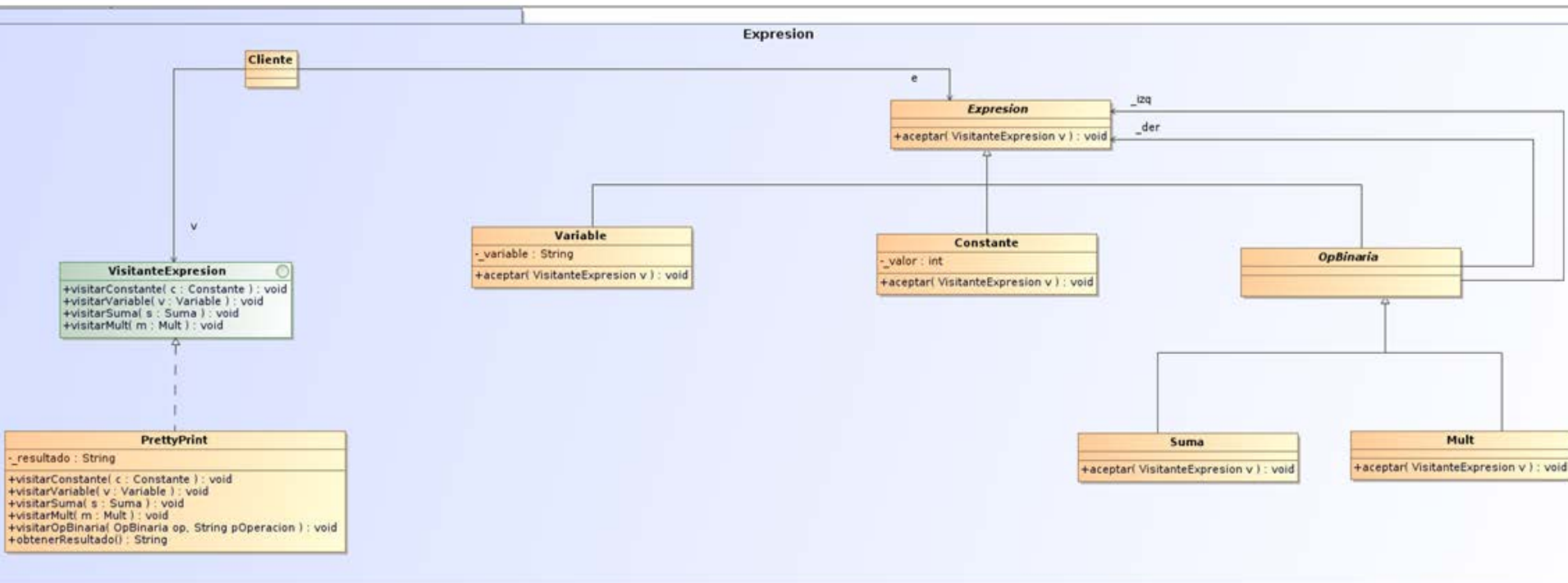
## Motivación



## Estructura

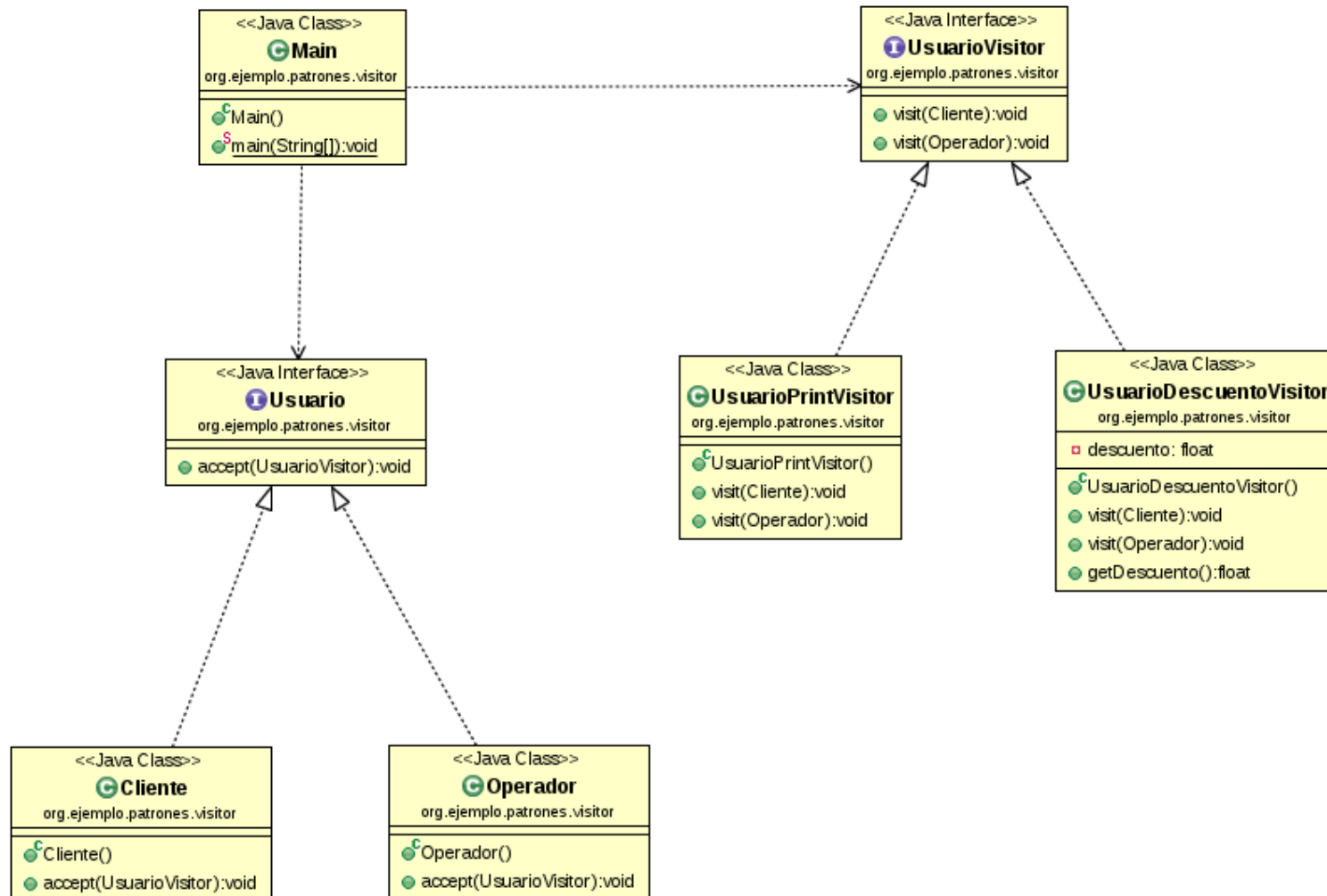


# Ejemplo 1





# Ejemplo 2



# Visitor

- **Aplicabilidad**

- Tenemos una **jerarquía de clases que representan objetos de propósito general** (p.e. nodos de un árbol de derivación de sintaxis) y podemos utilizarlo en diferentes aplicaciones, lo que implicaría añadir métodos en las clases de la jerarquía.
- Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y se quiere realizar operaciones sobre los objetos que dependen de las clases concretas.
- Las clases definiendo la estructura de objetos cambian con poca frecuencia, pero a menudo se definen nuevas operaciones sobre la estructura. Mejor definir las operaciones en clases aparte.