Diseño de Software

Patrones de Diseño de Software

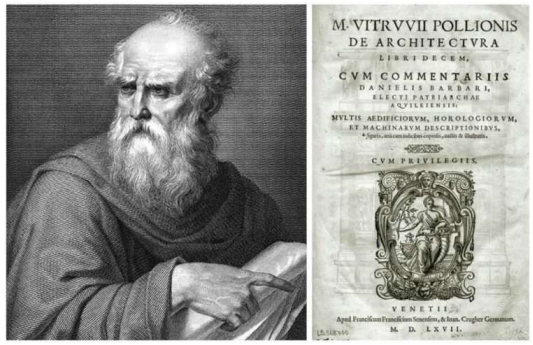
Mg. Marcela Daniele

Departamento de Computación

Facultad de Ciencias Exactas, Físico-Químicas y Naturales

Universidad Nacional de Río Cuarto

Diseño del Software



Marco Vitruvio Polión (siglo I a.C.), arquitecto e ingeniero romano, autor del primer manual de arquitectura de la humanidad, afirmaba que los edificios bien diseñados eran aquellos que tenían ***resistencia, funcionalidad y belleza****.*

Pressman sugiere aplicarlo al software…….

Resistencia*:* un programa no debe tener ningún error que impida su funcionamiento.

Funcionalidad*:* un programa debe se apropiado para los fines que persigue.

Belleza*:* la experiencia de usar el programa debe ser placentera.

Estos son los comienzos de una teoría del diseño de software.

* del libro de Pressman…..

Diseño del Software

El **diseño de software** es la etapa del desarrollo donde el ingeniero debe crear una representación o modelo de diseño del software, que proporcione de manera sólida y estable, todos los detalles de la arquitectura del software, las estructuras de datos, las interfaces y los componentes necesarios para implementar el sistema.

El **Modelo de Diseño** está constituido por:

1. **Diseño de Datos**
2. **Diseño Arquitectónico**
3. **Diseño de Interfaces**
4. **Diseño de Componentes**

Diseño del Software

- **Diseño de Datos o Clases:** transforma el modelo de clases de análisis en realizaciones de clases de diseño y en las estructuras de datos que se requieren para implementar el software.
- **Diseño Arquitectónico:** define la relación entre los elementos estructurales del software y las restricciones.
- **Diseño de Interfaces:** describe la forma de comunicarse el SW entre sí, la interoperabilidad con otros sistemas y con las personas que lo utilizan.
- **Diseño de Componentes:** transforma los elementos estructurales de la arquitectura del SW en una descripción procedimental de los componentes del software.

Diseño del Software

- El **diseño del software** es un proceso iterativo, donde los requisitos son traducimos o representados en un "plano" para la construcción del SW.

- ✓ El proceso de diseño inicia en un nivel alto de abstracción y por refinamientos sucesivos obtiene representaciones a niveles más bajos.

- ❖ La calidad de la evolución del diseño debe ser evaluada con una serie de revisiones de diseño.

- El **Diseño** debe:
  - Servir de guía, fácilmente legible y comprensible, para generadores de código y para el equipo de pruebas y soporte del SW.
  - Proporcionar una vista completa del comportamiento, funciones y datos del SW, desde una perspectiva de implementación.

Conceptos de Diseño del Software

- ABSTRACCION
- REFINAMIENTO
- MODULARIDAD
- ARQUITECTURA DEL SOFTWARE
- JERARQUIA DE CONTROL
- DIVISIÓN ESTRUCTURAL
- ESTRUCTURA DE DATOS
- PROCEDIMIENTO DE SOFTWARE
- OCULTACIÓN DE INFORMACIÓN
- INDEPENDENCIA FUNCIONAL:

  COHESIÓN y ACOPLAMIENTO

Conceptos de Diseño del Software

- **ABSTRACCION**

Una ***abstracción procedimental*** es una secuencia nombrada de instrucciones que tiene una función específica y limitada. Ej: Abrir una puerta. **Abrir** implica una secuencia de pasos procedimentales.

Una ***abstracción de datos*** es una colección nombrada de datos que describe un objeto de datos. **Puerta** es una abstracción de datos.

Una ***abstracción de control*** implica un mecanismo de control de programa sin especificar los datos internos.

- **REFINAMIENTO**

El ***refinamiento paso a paso*** es una estrategia de diseño descendente. Se desarrolla un programa, refinando sucesivamente los niveles de detalle procedimentales.

Conceptos de Diseño del Software

• MODULARIDAD

El SW se divide en componentes, llamados ***módulos***, que se integran para satisfacer los requisitos del problema.

Capacidad de descomposición modular. Usar un mecanismo sistemático para descomponer el problema en subproblemas, reduce la complejidad del problema, y obtiene una solución modular más efectiva.

Capacidad de empleo de componentes modulares. Ensamblar componentes de diseño (reusables) existentes, en un sistema nuevo.

Capacidad de comprensión modular. Si un módulo se puede comprender como una unidad autónoma (sin referencias a otros módulos) será más fácil de construir y de cambiar.

Continuidad modular. Si se cambian los módulos individuales, se minimizará el impacto de los efectos secundarios de los cambios.

Protección modular. Si en un módulo se produce una condición no deseada y sus efectos se limitan a ese módulo, se minimizará el impacto de los efectos secundarios inducidos por los errores.

Conceptos de Diseño del Software

• ARQUITECTURA DEL SOFTWARE

La arquitectura es la estructura jerárquica de los componentes (módulos), la manera en que interactúan y la estructura de datos que van a utilizar. Los patrones arquitectónicos permiten determinar formas generales para estructurar todo el sistema.

• JERARQUIA DE CONTROL

La relación de control entre módulos se expresa considerando una jerarquía entre módulos y de quién controla a quién.

• DIVISIÓN ESTRUCTURAL

La estructura del programa se puede dividir tanto horizontal como verticalmente.

• ESTRUCTURA DE DATOS

Es una representación de la relación lógica entre los datos, y determina una estructura de datos para el sistema.

Conceptos de Diseño del Software

- **PROCEDIMIENTO DE SOFTWARE**

Se debe proporcionar una especificación precisa de procesamiento de cada módulo, incluyendo la secuencia de sucesos, los puntos de decisión, las operaciones repetitivas y la organización de datos.

- **OCULTACIÓN DE INFORMACIÓN**

Sugiere que los módulos se caractericen por las decisiones de diseño que cada uno oculta al otro.

- **INDEPENDENCIA FUNCIONAL:**
  COHESIÓN y ACOPLAMIENTO

Crear módulos cohesivos que tengan una única función, y con poca interacción con las funciones que realizan otros módulos. El **acoplamiento** depende de la complejidad de interconexión entre módulos. En el diseño del software, intentamos conseguir el acoplamiento más bajo posible.

Patrones de Diseño de Software

¿ QUÉ ES UN PATRON DE DISEÑO DE SOFTWARE ?

Un **Patrón de Diseño** define una

solución general, estándar y reusable

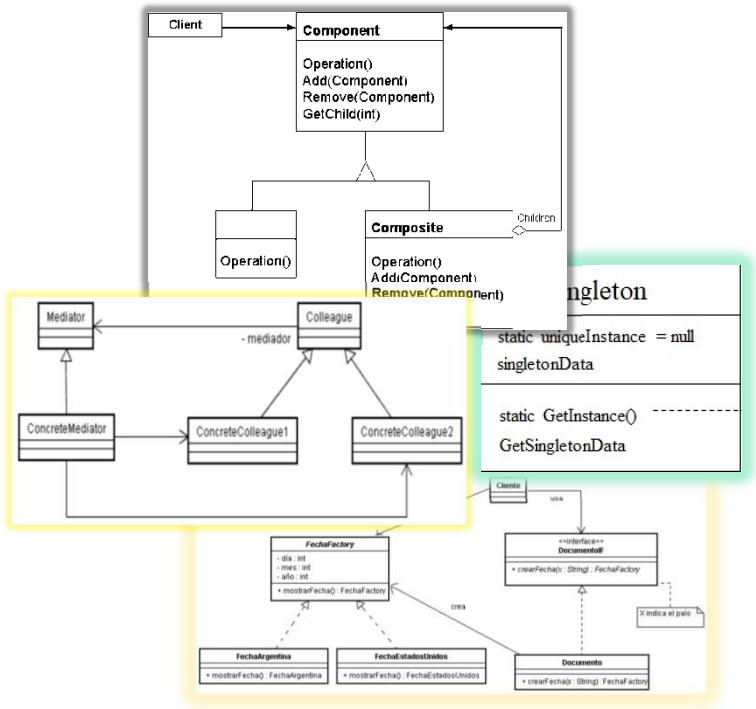
para ser aplicada a un

problema recurrente.

- En diferentes dominios, circunstancias, contextos, y en combinación con otros, pueden presentarse situaciones a resolver, que muestran características y comportamientos similares.
- Una solución estándar podrá servir para aplicar a problemas similares.
- Una solución reusable permitirá garantizar una solución apropiada a un cierto tipo de problemas.

¿ QUÉ ES UN PATRON DE DISEÑO DE SOFTWARE ?

Un Patrón de Diseño se representa por una estructura y descripción de clases (objetos comunicándose entre sí).

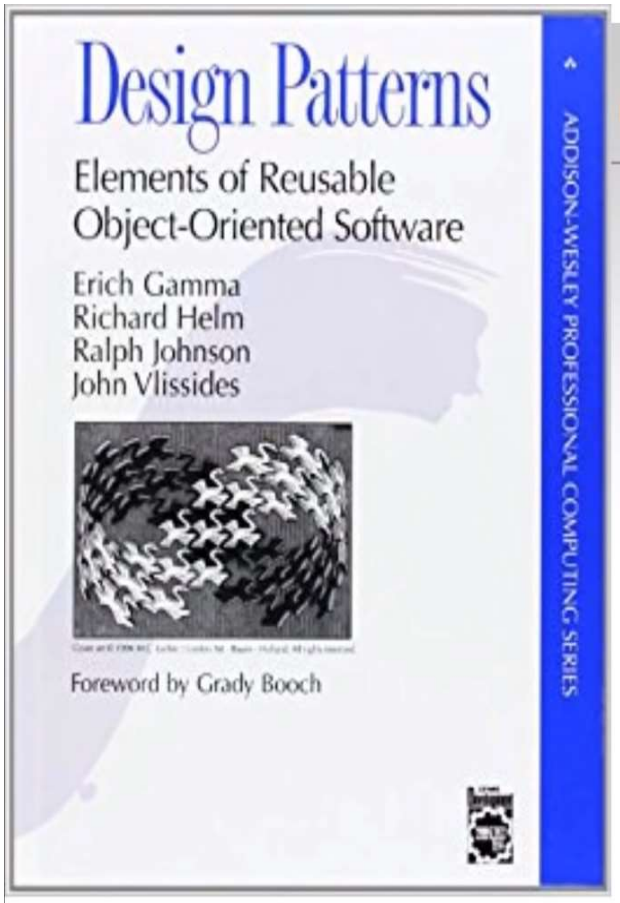


Para utilizar correctamente los patrones de diseño, se debe conocer y comprender muy bien el dominio del problema a resolver. Caso contrario, un uso inadecuado de un patrón podría retrasar el proyecto y llevar a una solución errónea.

¿ POR QUÉ UTILIZAR PATRONES DE DISEÑO ?

- Son soluciones probadas, que ayudan a resolver problemas en menos tiempo y sin errores.
- Diseñadores expertos documentan sus soluciones para que otros con menor experiencia las utilicen.
- Formalizan un vocabulario común entre diseñadores.
- Proporcionan catálogos de soluciones reusables para el diseño de sistemas de software.
- Evitan la búsqueda de soluciones a problemas ya conocidos y solucionados.
- Mejora la relación costo/beneficio, la rentabilidad, la calidad, …..
- Estandarizan la forma de realizar el diseño, de documentar las decisiones de diseño.
- Facilitan la capacitación y el aprendizaje a nuevos diseñadores, y su incorporación a equipos de desarrollo ya existentes.

PATRONES DE DISEÑO según GoF



OOP Design Patterns: The Gang of Four!

It all started around 1994 when a group of 4 IBM programmers: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides based on their coding experience , were able to observe and document a set of about 23 common problems and their best accepted solutions

< https://www.drupal.org/project/oop_design_patterns >

MitoCode

Elementos esenciales de un PD

El Nombre

- Identifica al Patrón
- Define un vocabulario común entre colegas

El Problema a resolver

- Describe cuando aplicar el patrón
- Explica el problema y su contexto

La Solución

- Describe los elementos que participan del diseño, sus relaciones, responsabilidades y colaboraciones.
- Define una descripción abstracta de un problema de diseño y la estructura de un grupo de elementos que lo resuelven. No describe una implementación concreta particular.

Las Consecuencias (positivas o negativas) de usarlo

- Resultados de aplicar el patrón.
- Impacto en la flexibilidad, extensibilidad y portabilidad de un sistema.

Forma general para describir PD (según GoF)

Nombre	¿Requiere explicación?
Intención	Brevemente, qué problema resuelve
También conocido como…	
Motivación	Escenario de ejemplo
Aplicabilidad	Cuándo debe ser utilizado
Estructura	Diagrama de clases de la solución
Participantes	Diccionario de las clases que participan
Colaboraciones	Relaciones entre las clases
Consecuencias	Ventajas e inconvenientes de su uso
Implementación	Técnicas recomendadas
Patrones relacionados	Usados por él o que lo usan a él

Clasificación de PD según su propósito (GoF)

- **Creacionales**

Resuelven problemas relativos a la creación/inicialización y configuración de objetos.

- **Estructurales**

Resuelven problemas relacionados a la estructuración o composición o de clases y objetos para formar estructuras más grandes.

- **Comportamiento**

Resuelven problemas relativos a la interacción o comunicación entre objetos y distribución de responsabilidades.

Clasificación de PD respecto a su propósito

Del Catálogo de Patrones del GOF veremos.....

Creacionales	Estructurales	Comportamiento
Singleton	Composite	Command
Abstract Factory	Proxy	Iterator
Factory Method		Mediator

Patrones de Diseño de Software de la categoría

CREACIONAL

Resuelven problemas relativos a la creación/inicialización y configuración de objetos.

- **Singleton**
- **Abstract Factory**
- **Factory Method**

Nombre: **SINGLETON** (Creacional)

Problema a resolver

se necesita UNA y solo UNA instancia de un objeto, y el acceso global al mismo.

SINGLETON

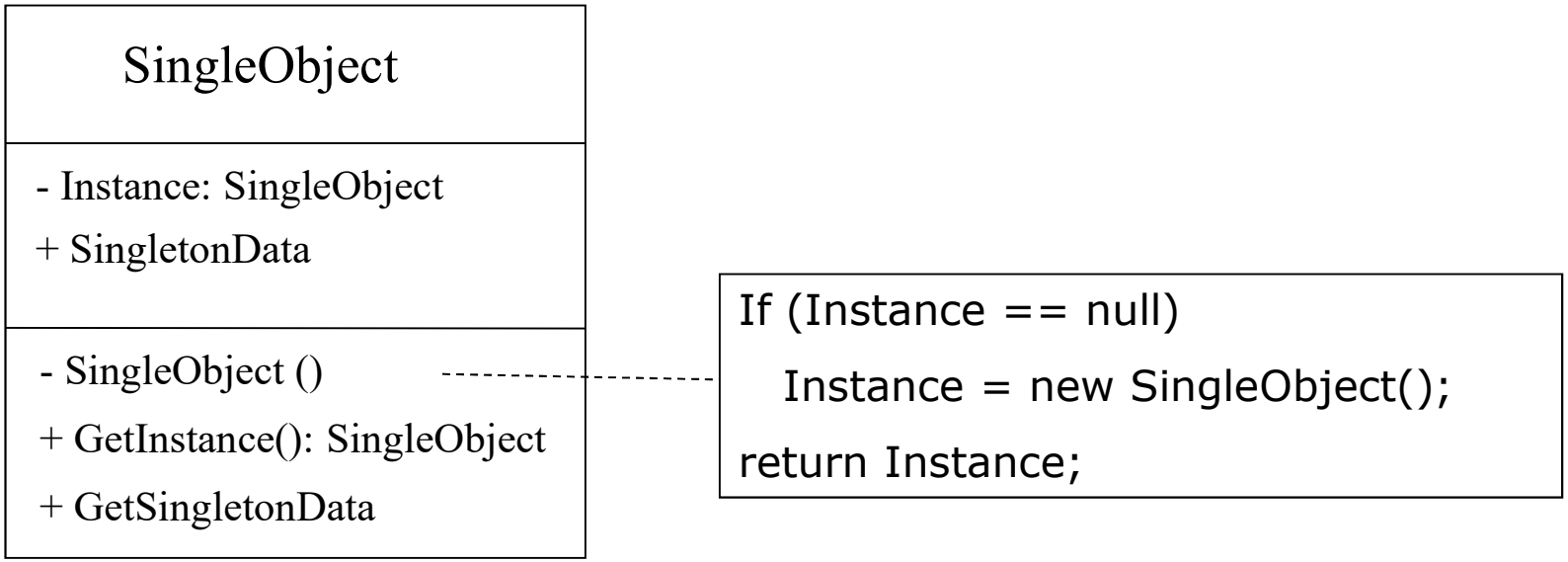
- Propone una solución que permite asegurar la creación de **una única instancia de la clase**
- Todos los objetos que la utilicen accederán de **forma global** a la misma y única instancia creada

Ejemplos

- ✓ Un *spooler* de impresión en un sistema, con una o más impresoras
- ✓ Un sistema de archivos. Una base de datos
- ✓ Un único registro de la configuración del sistema
- ✓ Una única estructura con los datos generales de una empresa

Nombre: **SINGLETON** (Creacional)

Estructura - Implementación



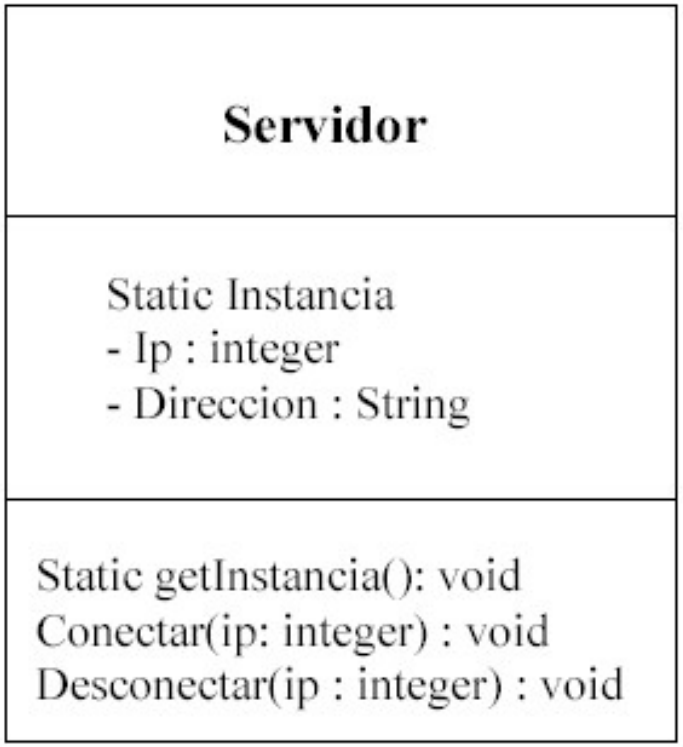
El constructor **SingleObject** se define *Private Static*. *Private* asegura que se creará una única instancia. *Static* que permitirá el acceso global.

GetInstance(): método estático que permite a los clientes acceder de manera global a **Instance,** única instancia de la clase.

SINGLETON - ejemplos

Problema:

Tenemos un servidor de Internet que debe ser *único* con las funciones de conectar o desconectar a direcciones ip de servidores externos.



Nombre: **FACTORY METHOD** (Creacional)

Problema a resolver

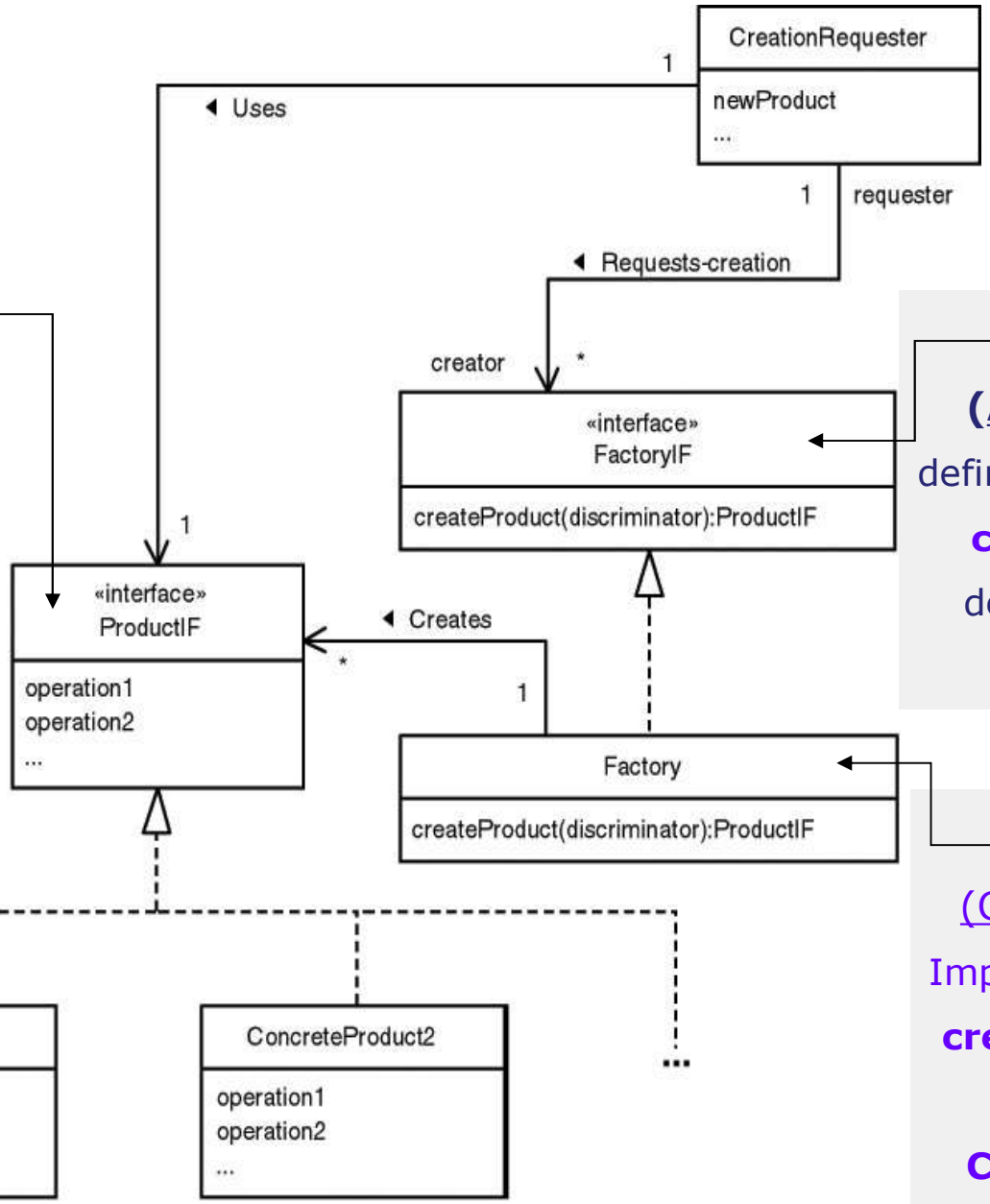
- Cuando una clase no puede anticipar el tipo de objetos que debe crear.
- Cuando una clase quiere que sus subclases especifiquen los objetos que deben crear.
- Cuando hay clases que delegan las responsabilidades en una o más subclases.

FACTORY METHOD

- Define una interfaz (**creador abstracto**) para la creación de objetos,
- Delega en las subclases (**creador concreto)** la decisión de la clase a instanciar o la creación definitiva

FACTORY METHOD (Creacional)

Estructura



ProductIF
define la interfaz de los objetos creados por el método de fabricación

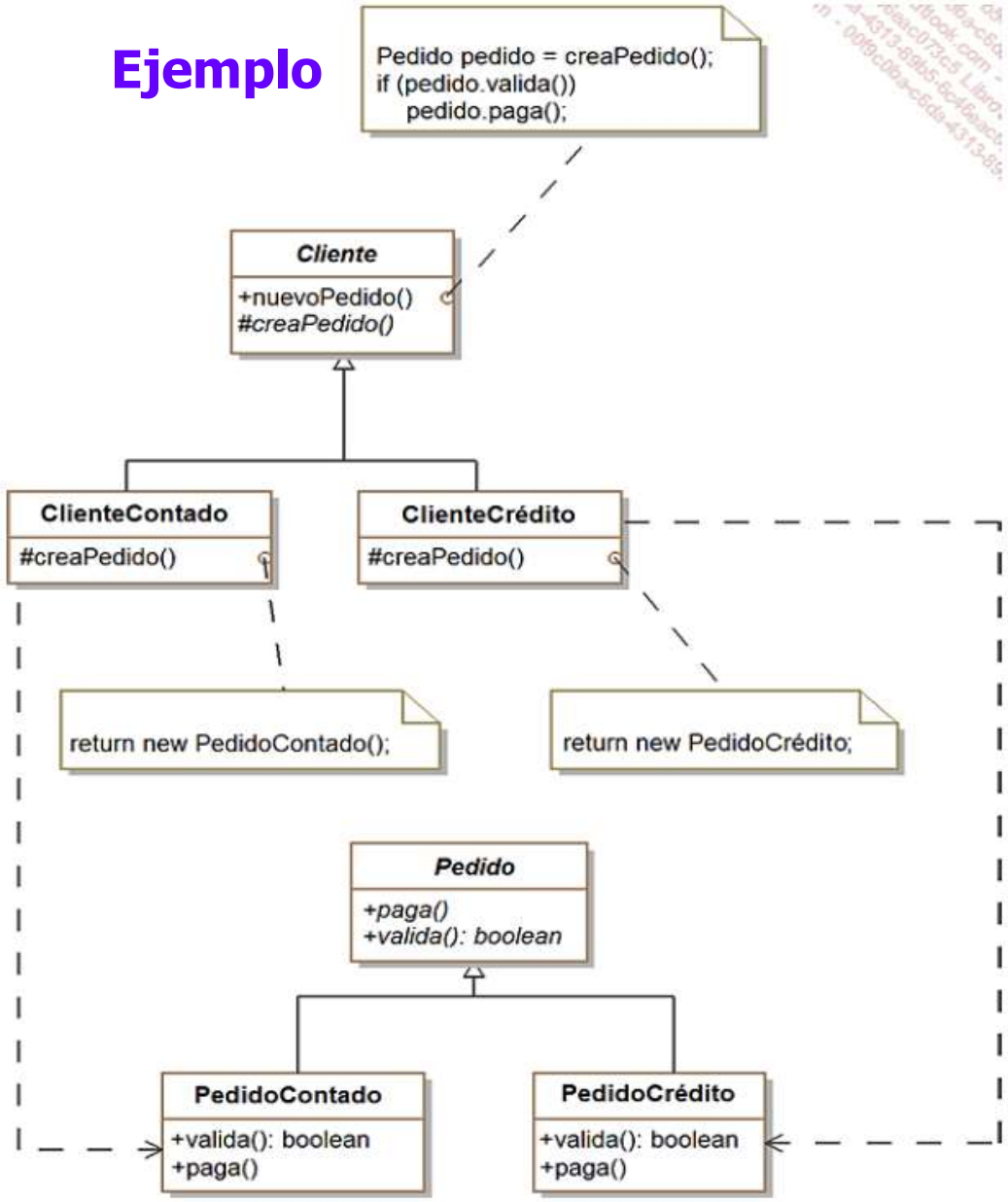
FactoryIF
(AbstractCreator)
define el método fábrica **createProduct** → devuelve un objeto ProductIF

ConcreteProduct
implementa **ProductIF**

Factory
(ConcreteCreator)
Implementa el método **createProduct,** crea instancias de **ConcreteProduct**

FACTORY METHOD (Creacional)

Ejemplo



Pedido: Define la interfaz de los objetos creados por el método de fabricación.

PedidoContado / **PedidoCrédito**: implementan **Pedido** según tipo de cliente.

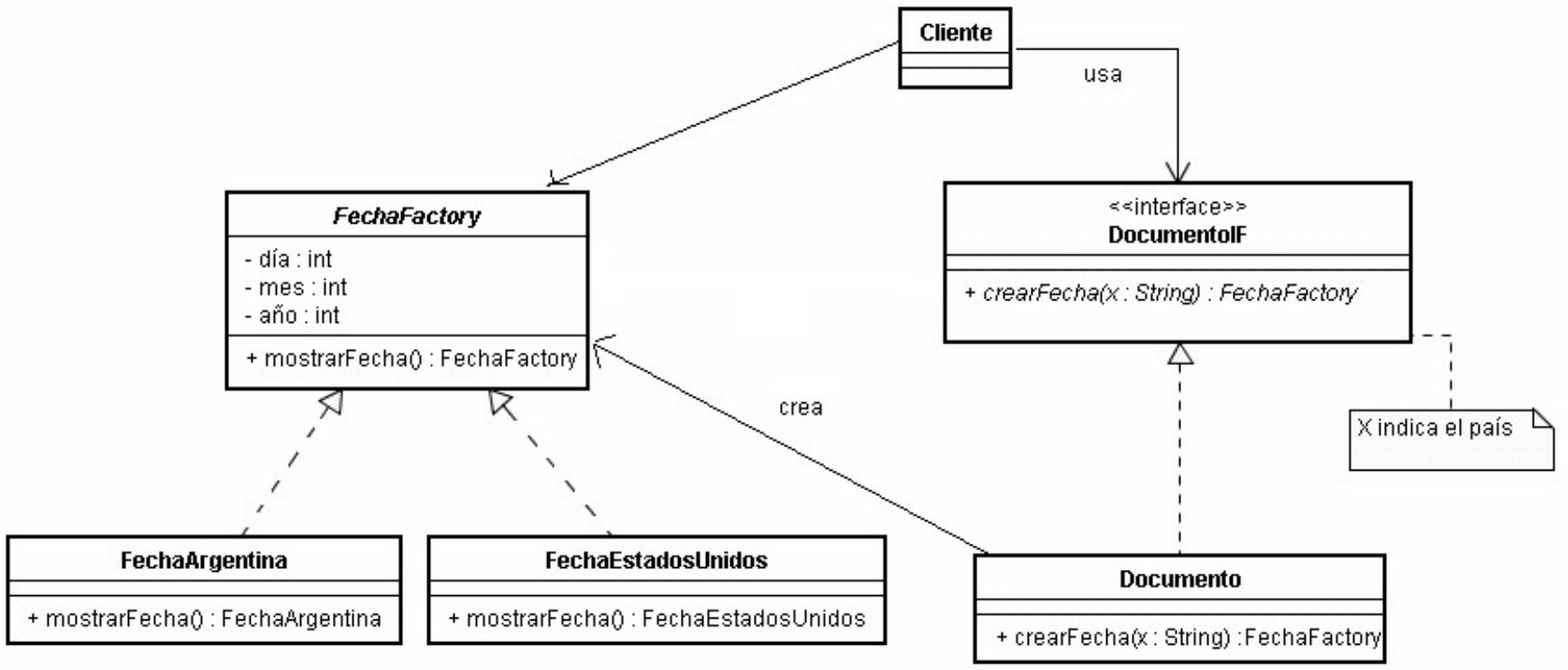
Cliente: declara el método de fabricación, **creaPedido()**

Cliente Contado: Implementa el método **creaPedido,** si el cliente paga de contado

Cliente Crédito: Implementa el método **creaPedido,** si el cliente paga a crédito

FACTORY METHOD (Creacional)

Ejemplo



Nombre: **ABSTRACT FACTORY** (Creacional)

Provee una interface que permite crear familias de objetos relacionados o dependientes,
sin conocer sus clases concretas responsables de la instanciación.

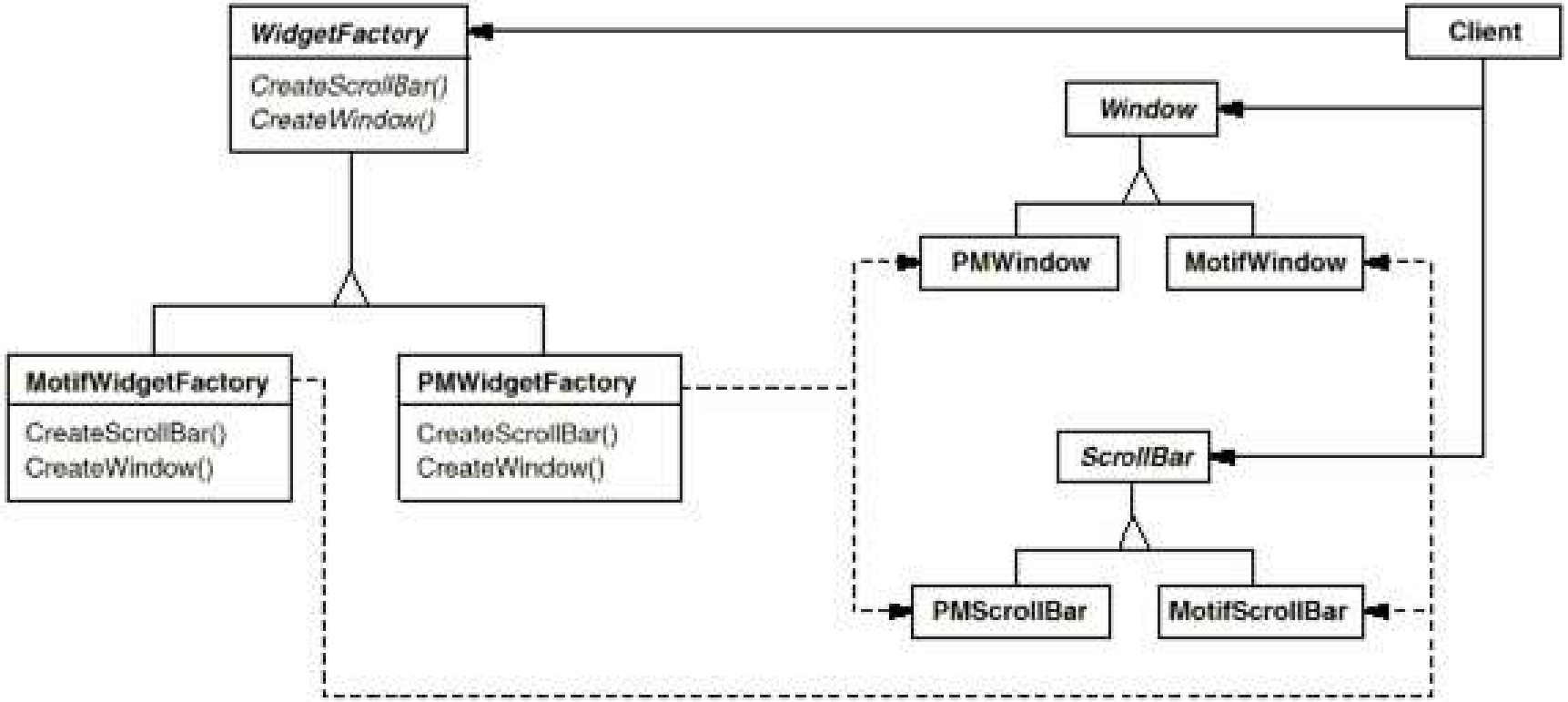
- *Es principalmente usado cuando, es necesario organizar un conjunto de elementos que trabajan con productos diferentes, pero muy relacionados de alguna manera.*
- *Evita crear clases para cada caso específico o concreto.*

Ejemplos:

- Construir un framework de interfaces de usuario que trabaje con múltiples sistemas de ventanas, tal como MS-Windows, Openview, PM, Motif o MacOS. Cada plataforma debe trabajar con sus características nativas.
- Una empresa fabrica computadoras con diferentes arquitecturas. Cada arquitectura, posee un conjunto distinto de componentes pero con similares funcionalidades. Diseñar el diagnóstico de los componentes principales de las computadoras que fabrica la empresa X.

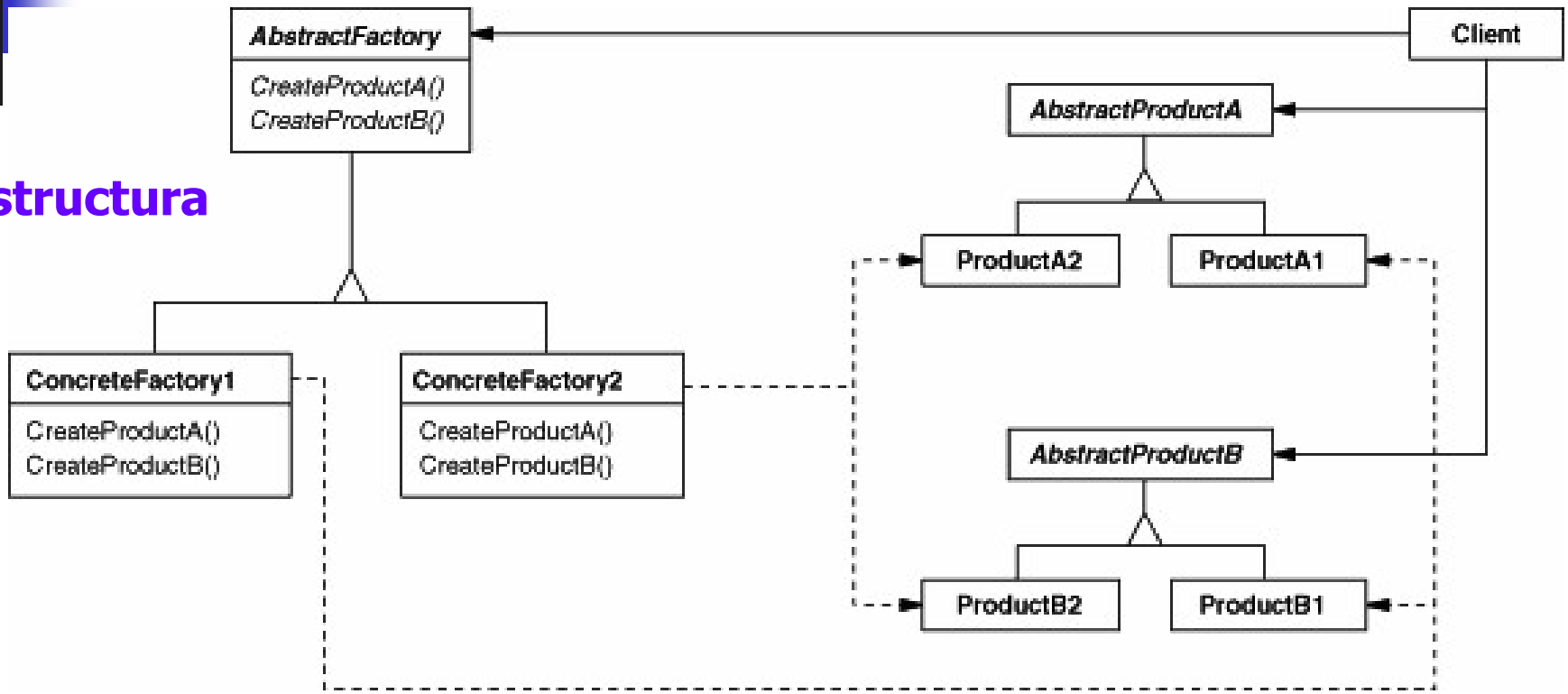
ABSTRACT FACTORY (Creacional)

Ejemplo



ABSTRACT FACTORY (Creacional)

Estructura



Client Sólo conoce las clases abstractas.

AbstractFactory Define métodos abstractos para crear instancias de clases Producto concretas.

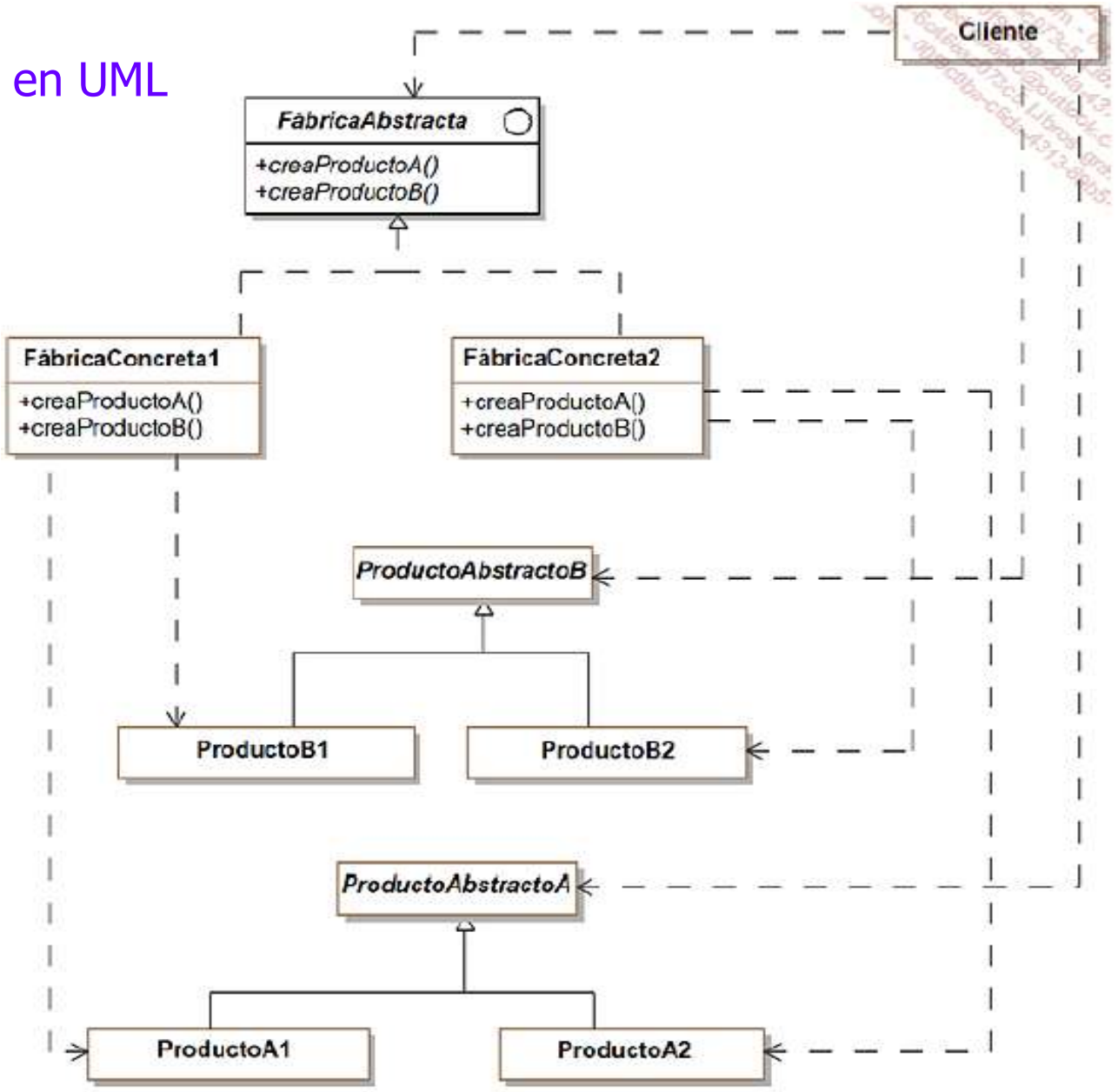
ConcreteFactory1, ***ConcreteFactory2*** Implementan los métodos definidos por la superclase **AbstractFactory** para crear instancias de clases Producto concretas.

AbstractProductA, ***AbstractProductB*** Clases abstractas que corresponden a una característica de un producto con la que sus subclases concretas trabajarán.

ProductA1, ***ProductA2***, ***ProductB1***, ***ProductB2*** Clases concretas.

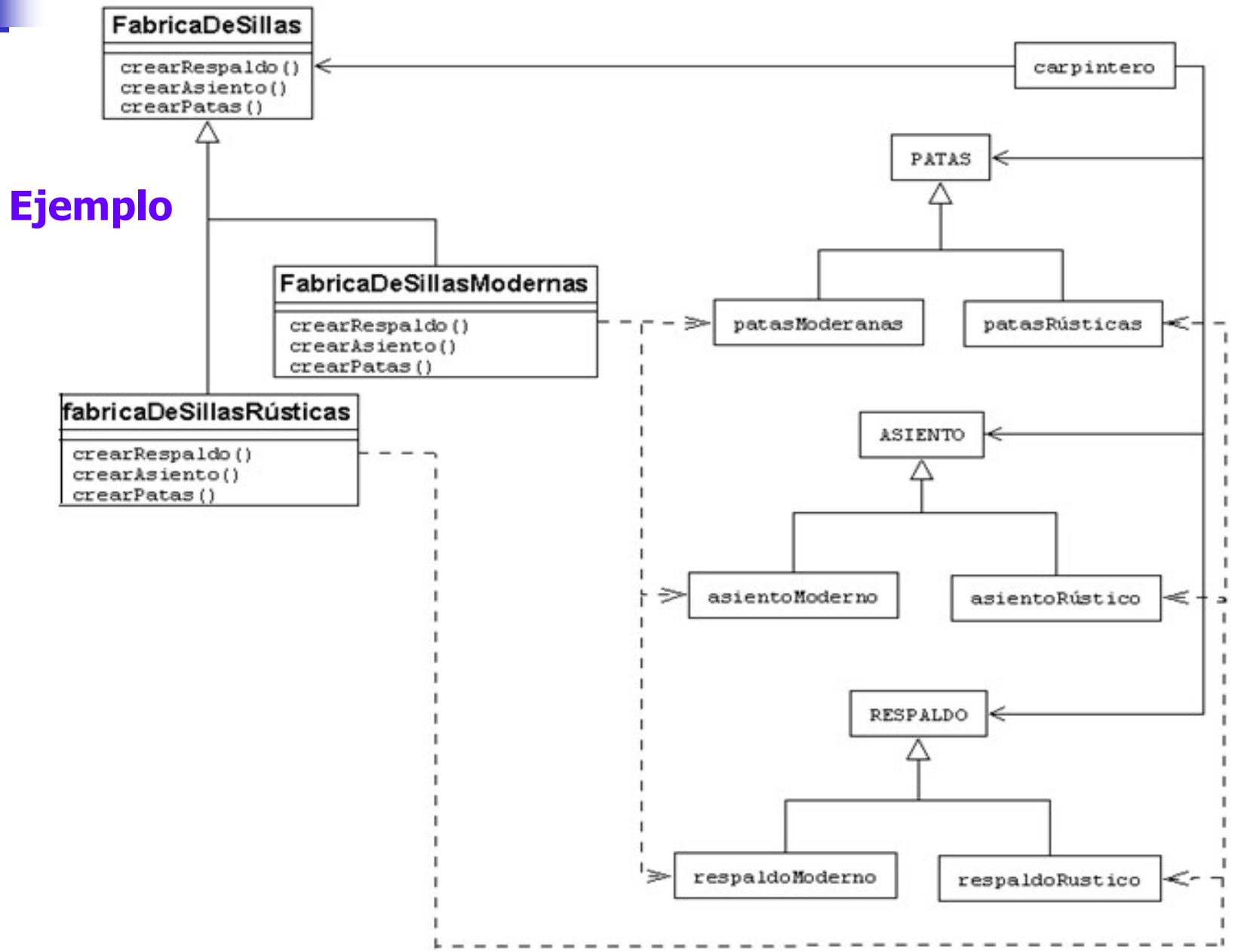
ABSTRACT FACTORY (Creacional)

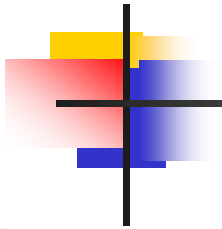
Estructura en UML



ABSTRACT FACTORY (Creacional)

Ejemplo





Patrones de Diseño de Software de la categoría

ESTRUCTURAL

Resuelven problemas relacionados a la estructuración o composición o de clases y objetos para formar estructuras más grandes.

- **Composite**
- **Proxy**

Nombre: **COMPOSITE** (Estructural)

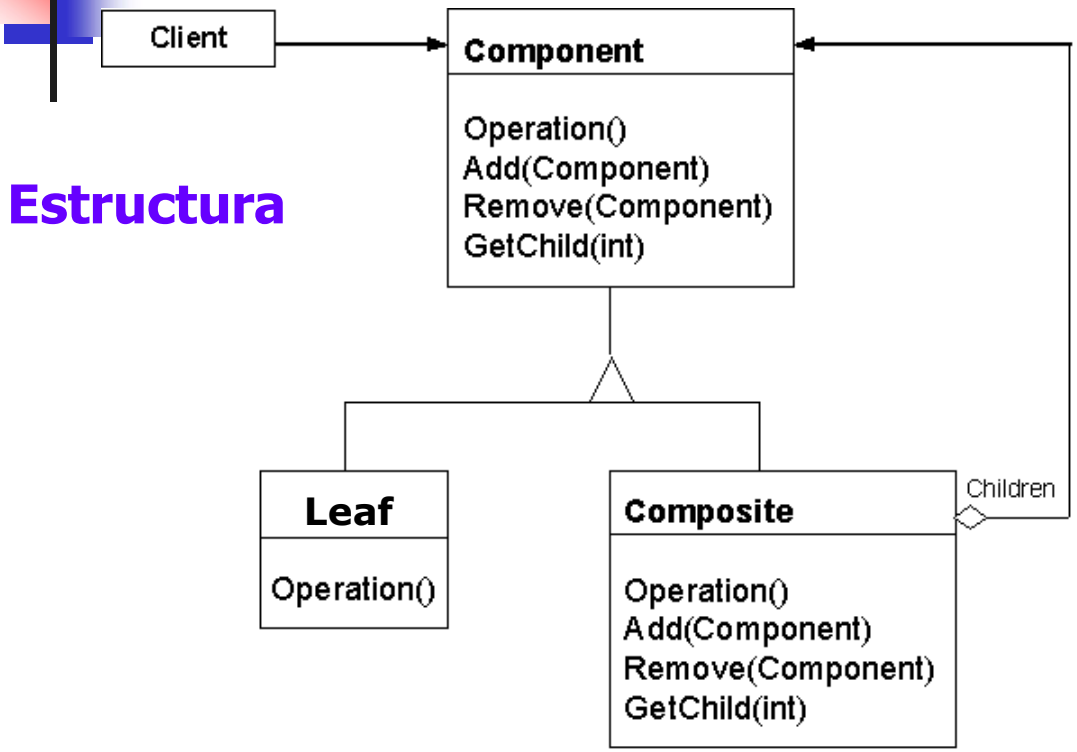
Permite representar jerarquías de objetos.

Una estructura de árbol del objetos simples y compuestos, "todo-parte".

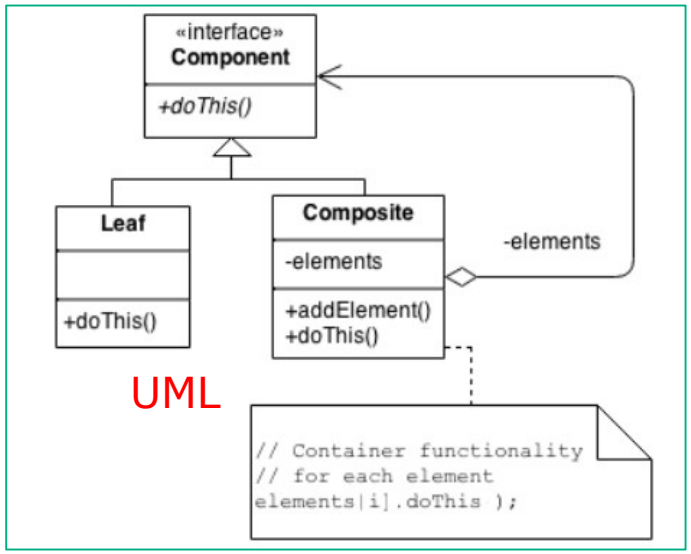
Composite ofrece un marco de diseño de una composición de objetos de profundidad variable.

- *Se utiliza cuando se requiere administrar, indistintamente, objetos individuales o composiciones de objetos, y tratarlos como una estructura compuesta uniforme.*
- *Cuando es conveniente construir objetos de complejidad mayor, mediante otros más sencillos de forma recursiva, formando una estructura similar a un árbol.*

COMPOSITE (Estructural)



Estructura



Client: Crea los objetos

Component (Abstracto): Clase común a todos los objetos, define los métodos u operaciones que realizan tanto los objetos individuales como los compuestos.

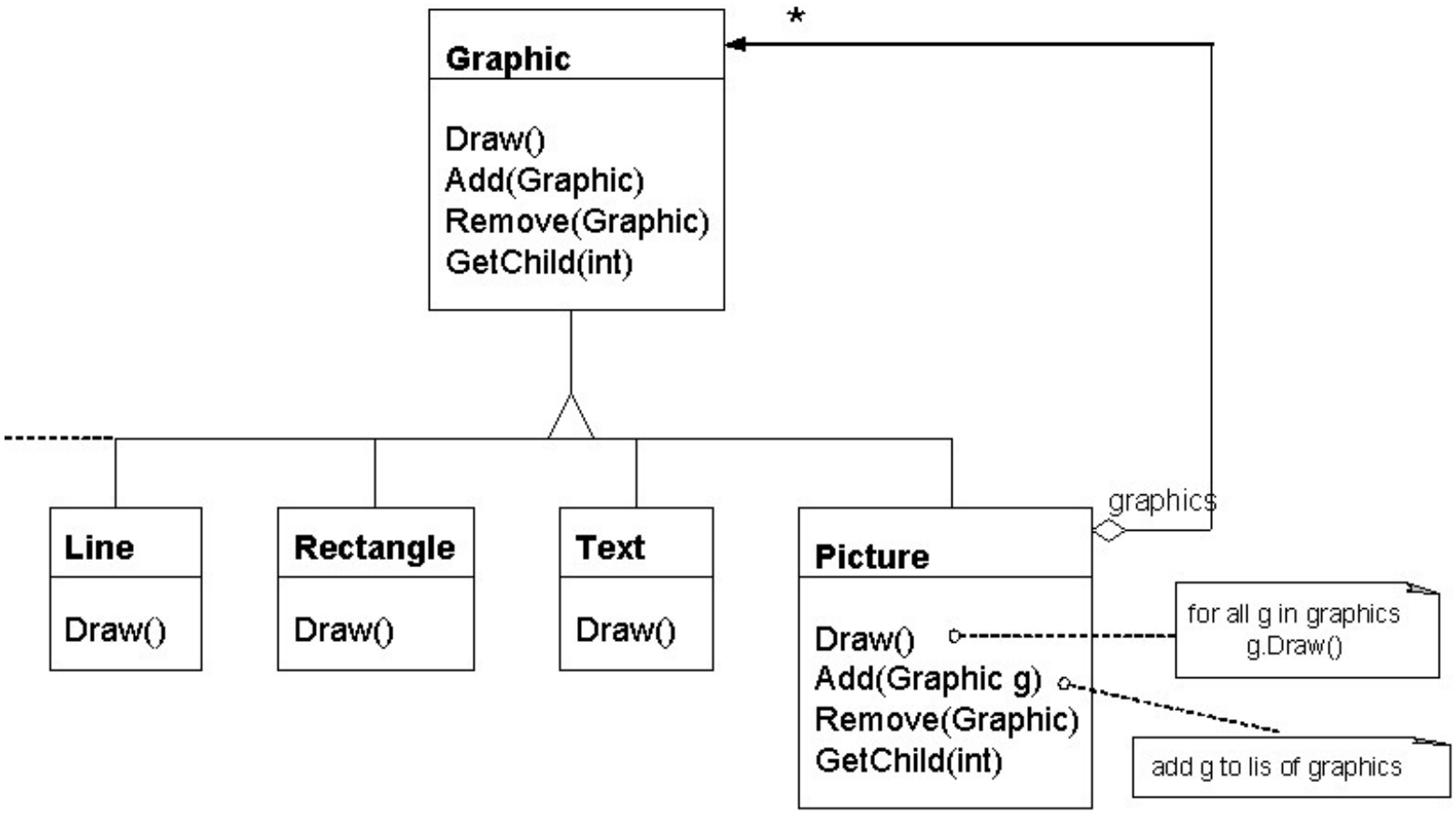
Composite (Abstracto): métodos propios de los objetos compuestos, **Add(Component)** añade componentes tanto individuales como compuestos, **Remove(Component)** para eliminarlos, **GetChild(n)** para recuperar los distintos objetos que lo componen.

Leaf: componente que representa las clases individuales, no compuestas.

COMPOSITE (Estructural)

Ejemplo

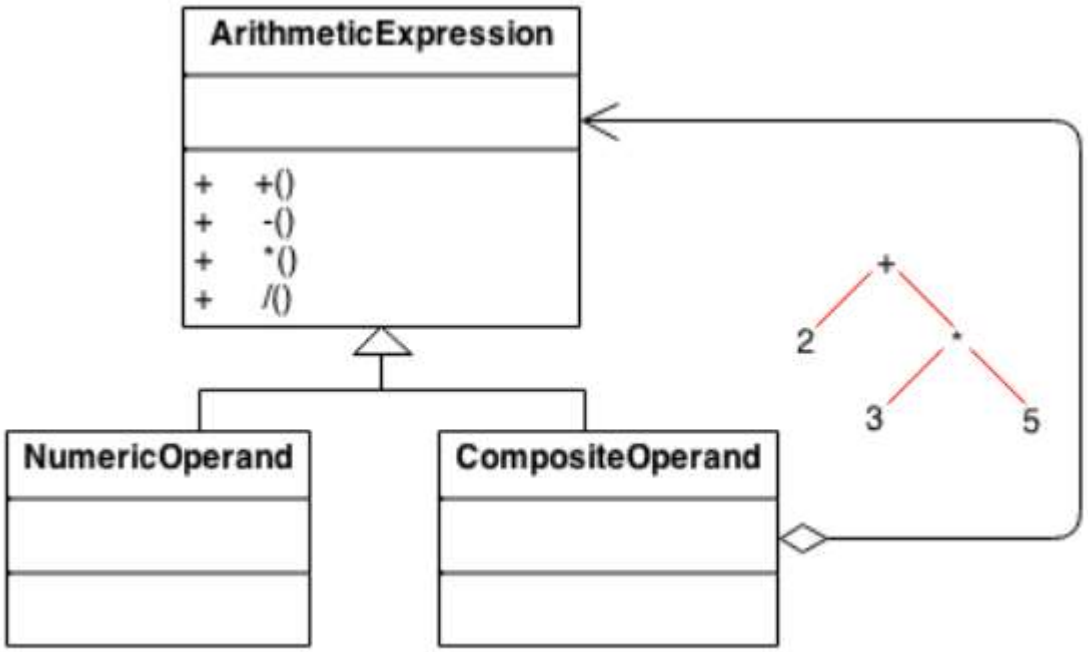
Un gráfico está compuesto por textos, rectángulos, líneas, figuras, etc. Una figura es también un gráfico.



COMPOSITE (Estructural)

Ejemplo

Una expresión aritmética consiste en un operando, un operador (+ - * /) y otro operando. El operando puede ser un número u otra expresión aritmética. Por lo tanto, 2 + 3 y (2 + 3) + (4 * 6) son expresiones válidas.



Nombre: **PROXY** (Estructural)

Un objeto sustituye a otro para controlar el acceso al mismo.

El objeto que realiza la sustitución posee la

misma interface que el sustituido,

esto lo vuelve transparente para el cliente que accede

- *Se necesita resguardar el espacio de la memoria, y se requiere diferir la creación de objetos hasta que el cliente lo requiera.*
- *Se requiere acceder a un objeto que está en un sistema remoto.*
- *Se debe controlar el acceso a un objeto que comparten múltiples clientes*

PROXY (Estructural)

Proxy Virtual

Se retrasa la instanciación de un objeto hasta que es necesario utilizarlo. Un objeto proxy lo sustituye ofreciendo la misma interfaz, y cuando es necesario solicita al objeto real la información que necesita.

Proxy Remoto

El objeto original está en un sistema remoto. Un objeto proxy sustituye el objeto que está en un sistema remoto, permitiendo controlar el acceso al mismo.

Proxy de Protección

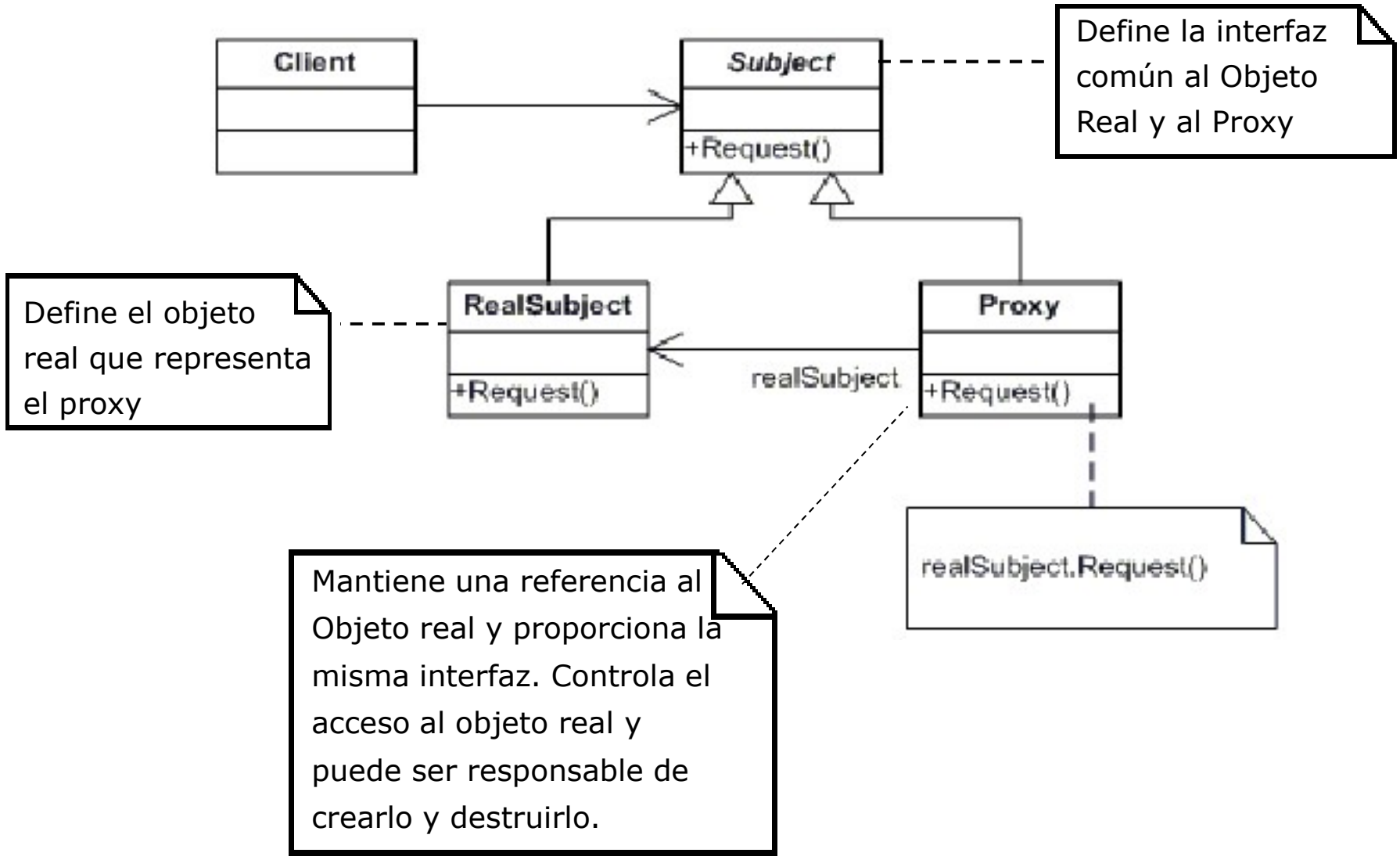
Control de permisos para el acceso a un objeto.

Proxy de Sincronización

Controla accesos de múltiples clientes a un recurso.

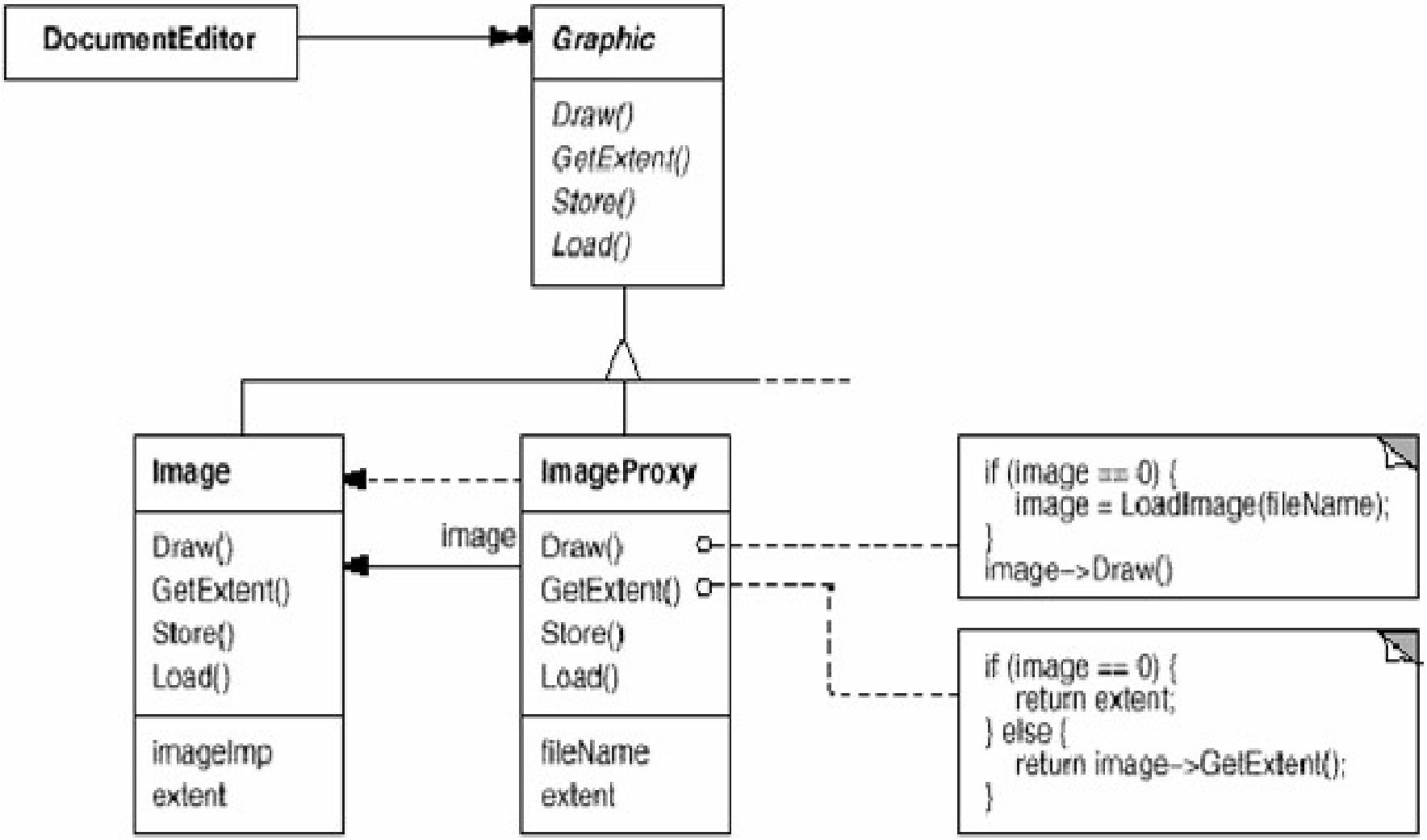
PROXY (Estructural)

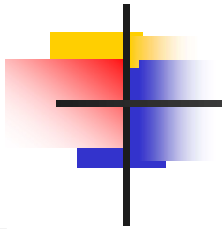
Estructura



PROXY (Estructural)

Ejemplo





Patrones de Diseño de Software de la categoría

COMPORTAMIENTO

Resuelven problemas relativos a la interacción o comunicación entre objetos y distribución de responsabilidades

- **Command**
- **Mediator**
- **Iterator**

Nombre: **COMMAND** (Comportamiento)

Encapsula una solicitud como un objeto.

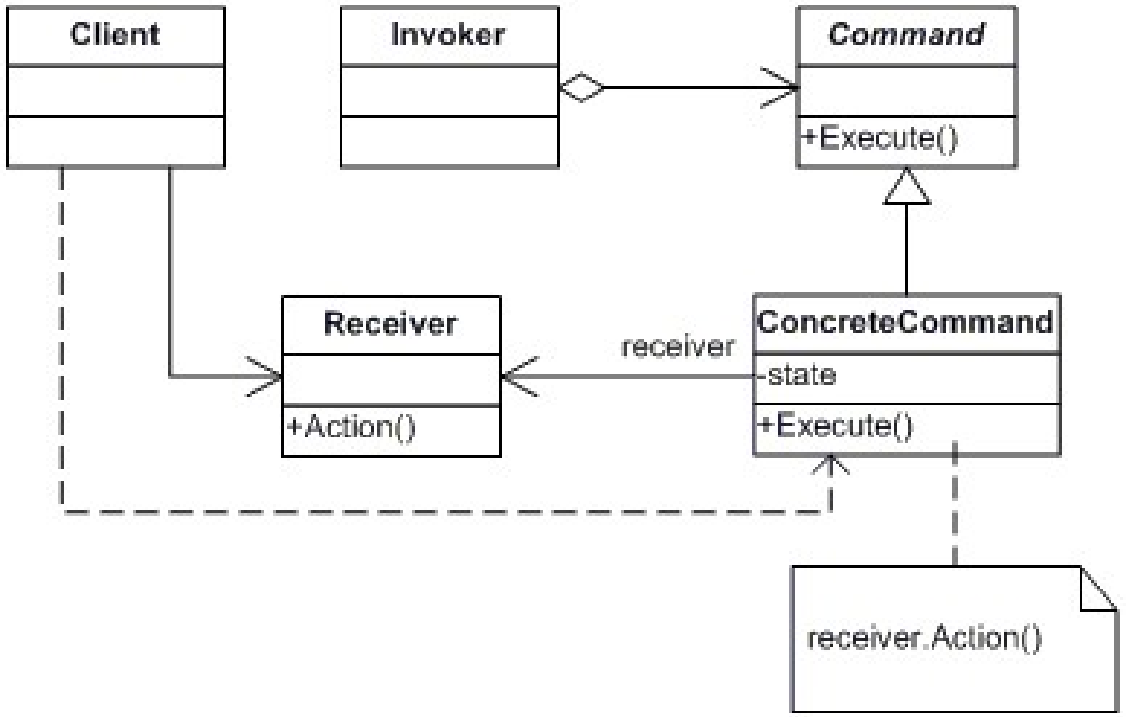
Facilita la parametrización de clientes con diferentes peticiones, el encolado de las mismas y su reordenación.

Permite implementar el hacer/deshacer (do/undo) mediante métodos.

- *Presenta una forma sencilla de implementar un sistema basado en comandos facilitando su uso y ampliación.*
- *Se independiza la invocación de los comandos de la implementación de los mismos.*
- *Los comandos se tratan como objetos, entonces se puede realizar herencia o composiciones de comandos (Composite).*

COMMAND (Comportamiento)

Estructura



Receiver: es el receptor que conoce cómo ejecutar un comando en particular.

Client: responsable de crear un objeto **ConcreteCommand** y asignarle un **Receiver**

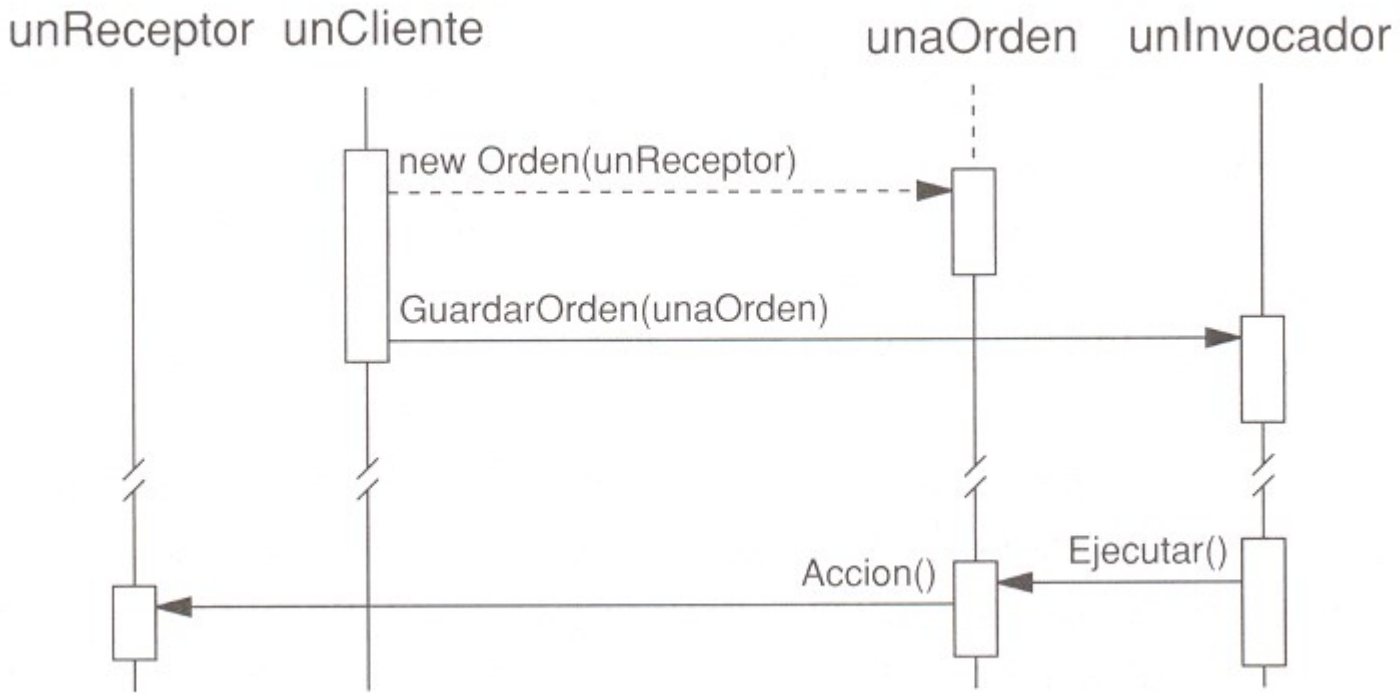
Command es la clase abstracta que declara una interfaz para ejecutar una operación (Execute())

ConcreteCommand: define un enlace entre un objeto receptor (Receiver) y una acción (Command)

Invoker: Responsable de solicitar a Command que ejecute la petición. Este objeto crea el comando concreto (ConcreteCommand) e invoca a Execute()

COMMAND (Comportamiento)

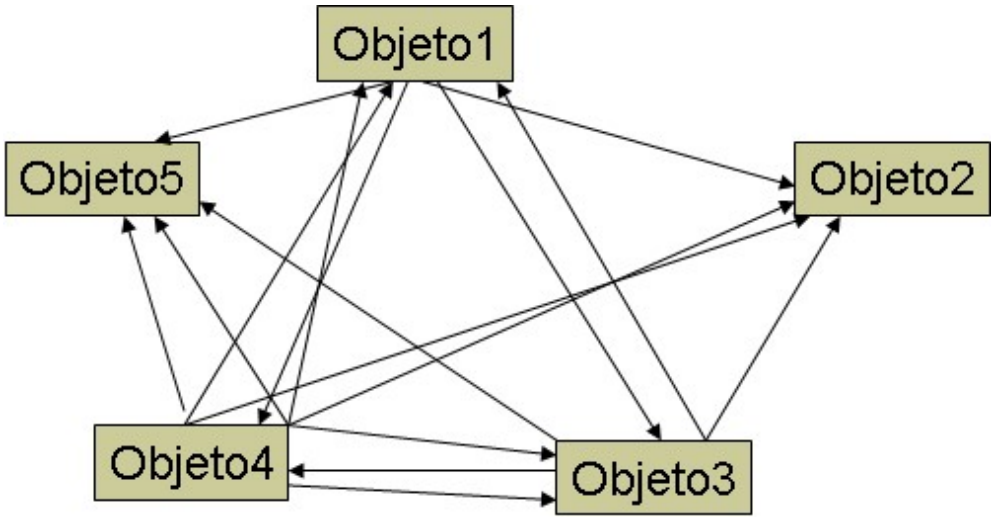
El siguiente diagrama de secuencia ayuda a visualizar y comprender la colaboración entre los objetos. (Orden refiere a Command)



Nombre: **MEDIATOR** (Comportamiento)

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.

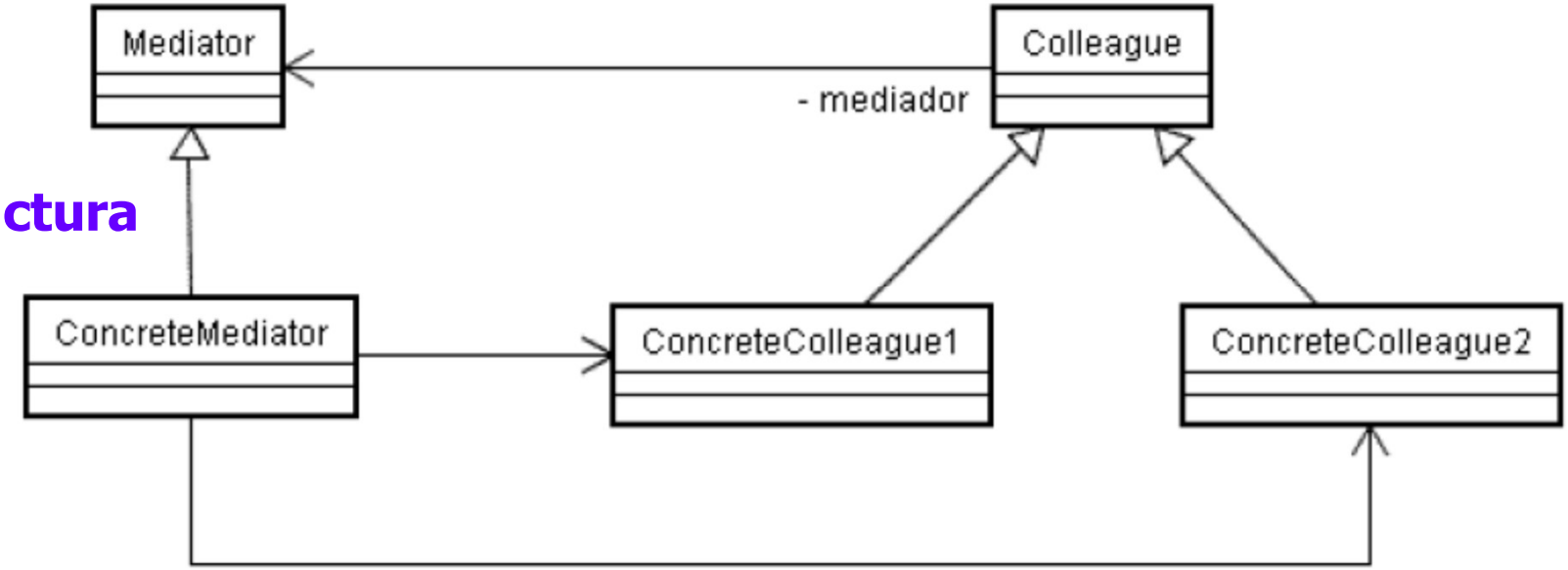
Cuando muchos objetos interactúan con otros, se puede formar una estructura muy compleja, con demasiadas conexiones. En un caso extremo cada objeto puede conocer a todos los demás.



Nombre: **MEDIATOR** (Comportamiento)

- Encapsula el comportamiento de todo un conjunto de objetos en un solo objeto.

Estructura



- **Mediator**: define una interfaz para comunicarse con los objetos colegas.
- **ConcreteMediator** : Los objetos envían y reciben peticiones a través del **mediador**, el cual implementa el comportamiento cooperativo encaminando esas peticiones a los objetos apropiados.
- **Colleagues (colegas)**: Cada colega conoce su mediador, y usa a este para comunicarse con otros colegas.

MEDIATOR - Ejemplo

Ejemplo

Modelar las rutas aéreas en un país.

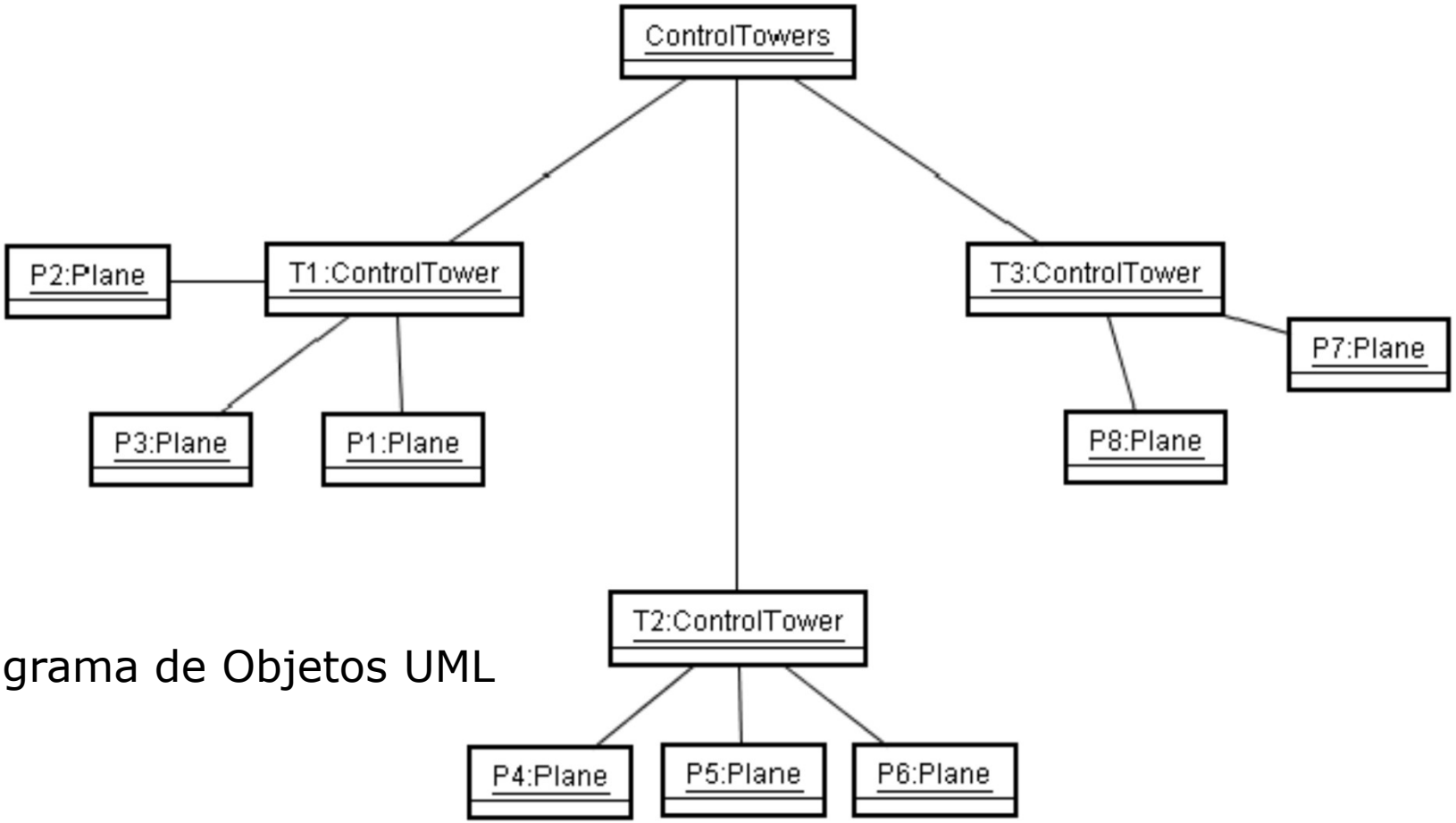


Diagrama de Objetos UML

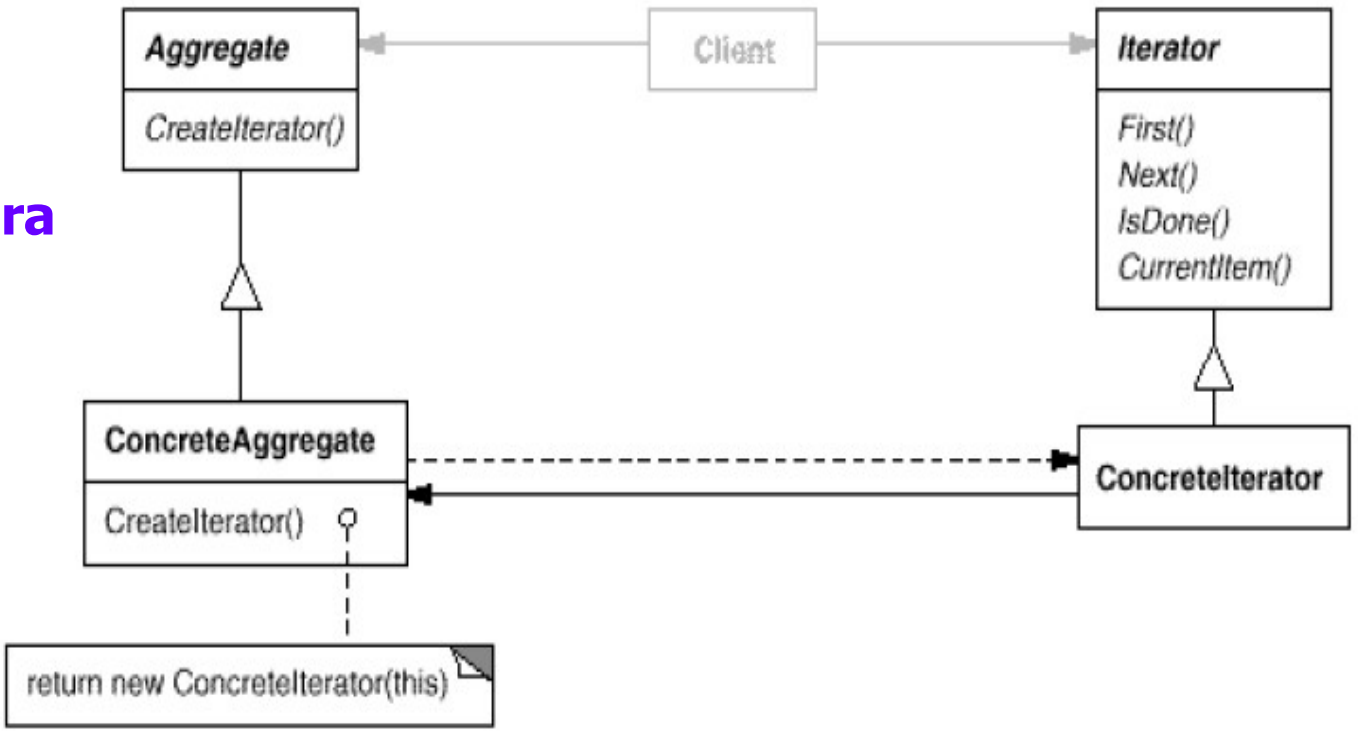
Nombre: **ITERATOR** (Comportamiento)

Provee un camino para acceder a los elementos de una colección secuencial de objetos sin exponer la representación interna de dicho recorrido.

- Además, puede ser necesario hacer recorridos de diferentes formas sobre una misma estructura.
- Para proveer una interface única que permita recorrer diferentes estructuras de datos.

ITERATOR (Comportamiento)

Estructura



Iterator: define una interface de acceso y recorrido de los elementos de una lista.

ConcreteIterator: implementa la interface Iterator y mantiene o guarda el recorrido del ítem corriente

Aggregate: (colección) define una interface que permite crear objetos de tipo Iterator.

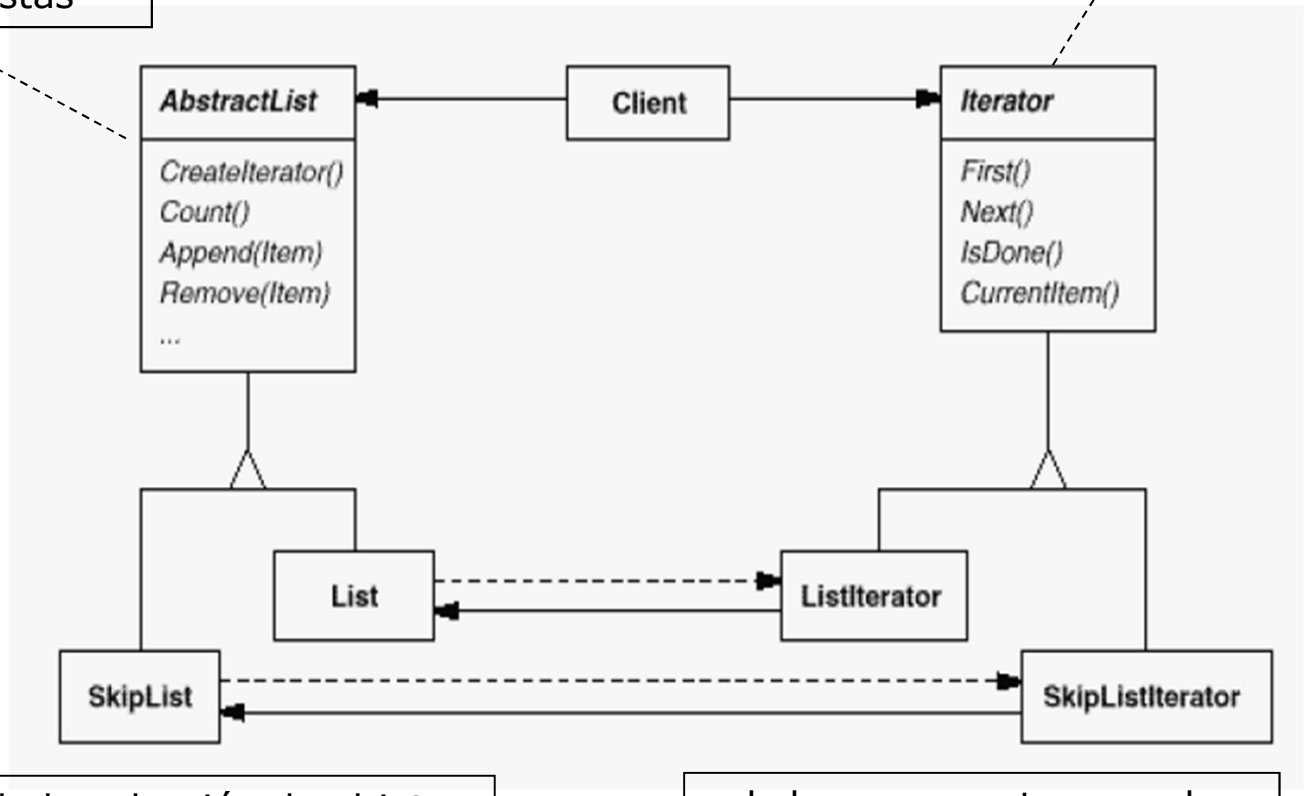
ConcreteAggregate: implementa la interface de creación de objetos Iterator y retorna una instancia apropiada de la clase ConcreteIterator

ITERATOR (Comportamiento)

Ejemplo: Es necesario recorrer listas secuencialmente, y además, de una manera especial llamada SkipList. Estas son estructuras de datos probabilísticas con características similares a los árboles balanceados.

Interface común para la manipulación de listas

interface común de iteración



tipo concreto de la colección de objetos

subclases concretas para las diferentes implementaciones

Bibliografía

✓ Erich Gamma, Richard Helm, Ralph Johnson, John Vlissedes. Design Patterns, Elements of Reusable Object/Oriented Software. Addison-Wesley, 1995.

✓ Mark Grand. Patterns in Java, Volume 1. A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition, John Wiley & Sons © 2002.

✓ Laurent Debrauwer. Patrones de Diseño en Java. 2013.

✓ Recomendados para ejemplos en UML e implementaciones en JAVA:

https://www.tutorialspoint.com/design_pattern

https://sourcemaking.com/design_patterns

✓Pressman, Roger S.. Software Engineering. A Practitioner´s Approach. 8th edition. McGraw Hill. 2015. Chapter 12.