Patrones de Diseño

Ingeniería del Software I

Carlos Blanco
Universidad de Cantabria

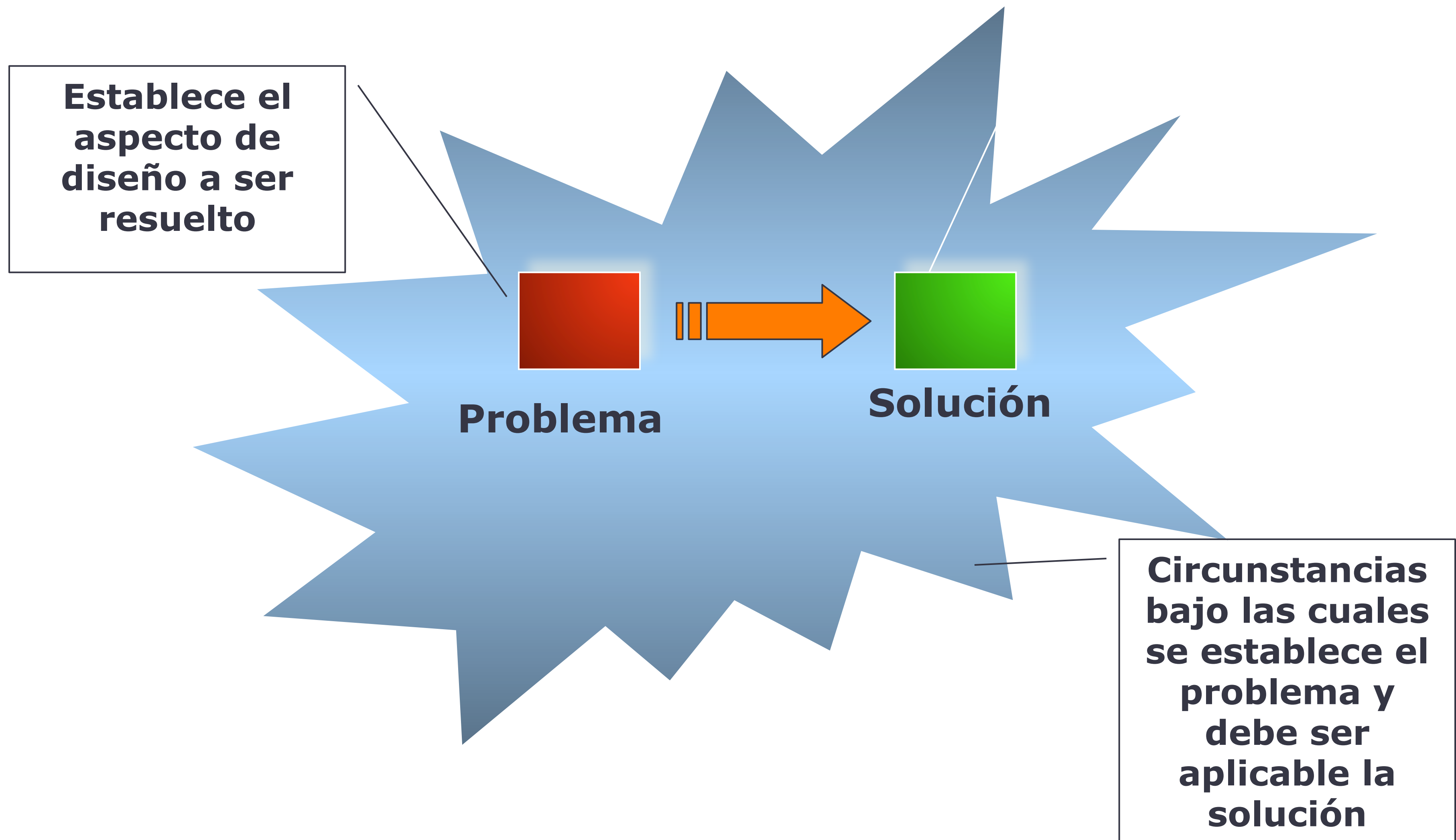
Índice

- Introducción
- Clasificación
- Patrones Creacionales
- Patrones Estructurales
- Patrones de Comportamiento
- Antipatrones

Introducción

- Un patrón es una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en algún campo
- Propuestos por el Gang of Four (Gamma, Helm, Johnson y Vlissides).
  - *Design Patterns. Elements of Reusable Object-Oriented Software* - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - Addison Wesley (GoF- Gang of Four). Addison Wesley. 1994.
- Son un esqueleto básico que cada diseñador adapta a las peculiaridades de su aplicación.

Introducción



Introducción

- Elementos de un patrón de diseño
  - Nombre:
    - Para describir el problema y la solución en una o dos palabras.
    - Un buen nombre facilita la comunicación entre los desarrolladores
  - Problema:
    - Especifica el problema y su contexto
    - Lista de condiciones para que pueda aplicarse el patrón

Introducción

- Elementos de un patrón de diseño
  - Solución:
    - Describe que componen el diseño sus relaciones, responsabilidades, y colaboraciones.
    - No describe una implementación en particular
  - Consecuencias:
    - Son los resultados de aplicar el patrón
    - Son criticas para evaluar las alternativas
    - Entender los costos y beneficios de aplicar el patrón

Clasificación

- Según el Propósito: Que hace el Patrón?
  - Creacional: creación de objetos
  - Estructural: composición de clases y objetos
  - de Comportamiento: interacción entre objetos
- Según el Ámbito: Clases u Objetos?
  - De Clases
  - De Objetos

Clasificación

		Ámbito	
		Clase	Objeto
Propósito	Creacional	Método Fábrica	Fábrica Abstracta, Constructor, Prototipo, Singleton
	Estructural		Adaptador, Puente, Compuesto, Decorador, Fachada, Peso Mosca, Apoderado
	De Compor-tamiento	Intérprete, Método Plantilla	Cadena de Responsabilidad, Comando, Iterador, Mediador, Memento Observador, Estado Estrategia, Visitante

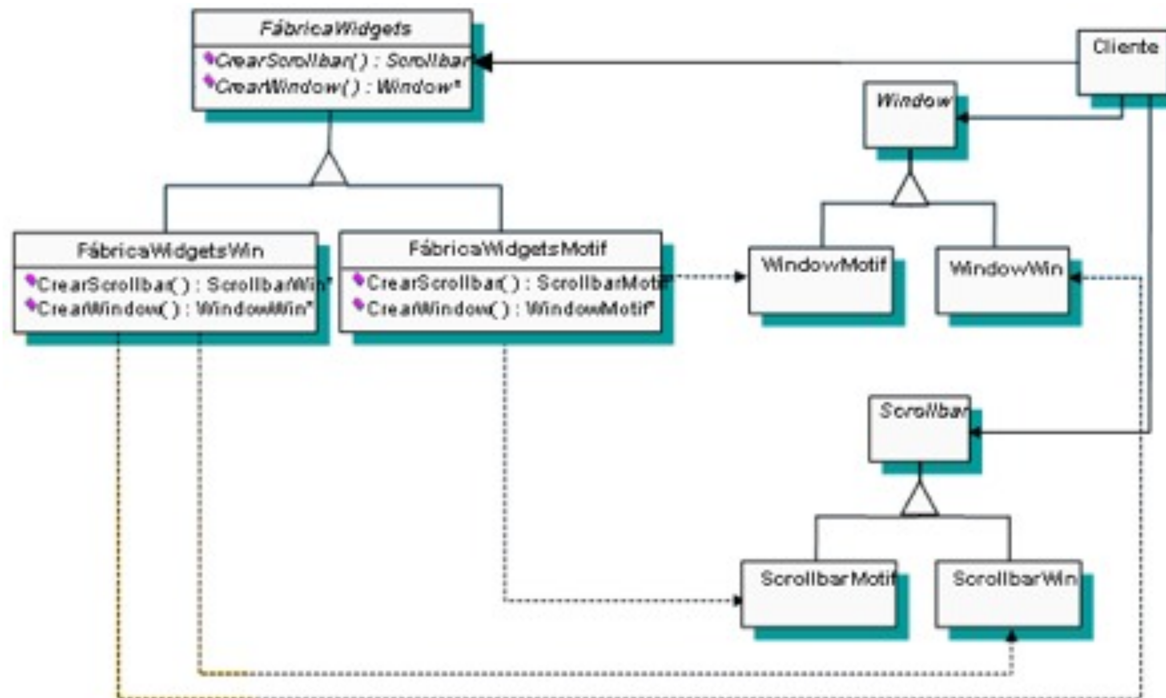
Fábrica Abstracta

ABSTRACT FACTORY

- Intención:
  - Proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas
- También conocido como KIT
- Motivación:
  - Crear aplicaciones independientes del sistema de interfaz de usuario

Fábrica Abstracta

ABSTRACT FACTORY



Fábrica Abstracta

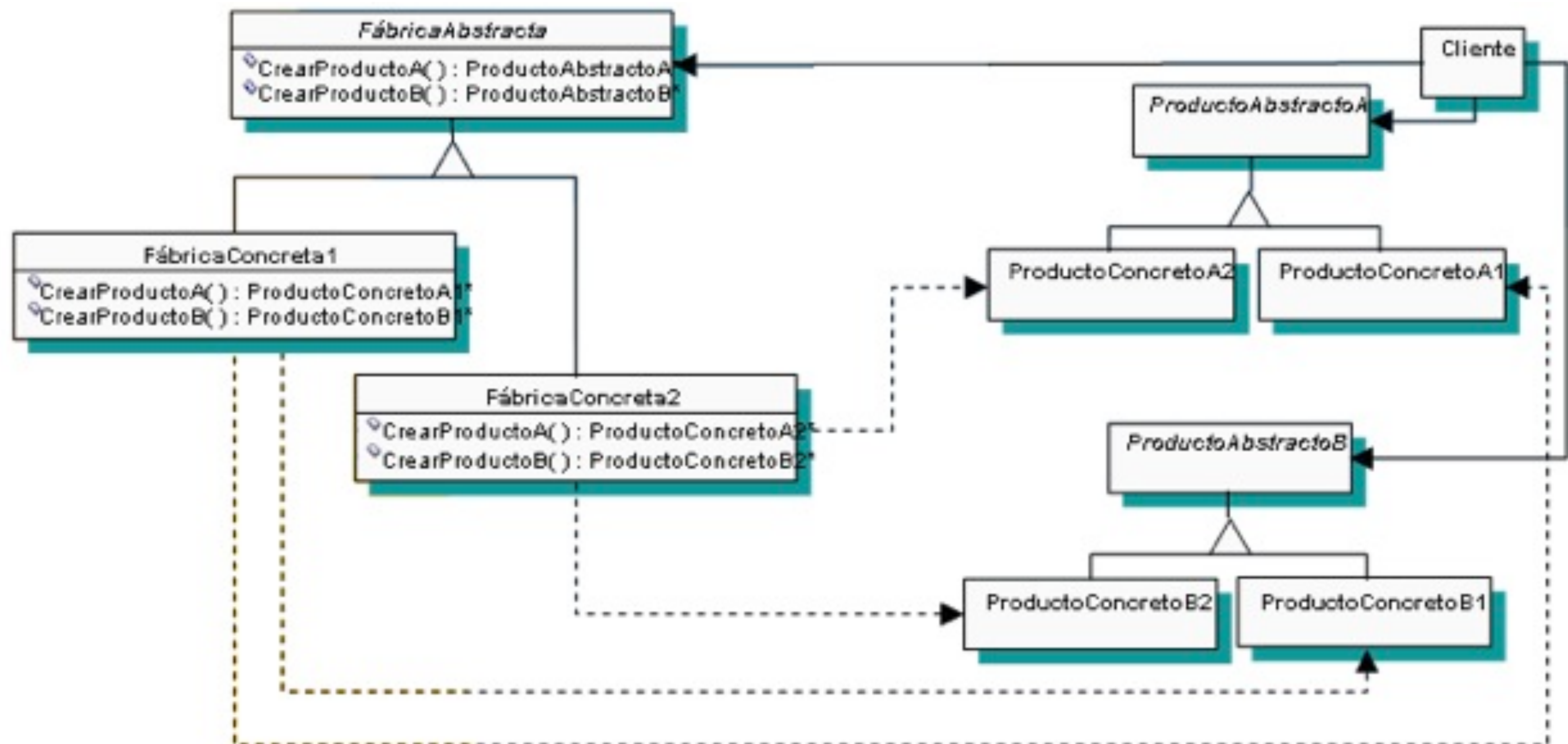
ABSTRACT FACTORY

- Aplicabilidad: utilizar cuando...
  - Un sistema deba ser independiente de cómo se crean, componen o representan sus productos
  - Un sistema deba ser configurado con una de múltiples familias de productos
  - Una familia de productos se diseña para usarse a la vez y se quiere reforzar esa restricción

Fábrica Abstracta

ABSTRACT FACTORY

- Estructura



Fábrica Abstracta

ABSTRACT FACTORY

- Participantes
  - FábricaAbstracta:
    - declara una interfaz para operaciones que crean productos abstractos
  - FábricaConcreta:
    - implementa las operaciones para crear los productos concretos
  - ProductoAbstracto:
    - declara una interfaz para un tipo de producto
  - ProductoConcreto:
    - define un producto para que pueda ser creado por la correspondiente fábrica concreta
  - Cliente:
    - utiliza sólo las interfaces declaradas por FábricaAbstracta y ProductoAbstracto

Fábrica Abstracta

ABSTRACT FACTORY

- Ventajas e Inconvenientes
  - Aisla los clientes de las implementaciones, ya que sólo usan la interfaz. Las dependencias se encapsulan en las fábricas concretas.
  - Facilita el cambio de familias de productos, ya que la clase concreta sólo aparece una vez en el código de la aplicación y es fácil cambiarla.
  - Favorece la consistencia entre productos, ya que favorece que sólo se use una familia de productos a la vez.
  - Es difícil soportar nuevos productos, ya que afecta a la interfaz de la fábrica abstracta y por lo tanto a la de todas sus subclases.

Constructor

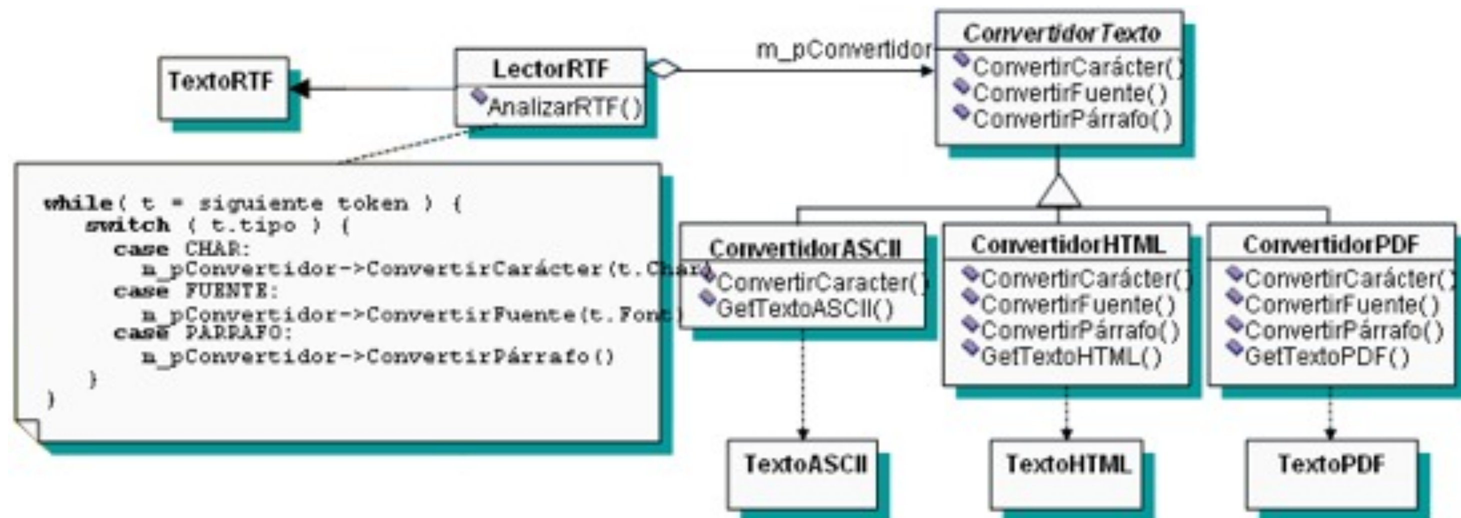
BUILDER

- Intención:
  - Construir de manera flexible un objeto complejo a partir de otro objeto complejo en una serie de pasos

- También conocido como Builder

Constructor

BUILDER



Constructor

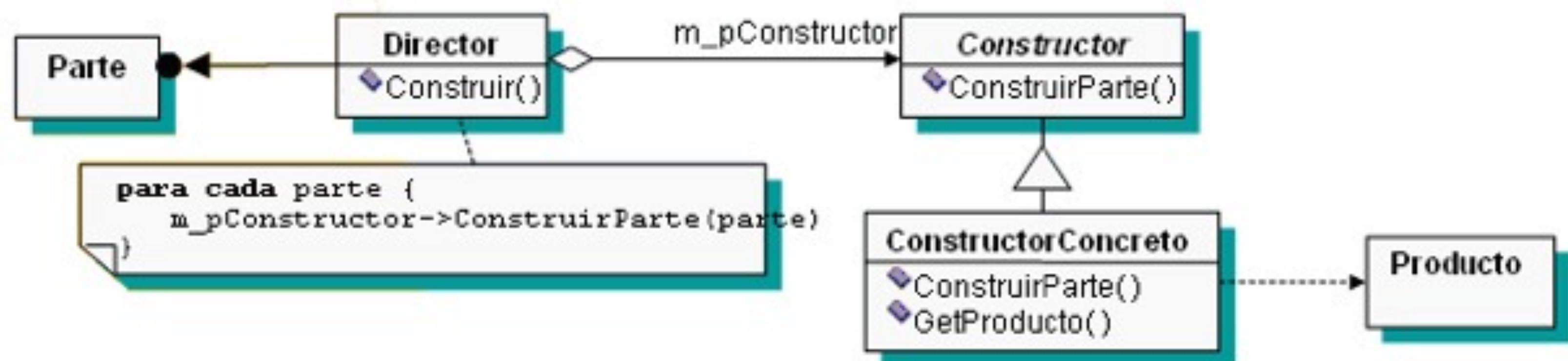
BUILDER

- Aplicabilidad: utilizar cuando...
  - El algoritmo para crear un objeto complejo deba ser independiente de las partes que componen el objeto y de cómo se ensamblan
  - El proceso de construcción deba permitir distintas representaciones para el objeto que es construido

Constructor

BUILDER

- Estructura



Constructor

BUILDER

- Participantes
  - Constructor:
    - especifica una interfaz abstracta para crear un objeto complejo Producto
  - ConstructorConcreto:
    - construye y ensambla las partes del Producto implementando la interfaz de Constructor
  - Director:
    - obtiene las partes que forman el Producto y lo construye usando la interfaz de Constructor
  - Parte:
    - representa las partes que se usan para construir el Producto
  - Producto:
    - representa el objeto complejo en construcción

Constructor

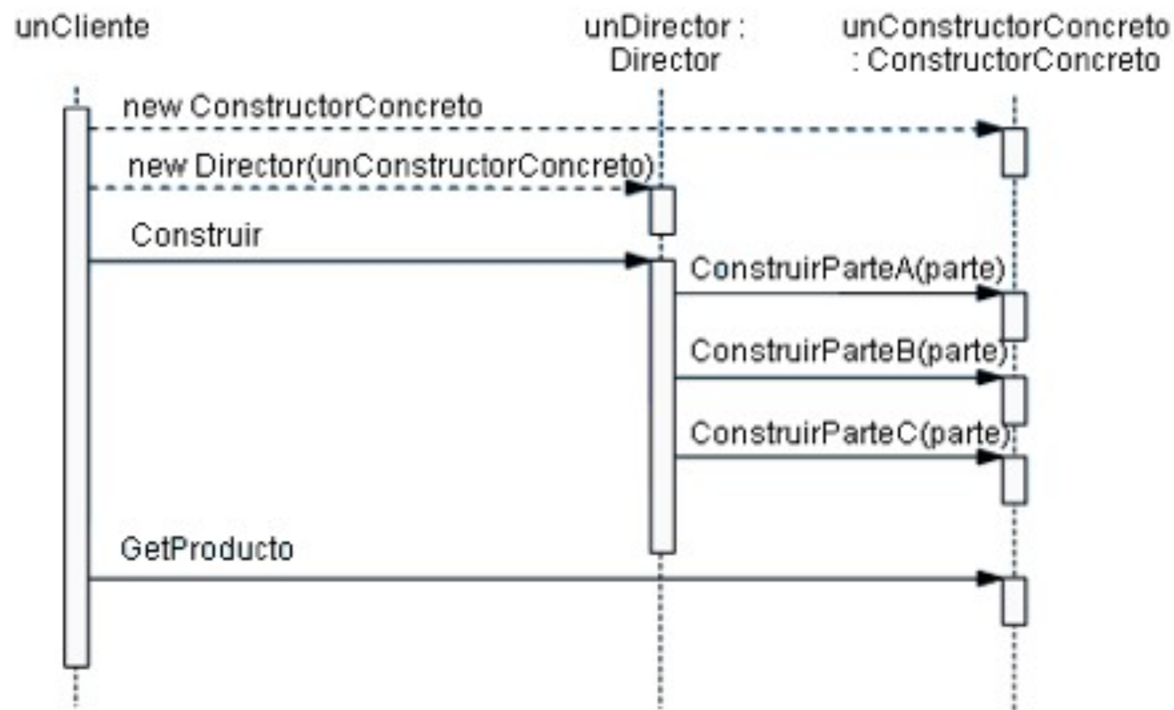
BUILDER

- Colaboraciones
  - El cliente crea el objeto director y lo configura con el objeto constructor deseado
  - El director avisa al constructor cuando una parte del producto tiene que ser construida
  - El constructor gestiona las peticiones del director y añade partes al producto

Constructor

BUILDER

- Colaboraciones
  - El cliente obtiene el producto del constructor



Constructor

BUILDER

- Ventajas e Inconvenientes
  - Desacopla al director del constructor, permitiendo nuevas representaciones de objetos cambiando el constructor
  - Aisla a los clientes de la estructura interna del producto
  - Permite controlar la construcción paso a paso de un objeto complejo

Singleton

SINGLETON

- Intención:
  - Asegurarse de que una clase tiene una sola instancia y proporcionar un punto de acceso global a ella
- También conocido como de Instancia Única
- Motivación
  - Algunas clases sólo pueden tener una instancia
  - Una variable global no garantiza que sólo se instancie una vez

Singleton

SINGLETON

- Aplicabilidad: utilizar cuando...
  - Deba haber exactamente una instancia de una clase y deba ser accesible a los clientes desde un punto de acceso conocido
- Estructura

Singleton

SINGLETON

- Participantes
  - Singleton:
    - define una operación GetInstancia que permite a los clientes acceder a su instancia única y además es responsable de su creación de Constructor
- Colaboraciones
  - Los clientes acceden a la única instancia solamente a través de la operación GetInstancia

Singleton

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 *templates*

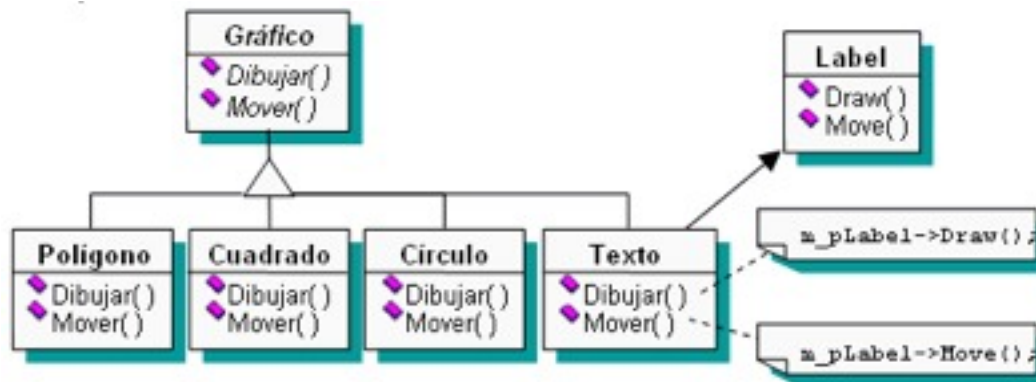
Adaptador

ADAPTER

- Intención:
  - Convertir la interfaz de una clase en otra interfaz esperada por los clientes. Permite que clases con interfaces incompatibles puedan comunicarse.
- También conocido como Wrapper
- Motivación
  - Extender un editor gráfico con clases de otro toolkit

Adaptador

ADAPTER



Adaptador

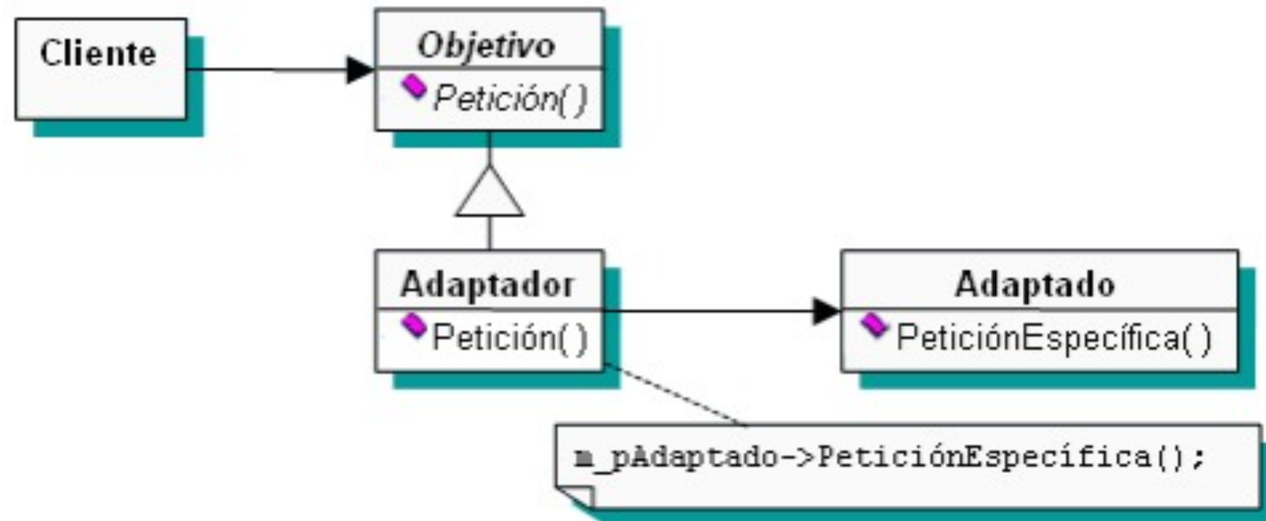
ADAPTER

- Aplicabilidad: utilizar cuando...
  - Se quiera utilizar una clase ya existente y su interfaz no se corresponda con la interfaz que se necesita
  - Se quiera envolver código no orientado a objeto con forma de clase

Adaptador

ADAPTER

- Estructura



Adaptador

ADAPTER

- Participantes
  - Objetivo:
    - define la interfaz específica que usa el cliente
  - Cliente:
    - colabora con objetos que respetan la interfaz de Objetivo
  - Adaptado:
    - define la interfaz existente que necesita ser adaptada
  - Adaptador:
    - adapta la interfaz de Adaptado a la interfaz de

Adaptador

ADAPTER

- Colaboraciones
  - Los clientes envían sus peticiones a una instancia de Adaptador que la reenvía a otra de Adaptado
- Ventajas e inconvenientes
  - Una misma clase adaptadora puede adaptar a una clase adaptada y a todas sus subclases
  - La adaptación no siempre consiste en cambiar el nombre y/o los parámetros de las operaciones, puede ser más compleja

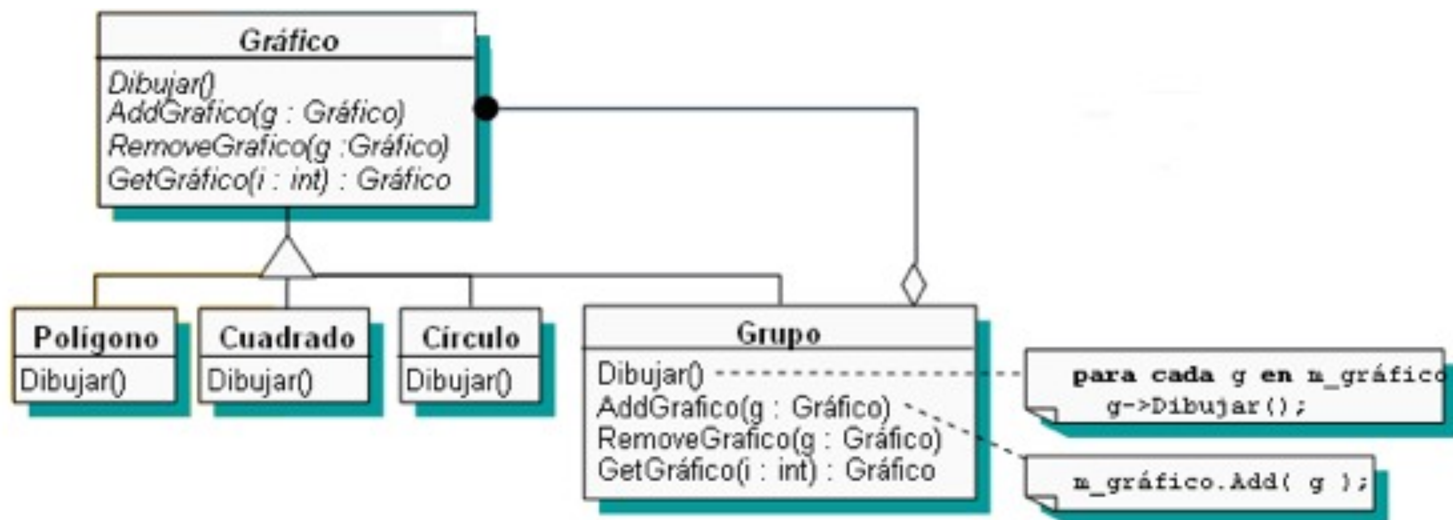
Compuesto

COMPOSITE

- Intención:
  - Componer objetos en jerarquías *todo-parte* y permitir a los clientes tratar objetos simples y compuestos de manera uniforme
- También conocido como Composite
- Motivación
  - Tratar grupos de objetos en un editor gráfico

Compuesto

COMPOSITE



Compuesto

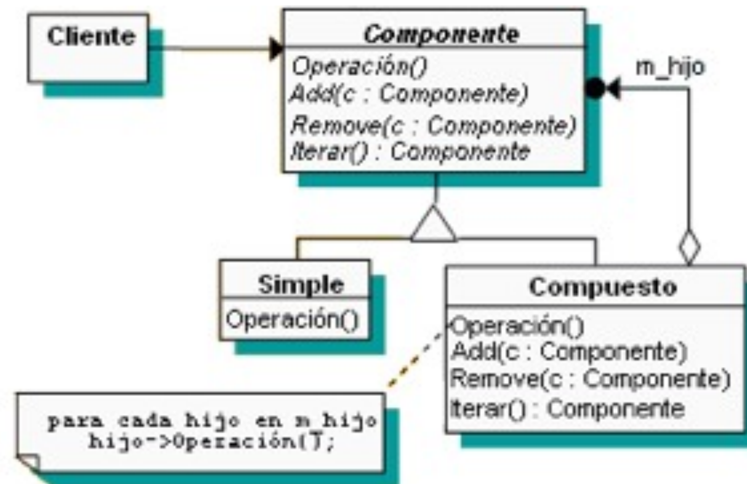
COMPOSITE

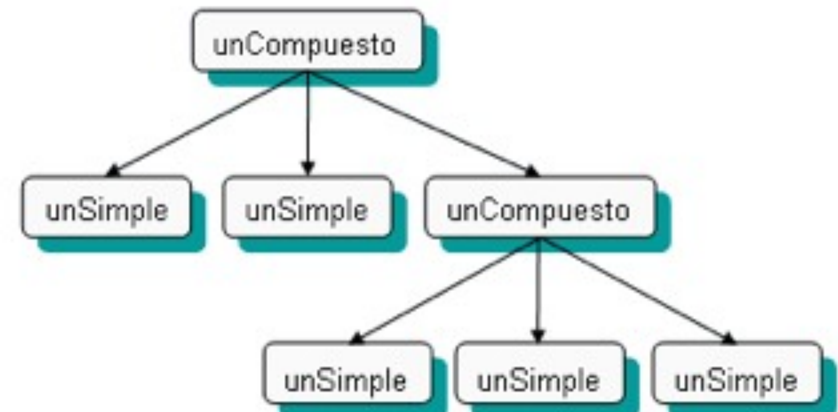
- Aplicabilidad: utilizar cuando...
  - Se quiera representar jerarquías de objetos *todo-parte*
  - Se quiera que los clientes puedan ignorar la diferencia entre objetos simples y compuestos y tratarlos uniformemente

Compuesto

COMPOSITE

- Estructura





Compuesto

COMPOSITE

- Participantes
  - Componente:
    - Declara la interfaz para la composición de objetos, implementa acciones por defecto cuando es oportuno y, opcionalmente, accede al padre en la jerarquía
  - 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

Compuesto

COMPOSITE

- Ventajas e inconvenientes
  - Permite tratamiento uniforme de objetos simples y complejos así como composiciones recursivas
  - Simplifica el código de los clientes, que sólo usan una interfaz

Compuesto

COMPOSITE

- Ventajas e Inconvenientes
  - Facilita añadir nuevos componentes sin afectar a los clientes
  - Es difícil restringir los tipos de hijos

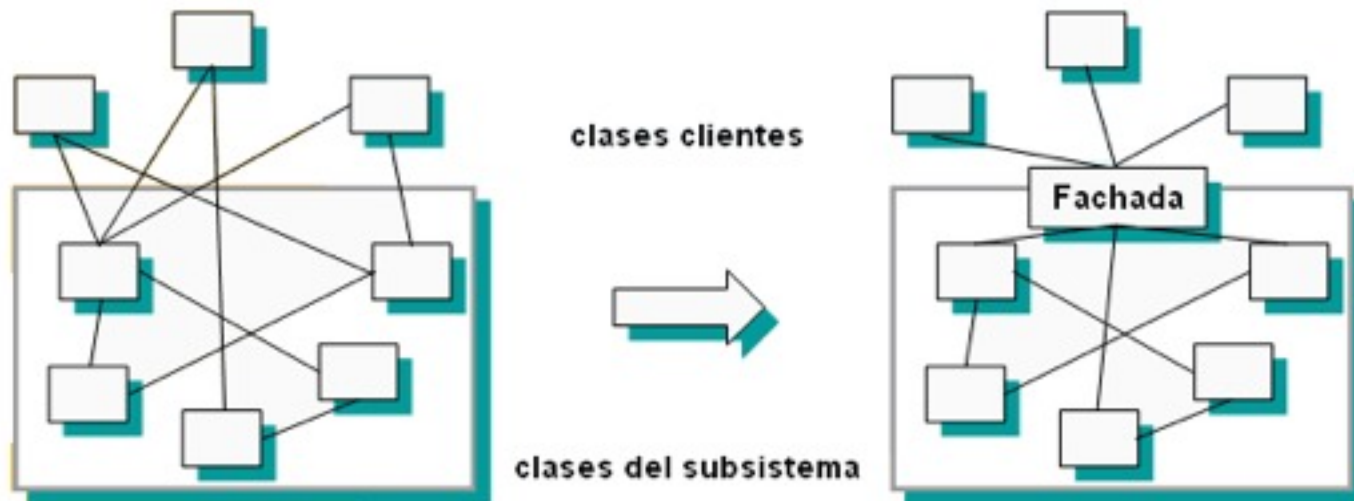
Fachada

FACADE

- Intención:
  - Proporcionar una interfaz unificada para un conjunto de interfaces en un subsistema, haciéndolo más fácil de usar.
- Motivación
  - Reducir la complejidad y minimizar dependencias

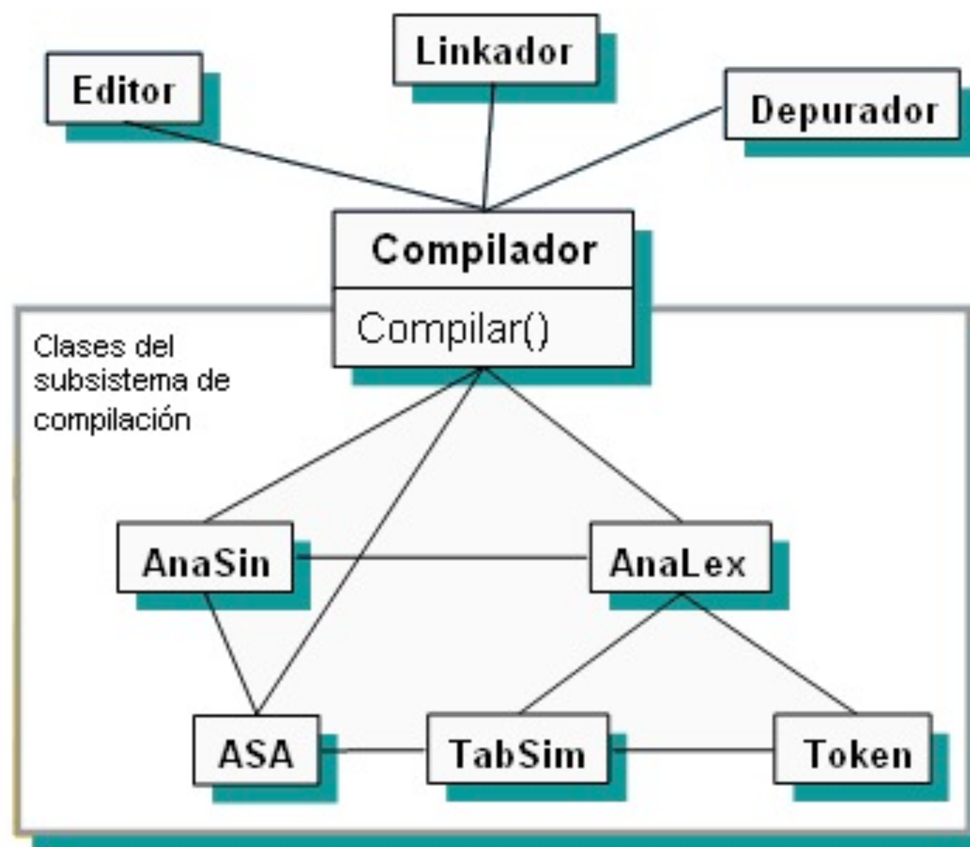
Fachada

FACADE



Fachada

FACADE



Fachada

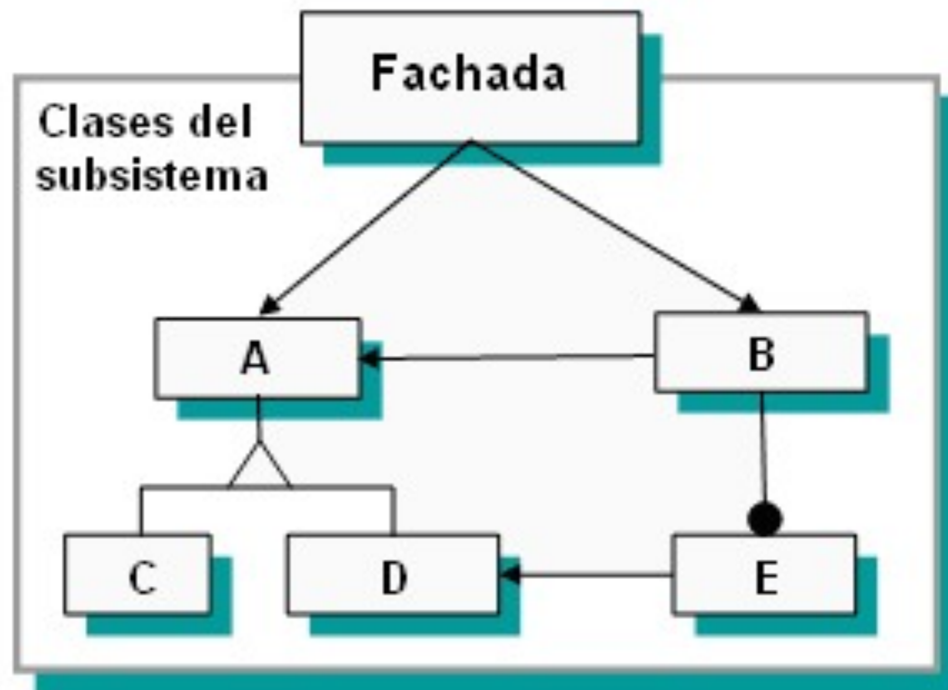
FACADE

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

Fachada

FACADE

- Estructura



Fachada

FACADE

- Participantes
  - Fachada:
    - conoce las clases del subsistema y delega las peticiones de los clientes en los objetos del subsistema
  - Clases del subsistema:
    - implementan la funcionalidad del subsistema y llevan a cabo las peticiones que les envía la fachada, aunque no la conocen

Fachada

FACADE

- Colaboraciones
  - Los clientes se comunican con el subsistema a través de la fachada, que reenvía las peticiones a los objetos del subsistema apropiados y puede realizar también algún trabajo de traducción
  - Los clientes que usan la fachada no necesitan acceder directamente a los objetos del sistema

Fachada

FACADE

- Ventajas e inconvenientes
  - Oculta a los clientes de la complejidad del subsistema y lo hace más fácil de usar
  - Favorece un acoplamiento débil entre el subsistema y sus clientes, ayuda a dividir un sistema en capas y reduce dependencias de compilación
  - No evita que se usen las clases del sistema si es necesario, dejando la puerta del subsistema abierta por si es necesario

Proxy

PROXY

- Intención:
  - Proporcionar un representante para controlar el acceso a un objeto .
- También conocido como:
  - *Surrogate*, Embajador, Representante, *Smart Pointer*

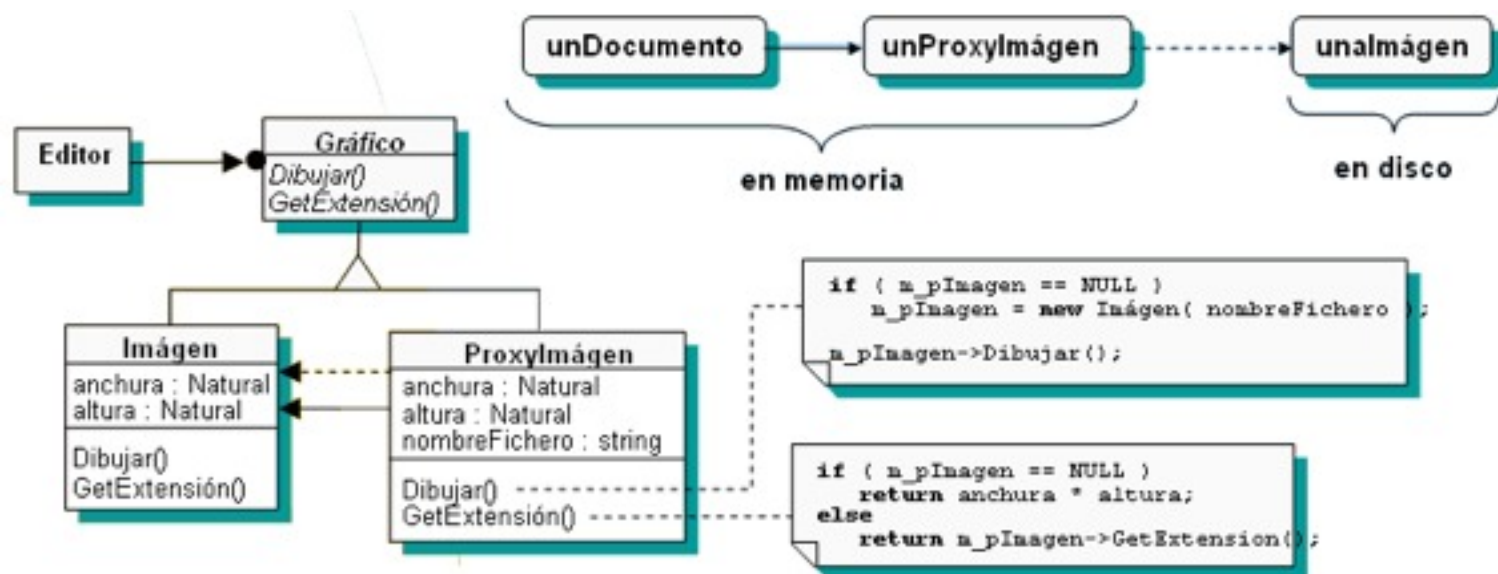
Proxy

PROXY

- Motivación
  - Procesador de texto capaz de incluir gráficos. Necesita conocer tamaño y posición de los gráficos pero quiere cargarlos bajo demanda, cuando tenga que visualizarlos en pantalla
  - El proxy, que conoce el tamaño y la posición de la imagen, responde a todas las peticiones que pueda hasta que no le queda más remedio que cargar la imagen desde el disco
  - Una vez que ha creado la imagen ya no *da la cara* por ella y le reenvía las peticiones directamente

Proxy

PROXY



Proxy

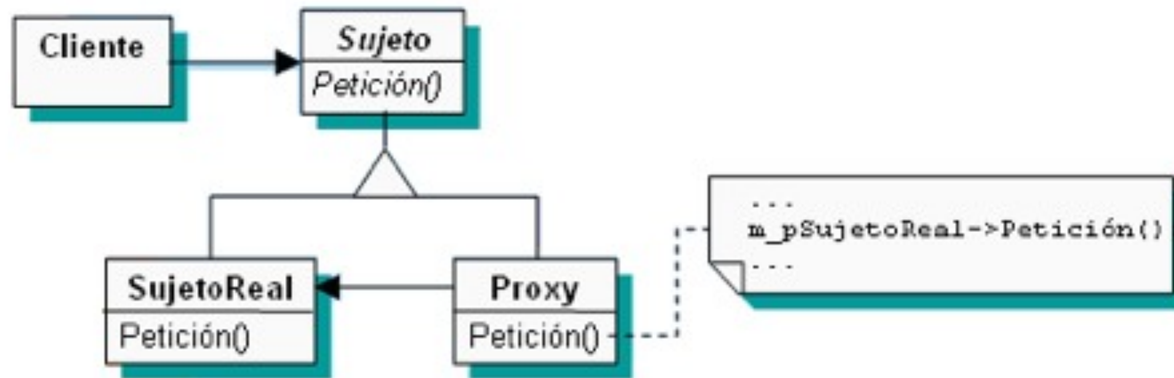
PROXY

- Aplicabilidad: utilizar cuando...
  - Se necesite una forma más versátil o sofisticada de referencia a un objeto que un simple puntero. Hay varios tipos de proxy:
    - **Proxy remoto**: es un representante local para un objeto de otro proceso, posiblemente en otra máquina. También se le denomina *embajador*.
    - **Proxy virtual**: crea objetos costosos bajo demanda.
    - **Proxy de protección**: controla el acceso al objeto original cuando hay distintos niveles de acceso.

Proxy

PROXY

- Estructura



Proxy

PROXY

- Participantes
  - Sujeto:
    - define la interfaz común para SujetoReal y Proxy
  - Sujeto real:
    - define el objeto real al que Proxy representa
  - Proxy
    - representa a un objeto real y además...
      - mantiene una referencia para acceder al sujeto real, controla el acceso y puede ser responsable de crearlo y destruirlo
      - tiene una interfaz idéntica a la de Sujeto para que ambos sean intercambiables

Proxy

PROXY

- los ***proxys remotos*** deben codificar las peticiones y enviárselas al sujeto real remoto
- los ***proxys virtuales*** pueden tener un caché con información sobre el sujeto real para evitar accesos innecesarios
- los ***proxys de protección*** comprueban que el cliente tenga permisos de acceso para realizar una petición

- Colaboraciones
  - El proxy reenvía las peticiones de los clientes al sujeto real cuando es conveniente, dependiendo del tipo de proxy

Proxy

PROXY

- Ventajas e Inconvenientes
  - El proxy remoto oculta espacios de direcciones diferentes
  - El proxy virtual evita consumir recursos hasta que no es necesario
  - El proxy de protección y los punteros inteligentes permiten realizar tareas internas (recolección de basura, controlar accesos, etc.)

Método Plantilla

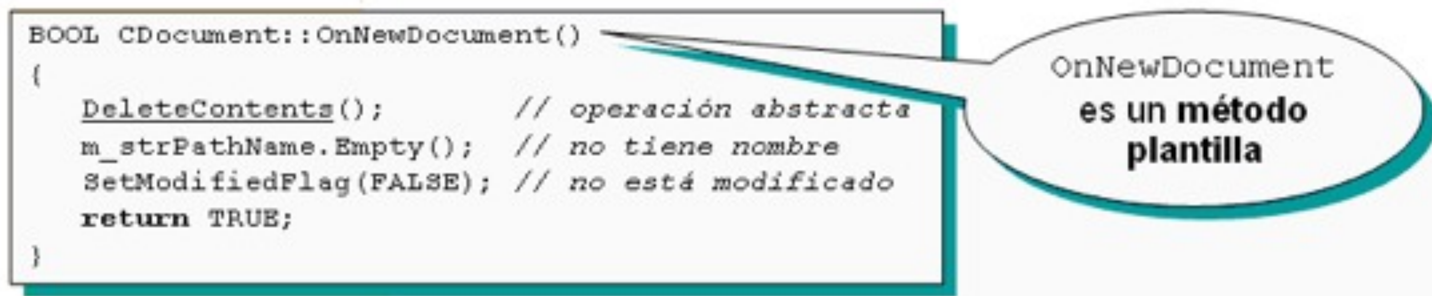
TEMPLATE METHOD

- Intención:
  - Definir el esqueleto de un algoritmo en una operación delegando algunos pasos en las subclases
- También conocido como: Template

Método Plantilla

TEMPLATE METHOD

- Motivación
  - FW de Visual C++: en las aplicaciones SDI, el documento debe *limpiarse* cuando se selecciona la opción Nuevo del menú Archivo
  - La clase abstracta CDocument no sabe como limpiar todas los posibles clases de documentos, así que declara una operación abstracta DeleteContents que las subclases deben implementar



Las operaciones que incluyen en su código llamadas a operaciones abstractas que deben ser redefinidas por las subclases se denominan **métodos plantilla**

Método Plantilla

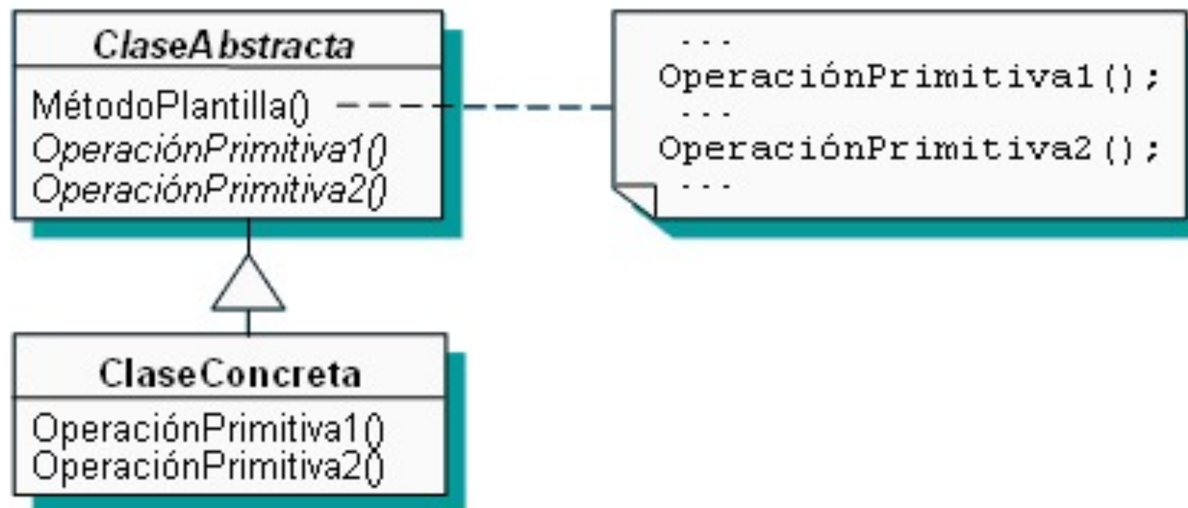
TEMPLATE METHOD

- Aplicabilidad: utilizar cuando...
  - Implementar las partes fijas de un algoritmo una sola vez y dejar que las subclases implementen las partes variables
  - Sacar factor común la conducta similar entre varias subclases: la parte común en un método plantilla, la parte distinta en operaciones abstractas que deben redefinirse
  - Controlar las ampliaciones que puedan hacer las subclases: sólo pueden cambiar las operaciones abstractas

Método Plantilla

TEMPLATE METHOD

- Estructura



Método Plantilla

TEMPLATE METHOD

- Participantes
  - ClaseAbstracta:
    - implementa un método plantilla que define el esqueleto de un algoritmo y define operaciones primitivas abstractas que implementan las subclases concretas para definir ciertos pasos del algoritmo
  - ClaseConcreta:
    - implementa las operaciones primitivas para realizar los pasos del algoritmo que son específicos de la subclase

Método Plantilla

TEMPLATE METHOD

- Colaboraciones
  - Las clases concretas confían en que la clase abstracta implemente la parte fija del algoritmo

Método Plantilla

TEMPLATE METHOD

- Ventajas e Inconvenientes
  - Favorece la reutilización del código (toolkits y FWs). Los métodos plantilla suelen llamar a los siguiente tipos de operaciones:
    - Operaciones concretas de la propia clase abstracta o de clases clientes
    - Operaciones primitivas (abstractas), como los métodos fábrica
    - **Operaciones gancho** (*hook operations*), que proporcionan una conducta por defecto (normalmente no hacer nada) que las subclases pueden ampliar si es necesario. Es importante especificar qué operaciones son ganchos (***pueden*** redefinirse) y cuales son abstractas (***deben*** redefinirse)

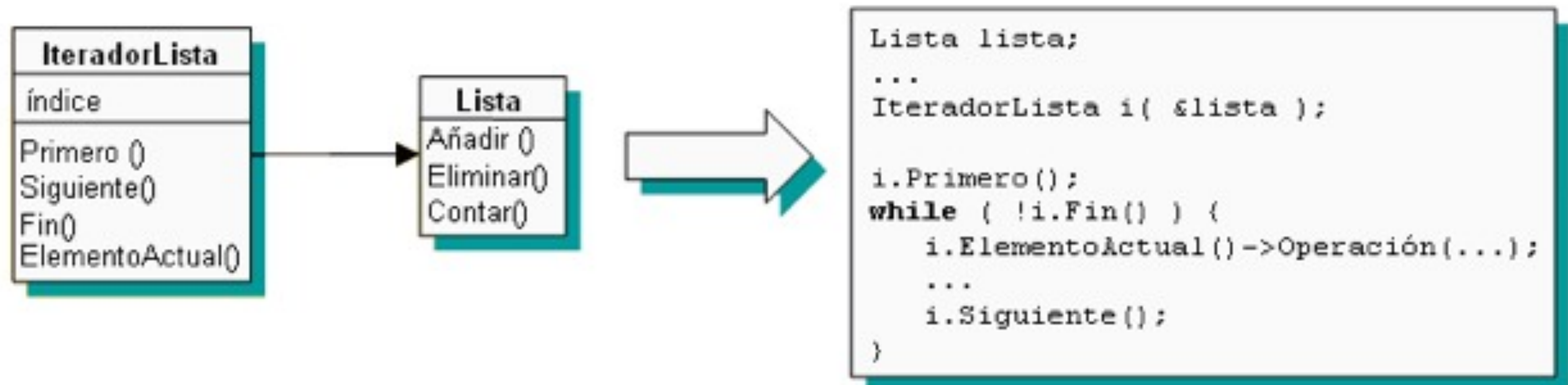
Iterador

ITERATOR

- Intención:
  - Proporcionar una forma de acceder secuencialmente a los elementos de un agregado sin revelar su representación interna
- Motivación
  - Iterar sobre los elementos de una lista sin ampliar la interfaz de la lista y permitiendo recorridos simultáneos

Iterador

ITERATOR



Iterador

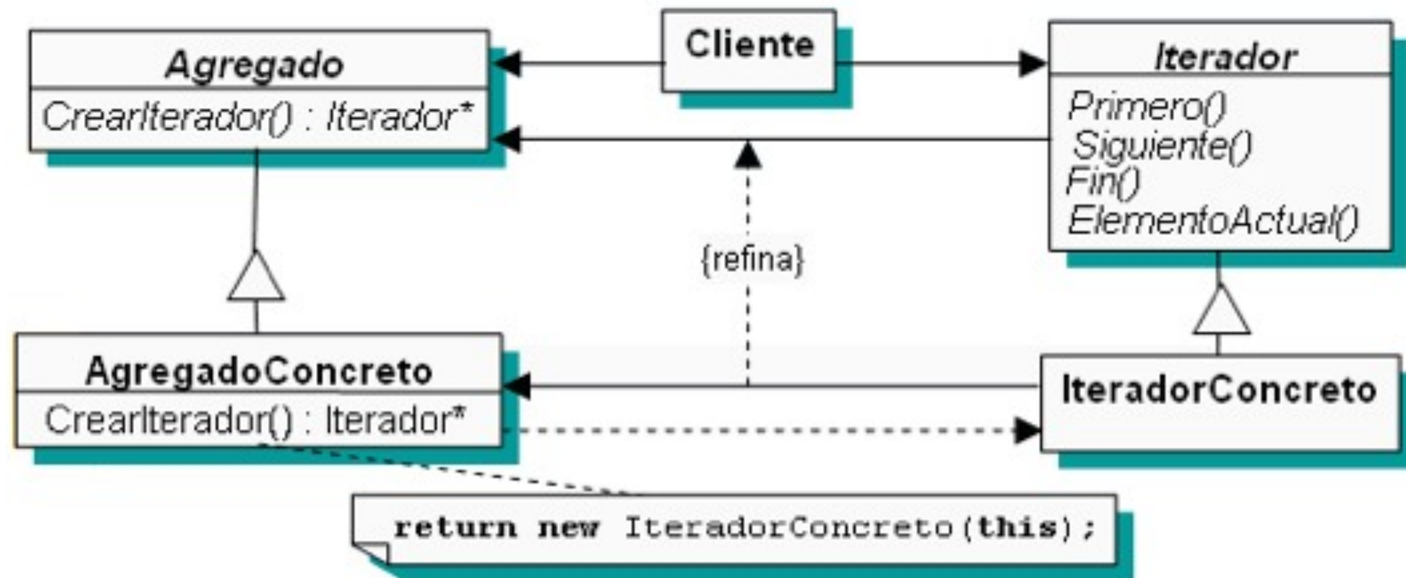
ITERATOR

- Aplicabilidad: utilizar cuando...
  - Acceder al contenido de un objeto agregado sin revelar su representación interna
  - Permitir múltiples tipos de recorridos sobre objetos agregados
  - Proporcionar una interfaz uniforme para recorrer diferentes estructuras de agregación (*iteración polimórfica*)

Iterador

ITERATOR

- Estructura



Iterador

ITERATOR

- Participantes
  - Iterador:
    - define una interfaz para acceder a los elementos del agregado y recorrerlos
  - IteradorConcreto:
    - implementa la interfaz de Iterador y mantiene la posición actual del recorrido
  - Agregado:
    - define una interfaz para crear un objeto iterador
  - AgregadoConcreto:
    - implementa la interfaz de creación del iterador para devolver una instancia apropiada de IteradorConcreto

Iterador

ITERATOR

- Colaboraciones
  - Un iterador concreto mantiene el objeto actual del recorrido y es capaz de calcular el siguiente objeto
- Ventajas e Inconvenientes
  - Permite variaciones en el recorrido del agregado encapsulando los algoritmos de recorrido en distintas subclases de iteradores
  - Simplifica la interfaz del agregado al no incluir las operaciones de iteración
  - Permite recorridos simultáneos con varios iteradores a la vez

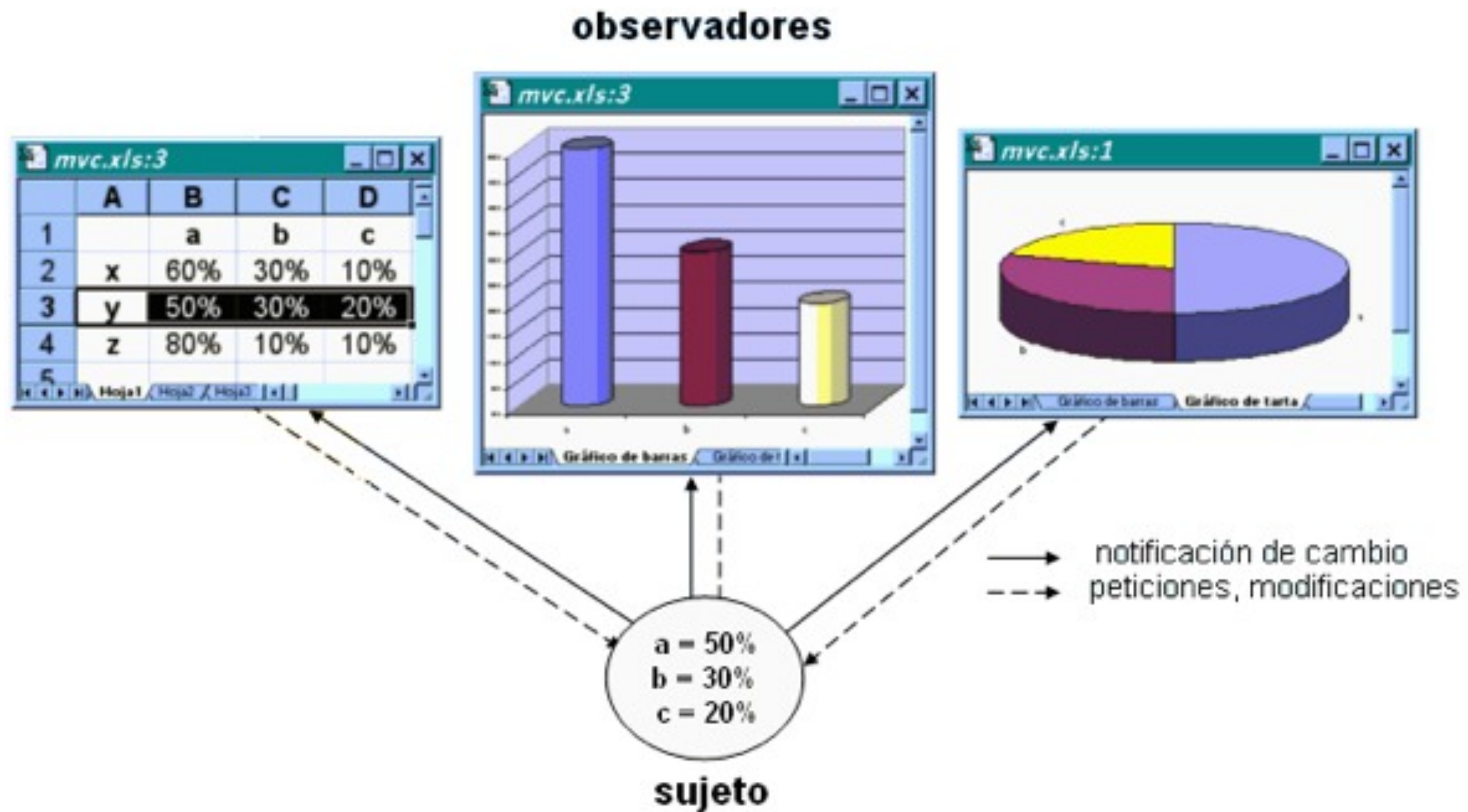
Observador

OBSERVER

- Intención:
  - Definir una dependencia *1:n* de forma que cuando el objeto *1* cambie su estado, los *n* objetos sean notificados y se actualicen automáticamente
- Motivación
  - En un toolkit de GUI, separar los objetos de presentación (vistas) de los objetos de datos, de forma que se puedan tener varias vistas sincronizadas de los mismos datos (editor-suscriptor)

Observador

OBSERVER



Observador

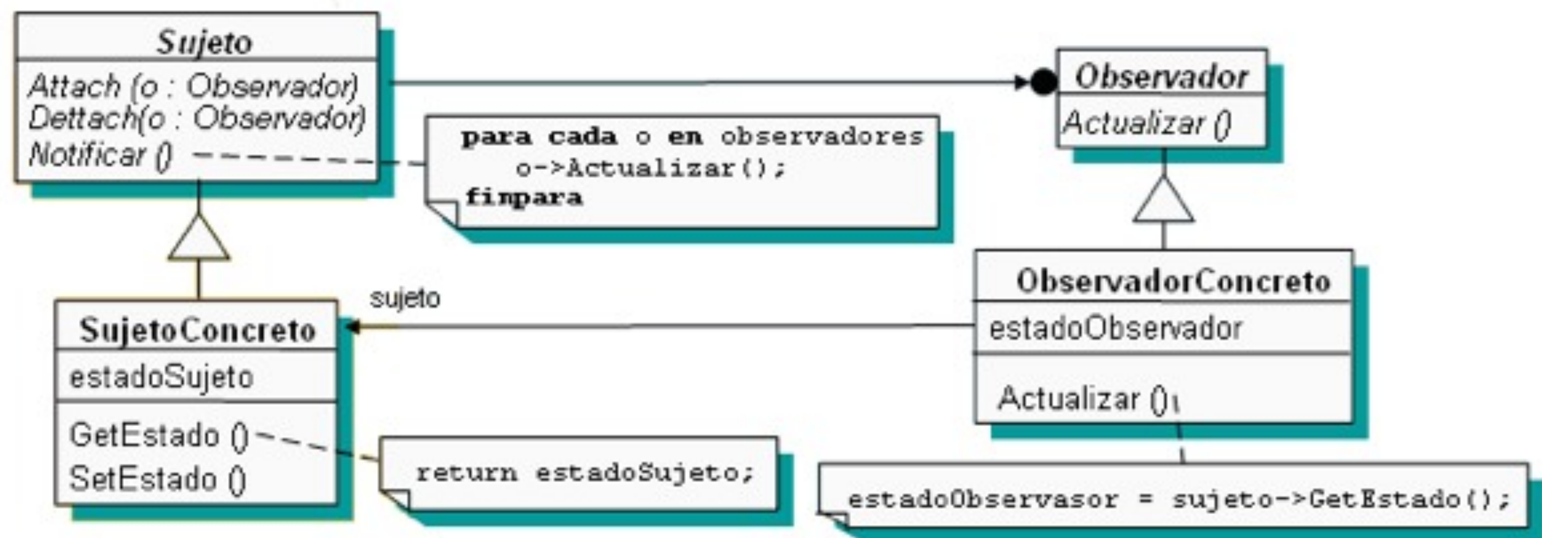
OBSERVER

- Aplicabilidad: utilizar cuando...
  - Un cambio en un objeto requiera cambiar otros y no se sepa cuantos objetos necesitan cambiar
  - Un objeto deba ser capaz de notificar a otros sin conocer su clase concreta, evitando así acoplarlos

Observador

OBSERVER

- Estructura



Observador

OBSERVER

- Participantes
  - Sujeto:
    - conoce a sus observadores y proporciona una interfaz para suscribirlos y desuscribirlos
  - Observador:
    - define una interfaz para actualizar los objetos que deban ser notificados de cambios en el sujeto
  - SujetoConcreto:
    - envía una notificación a sus observadores cuando cambia su estado
  - ObservadorConcreto:
    - mantiene una referencia a un sujeto (SujetoConcreto), almacena parte del estado del sujeto e implementa la interfaz de actualización de Observador para mantener su estado consistente con el del sujeto

Observador

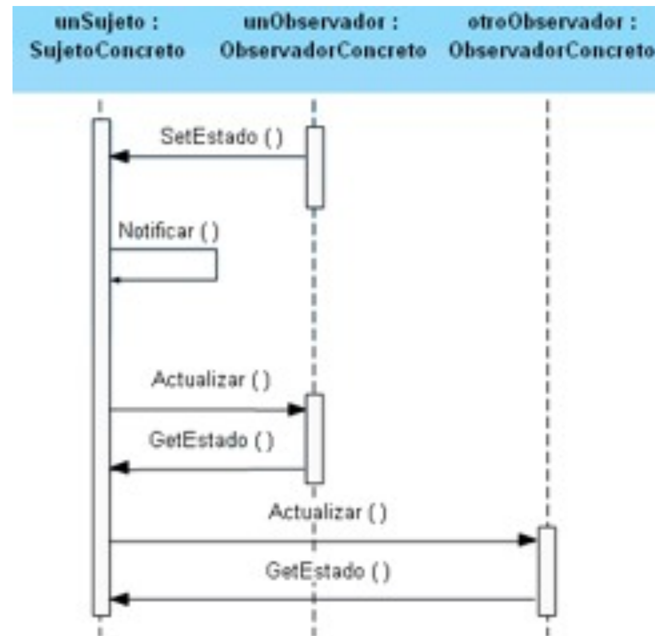
OBSERVER

- Colaboraciones
  - SujetoConcreto notifica a sus observadores cuando ocurre un cambio que pueda hacer inconsistente el estado de los observadores con el suyo propio.

Observador

OBSERVER

- Cuando se le informa del cambio, los observadores pueden solicitar información al sujeto para actualizar su estado



Observador

OBSERVER

- Ventajas e Inconvenientes
  - Permite reutilizar sujetos y observadores por separado así como añadir nuevos observadores sin modificar el sujeto o los otros observadores
  - El acoplamiento abstracto entre el sujeto y el observador ayuda a la división en niveles del sistema
  - Puede que cambios pequeños para unos observadores representen grandes cambios en otros, que además pueden tener problemas para detectar qué es lo que ha cambiado

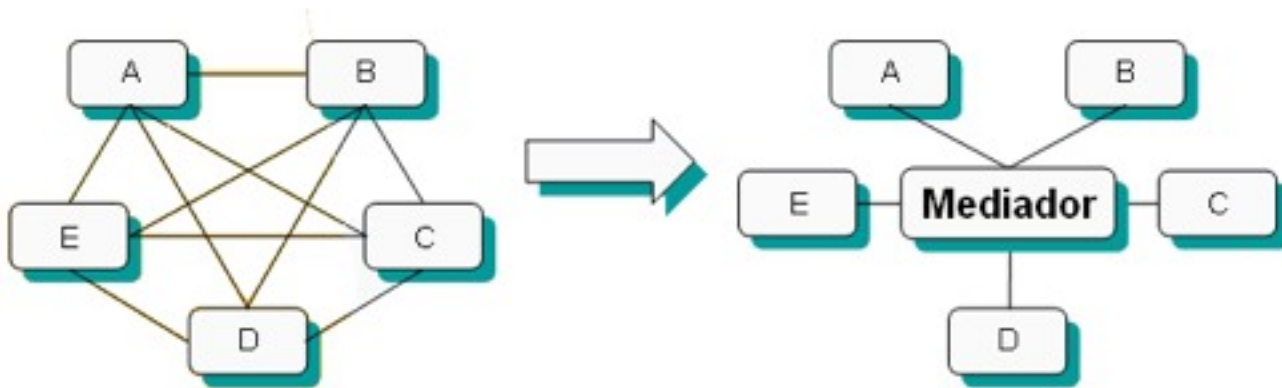
Mediador

MEDIATOR

- Intención:
  - Definir un objeto que encapsule como interactúan un conjunto de objetos, evitando que tengan que conocerse entre ellos y permitiendo cambiar la interacción de forma independiente
- Motivación
  - Que cada objeto encapsule toda su conducta suele ser bueno, aunque en ciertos casos implica un número excesivo de enlaces

Mediador

MEDIATOR



Mediador

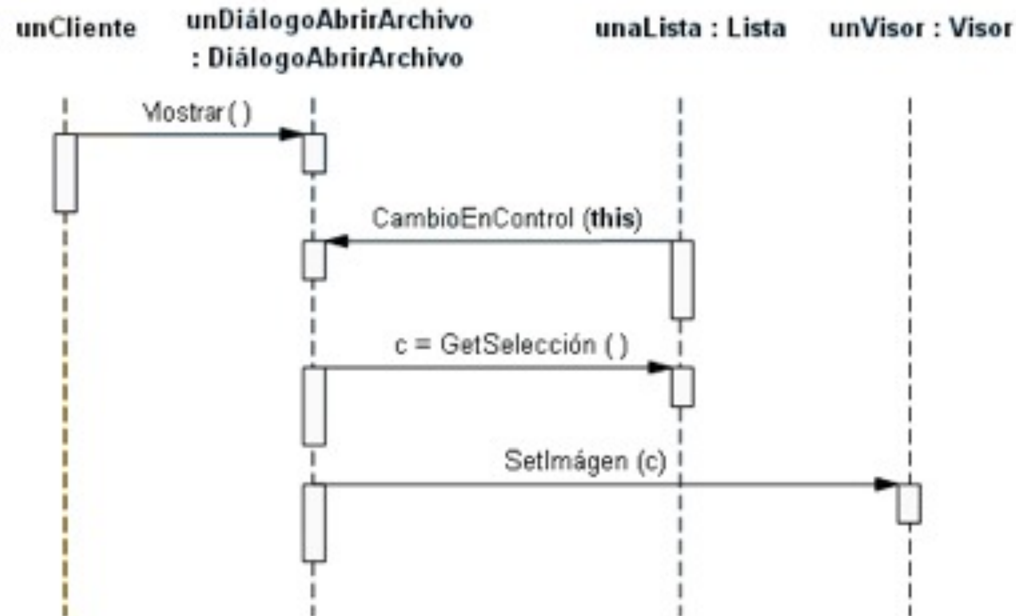
MEDIATOR

- Un diálogo para abrir un archivo tiene muchos objetos interdependientes: se evita derivar cada uno

Mediador

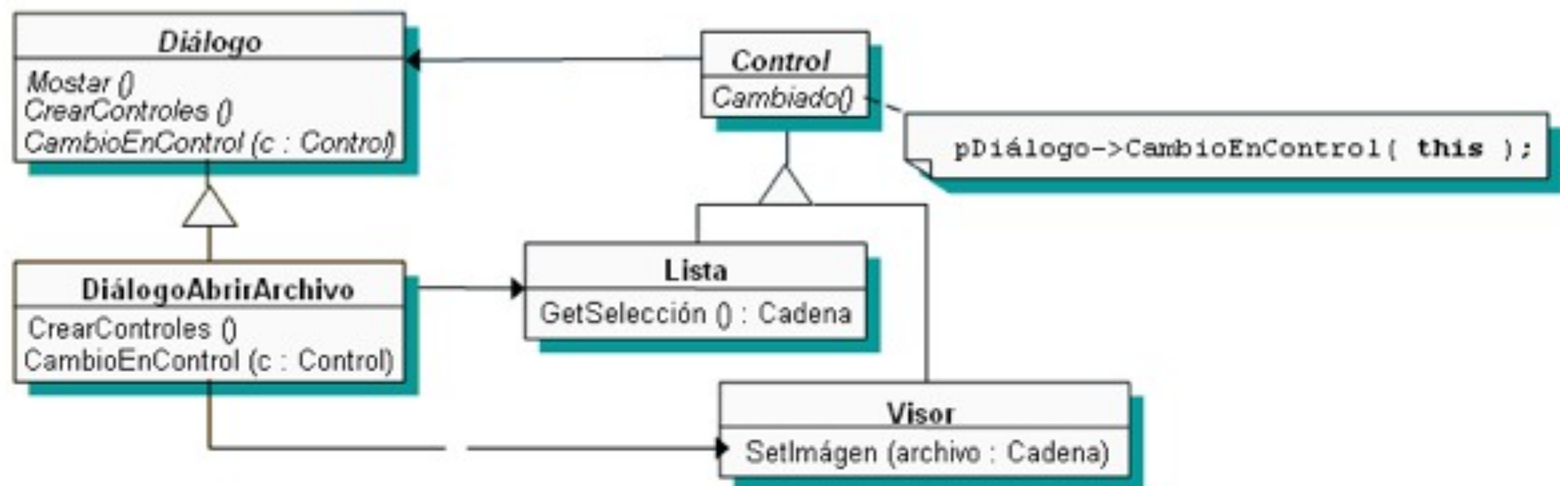
MEDIATOR

- Ejemplo de interacción: cambiar el nombre del fichero seleccionado y mostrarlo en el visor



Mediador

MEDIATOR



Mediador

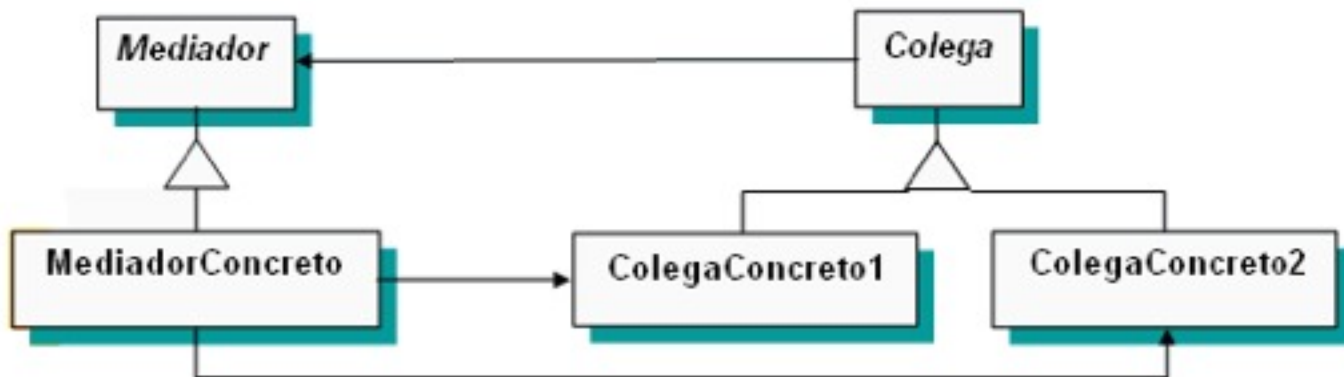
MEDIATOR

- Aplicabilidad: utilizar cuando...
  - Un conjunto de objetos se comunique de una forma bien definida pero compleja, con interdependencias complicadas
  - Reutilizar un objeto sea difícil porque tenga que tener referencias a muchos objetos para comunicarse con ellos

Mediador

MEDIATOR

- Estructura



Mediador

MEDIATOR

- Participantes
  - Mediador:
    - define una interfaz para comunicarse con los objetos colegas
  - MediadorConcreto:
    - implementa la conducta cooperativa coordinando los colegas, a los que conoce y mantiene actualizados
  - Clases colegas:
    - cada clase colega conoce a su mediador y se comunica con él cuando en otras circunstancias lo hubiera tenido que hacer con otro colega

Mediador

MEDIATOR

- Colaboraciones
  - Los objetos colegas envían y reciben peticiones del mediador, que es el que implementa la conducta cooperativa redirigiendo las peticiones entre los colegas apropiados

Mediador

MEDIATOR

- Ventajas e Inconvenientes
  - Evita derivar nuevas clases de colegas: para cambiar la conducta sólo hay que derivar un nuevo mediador
  - Desacopla colegas, permitiendo que cambien independientemente
  - Abstrae la cooperación entre objetos y la encapsula en el mediador, con lo que es más fácil entender como funciona el sistema
  - Centraliza el control en el mediador, que es un objeto complejo y difícil de mantener

Antipatrones de Diseño

- **Clase Gorda**: Dotar a una clase con demasiados atributos y/o métodos, haciéndola responsable de la mayoría de la lógica de negocio.
- **Fábrica de combustible (gas factory)**: Diseñar de manera innecesariamente compleja.
- **Acoplamiento secuencial (sequential coupling)**: Construir una clase que necesita que sus métodos se invoquen en un orden determinado.
- **BaseBean**: Heredar funcionalidad de una clase utilidad en lugar de delegar en ella.
- **Objeto todopoderoso (god object)**: Concentrar demasiada funcionalidad en una única parte del diseño (clase).
- **Poltergeist**: Emplear objetos cuyo único propósito es pasar la información a terceros objetos.
- **Singletonitis**: Abuso de la utilización del patrón singleton.
- ...