Unidad 4. Patrones de Diseño

Contenido

Introducción

- Una clase es un mecanismo para encapsulación.
- Engloba ciertos servicios proporcionados por los datos y comportamiento que es útil en el contexto de alguna aplicación.
- Una clase simple es muy raramente la solución completa a un problema real.
- Existe un cuerpo creciente de investigación para describir la manera en que las colecciones de clases trabajan juntas para **la solución de problemas**.

Patrones de Diseño

- Los patrones de diseño son un intento de coleccionar y catalogar las más pequeñas arquitecturas que son recurrentes en los sistemas **Orientado a Objetos**.
- Un patrón de diseño típicamente captura la solución a un problema que ha sido observado en muchos sistemas.
- Los diseños de patrones son a un nivel arquitectónico lo mismo que las frases en los lenguajes de programación.

Ventajas de los patrones

- Las ventajas del uso de patrones son evidentes:
- Conforman un amplio catálogo de problemas y soluciones
- Estandarizan la resolución de determinados problemas
- Condensan y simplifican el aprendizaje de las buenas prácticas
- Proporcionan un vocabulario común entre desarrolladores
- Evitan “reinventar la rueda”

Tipos de Patrones de Diseño

Creación:

Se preocupan del proceso de creación de un objeto

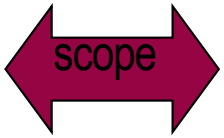
Estructurales:

Se preocupan de como las clases y objetos se componen para formar estructuras mas grandes.

Comportamiento:

Se preocupan con los algoritmos y la asignación de responsabilidades entre los objetos

De Clase tratan de la relación estática entre las clases y subclases



De Objeto trantan con la relación entre objetos que puede cambiar en tiempo de ejecución

Los elementos de los Patrones de Diseño

- Un nombre
- El problema que intenta resolver
  - Incluye las condiciones para que el patrón sea aplicable
- La solución al problema brindada por el patrón
  - Los elementos (clases objetos) involucrados, sus roles, responsabilidades, relaciones y colaboraciones
  - No un diseño o implementación concreta
- Las consecuencias de aplicar el patrón
  - Compromiso entre tiempo y espacio
  - Problemas de implementación y lenguajes
  - Efectos en la flexibilidad, extensibilidad, portabilidad

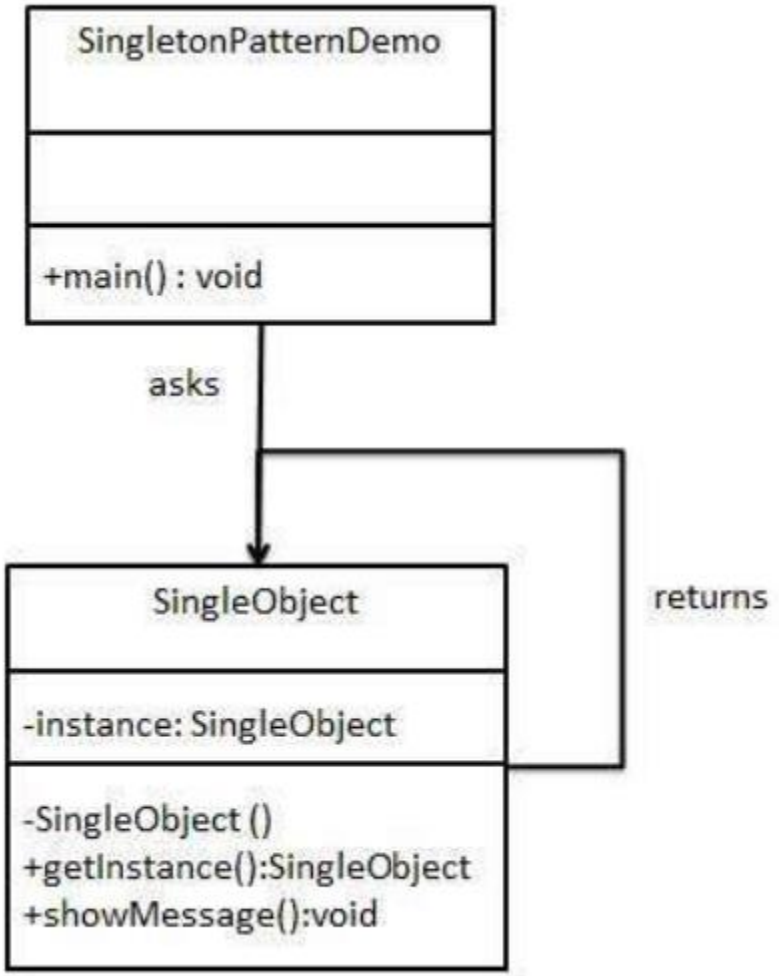
Resumen Patrones

Creacionales	Estructurales	Comportamiento
Singleton	Adapter	Chain of Responsability.
Factory Method	Proxy	Command
Abstract factory	Façade	Interpreter
Prototype	Composite	Iterator
Builder	Flyweight	Mediator
	Bridge	Memento
	Decorator	Observer
		State
		Strategy
		Template Method
		Visitor

Patrones de Diseño Creacionales

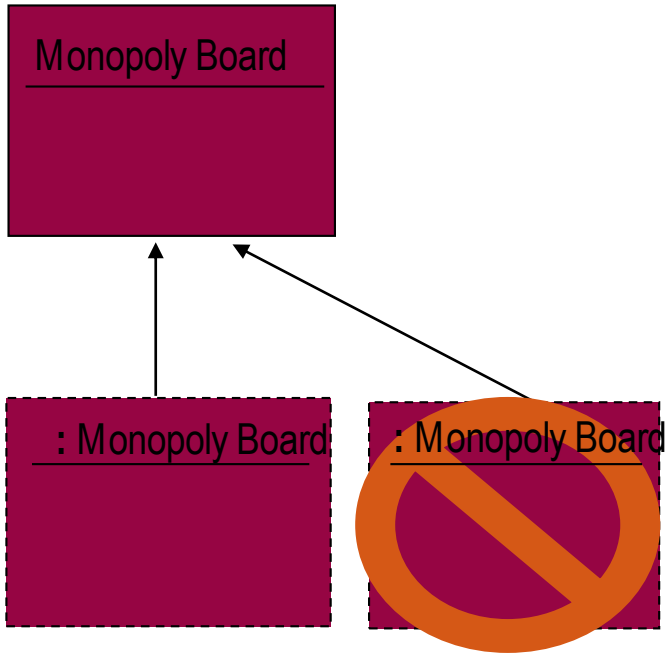
- Abstraen el proceso de instanciación:
  - Encapsulan el conocimiento acerca de que clase concreta se usa
  - Esconde como las instancias de estas clases son creadas y unidas
- Da gran cantidad de flexibilidad en que se crea, quien lo crea, como es creado, y cuando es creado.
- Un patrón de objeto creacional delega la instanciación a otro objeto

Singleton



Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella

Motivación



-Solo puede haber una cola de impresión, un sistema de archivos, un manejador de ventanas en una aplicación estándar

- Existe solo un tablero de juego en monopolio, un laberinto en el juego de Pacman

Participante

- Singleton:
  - Es responsable por crear y almacenar su propia instancia única.
  - Define una operación Instancia que permite al los clientes acceder su única instancia

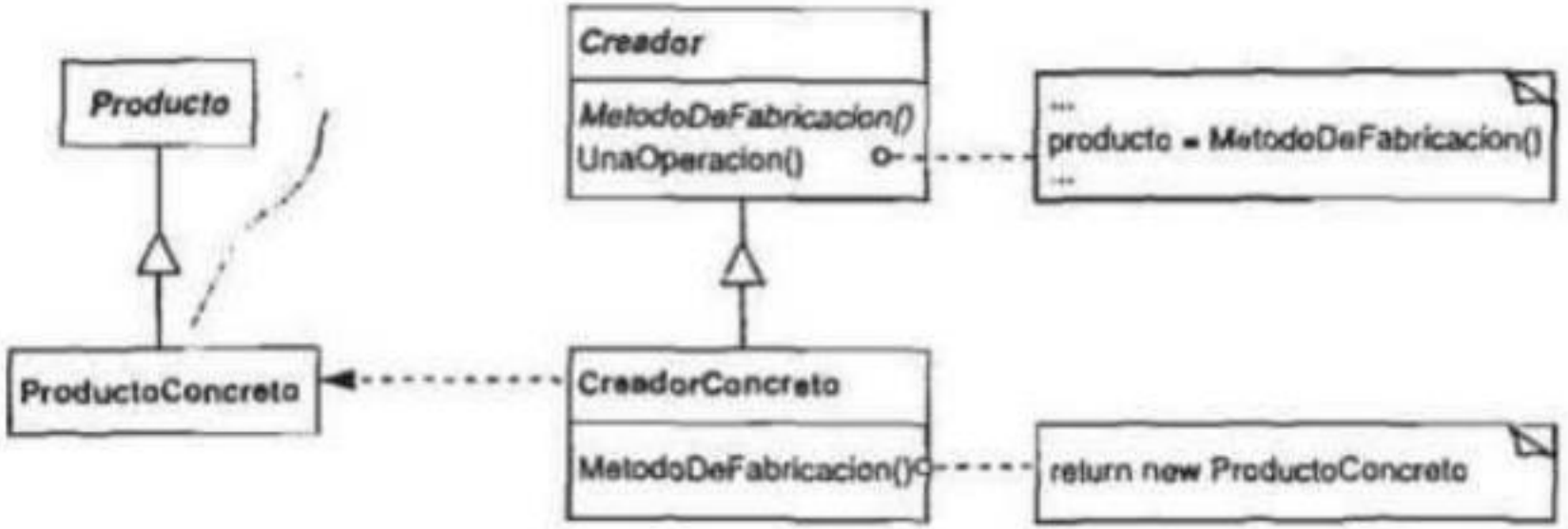
Colaboradores

- La operación Instancia a nivel de clase retorna o crea una sola instancia; un atributo a nivel de clase contiene un valor por defecto indicando que no hay una instancia todavía o una sola instancia

Consecuencias del Patrón Singleton

- Acceso controlado a una sola instancia: debido a que el Singleton encapsula su sola instancia, tiene control estricto sobre ella.
- Reduce el espacio de nombres: es una mejora a polucionar el espacio de nombres con variables globales que contendrán instancias únicas
- Permite refinamiento de operaciones y representación: la clase Singleton puede tener subclases y la aplicación pude ser configurada con una instancia de la clases necesitada en tiempo de ejecución
- Permite un número variable de instancias: el mismo acercamiento puede ser usado para controlar el número de instancias que existen; una operación que de acceso a la instancia debe ser proveída
- Mas flexible que usar una clase con operaciones solamente.

Factory (Virtual Constructor)



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

Participantes

- **Producto**: define la interface de los objetos que el método fábrica crea
- **ProductoConcreto**: implementa la interface Producto
- **Creador**
  - Declara el método fábrica, que retorna un objeto tipo Producto; pude definir una implementación por defecto
  - Puede llamar al método fábrica para crear objetos Producto
- **CreadorConcreto**: sobrescribe el método fábrica para devolver una instancia de ProductoConcreto

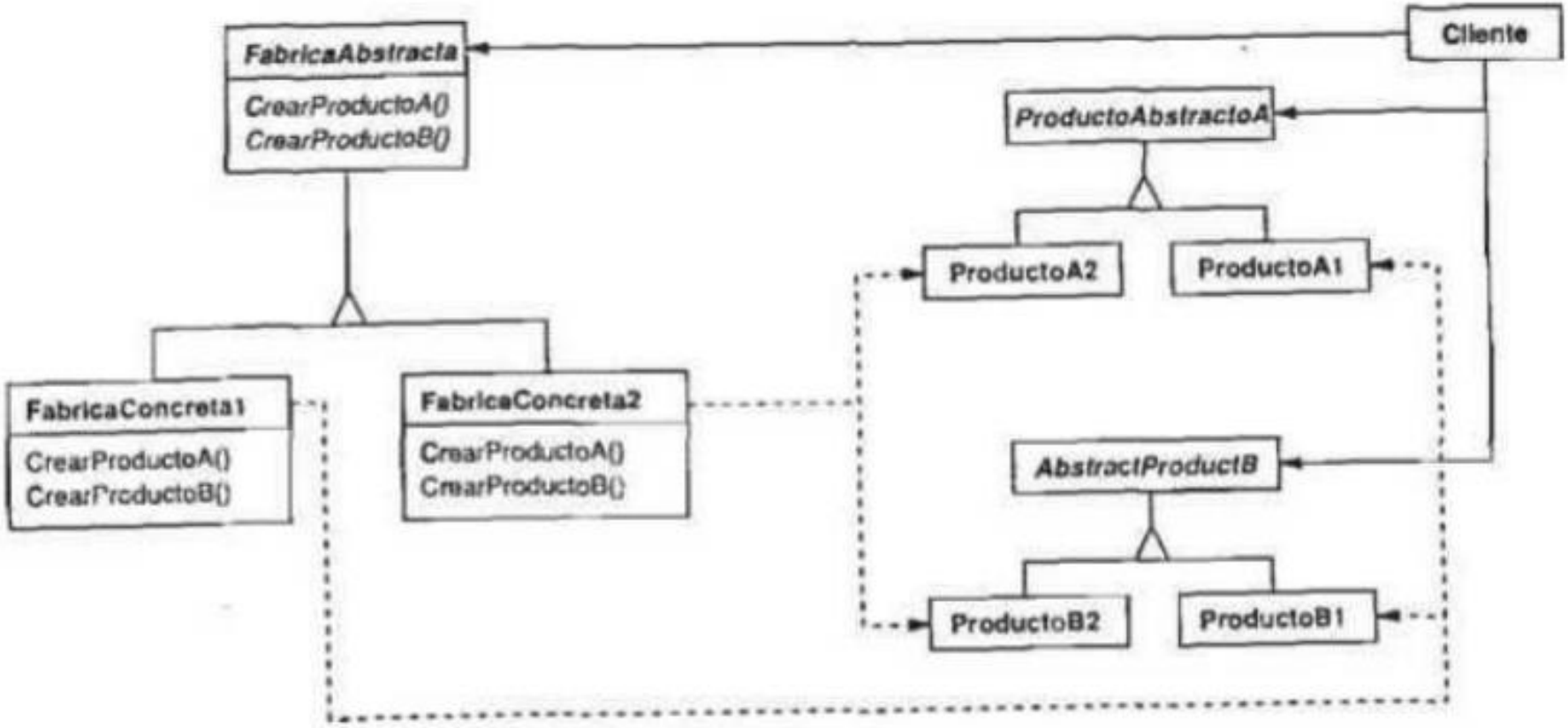
Colaboración

- El Creador se basa en sus subclases para definir el método fábrica que retorna una instancia de ProductoConcreto

Consecuencias

- Elimina la necesidad de unir clases específicas de la aplicación dentro de tu código
- Los clientes pueden tener que crear una subclase de Creador solo para crear un ProductoConcreto particular
- Provee ganchos para las subclases: el método fábrica da a las subclases un gancho para proveer una versión extendida de un objeto
- Conecta jerarquías paralelas de clases: un cliente puede usar un método fábrica para crear una jerarquía paralela de clases.

Abstract Factory (kit)



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

Participantes

- FabricaAbstracta
  - Declara una interfaz para operaciones que crean objetos producto abstractos.
- FabricaConcreta
  - Implementa las operaciones para crear objetos producto concretos.
- Producto Abstracto
  - Declara una interfaz para un tipo de objeto producto.
- Producto Concreto
  - Define un objeto producto para que sea creado por la fábrica correspondiente.
    - implementa la interfaz ProductoAbstracto.
- •Cliente
  - Sólo usa interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto

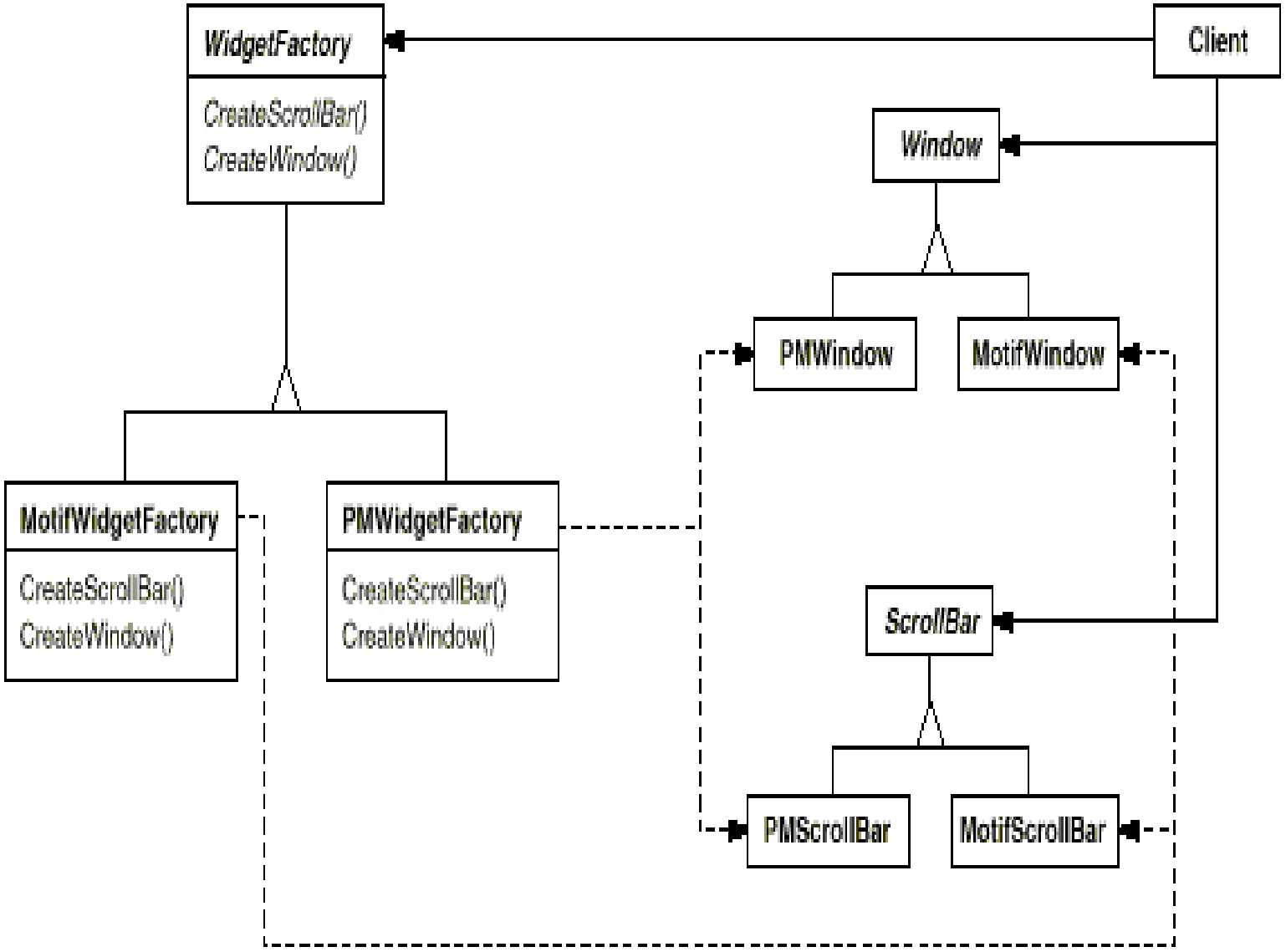
Colaboración

- Normalmente sólo se crea una única instancia de una clase FabricaConcreta en tiempo de eje- cución. Esta fábrica concreta crea objetos producto que tienen una determinada implementación.
- Para crear diferentes objetos producto, los clientes deben usar una fábrica concreta diferente.
- FabricaAbstracta delega la creación de objetos producto en su subclase FabricaConcreta.

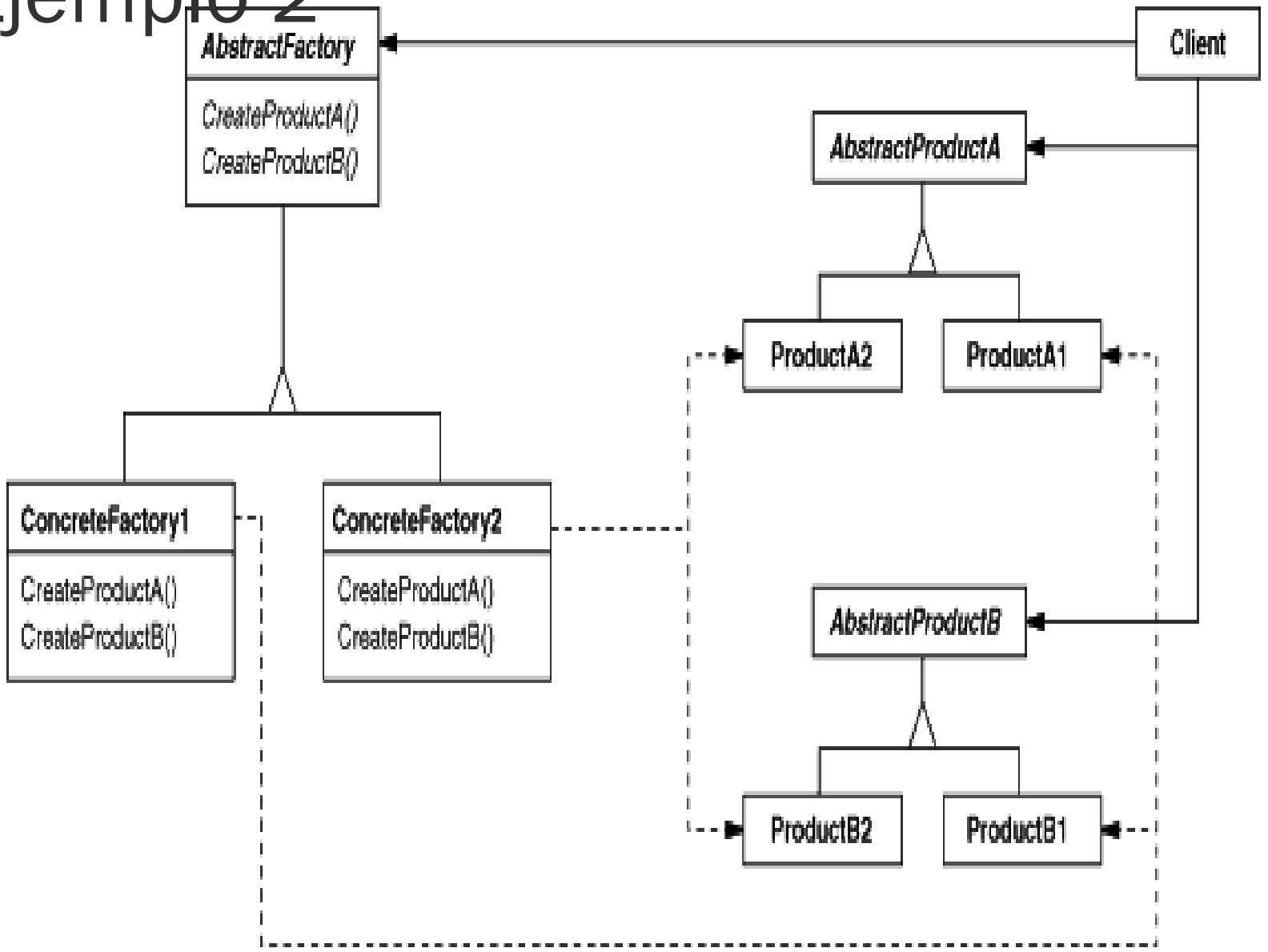
Aplicación

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

Ejemplo


WidgetFactory
CreateScrollBar()
CreateWindow()
Client
Window
PMWindow
MotifWindow
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
ScrollBar
PMScrollBar
MotifScrollBar

Ejemplo 2


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

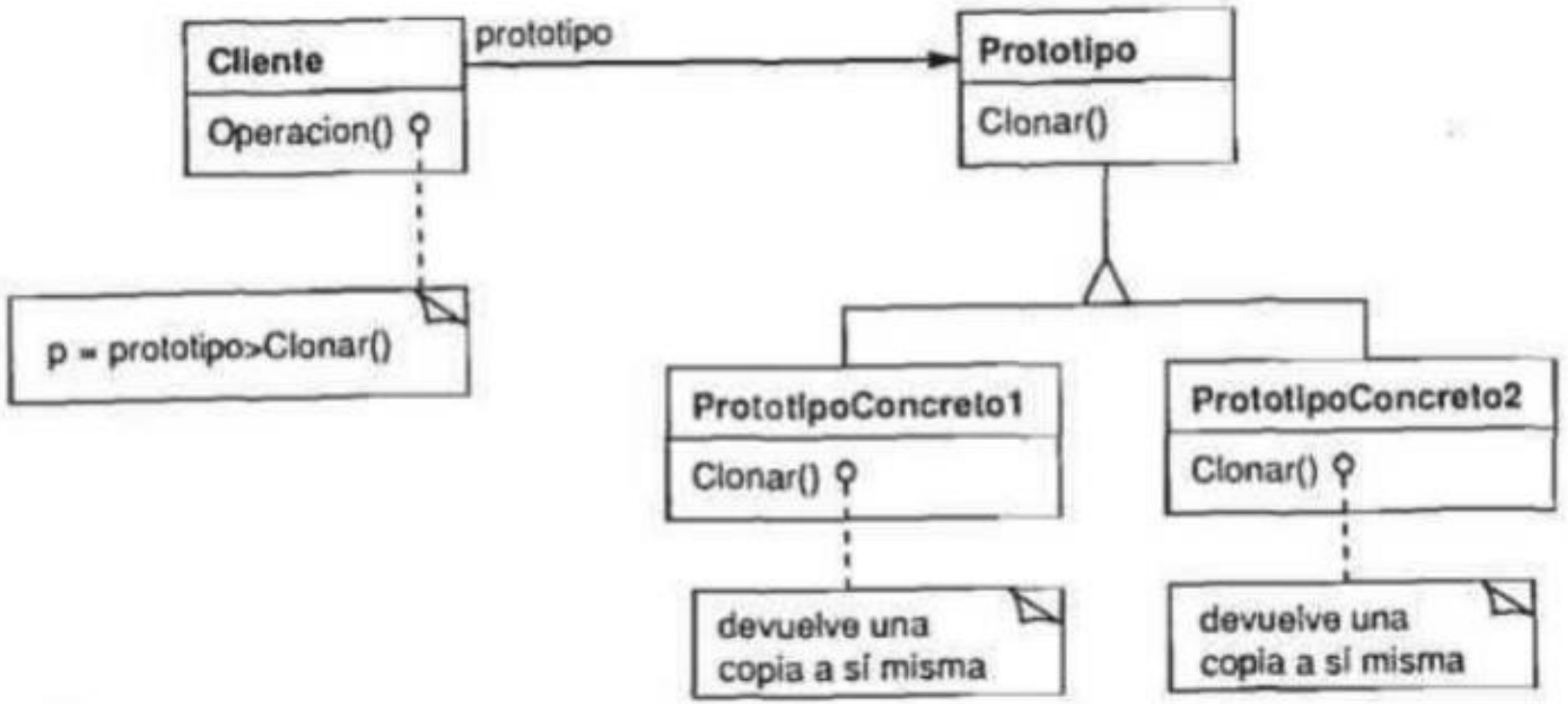
Consecuencias

- Clases concretas aisladas: la AbstractFactory encapsula la responsabilidad y el proceso de crear objetos producto, aísla al cliente de las clases con implementación; los clientes manipulan las instancias a través de sus interfaces abstractas; los nombres de las clases producto no aparecen en el código cliente.
- Hace fácil el intercambio de familias de productos: la clase ConcreteFactory aparece solamente una vez en la aplicación, esto es, donde es instanciada, así que es fácil de remplazar; dado que la fábrica abstracta crea una familia entera de productos, la familia completa pude cambiar de manera sencilla

Consecuencias

- Promueve la consistencia entre productos: cuando los productos en una familia son diseñados para trabajar juntos, es importante para la aplicación el usar los objeto de una sola familia; el patrón de fábrica abstracta hace fácil cumplir esto.
- -Soportar nuevos tipos de productos es difícil: extender las fábricas abstractas para producir una nueva clase de producto no es fácil porque el conjunto de los produtos que pueden ser creados es fija en la interface AbstractFactory; soportar nuevas clases de productos requiere extender la interfase lo que involucra cambiar la clase AbstractFactory y todas sus subclases

Prototype



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

Participantes

- **Prototipo**: declara una interfaz para clonarse a si mismo
- **PrototipoConcreto**: implementa una operación para clonarse a si mismo
- **Cliente**: crea un nuevo objeto al pedir al prototipo que se clone a si mismo

Colaboración

- Un cliente le pide a un prototipo que se clone.

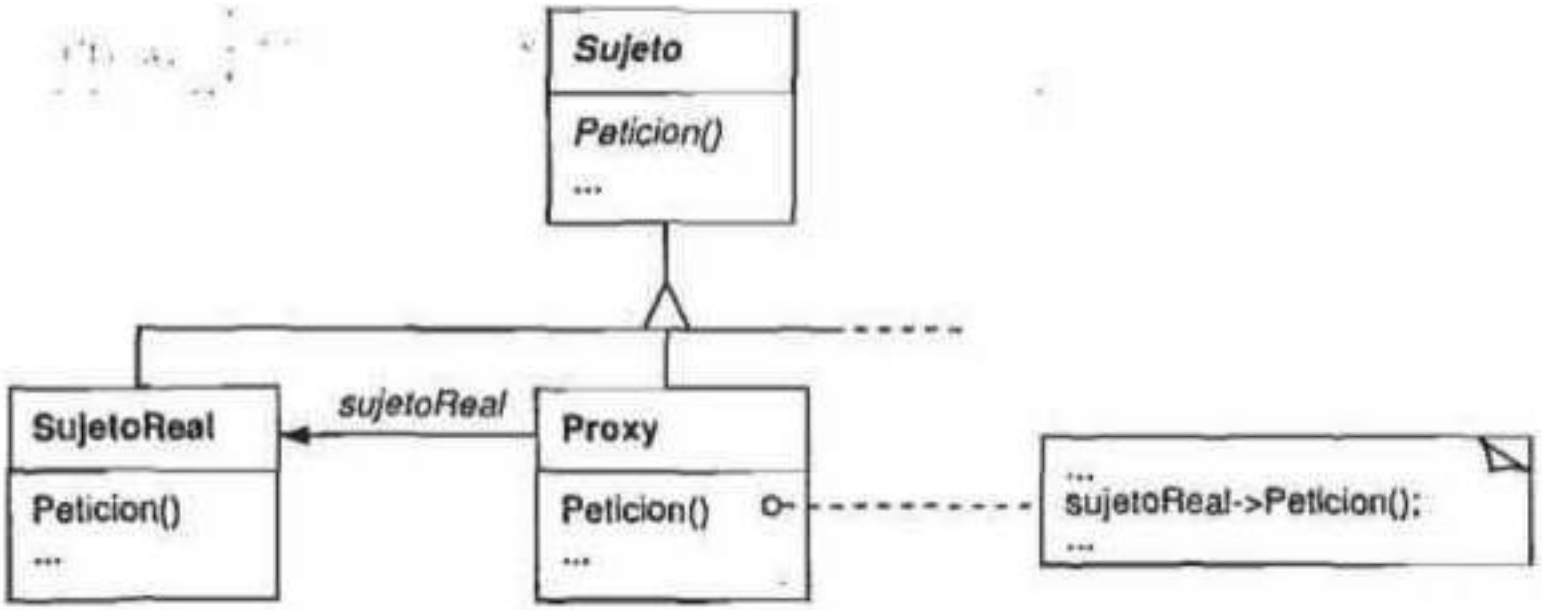
Consecuencias

- + Esconde del cliente las clases concretas del productos: El cliente puede trabajar con las clases específicas de la aplicación sin necesidad de modificarlas.
- + Los productos pueden ser añadidos y removidos en tiempo de ejecución: nuevos productos concretos pueden ser incorporados solo registrándolos con el cliente
- + Especificar nuevos objetos por valores cambiantes: nuevas clases de objetos pueden ser efectivamente definidos al instanciar una clase específica, llenar algunas de sus variables de instancia y registrarla como un prototipo
- + Especificar nuevos objetos por una estructura variante: estructuras complejas definidas por el usuario pueden ser registradas como prototipos también y ser usads una y otra vez al clonarlas

Consecuencias

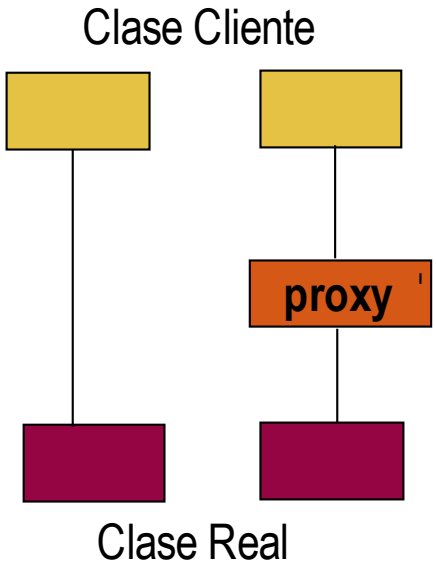
- + Reduce las subclases: al contrario que el Método Fábrica, que a menudo produce una jerarquía de clase creadora que asemeja al jerarquía de ProductosConcretos
- + Configura una aplicación con clases dinámicamente: cuando el ambiente de tiempo de ejecución soporta carga dinámica de clases, el patrón de prototipo es una llave para explotar esas facilidades en un lenguaje estático (los constructores de las clases cargadas dinámicamente no pueden ser accesadas de manera estática; en vez de esto el ambiente de tiempo de ejecución crea automáticamente una instancia del prototipo que la aplicación puede usar a través del manejador de prototipos)
- - Implementar la operación Clonar: es difícil cuando las clases en construcción existen o cuando internamente se incluye objetos que no soportan la copia o tiene referencias circulares

Proxy



Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste.

Problema



• un proxy remoto provee una representación local de un objeto en un espacio de direcciones diferente
•Un proxy virtual crea objejos caros bajo demanda
•Un proxy de protección controla el acceso al objeto original y es útil cuando los objetos tiene diferentes derechos de acceso.
•Una referencia inteligente es un remplazo de un puntero que realiza acciones adicionales cuando un objeto es accesado. Ej. Contar referencias

Participantes

- Proxy:
  - Mantiene una referencia que permite al proxy acceder al sujeto real
  - Provee una interfase idéntica a la del Sujeto de tal manera que el proxy puede sustituir al objeto real
  - Controla el acceso al sujeto real y puede ser responsable de su creación y borrado.
  - Los Proxies remotos son responsables por la codificación de un requerimiento y sus argumentos y por enviar el requerimiento al sujeto real en otro espacio de direcciones.
  - Los proxies Virtuales pueden guardar información acerca del sujeto real de tal manera que puede posponer el acceso a él.
  - Los proxies de protección verifica que el que llama tiene acceso y permiso para llevar a cabo la petición

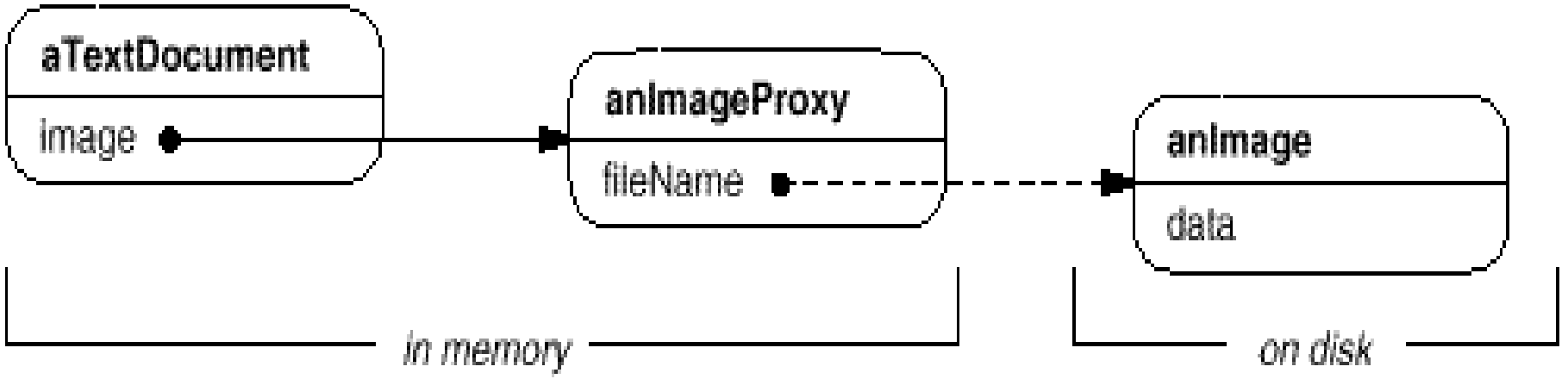
Participantes

- Sujeto:
  - Define una interfaz común para el SujetoReal y Proxy, de tal manera que el Proxy puede ser usado en cualquier lugar donde se pida un SujetoReal
- SujetoReal
  - Define el objeto real que el proxy representa

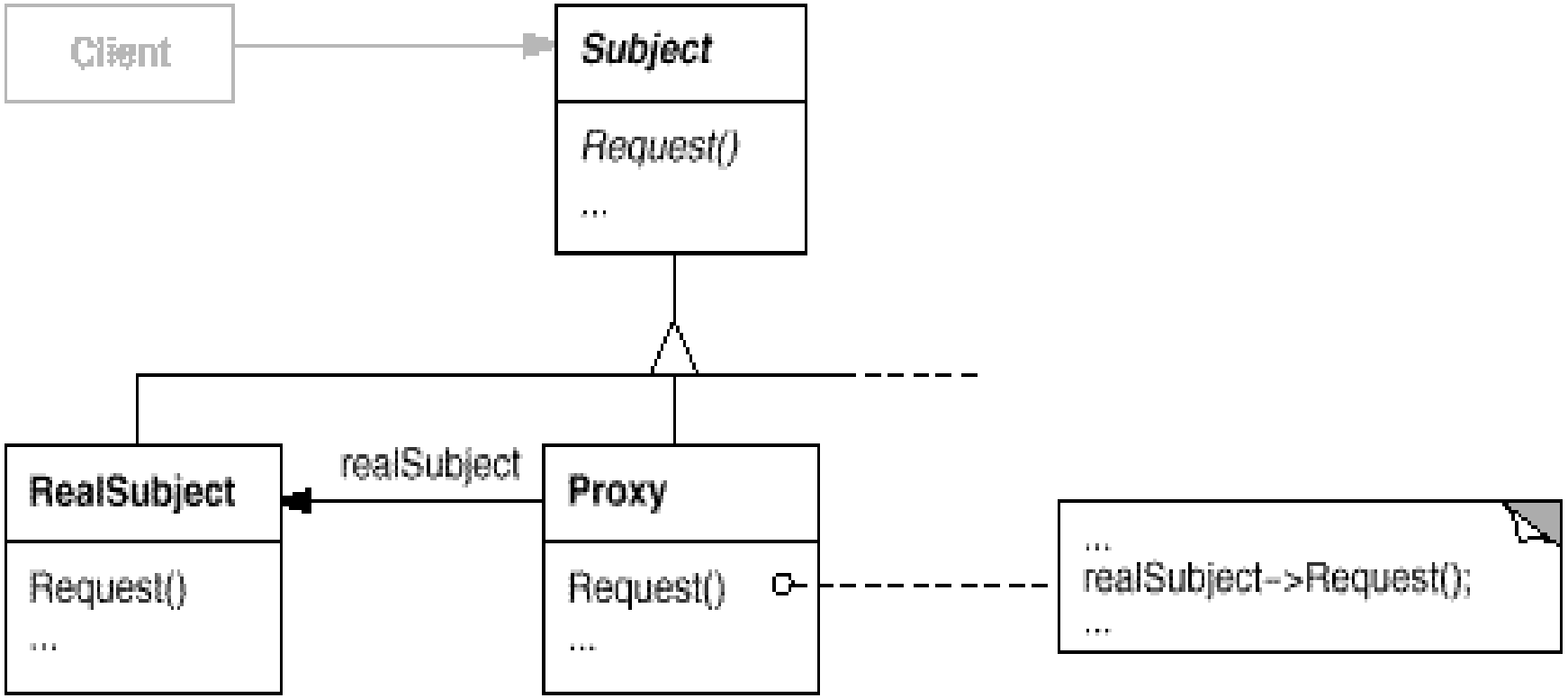
Colaboración

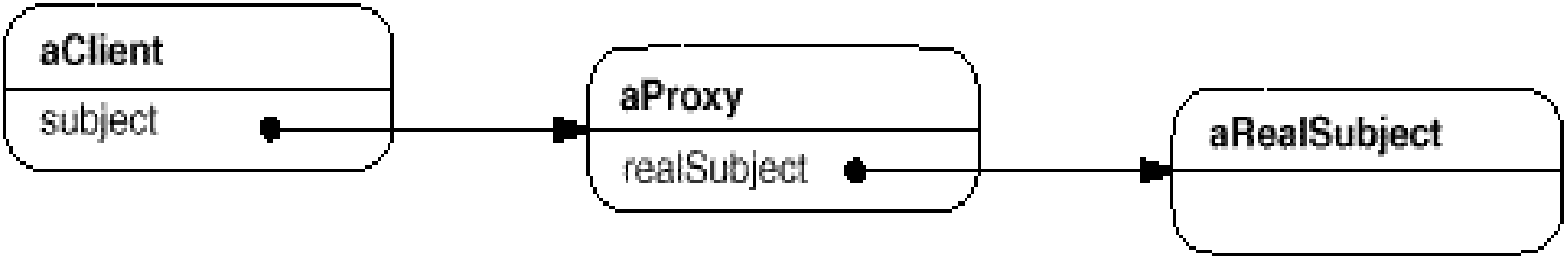
- El Proxy envía la petición al SujetoReal cuando es apropiado (depende del tipo de proxy)

Ejemplo1


aTextDocument
image
anImageProxy
fileName
anImage
data
in memory
on disk

Ejemplo 2

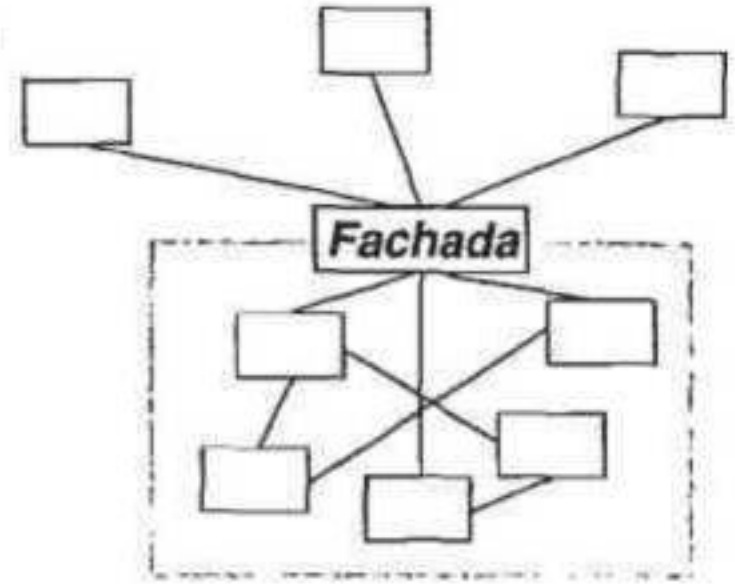

Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...


aClient
subject
aProxy
realSubject
aRealSubject

Consecuencias de Patrón Proxy

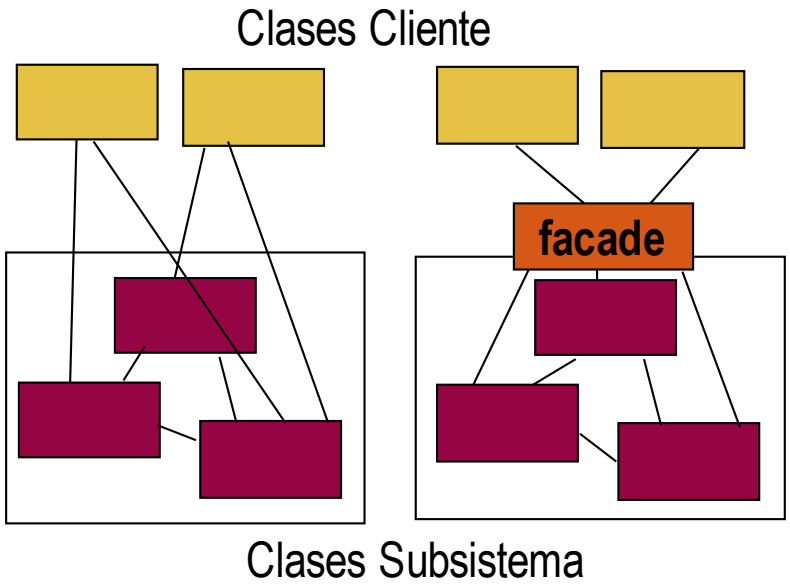
- El patrón Proxy introduce un nivel de indirección cuando se accede al objeto. Esta indirección tiene varios usos:
  - Un proxy remoto puede esconder el hecho de que un objeto reside en un espacio de direcciones diferente
  - Un proxy virtual puede ejecutar optimizaciones
  - Tanto el proxy de protección como los punteros inteligentes permiten mantenimiento adicional
- El patrón de proxy puede ser usado para implementar "copia-al-escbir" : para evitar copiar innecesariamente objetos granes, el sujeto real es referenciado; los requerimientos de cada copia incrementan este contador, pero solo cuando un cliente pide una operación que modifica al sujeto, el proxy lo copia.

Facade



Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

El Problema



- **Provee una interfaz simple a un subsistema complejo**
- **Desacopla un subsistema de sus clientes y otros subsistemas**
- **Crea capas de subsistemas al proveer una interfaz para cada nivel del subsistema.**

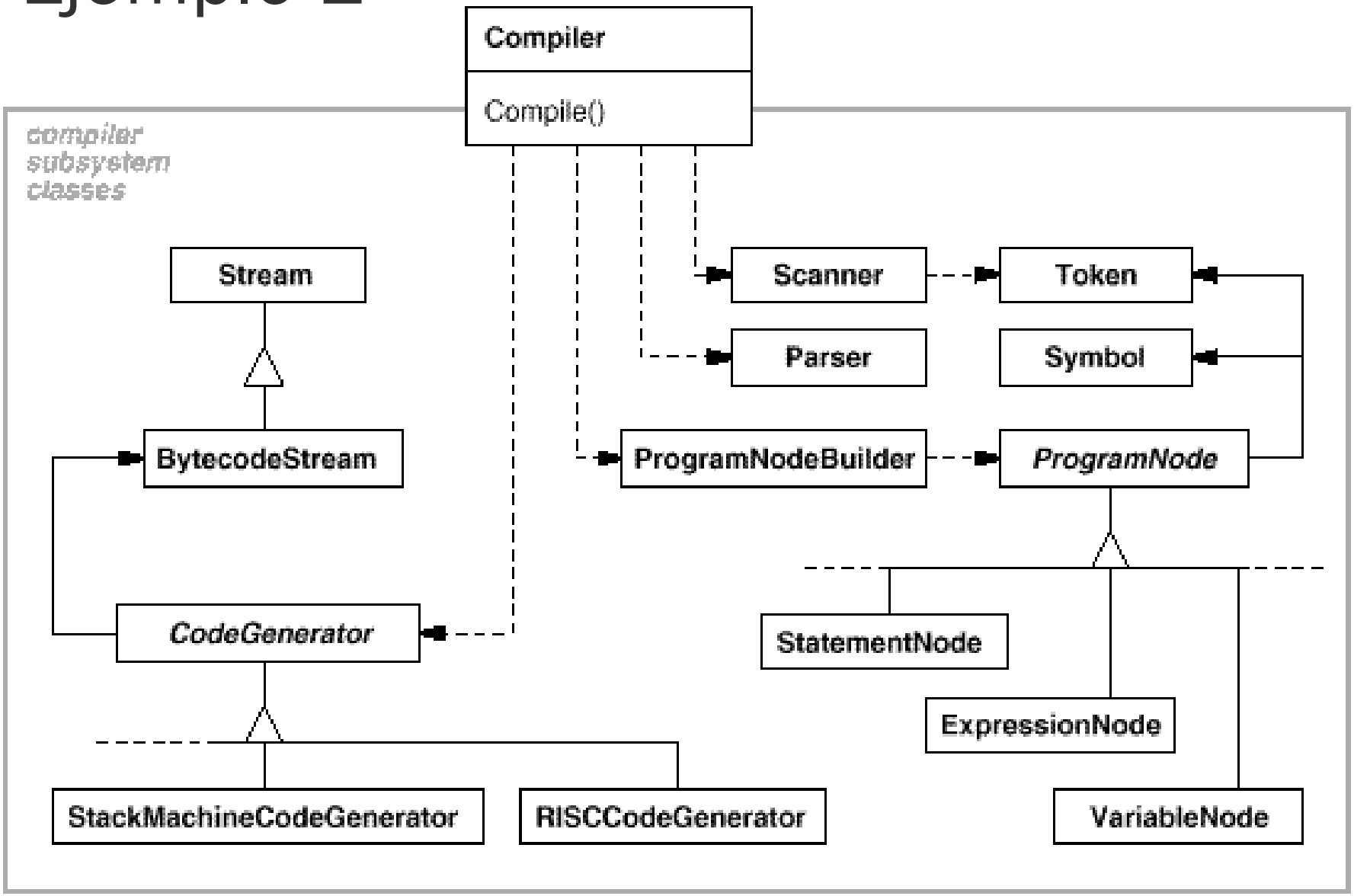
Participantes

- Fachada:
  - Sabe que clases del subsistemas son responsables para cada petición
  - Delega las peticiones de los clientes a los subsistemas apropiados
- Clases del Subsistema
  - Implementan la funcionalidad del subsistema
  - Manejan el trabajo asignado por el objeto Fachada
  - No tienen conocimiento del objeto Fachada. Ej.: no mantienen una referencia a él.

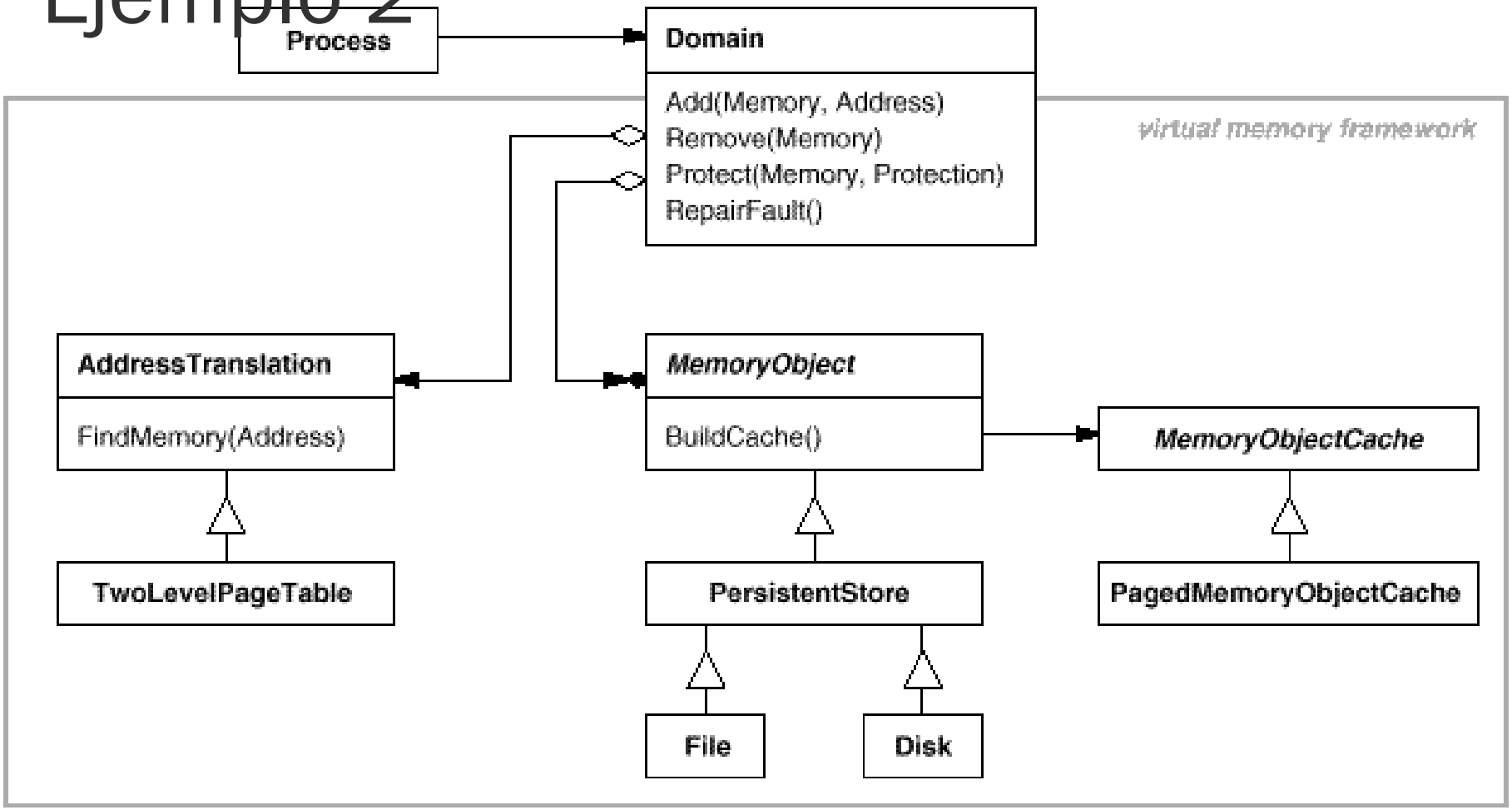
Colaboración

- Los clientes envía su petición a Fachada la que a su vez se lo envía al objeto apropiado en el subsistema

Ejemplo 1


Compiler
Compile()
compiler subsystem classes
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

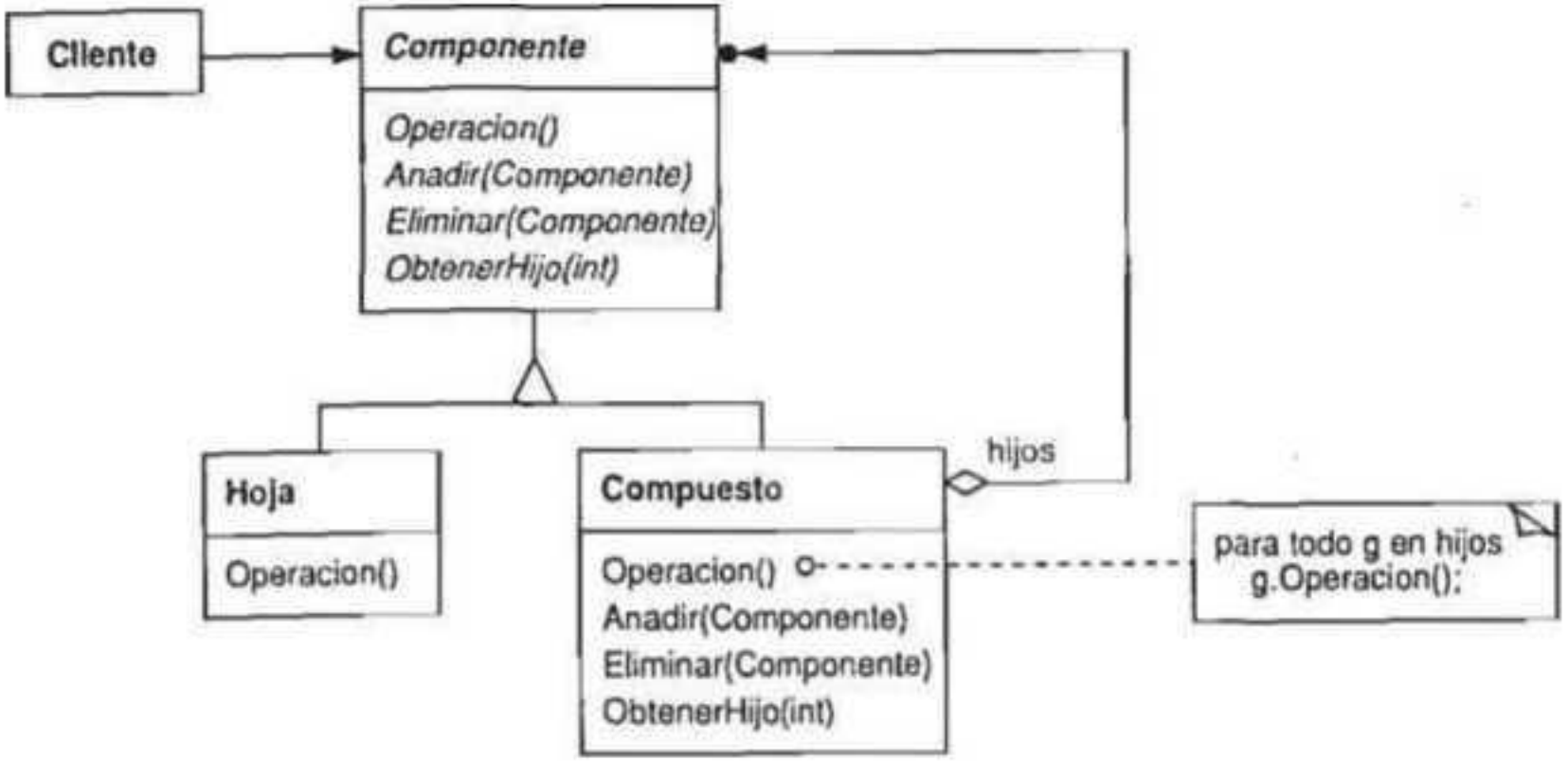
Ejemplo 2


Process
Domain
Add(Memory, Address)
Remove(Memory)
Protect(Memory, Protection)
RepairFault()
virtual memory framework
AddressTranslation
FindMemory(Address)
MemoryObject
BuildCache()
MemoryObjectCache
TwoLevelPageTable
PersistentStore
PagedMemoryObjectCache
File
Disk

Consecuencias

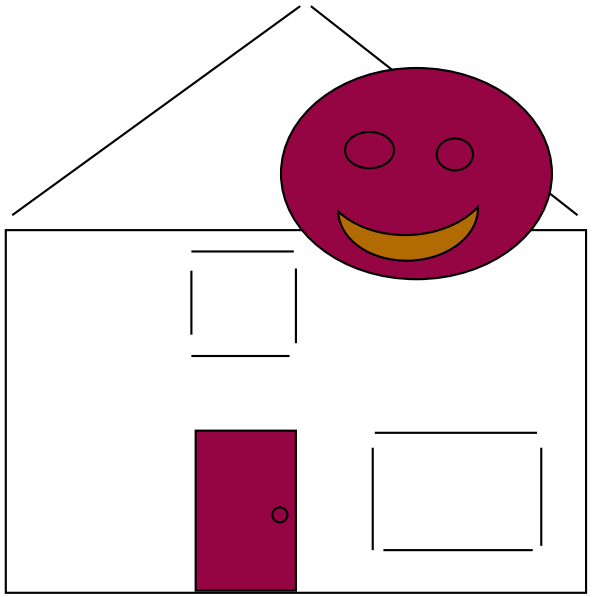
- + Escuda a los clientes de los componentes del subsistema reduciendo el número de objetos con que los clientes tienen que trabajar, haciendo al subsistema más fácil de usar
- + Promueve el aparejamiento débil entre el subsistema y sus clientes, permitiendo que los componentes del subsistema cambien sin afectar a los clientes.
- + Reduce las dependencias de compilación en sistemas grandes de software
- + No previene que las aplicaciones usen las clases del subsistemas si así lo necesitaren

Composite



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

El Problema



- Una herramienta de dibujo que permite a los usuarios construir diagramas complejos a partir de elementos simples
-Árboles con nodos heterogéneos, ej. El árbol de parsing de un programa

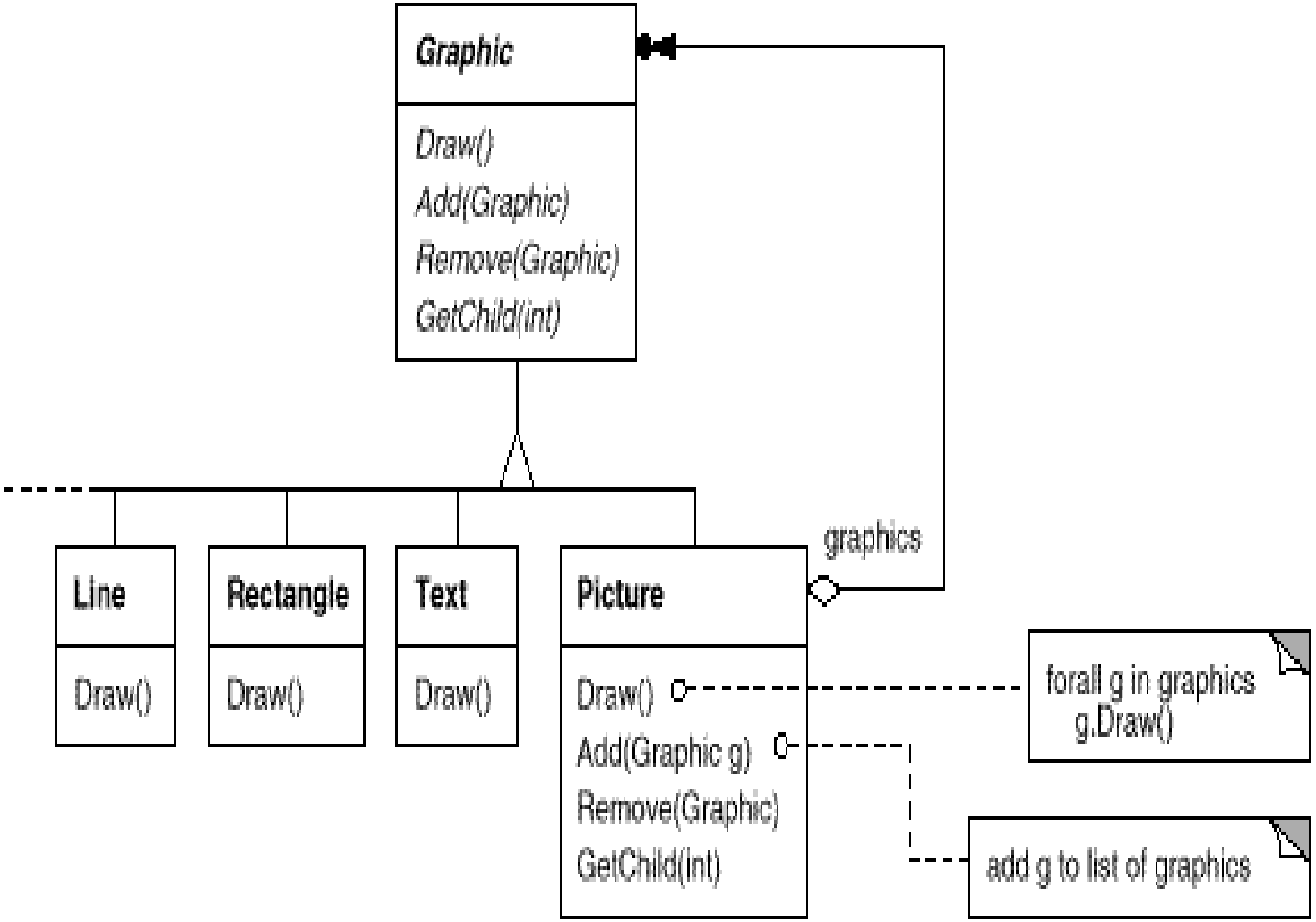
Participantes

- Componentes: declaran al interfaz para objetos en la composición, implementa el comportamiento por defecto para la interface común de todos los objetos, declara una interfaz para acceder y manejar los componentes hijo, opcionalmente define/implementa una interfaz para acceder al componente padre.
- Hoja: define el comportamiento para objetos primitivos de la composición
- Composición: define el comportamiento para los componentes que tengan hijos, guarda los componentes hijos, implemente acceso a los hijos y administran las operaciones en la interfaz del componente
- Cliente: manipula llos objetos en la composición a través de la interfaz componente

Colaboracion

- Los Clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta.
- SI el recipiente es una Hoja, la petición se trata correctamente.
- Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

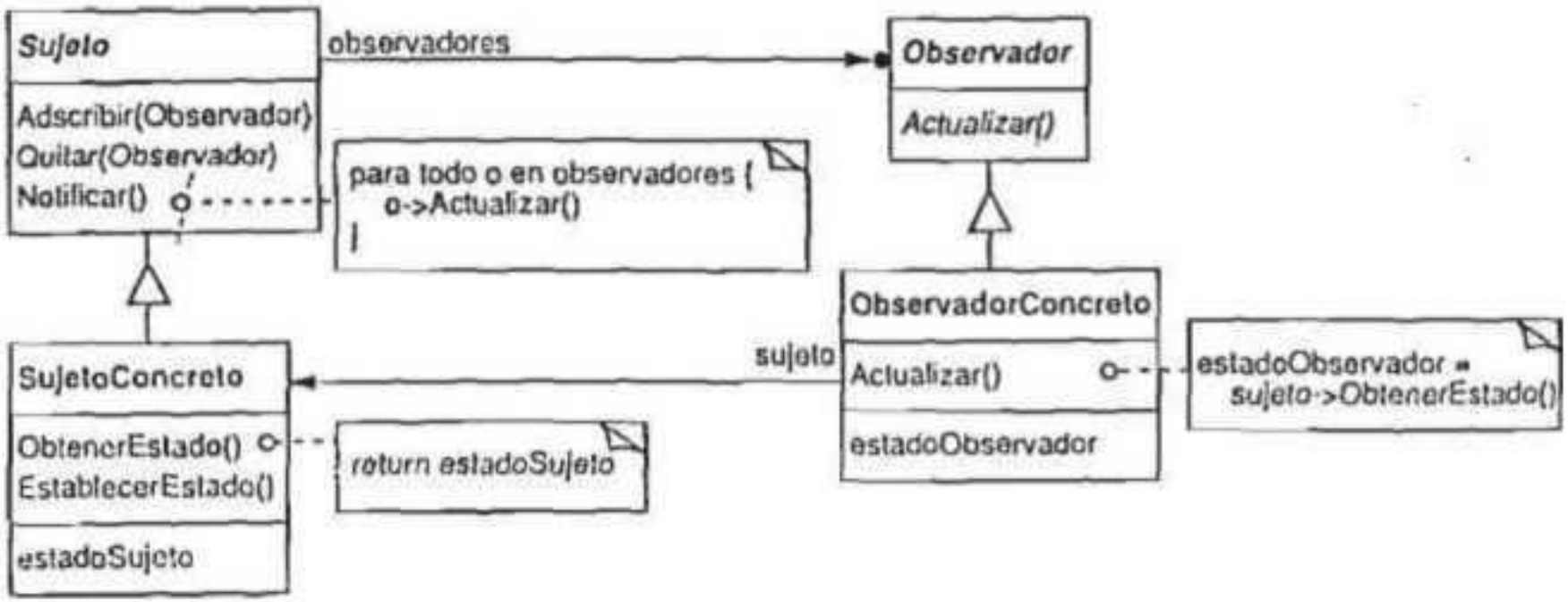
Ejemplo


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
graphics
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
forall g in graphics
g.Draw()
add g to list of graphics

Consecuencias

- + Hace al cliente más simple: los clientes pueden tratar a las estructuras compuestas y a los individuos de manera uniforme, los clientes normalmente no necesitan saber y no les debería importar si están tratando con una hoja o una composición.
- + Hace más fácil añadir un nuevo tipo de componentes: el código del cliente trabaja automáticamente con las composiciones u hojas recién definidas
- - Puede volver al diseño muy general: la desventaja de hacer sencillo el añadir nuevos componentes es que eso dificulta restringir los componentes de una composición, algunas veces en la composición se desea solo cierto tipo de hijos. Con el patrón Composición no se pude confiar en que el sistema hará cumplir esto.

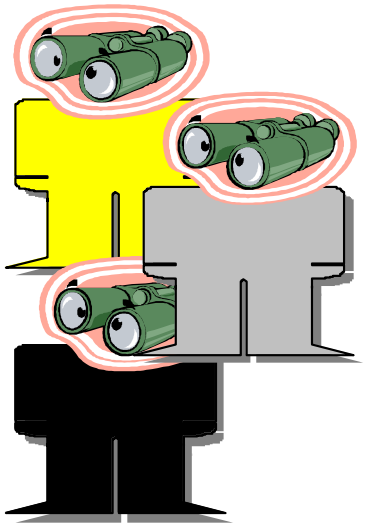
Observer



Asume una relación uno a muchos entre objetos, donde uno cambia y los dependientes deben actualizarse

El Problema

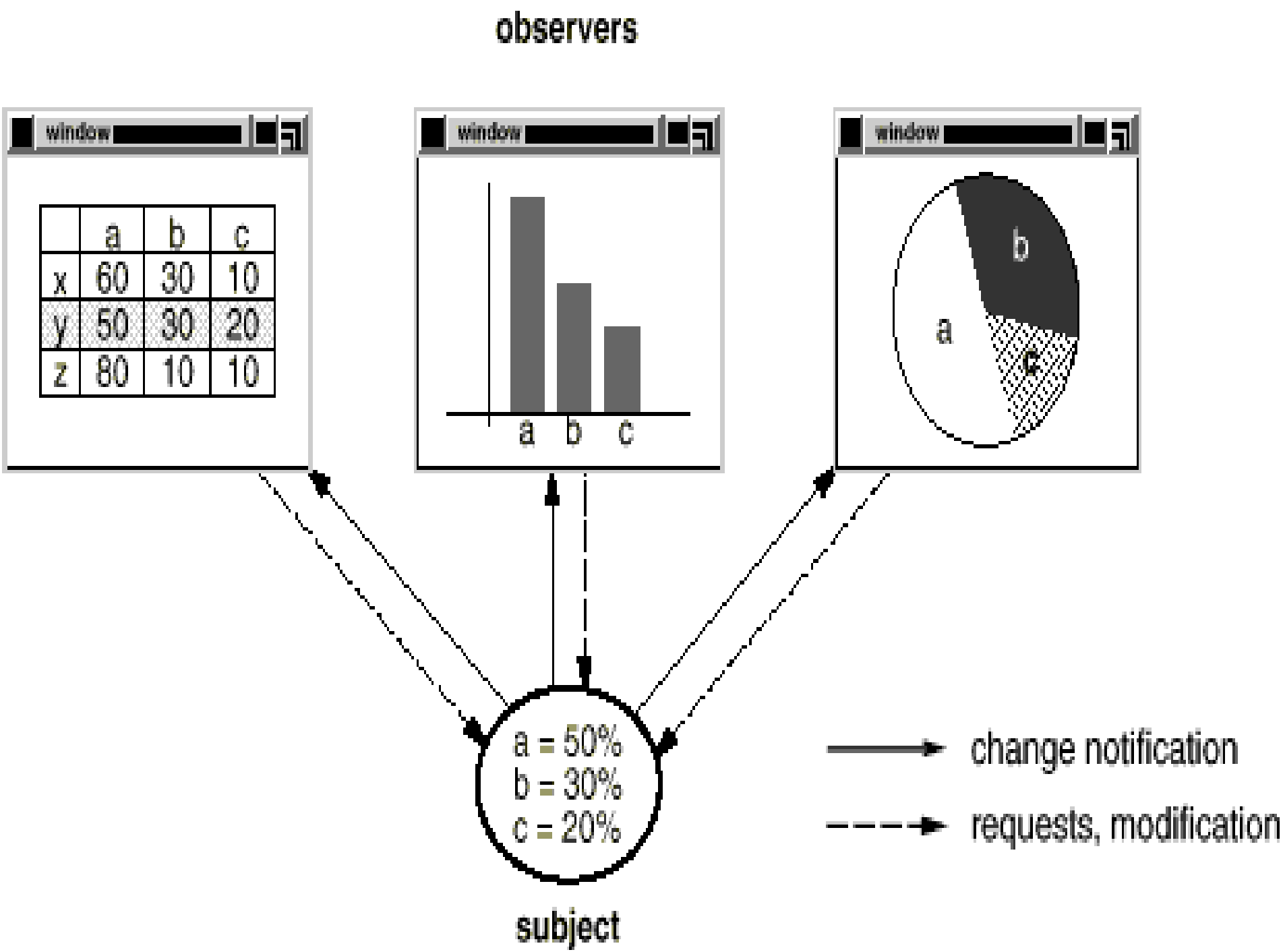
Sujeto



Observadores

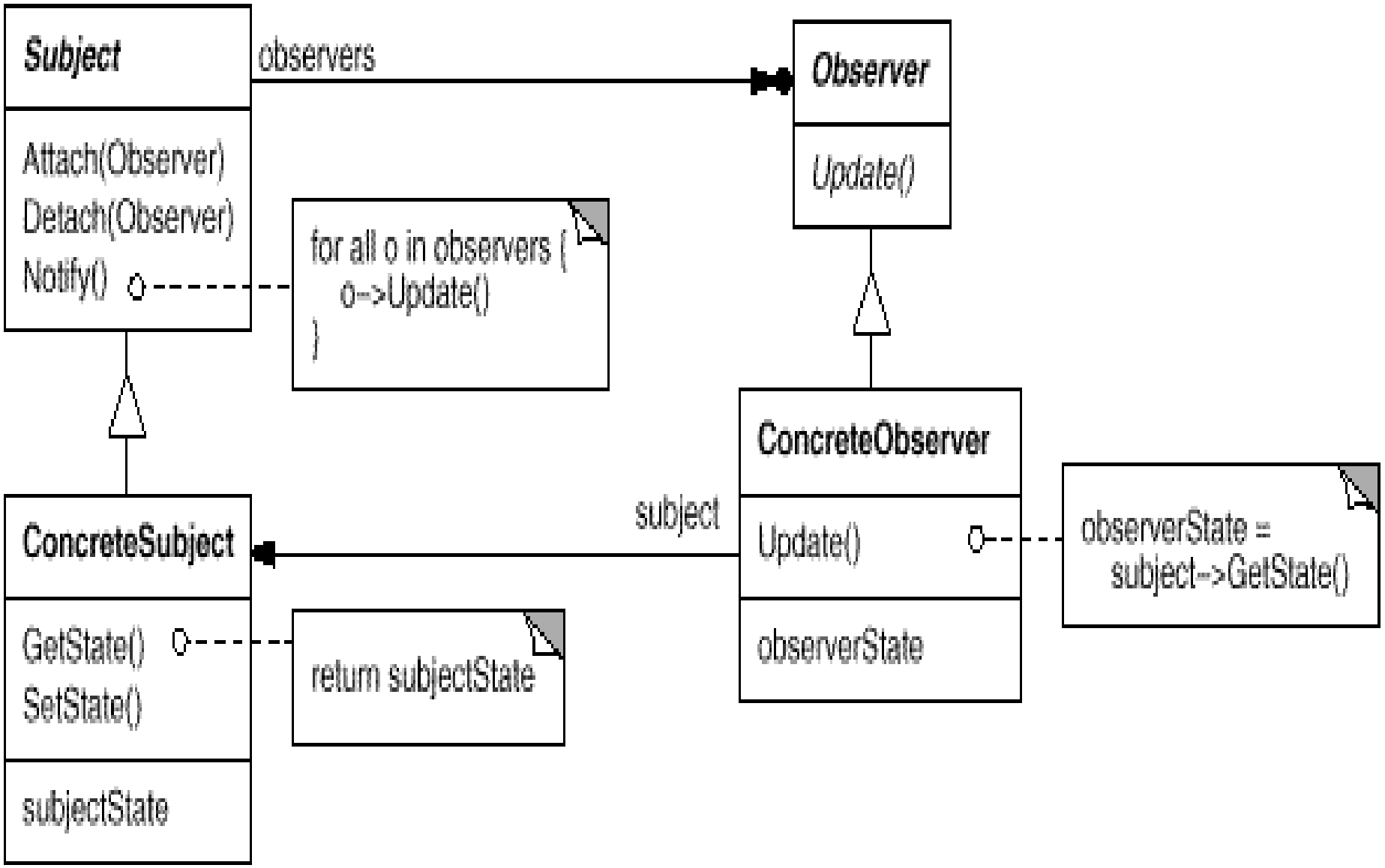
-Diferentes tipos de GUI mostrando los mismos datos
- Diferentes ventanas mostrando diferentes vistas de un mismo modelo.

También conocido como: Dependants, Publish-Subscribe


observers
window
window
window
a
b
c
x
y
z
60
30
10
50
30
20
80
10
10
a
b
c
a = 50%
b = 30%
c = 20%
change notification
requests, modification
subject

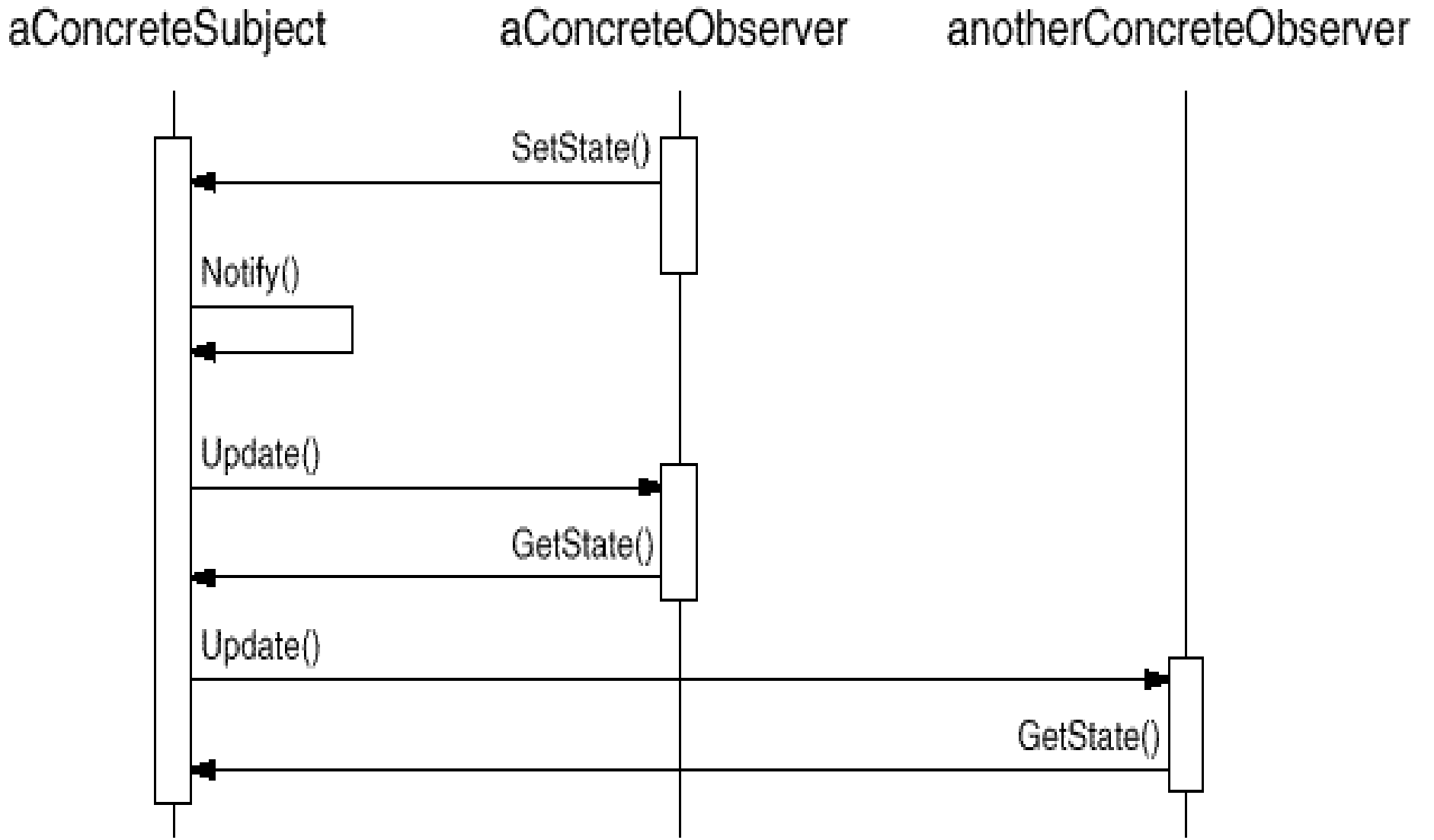
Participantes

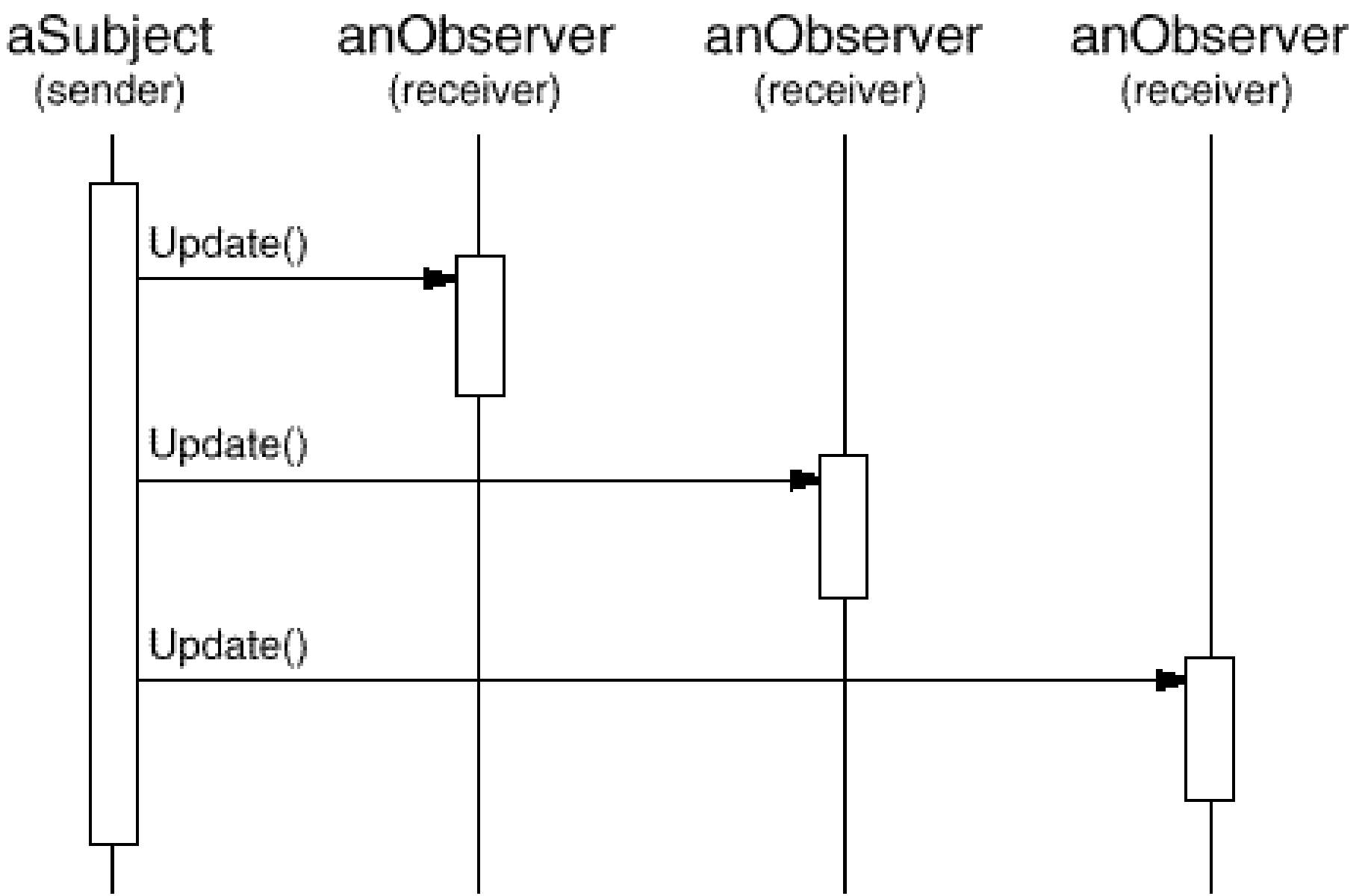
- Sujeto: conoce sus observadores, provee una interfaz para añadir (subscribir) y eliminar (cancelar) observadores y provee un método *notificar* que llama a *actualizar* en todos los observadores
- Observador: provee una interfase *actualizar*
- SujetoConcreto: mantiene el estado relevante para la aplicación, provee métodos para obtener y setear ese estado, llama a *notificar* cuando el estado es cambiado
- ObservadorConcreto: mantiene una referencia al sujeto concreto, guarda el estado que se mantiene consistente con el del sujeto e implementa la interfaz *actualizar*


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState

Colaboración

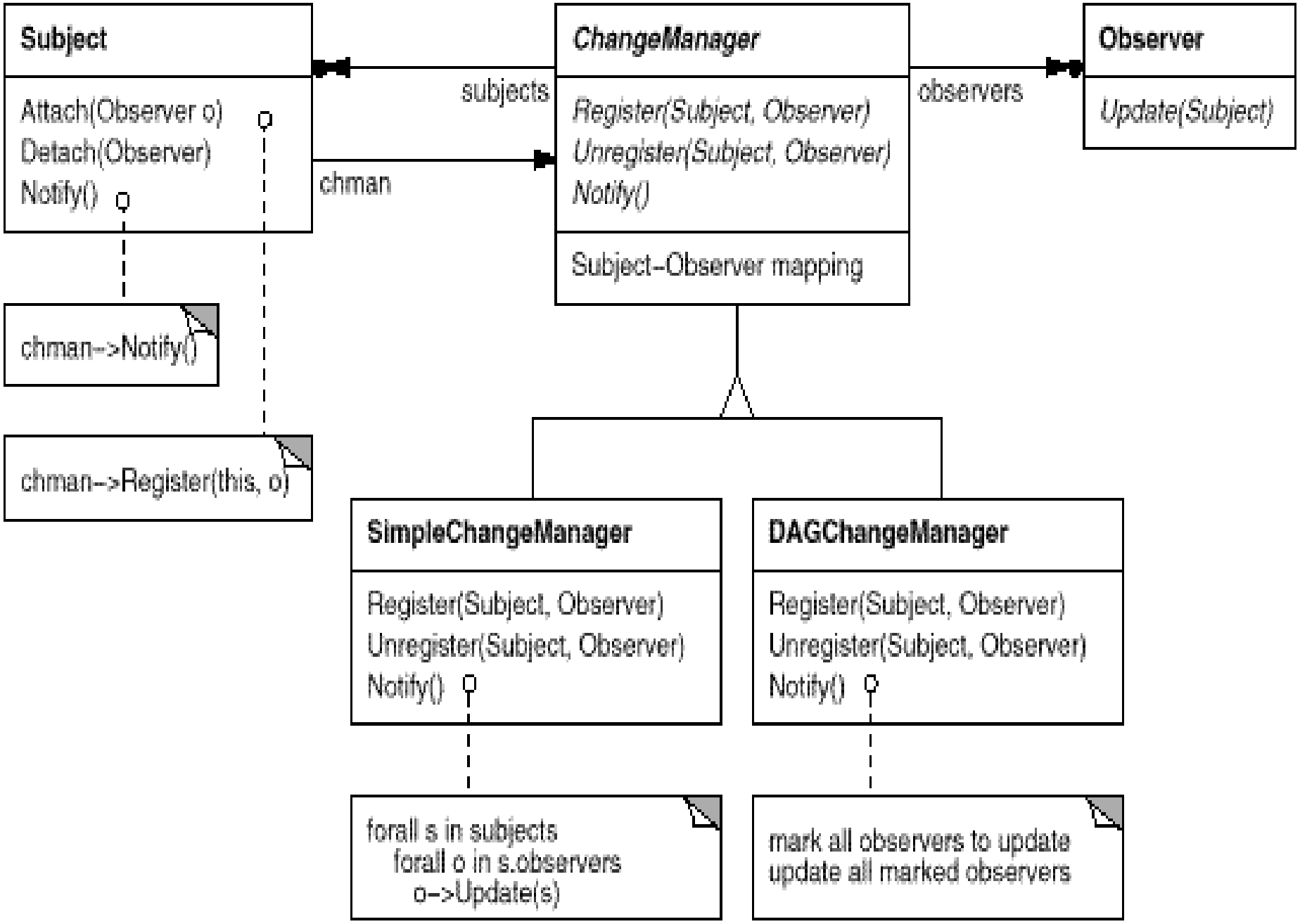
- El SujetoConcreto notifica a sus observadores cada vez que se produce un cambio que pueda volver el estado de los observadores inconsistente.
- Después de ser informados de un cambio, el Observador Concreto puede preguntar al Sujeto acerca de la información concerniente a su estado y entonces reconciliar su propio estado con el del Sujeto.
- El cambio en el SujetoConcreto pude ser iniciado o uno de los ObservadoresConcretos por algún otro objeto de la aplicación


aConcreteSubject
aConcreteObserver
anotherConcreteObserver
SetState()
Notify()
Update()
GetState()
Update()
GetState()

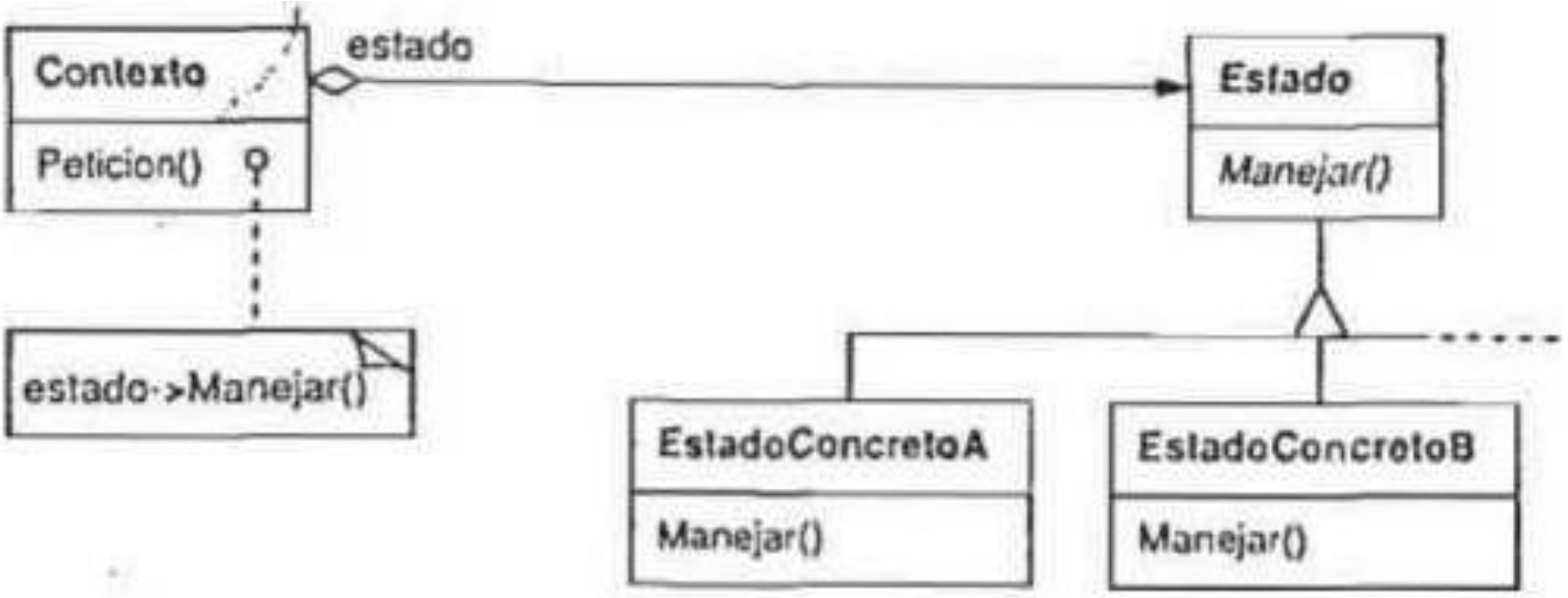

aSubject
(sender)
anObserver
(receiver)
anObserver
(receiver)
anObserver
(receiver)
Update()
Update()
Update()

Consecuencias

- + Aparejamiento abstracto y mínimo entre el Sujeto y el Observador: El sujeto no conoce la clase concreta del observador, las clases sujetoconcreto y observadorconcreto pueden ser reusadas independientemente, los sujetos y observadores pueden incluso pertenecer a diferentes capas de abstracción del sistema
- + Soporte para comunicación broadcast: la notificación enviada por el sujeto no necesita especificar un receptor, se transmitirá a todos las partes interesadas (subscritas)
- - Actualizaciones no esperadas: los observadores no tienen conocimiento de la existencia de los demás, una pequeña operación pueda causar una cascada de actualizaciones innecesarias.

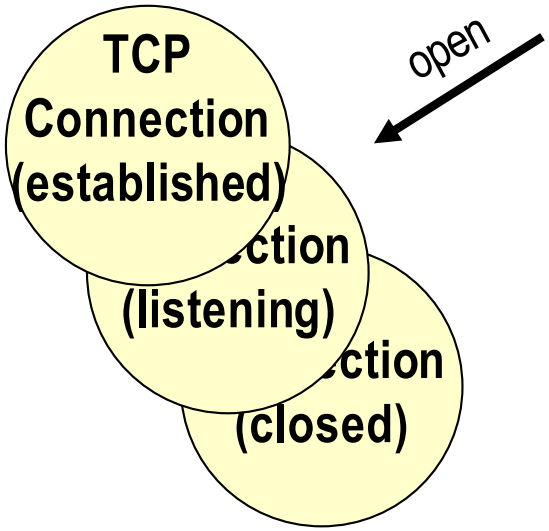

Subject
Attach(Observer o)
Detach(Observer)
Notify()
subjects
chman
ChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
Subject–Observer mapping
observers
Observer
Update(Subject)
chman->Notify()
chman->Register(this, o)
SimpleChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
DAGChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
forall s in subjects
forall o in s.observers
o->Update(s)
mark all observers to update
update all marked observers

State



Permite a un objeto cambiar su comportamiento cuando cambia de estado interno. El objeto parece como que ha cambiado su clase

El Problema



- Objetos que implementan protocolos de conexión reaccionan diferente a los mismos mensajes dependiendo si la conexión es establecida o no.
- Un programa de dibujo interactivo reacciona diferente a un click del ratón dependiendo de la herramienta que esta seleccionada de una paleta.

Aplicabiliad y Solucón del Patrón Estado

- El comportamiento de un objeto depende de su estado y este debe cambiar en tiempo de ejecución.
- Las operaciones tienen grandes condicionales multiparte que dependen del estado del objeto.
- Varias operaciones contienen la misma estructura condicional basada en uno o más constantes enumeradas representando el estado en que esta el objeto.
- El patrón de estado pone cada rama del condicional en una clase diferente de tal manera que un estado del objeto pude ser tratado como un objeto en si mismo que pude variar independientemente de otros objetos

Participantes del Patrón Estado

- Contexto
  - Define la interfase de interés para los clientes
  - Mantiene una instancia de algún EstadoConcreto que define el estado actual
- Estado
  - Define una interfase para encapsular el comportamiento asociado con un estado particular del Contexto
- Subclases de EstadoConcreto
  - Cada subclase implementa un comportamiento asociado con el estado del Contexto

Colaboraciones del Patrón Estado

- Contexto delega peticiones específicas para cada estado para el actual objeto EstadoConcreto
- Un contexto puede pasarse a si mismo como un argumento al objeto Estado que maneja la petición permitiendo que el objeto Estado acceda al Contexto de ser necesario.
- Contexto es la interfaz primaria para los clientes. Los Clientes pueden configurar el contexto con objetos Estados. Una vez que el contexto está configurado, el cliente no tiene que tratar con el objeto Estado directamente
- Ya sea el Contexto o las subclases del EstadoConcreto pueden decidir que estado sigue a otro.

Consecuencias del Patrón Estado

- + Localiza el comportamiento específico para el estado y particiona el comportamiento en diferentes estado de tal manera que nuevos estados y transiciones pueden ser añadidos fácilmente con solamente añadir una nueva subclase de EstadoConcreto
- +/- Procedimientos largos conteniendo un sentencias condicionales largas son evitadas, pero el número de clases incremente y el todo es menos compacto que la clase simple.
- + Hace las transiciones de estado explícita y protege el contexto de inconsistencias internas debido a que la actualización de estado es atómica.
- + Cuando el objeto Estado no necesita variables de instancia pueden ser compartidos; son esencialmente flyweights sin estado intrínseco y solo comportamiento.