Tema 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

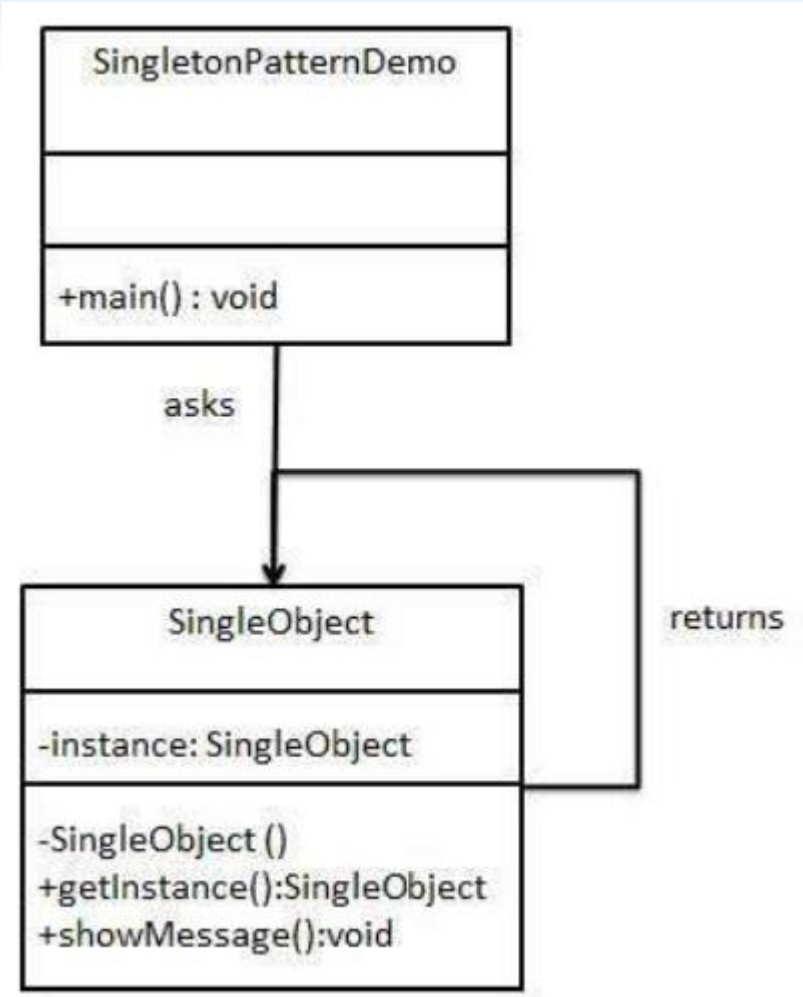
Resumen Patrones

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

Patrones de Diseño Creacionales

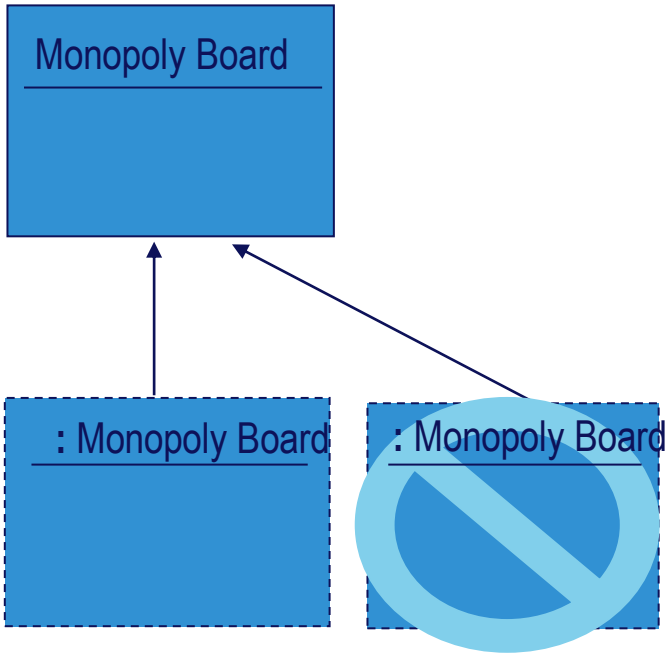
- Abstraen el proceso de instanciación:
  - Encapsulan el conocimiento acerca de que clase concreta se usa
  - Esconde como las instancias de estas clases son creadas y unidas
- Da gran cantidad de flexibilidad en que se crea, quien lo crea, como es creado, y cuando es creado.
- Un patrón de objeto creacional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

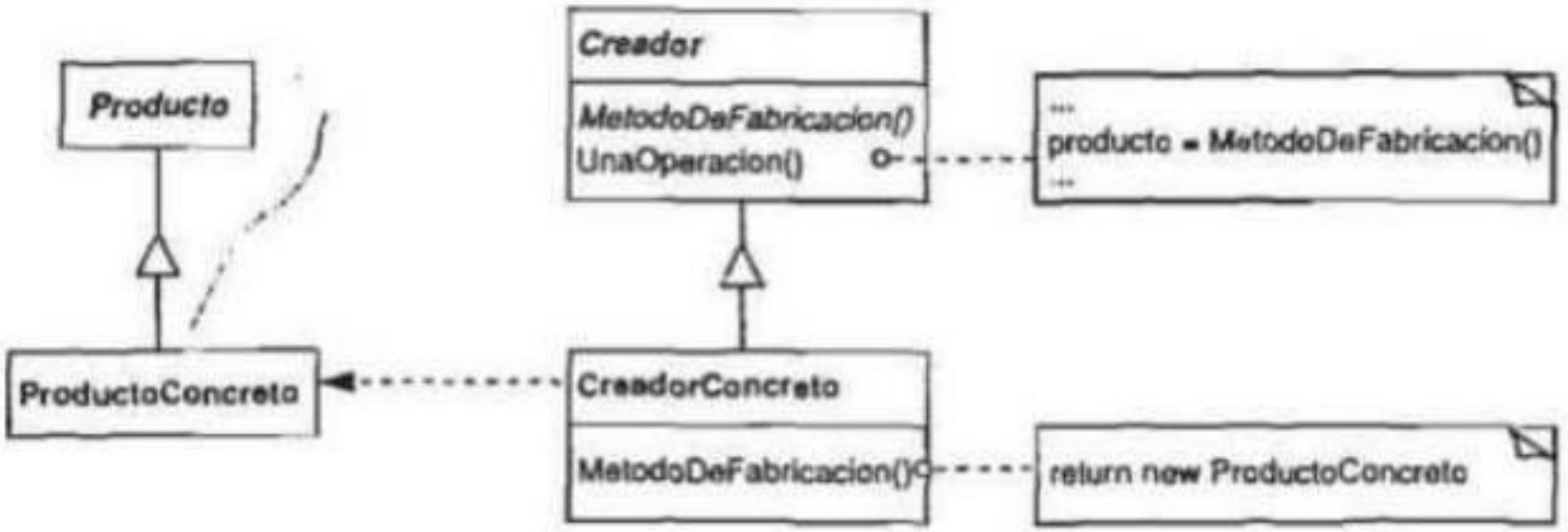
Colaboradores

- 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

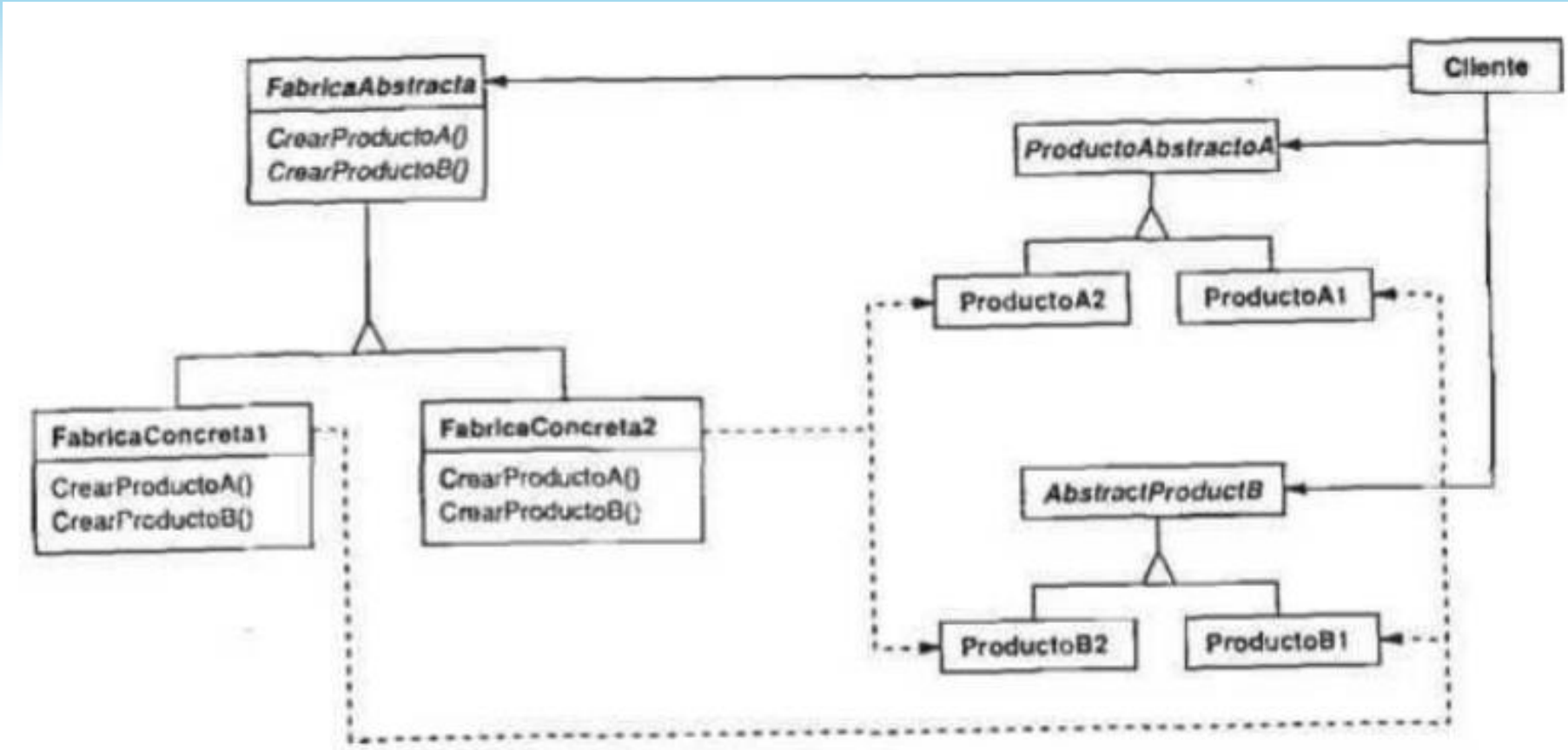
Colaboración

- 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

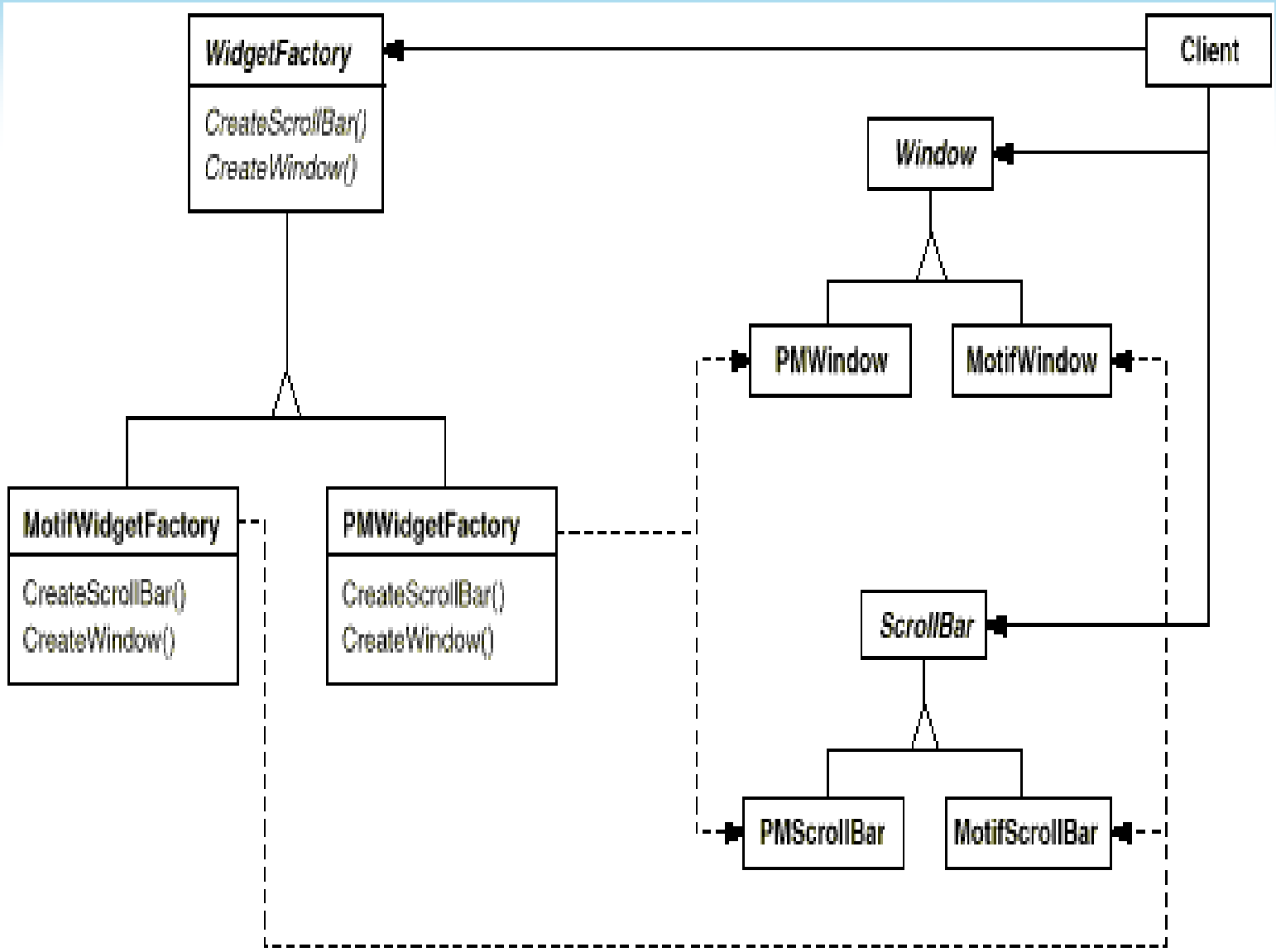
Colaboración

- 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.

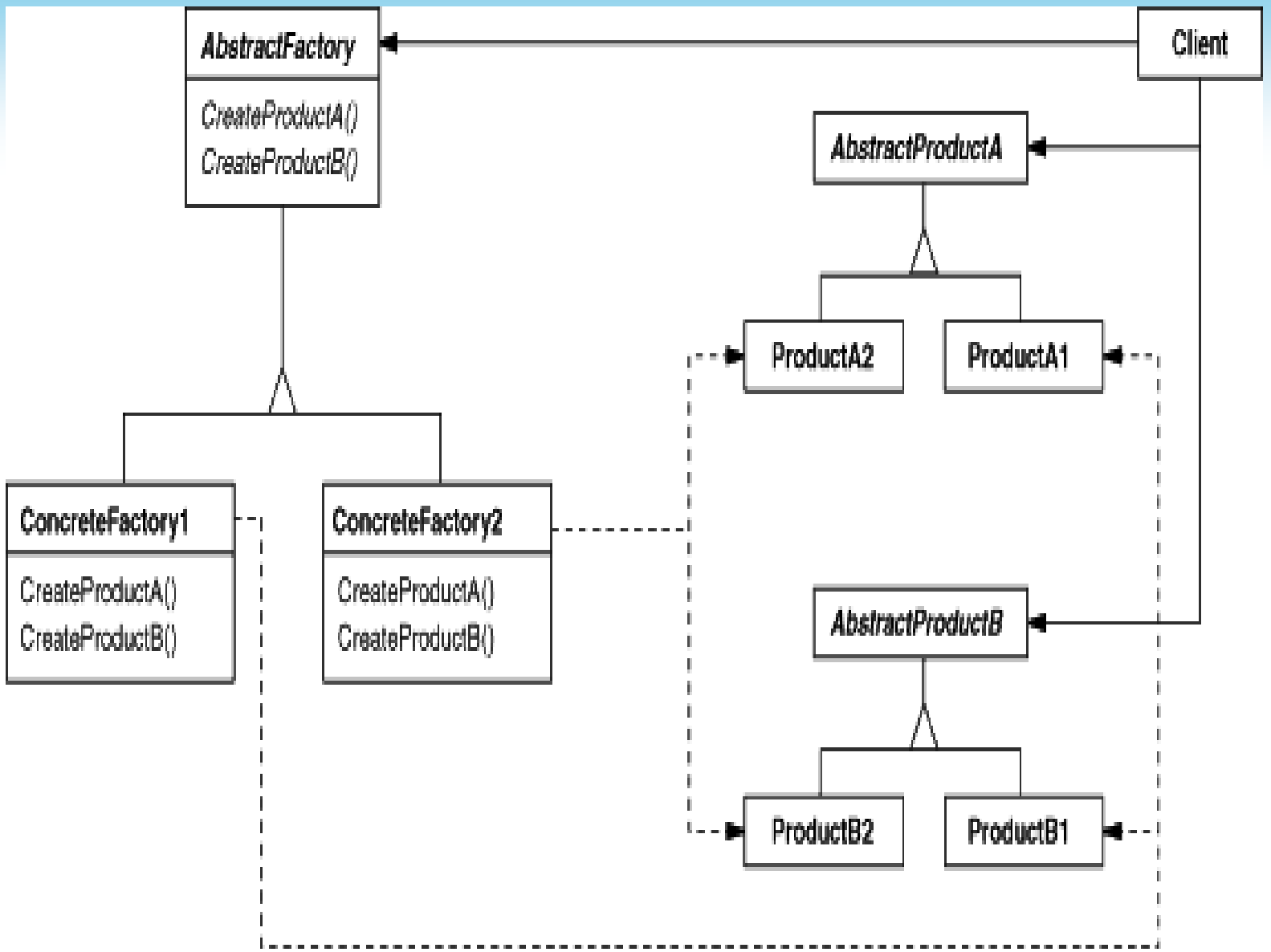
Aplicación

- 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

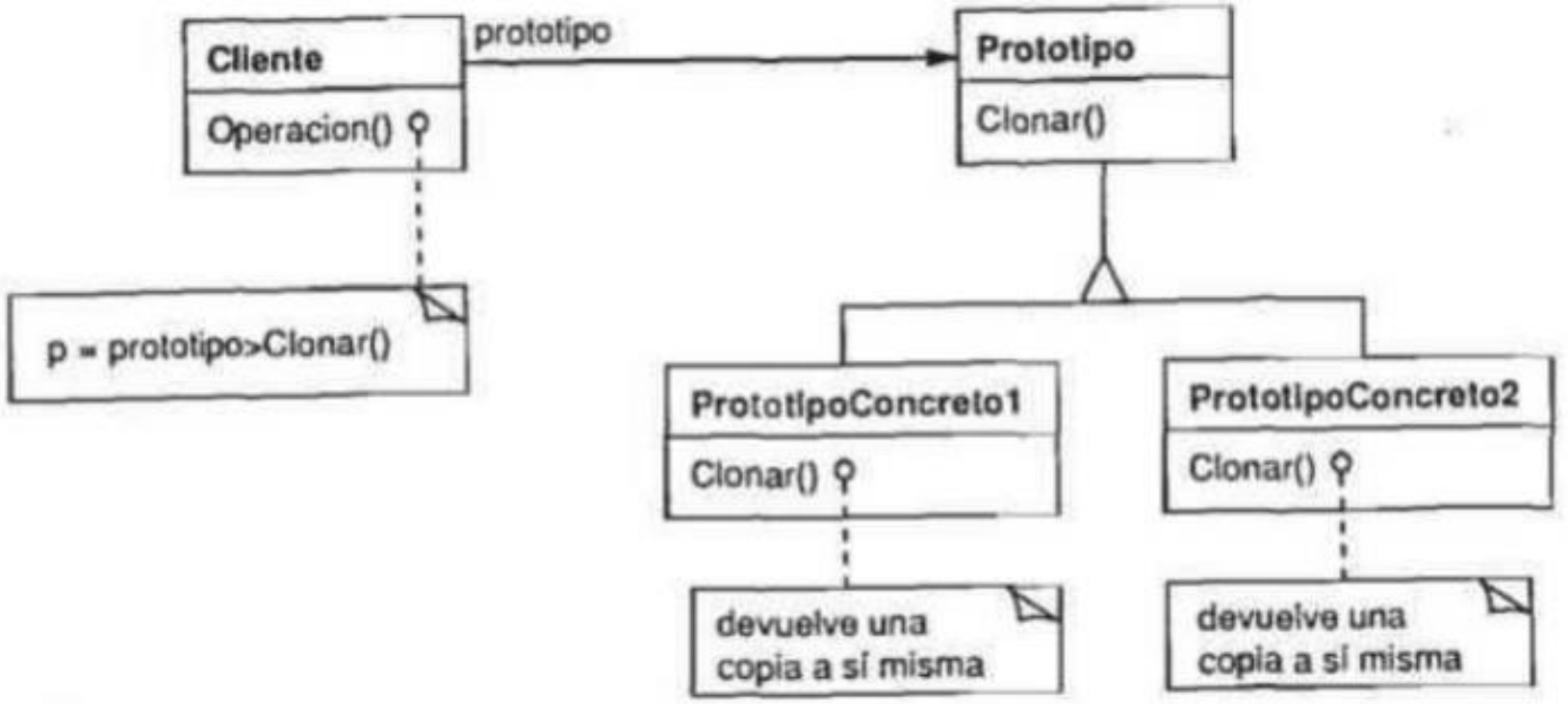
Consecuencias

- 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.

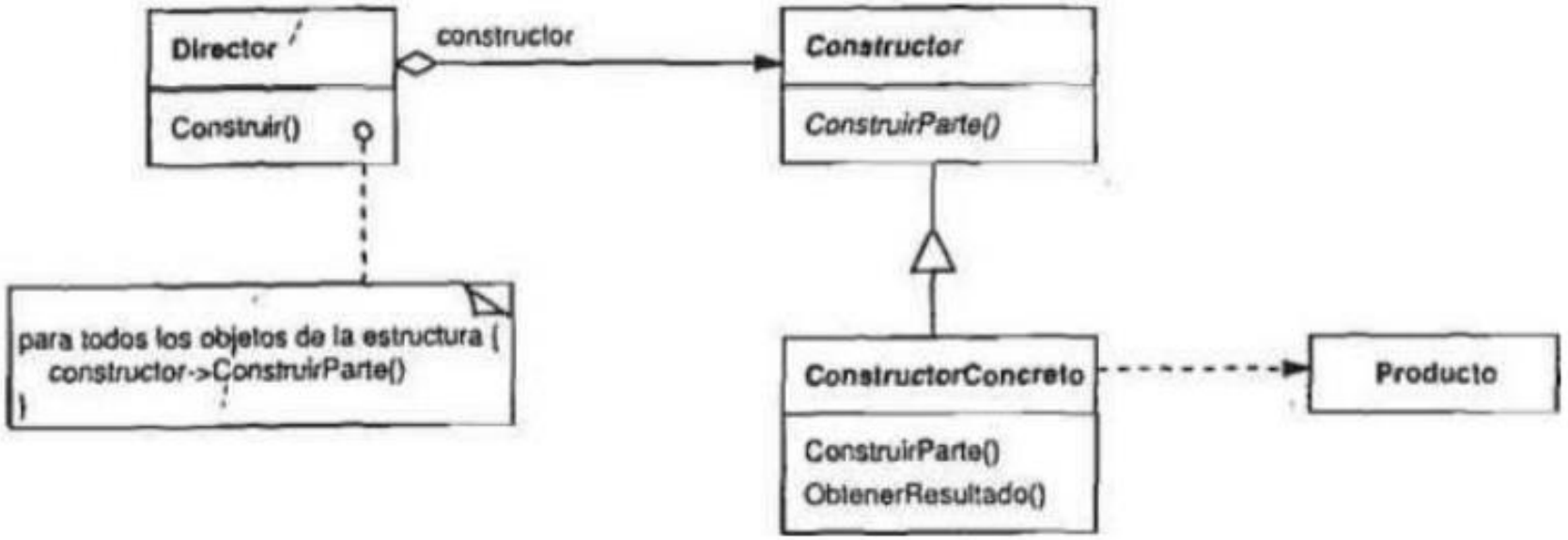
Consecuencias

- + 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

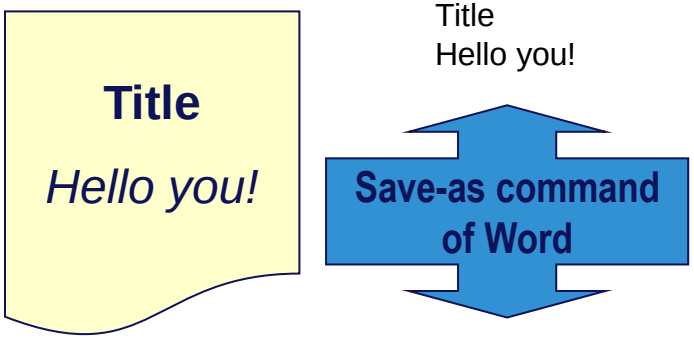
- + Reduce las subclases: al contrario que el Método Fábrica, que a menudo produce una jerarquía de clase creadora que asemeja al jerarquía de ProductosConcretos
- + Configura una aplicación con clases dinámicamente: cuando el ambiente de tiempo de ejecución soporta carga dinámica de clases, el patrón 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

Builder



Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Problema



- Un lector RTF que puede convertir en varios formatos

- Un parser que produce un árbol de parsing complejo

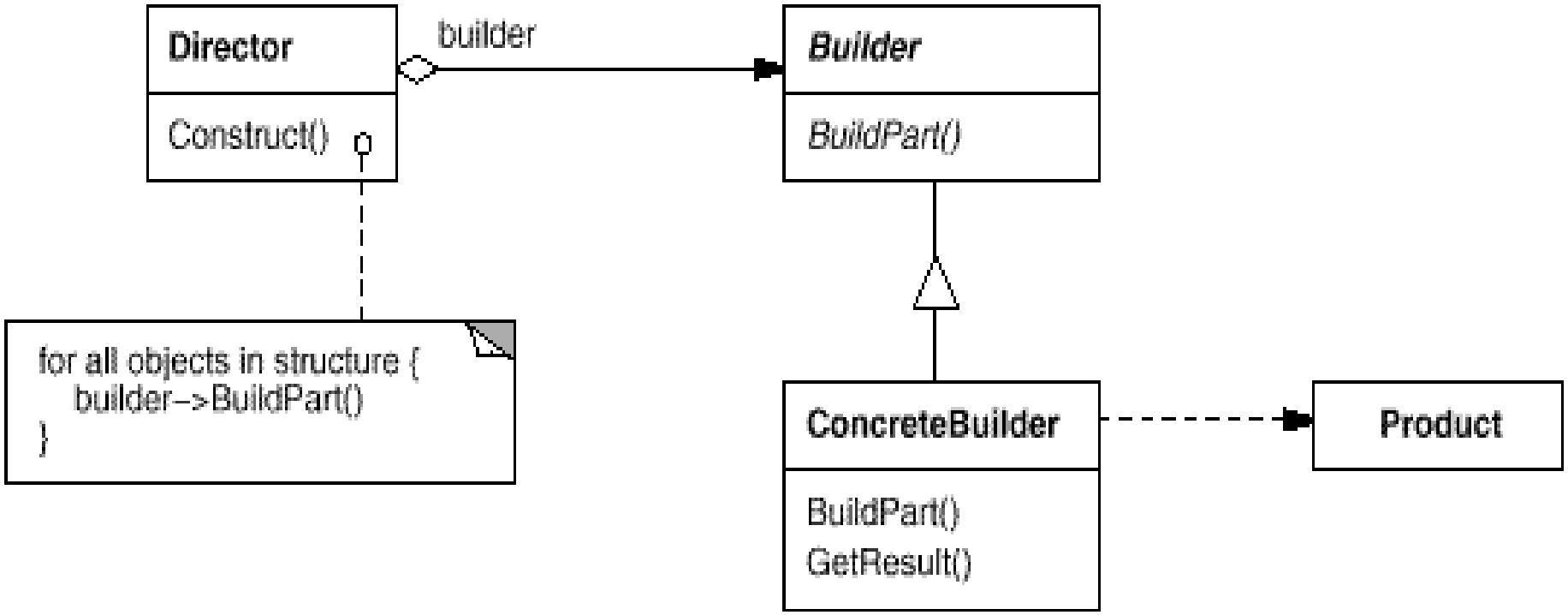
```
<B><FONT FACE="Arial" SIZE=6><P>Title</P>
</B></FONT><FONT FACE="Times">
</FONT><I><FONT FACE="Arial"><P>Hello you!</P></I></FONT></BODY>
</HTML>
```

Consecuencias del Patrón Constructor

- + Permite variar la representación interna del producto: los directores usan la interfaz abstracta provista por el constructor para construir el producto; para cambiar la representación del producto, solo se debe hacer un nuevo tipo de constructor.
- + Permite el reuso de los Constructores Concretos: todo el código para la construcción y representación esta encapsulado; diferentes directores usan el mismo Constructor Concreto
- + Da un control más fino sobre el proceso de construcción: En otros patrones creacionales, la construcción se realiza en un solo paso; aquí el producto es construido paso a paso y bajo la guía del director dando un control más fino sobre la

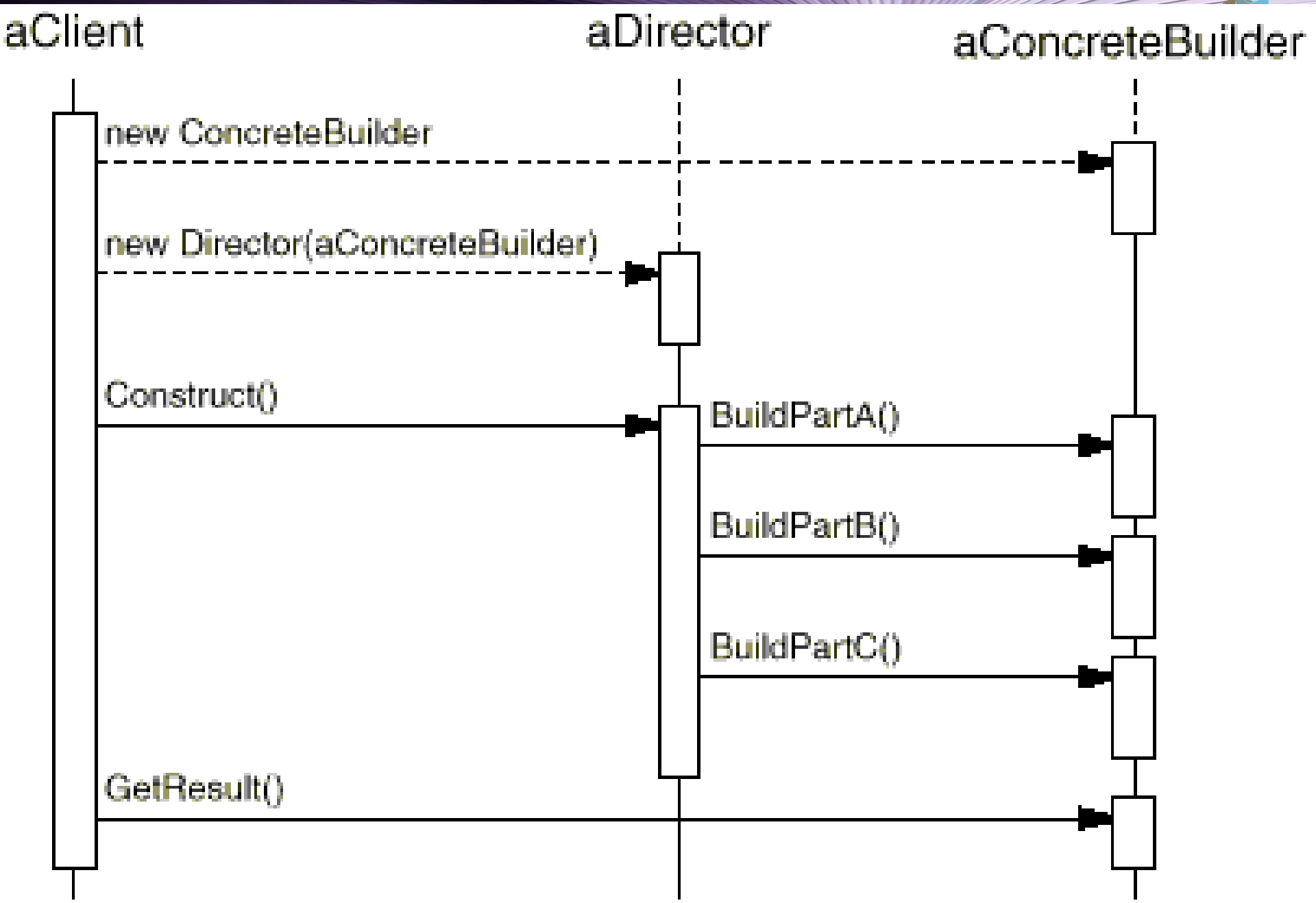
Participantes

- Constructor: especifica una interfaz abstracta para crear partes de un Producto
- ConstructorConcreto:
  - Construye y ensambla las partes de un Producto al implementar la interfase Constructor
  - Define y sigue la pista de la representación que crea
  - Provee una interfase para recuperar el producto
- Director: construye un objeto usando la interfaz del Constructor
- Producto:
  - Representa el objeto complejo en construcción
  - Incluye clases que definen las partes constituyentes incluyendo las interfaces para ensamblar las partes en un resultado final


Director
builder
Builder
Construct()
BuildPart()
for all objects in structure {
builder->BuildPart()
}
ConcreteBuilder
Product
BuildPart()
GetResult()

Colaboración

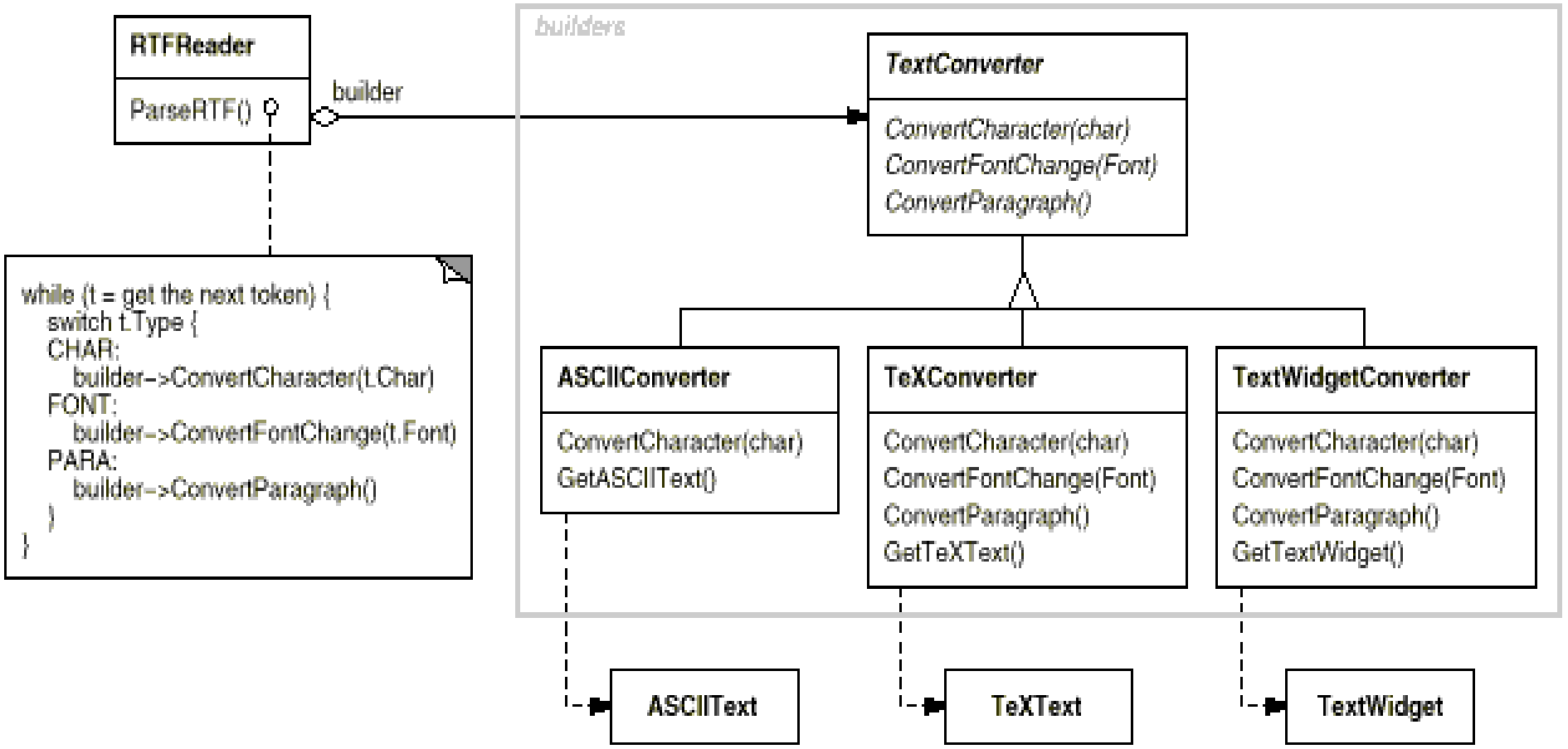
- El cliente crea al objeto Director y lo configura con el objeto Constructor deseado.
- El Director notifica al constructor que parte del producto debe ser construida
- El constructor maneja los requerimientos del director y añada partes al producto
- El cliente recupera el producto del constructor


aClient
aDirector
aConcreteBuilder
new ConcreteBuilder
new Director(aConcreteBuilder)
Construct()
BuildPartA()
BuildPartB()
BuildPartC()
GetResult()

Cuando utilizar

- Independizar el algoritmo de creación de un objeto complejo de las partes que constituyen el objeto y cómo se ensamblan entre ellas.
- Que el proceso de construcción permita distintas representaciones para el objeto construido, de manera dinámica.

Ejemplo


RTFReader
ParseRTF()
builder
while (t = get the next token) {
switch t.Type {
CHAR:
builder->ConvertCharacter(t.Char)
FONT:
builder->ConvertFontChange(t.Font)
PARA:
builder->ConvertParagraph()
}
}
builders
TextConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
ASCIIConverter
ConvertCharacter(char)
GetASCIIText()
TeXConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTeXText()
TextWidgetConverter
ConvertCharacter(char)
ConvertFontChange(Font)
ConvertParagraph()
GetTextWidget()
ASCIIText
TeXText
TextWidget

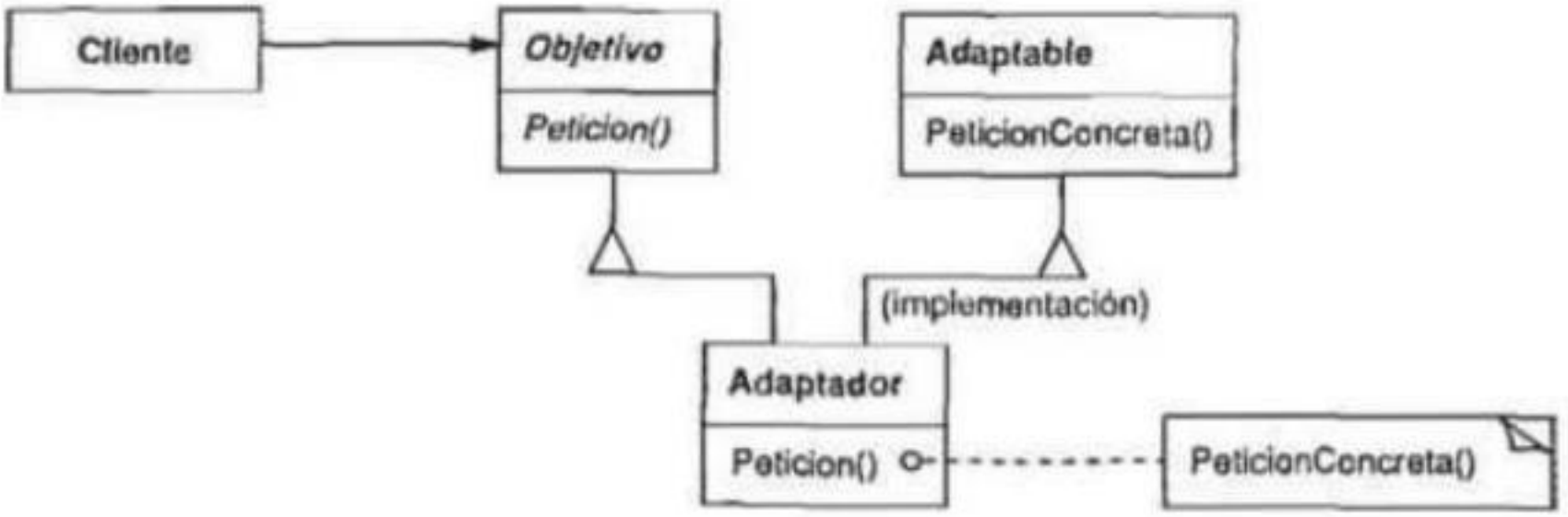
Concecuencias

- *Permite variar la representación interna de un producto.*
- *Aísla el código de construcción y representación.*
- El patrón Builder aumenta la modularidad al encapsular cómo se construyen y se representan los objetos complejos.
- Los clientes no ne- cesitan saber nada de las clases que definen la estructura intema del producto; dichas clases no aparecen en la interfaz del Constructor.

Patrones de Diseño Estructural

- Los patrones de diseño estructurales se preocupan de como las clases y objetos se unen para formar estructuras más grandes.
- Un patrón de clase estructural usa la herencia para componer interfaces o implementación; la composición es fijada en tiempo de diseño
- Un patrón de objeto estructural describe la manera de componer los objetos para crear nueva funcionalidad; la flexibilidad añadida a la composición de objetos viene de la agilidad de cambiar la composición en tiempo de ejecución

Adapter (Wrappers)



Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes. Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles.

Participantes

- Objetivo
  - Define la interfaz específica del dominio que usa el Cliente.
- Cliente
  - Colabora con objetos que se ajustan a la interfaz Objetivo.
- Adaptable
  - Define una interfaz existente que necesita ser adaptada.
- Adaptador
  - Adapta la interfaz de Adaptable a la interfaz Objetivo.

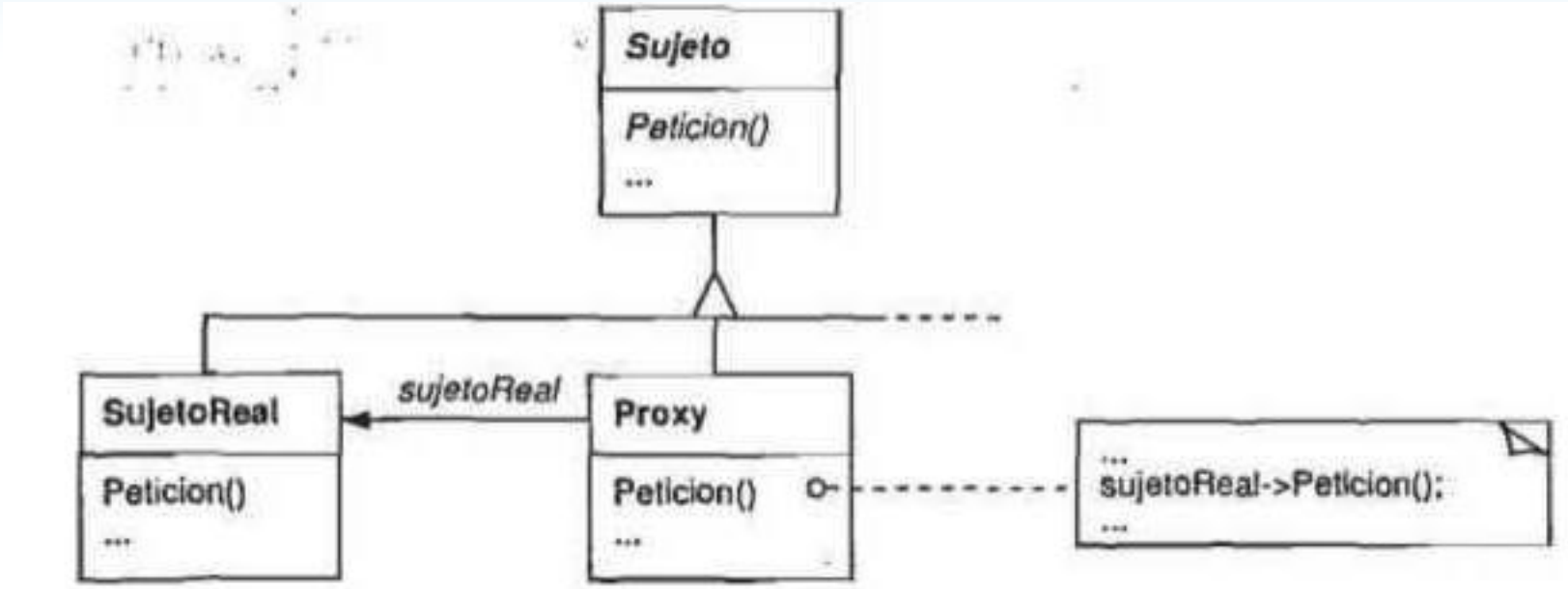
Colaboración

- Los clientes llaman a operaciones de una instancia de Adaptador. A su vez, el adaptador llama a operaciones de Adaptable, que son las que satisfacen la petición.

Consecuencias

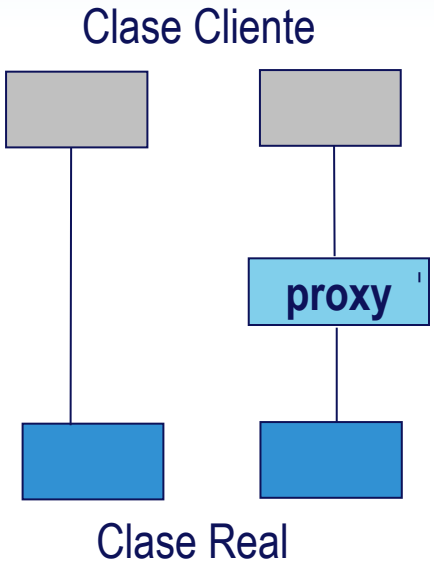
- Adapta una clase Adaptable a Objetivo, pero se refiere únicamente a una clase Adaptable concreta. Por tanto, un adaptador de clases no nos servirá cuando lo que queremos es adaptar una clase *y* todas sus subclases.
- Permite que Adaptador redefina parte del comportamiento de Adaptable, por ser Adaptador una subclase de Adaptable.
- Introduce un solo objeto, y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado.

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

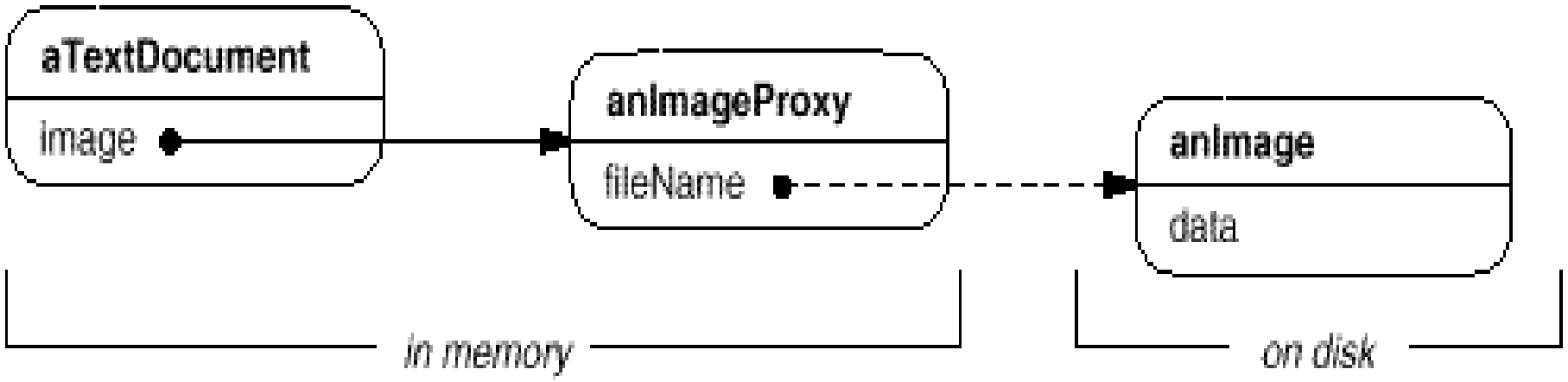
Participantes

- 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

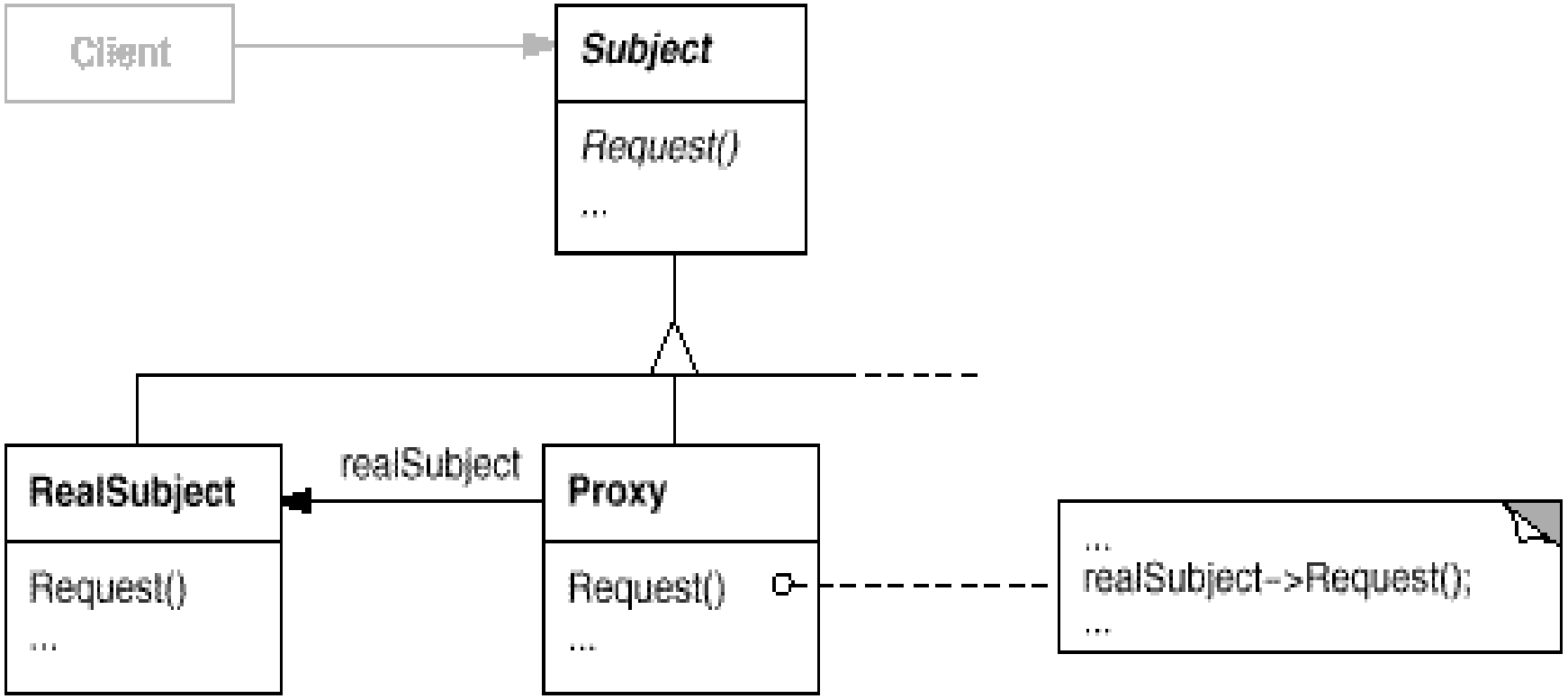
Colaboración

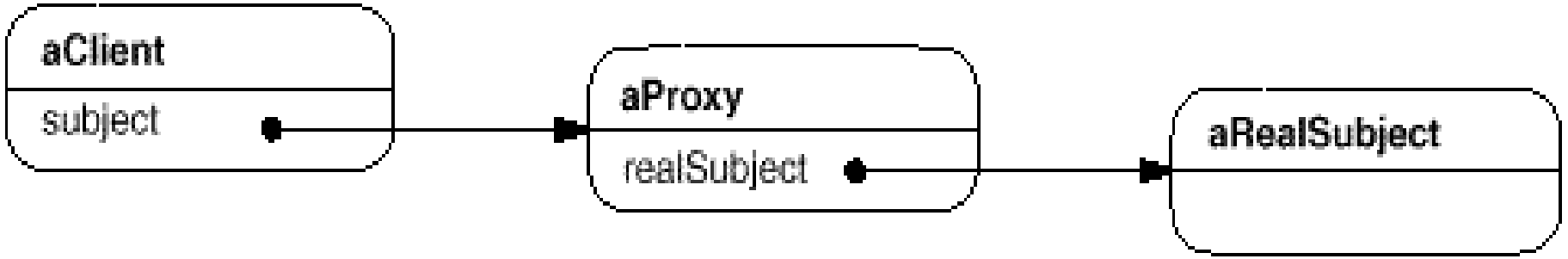
- El Proxy envía la petición al SujetoReal cuando es apropiado (depende del tipo de proxy)

Ejemplo1


aTextDocument
image
anImageProxy
fileName
anImage
data
in memory
on disk

Ejemplo 2

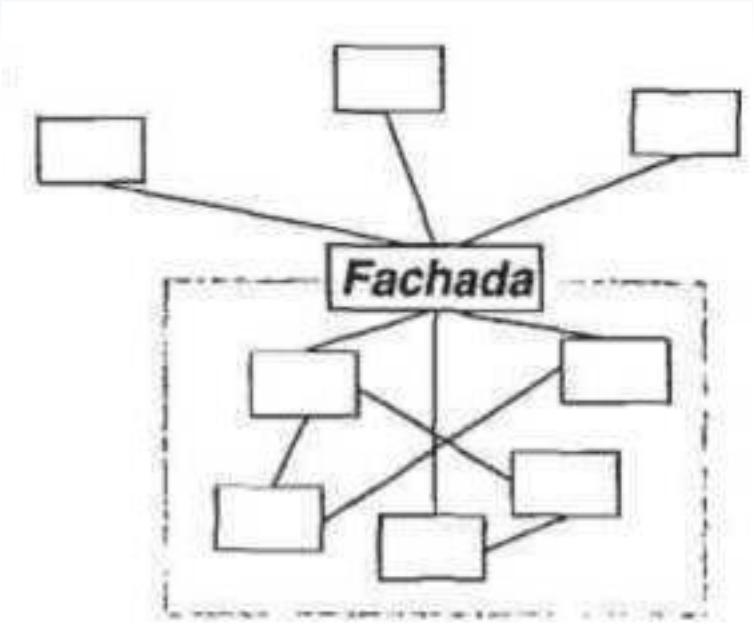

Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...


aClient
subject
aProxy
realSubject
aRealSubject

Consecuencias de Patrón Proxy

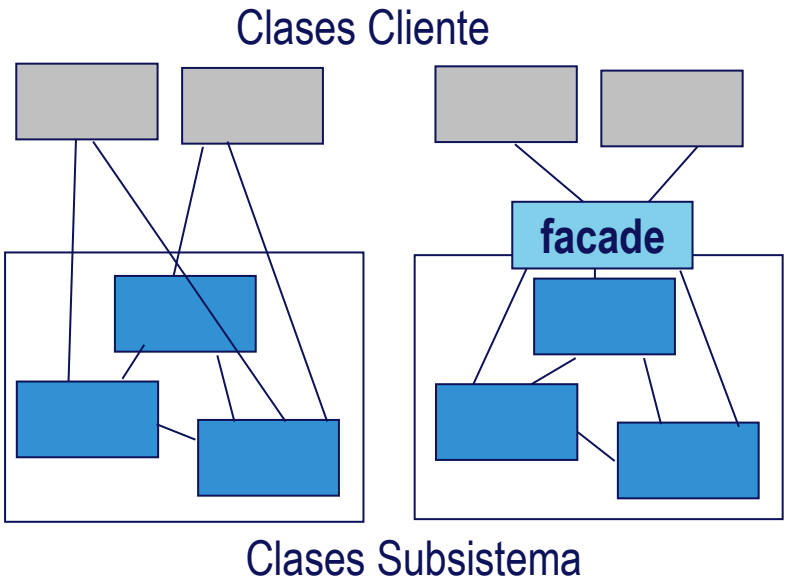
- El patrón Proxy introduce un nivel de indirección cuando se accede al objeto. Esta indirección tiene varios usos:
  - Un proxy remoto puede esconder el hecho de que un objeto reside en un espacio de direcciones diferente
  - Un proxy virtual puede ejecutar optimizaciones
  - Tanto el proxy de protección 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.**

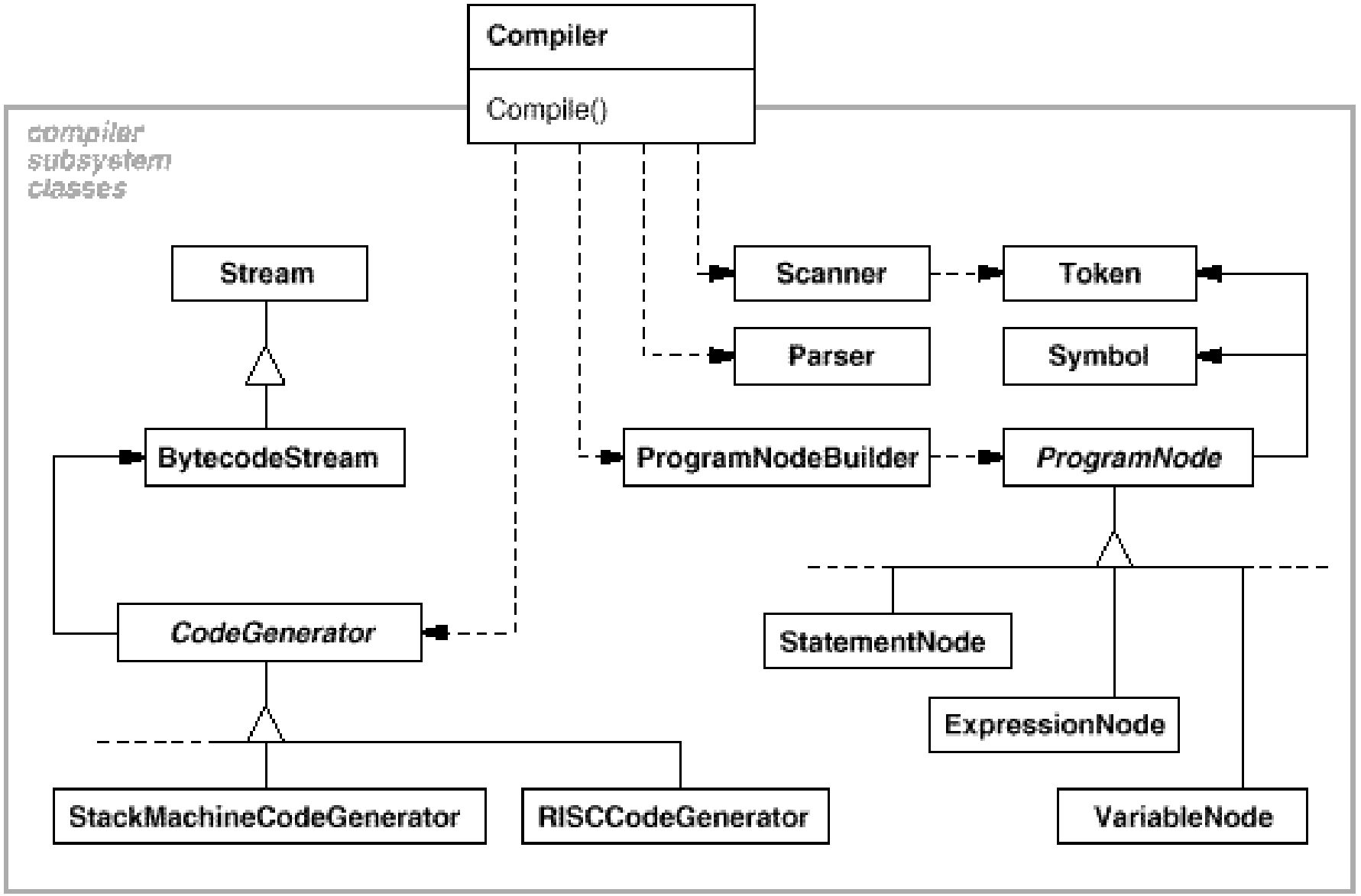
Participantes

- 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.

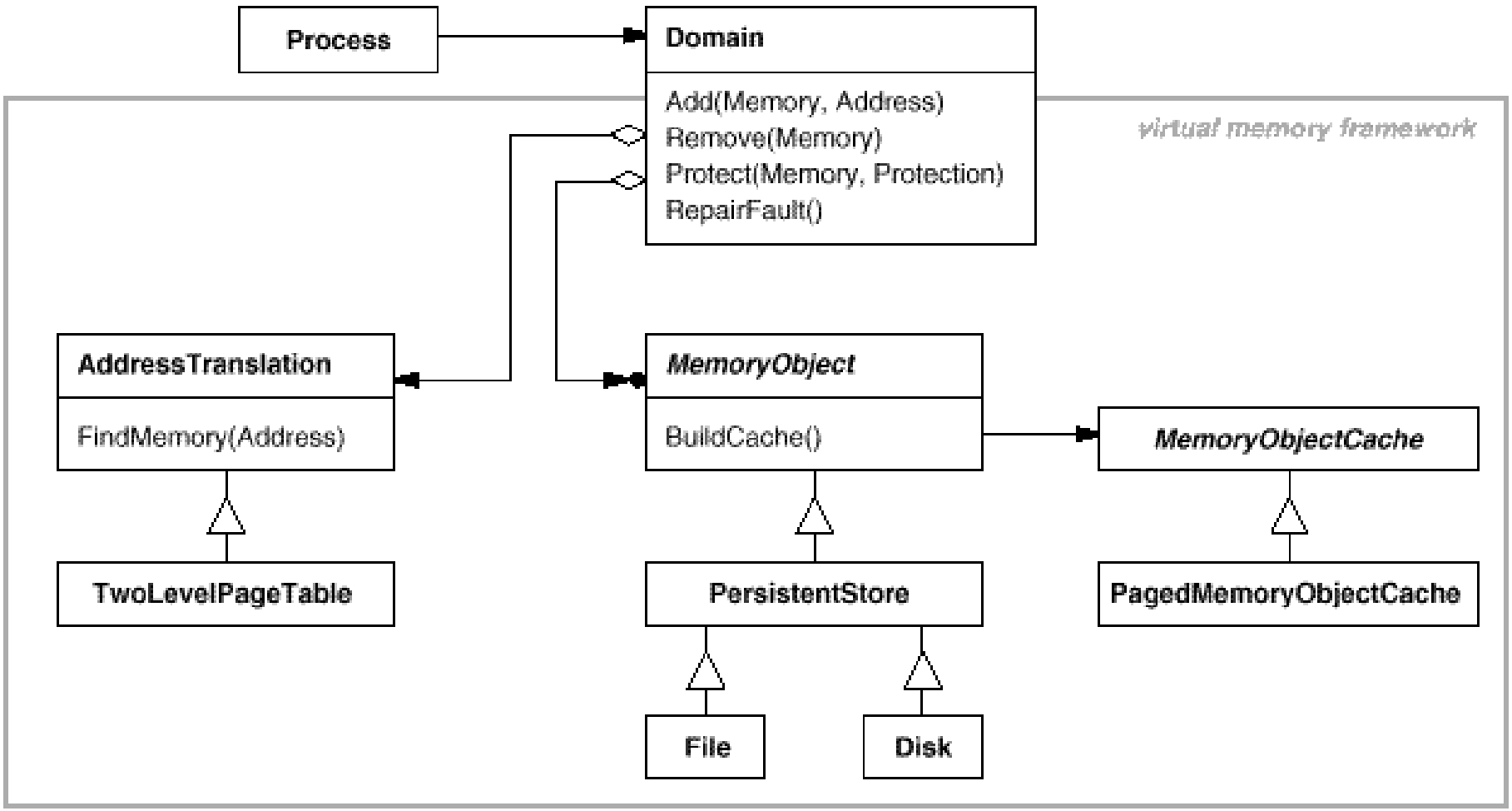
Colaboración

- 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

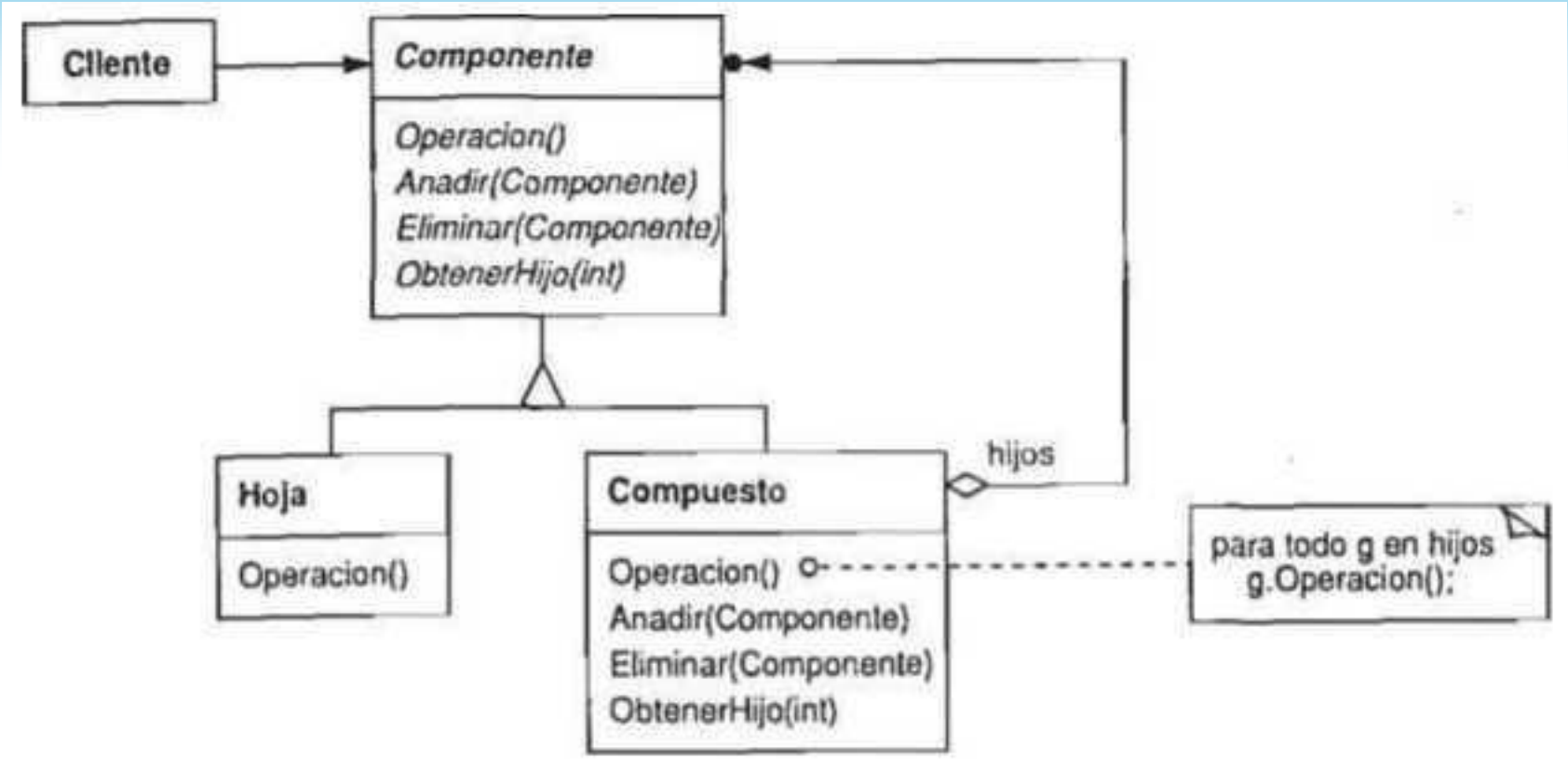
Ejemplo 2


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

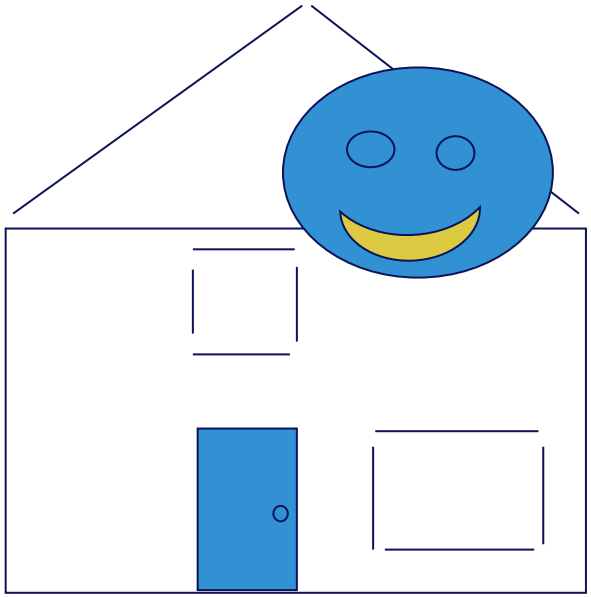
- + Escuda a los clientes de los componentes del subsistema reduciendo el número de objetos con que los clientes tienen que trabajar, haciendo al subsistema más fácil de usar
- + Promueve el aparejamiento débil entre el subsistema y sus clientes, permitiendo que los componentes del subsistema cambien 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

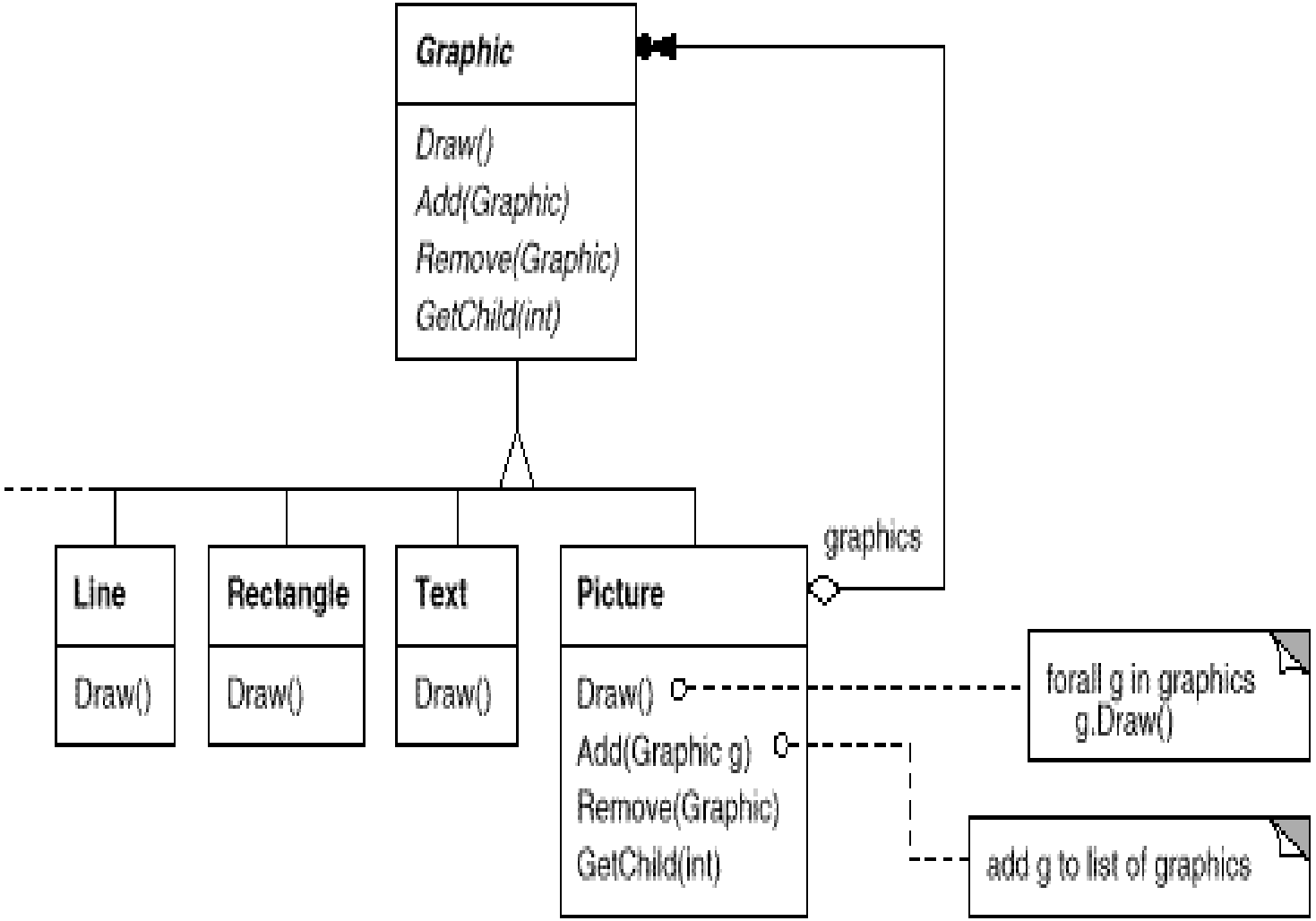
Participantes

- 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.

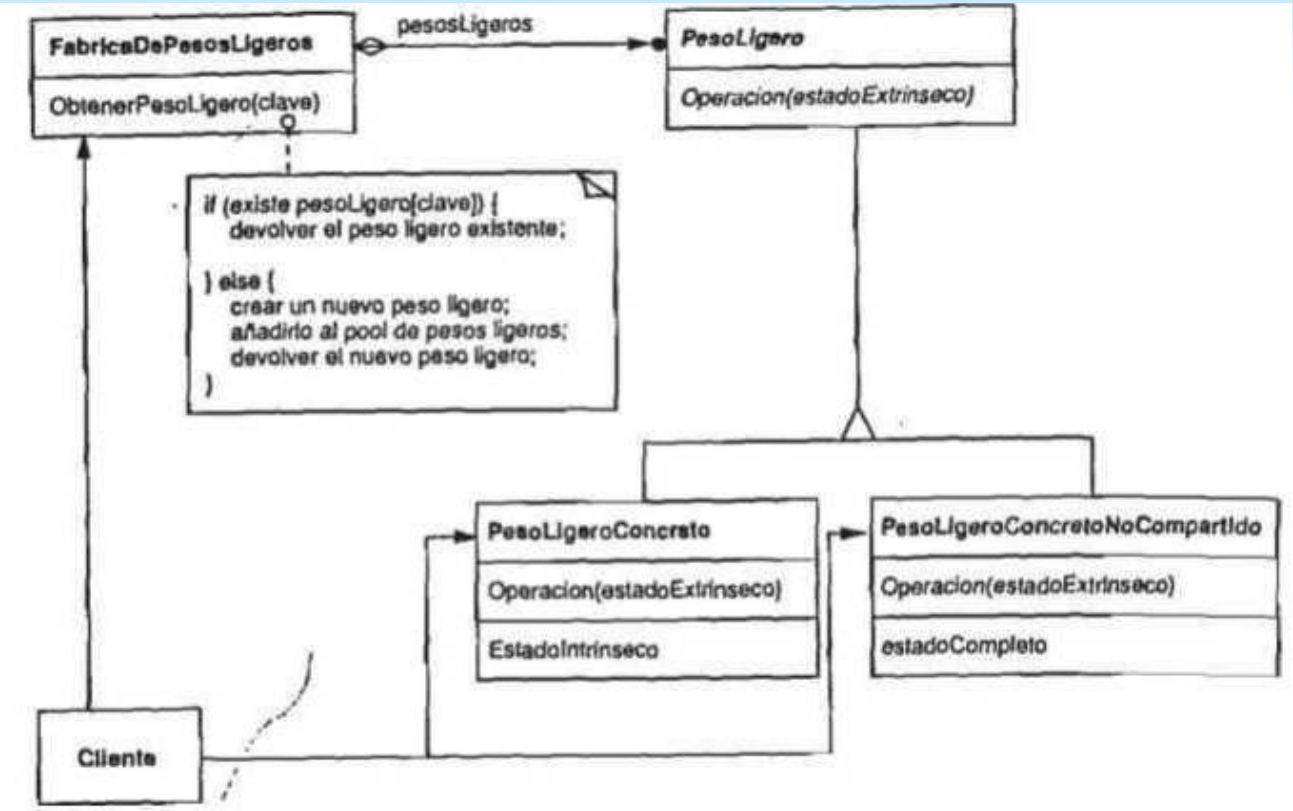
Ejemplo


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.

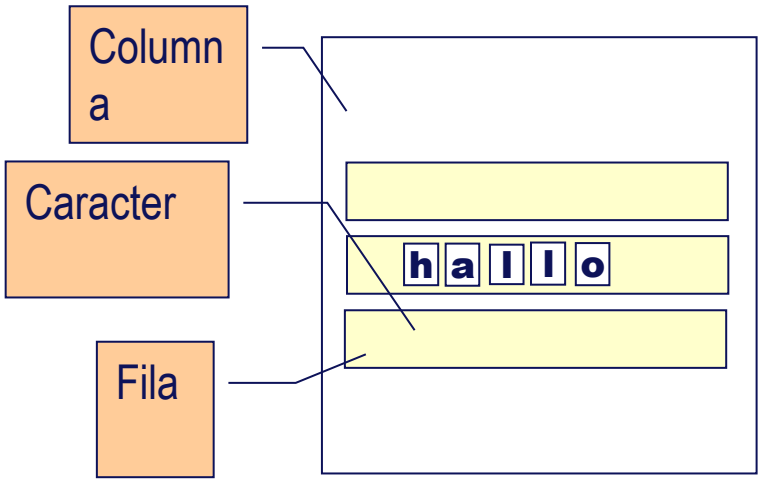
Flyweight



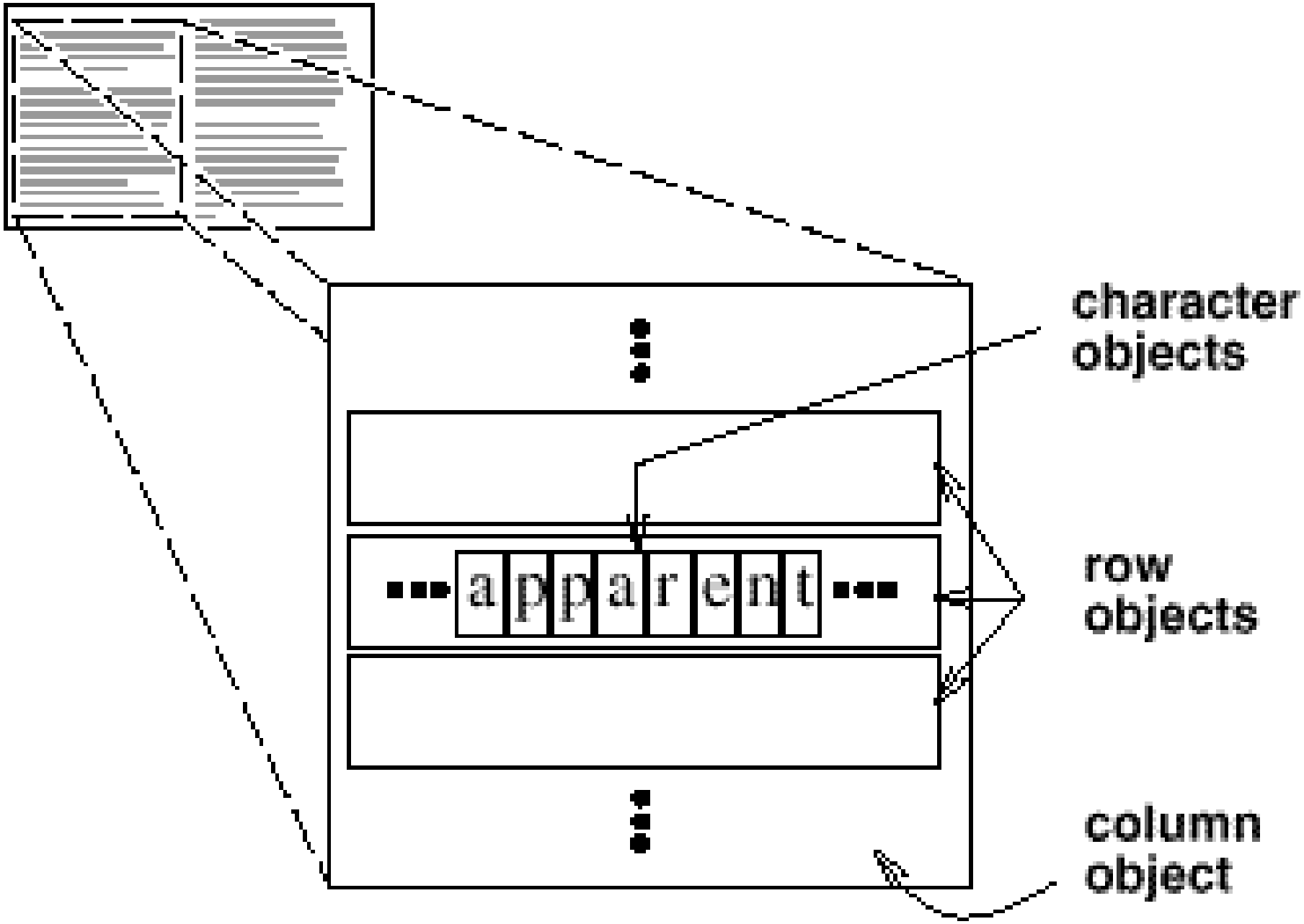
Usa compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

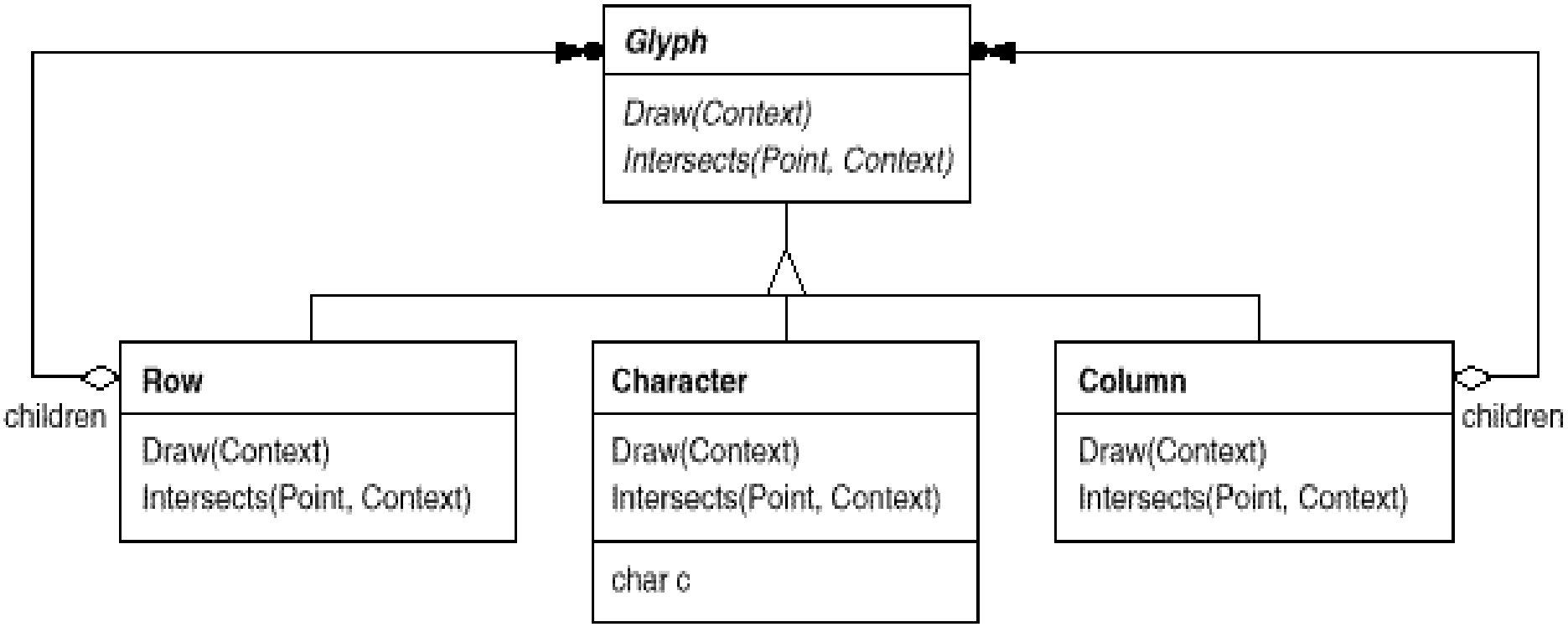
El Problema

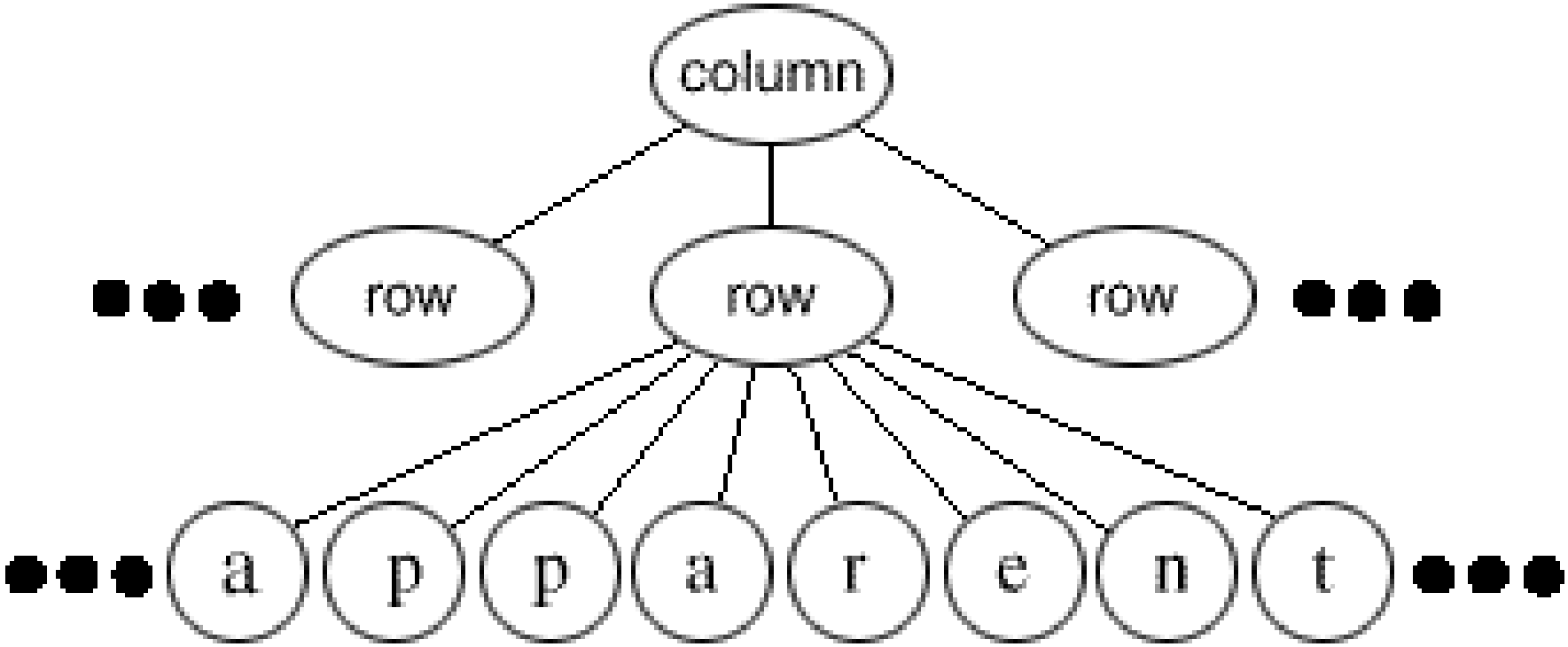
Algunas aplicaciones se benefician del uso de objetos en su diseño,pero implementaciones simples son prohibitivamente caras por el gran número de objetos

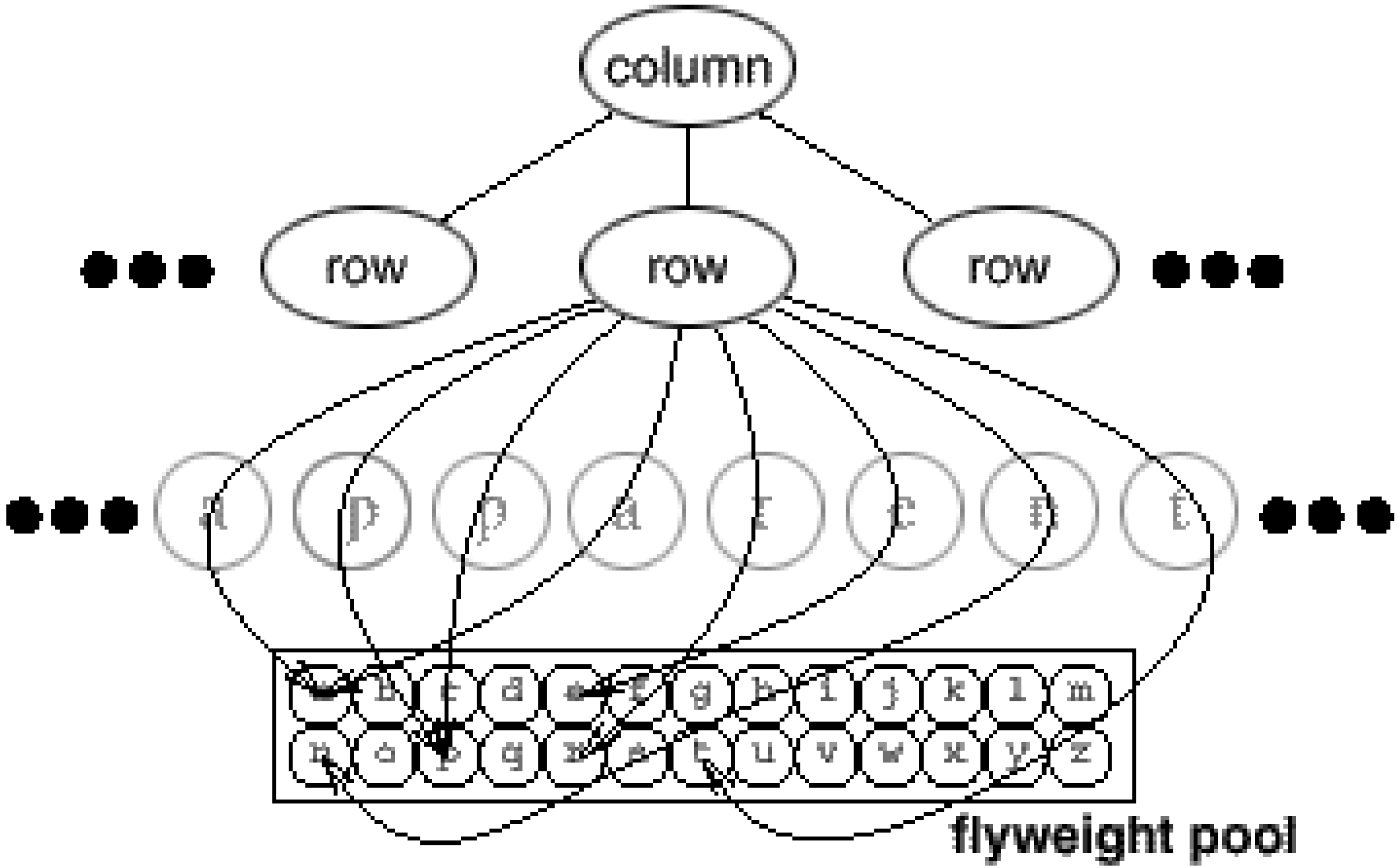


- **Use un objeto para cada caracter en el texto en el editor de texto**
- **Use un objeto posiciòn para cada elemento del GUI**


character objects
row objects
column object
apparent


Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children
children


column
row
row
row
a
p
p
a
r
e
n
t


column
row
row
row
flyweight pool

PesoMosca

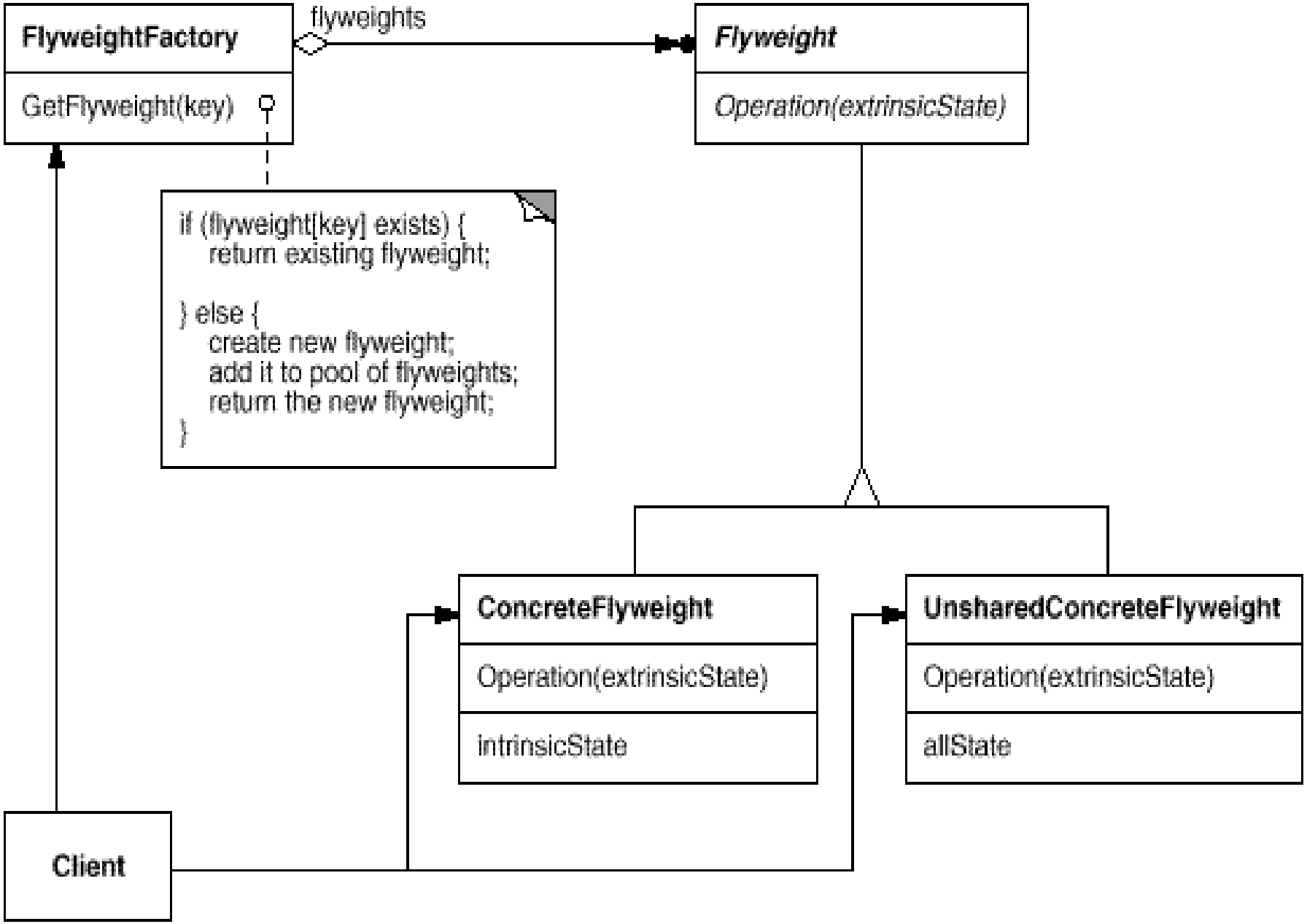
- Aplicar peso mosca si Todo lo siguiente es verdad:
  - Una aplicación usa un número grande de objetos
  - El costo de almacenar es alto debido a la cantidad de objetos
  - La mayoría de los objetos puede ser hecho intrinseco
  - Most objects can be made extrinsic
  - Muchos grupos de objetos pueden ser remplazados con relativamente pocos objetos compartidos una vez que su estado extrínseco es removido.
  - Las aplicaciones no dependen de la identidad del objeto

Participantes

- Flyweight
  - Declara una interfase a través de la cual flyweights pueden recibir y actuar según su estado extrínseco
- Concrete Flyweight
  - Implementa la interfaz del flyweight y añade almacenamiento para el estado intrínseco.
  - Un objeto flyweight concreto debe ser compartible; Ej. Dos estado debe ser intrínsico
- Unshared Concrete Flyweight
  - No todos las subclases flyweights necesitan ser compartidas, flyweight concretos no compartidos pueden tener objetos flyweight concretos a algún nivel.

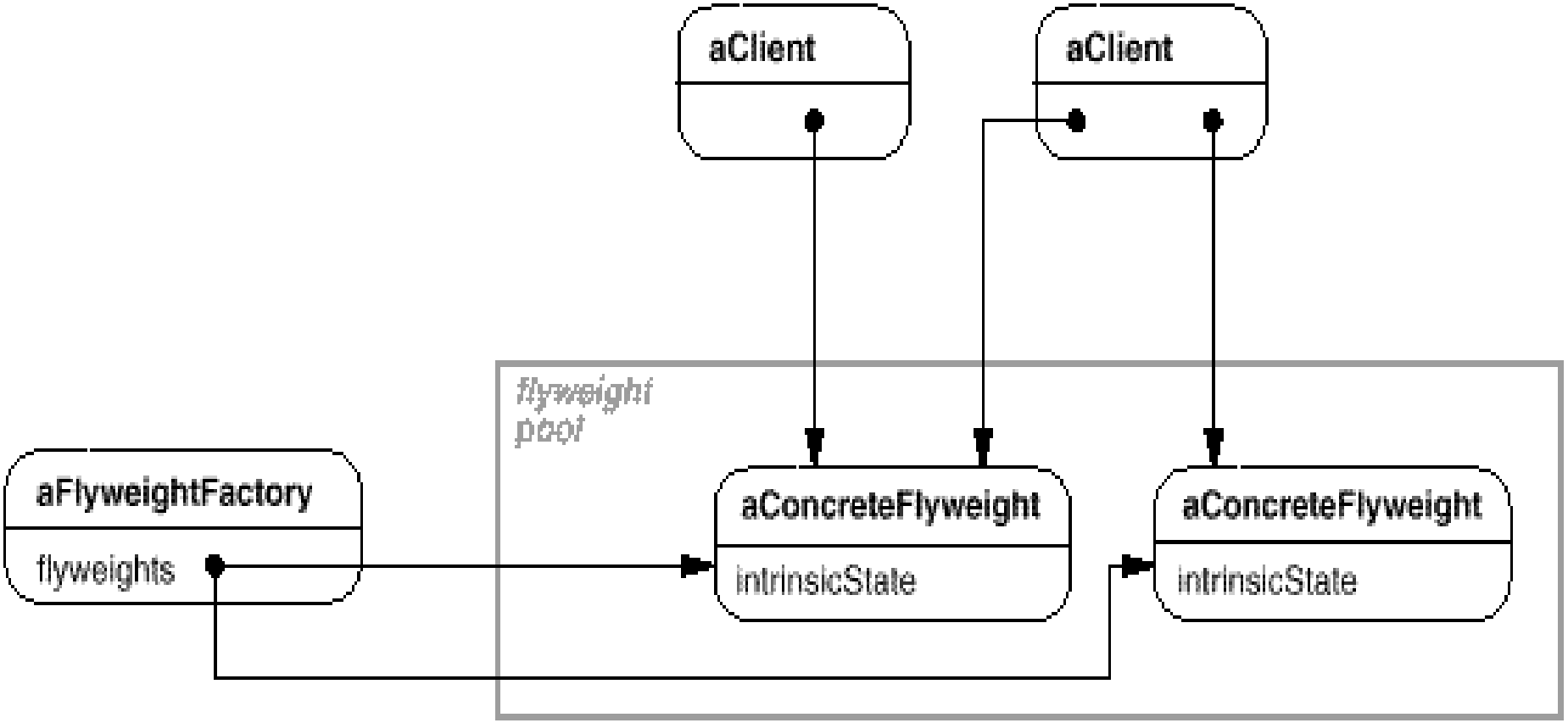
Participantes

- Fábrica Flyweight
  - Crea y maneja objetos flyweight
  - Asegura que los flyweights son compartidos adecuadamente; cuando un cliente pide un flyweight la fábrica flyweight devuelve un objeto existen de un repositorio o crea uno y lo añade al repositorio
- Cliente
  - Mantiene una referencia hacia el o los flyweights
  - Calcula o guarda el estado extrínseco de los flyweights


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Colaboraciones

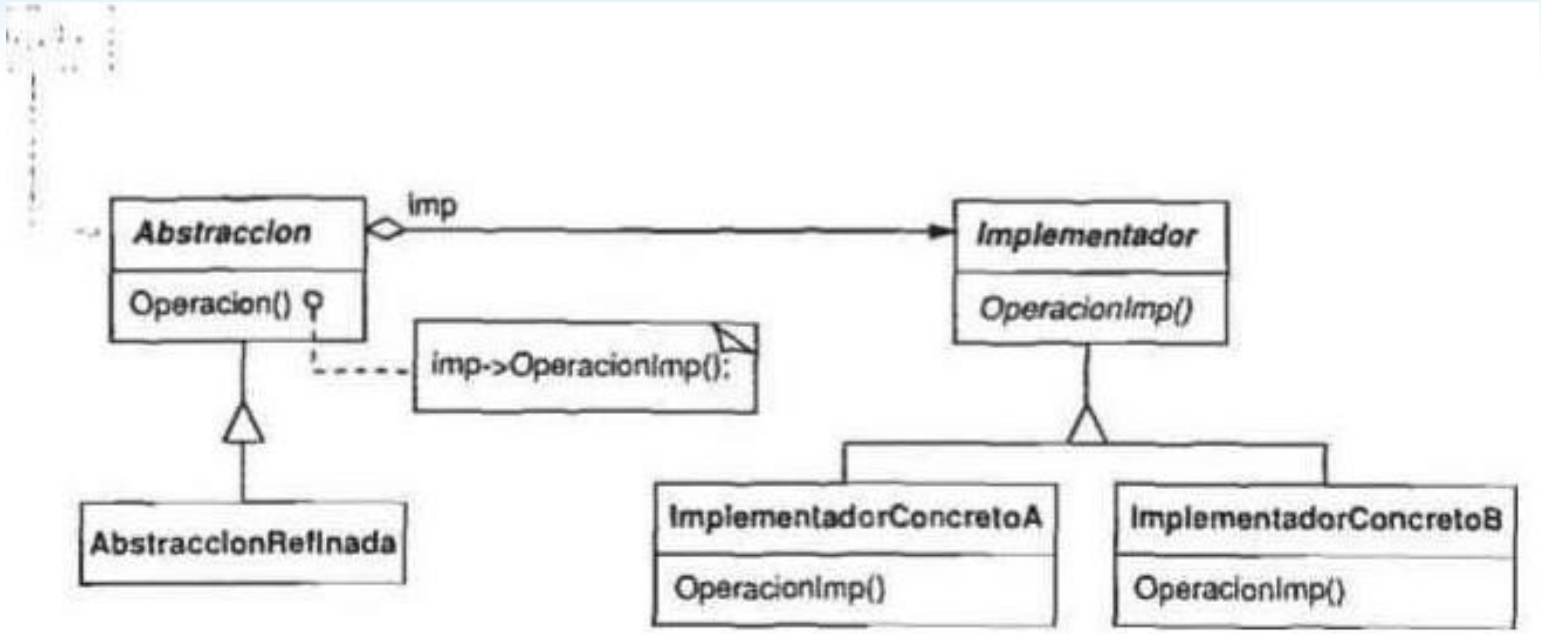
- El estado que un flyweight necesita para funcionar debe ser caracterizado como intrínseco o extrínseco. El estado intrínseco es guardado en el objeto flyweight concreto; el extrínseco esta guardado o computado por los objetos clientes. El cliente pasa este estado al flyweight cuando invoca operaciones.
- Los clientes no deben instanciar los flyweights concretos directamente. El cliente debe obtener los objetos flyweight concretos exclusivamente de la fábrica de flyweight para asegurar que se compartan adecuadamente


aClient
aClient
flyweight
pool
aFlyweightFactory
flyweights
aConcreteFlyweight
intrinsicState
aConcreteFlyweight
intrinsicState

Consecuencias

- - Los Flyweights pueden introducir costos en tiempo de ejecución asociados con la transferencia, búsqueda, y/o cálculo del estado extrínseco.
- + El incremento de los costos de tiempo de ejecución son compensados por ahorros de almacenamiento, el cual decrece
  - Tanto cuanto más flyweights son compartidos
  - Tanto cuanto la cantidad de estado intrínseco sea considerable
  - Tanto cuanto la cantidad de estado extrínseco es considerable pero puede ser calculado.
- - El patrón flyweight se combina con el patrón composición para construir un grafo con varios nodos hoja. Debido a la compartición los nodos hojas no pueden guardar a su padre lo cual ocasiona un impacto mayor en como los objetos de la jerarquía se comunican.

Bridge



Desacopla una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.

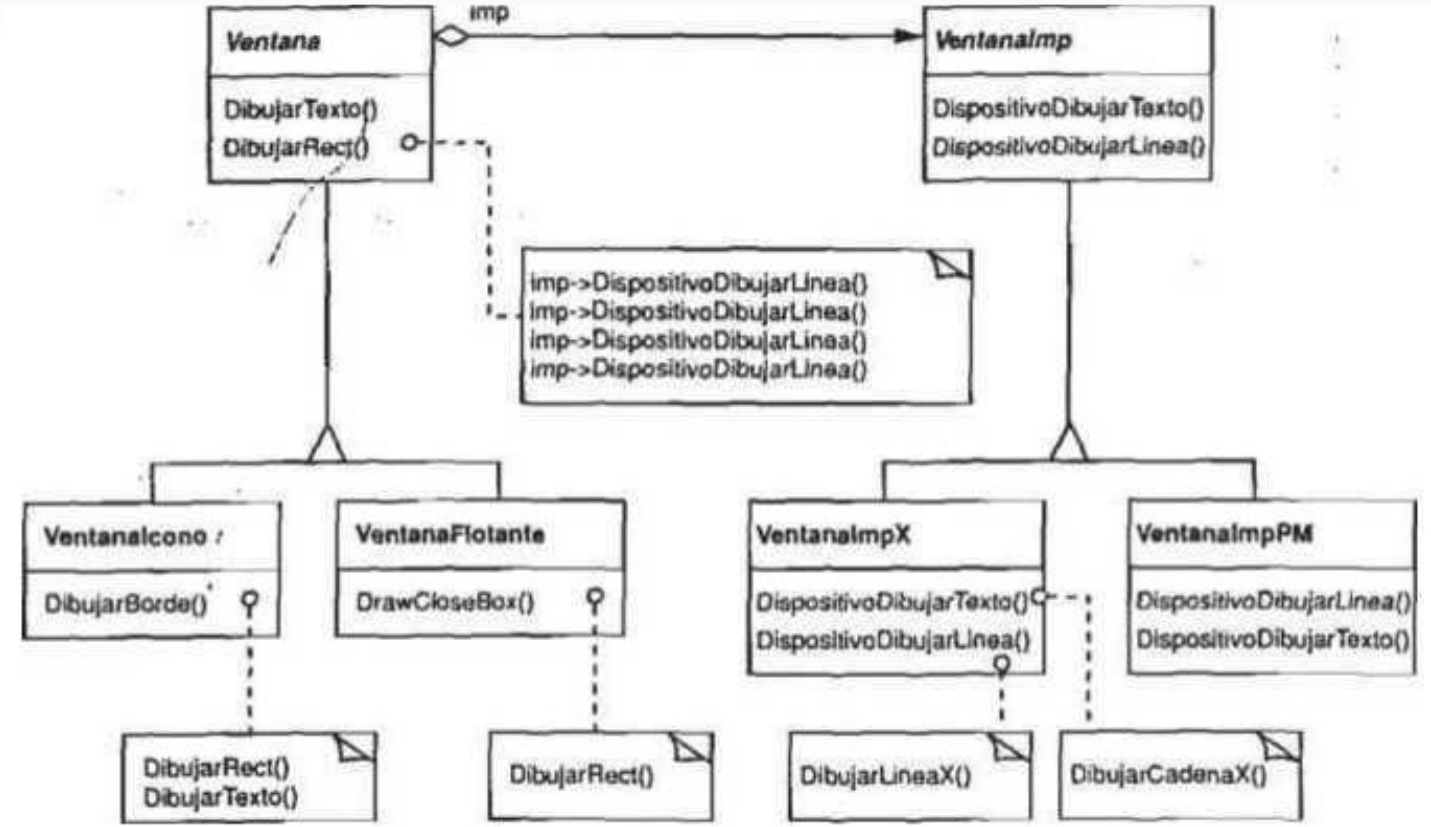
Participantes

- Abstracción
  - Define la interfaz de la abstracción.
  - Mantiene una referencia a un objeto de tipo Impleinemador.
- AbstraccionRefinada
  - Extiende la interfaz definida por Abstracción.
- Implementador
  - define la interfaz de las clases de Implementación.
- ImplementadorConcreto
  - Implementa la interfaz Implementador y define su implementaeión concreta.

Colaboradores

- Abstracción redirige las peticiones del cliente a su objeto Implementador.

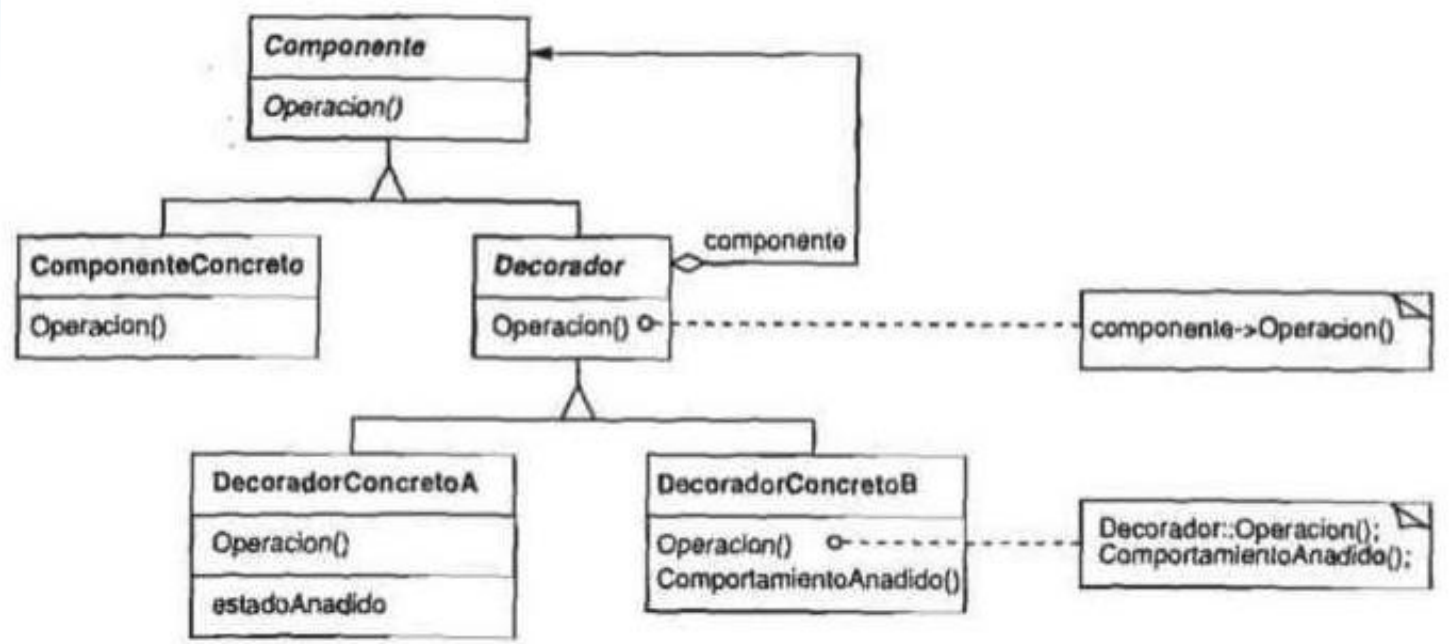
Ejemplo


Ventana
DibujarTexto()
DibujarRect()
imp
VentanaImp
DispositivoDibujarTexto()
DispositivoDibujarLinea()
imp->DispositivoDibujarLinea()
imp->DispositivoDibujarLinea()
imp->DispositivoDibujarLinea()
imp->DispositivoDibujarLinea()
VentanaIcono
DibujarBorde()
VentanaFlotante
DrawCloseBox()
VentanaImpX
DispositivoDibujarTexto()
DispositivoDibujarLinea()
VentanaImpPM
DispositivoDibujarLinea()
DispositivoDibujarTexto()
DibujarRect()
DibujarTexto()
DibujarRect()
DibujarLineaX()
DibujarCadenaX()

Consecuencias

- *Desacopla la interfaz y la implementación.* No une permanentemente una implementación a una interfaz, sino que la implementación puede configurarse en tiempo de ejecución. Incluso es posible que up objeto cambie su implementación en tiempo de ejecución.
- *Mejora la extensibilidad.* Podemos extender las jerarquías de Abstracción y de Implementador de forma independiente.
- *Oculta detalles de implementación a tos clientes.*

Decorator



Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa fle-xible a la herencia para extender la funcionalidad.

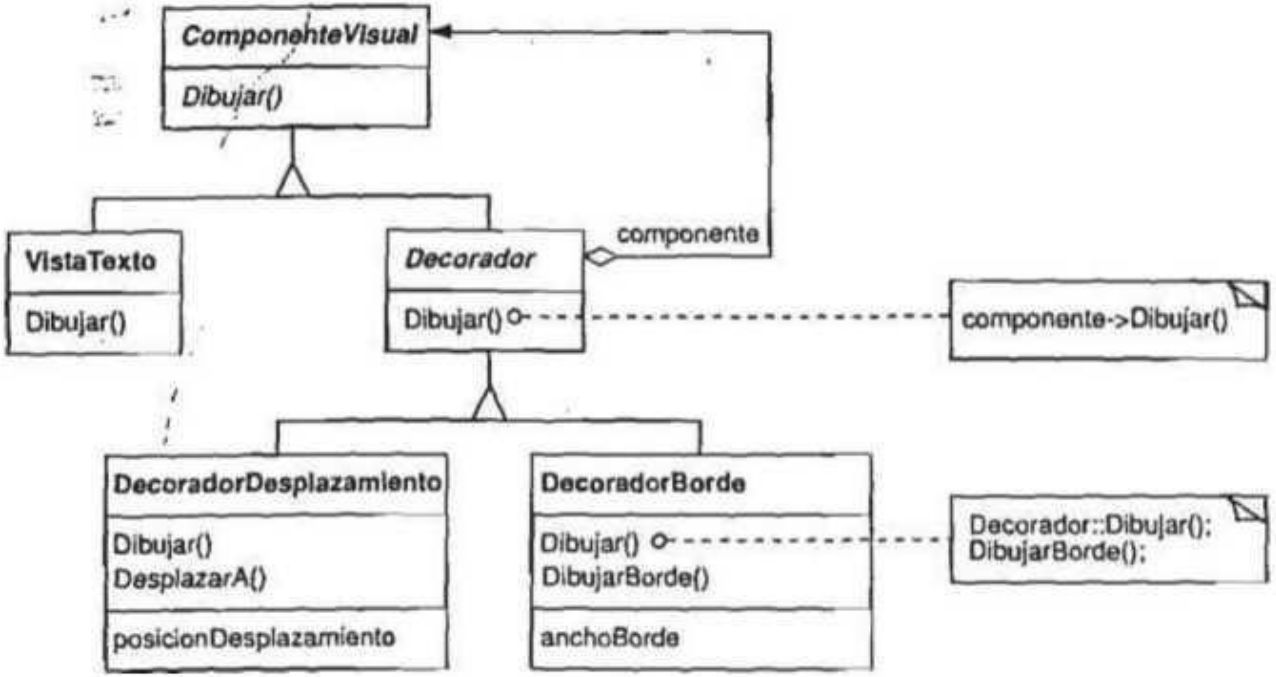
Participantes

- Componente
  - Define la interfaz para objetos a los que se puede añadir responsabilidades dinámicamente.
- ComponenteConcreto
  - Define un objeto al que se pueden añadir responsabilidades adicionales.
- Decorador
  - Mantiene una referencia a un objeto Componente y define una interfaz que se ajusta a la in- terfaz del Componente.
- DecoradorConcreto
  - añade responsabilidades al componente.

Colaboración

- El Decorador redirige peticiones a su objeto Componente. Opcionalmente puede realizar operaciones adicionales antes y después de reenviar la petición.

Ejemplo


ComponenteVisual
Dibujar()
VistaTexto
Dibujar()
Decorador
Dibujar()
componente
componente->Dibujar()
DecoradorDesplazamiento
Dibujar()
DesplazarA()
posicionDesplazamiento
DecoradorBorde
Dibujar()
DibujarBorde()
anchoBorde
Decorador::Dibujar();
DibujarBorde();

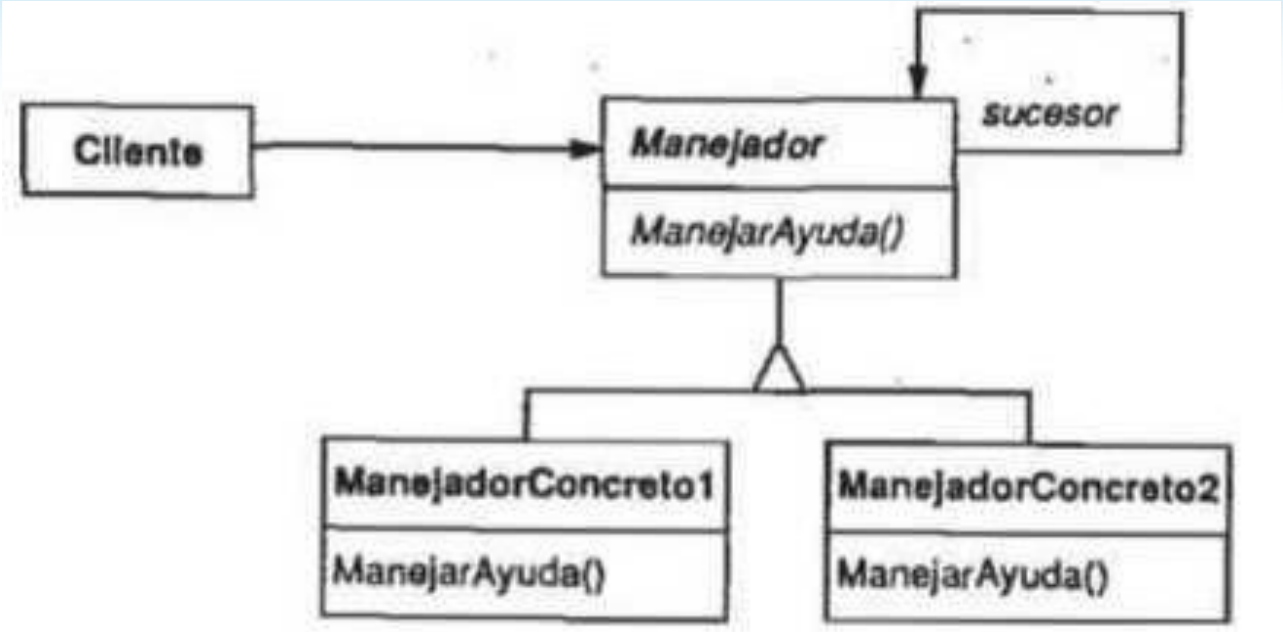
Consecuencias

- *Más flexibilidad que la herencia estática.* El patrón Decorador proporciona una manera más flexible de añadir responsabilidades a los objetos que la que podía obtenerse a través de la herencia (múltiple) estática.
- *Evita clases cargadas de funciones en la parte de arriba de la jerarquía.* El Decorador ofrece un enfoque para añadir responsabilidades que consiste en pagar sólo por aquello que se necesita.
- *Un decorador y su componente no son idénticos.* Un decorador se comporta como un revestimiento transparente.

Patrones de Diseño de Comportamiento

- Los patrones de diseño de comportamiento se preocupan acerca de los algoritmos y de la asignación de responsabilidades entre los objetos.
- Los patrones de clase de comportamiento usan herencia para distribuir el comportamiento entre las clases
- Los patrones de objeto de comportamiento usan la composición de objetos en vez de la herencia.; algunos describen como un grupo de objetos coopera para llevar a cabo una tarea que un objeto simple no puede ejecutar por si mismo; otros tratan acerca de la encapsulación de comportamiento en un objeto y la delegación de peticiones a él.

Chain of Responsibility

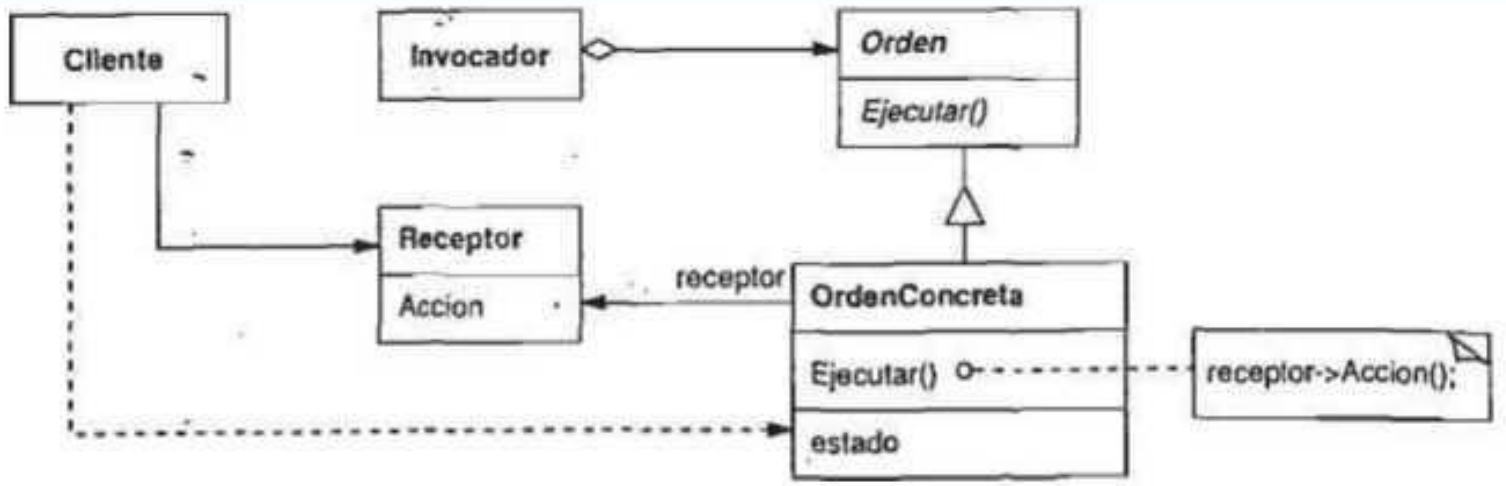


Evita acoplar el emisor de una petición a su receptor, dando a más de un objeto la posibilidad de responder a la petición. Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.

Participantes

- Manejador
  - Define una interfaz para tratar las peticiones.
  - implementa el enlace al sucesor.
- ManejadorConcreto
  - trata las peticiones de las que es responsable.
  - puede acceder a su sucesor.
  - si el ManejadorConcreto puede manejar la petición, lo hace; en caso contrario la reenvía a su sucesor.
- Cliente
  - inicializa la petición a un objeto ManejadorConcreto de la cadena.

Command



Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones, y poder deshacer las operaciones.

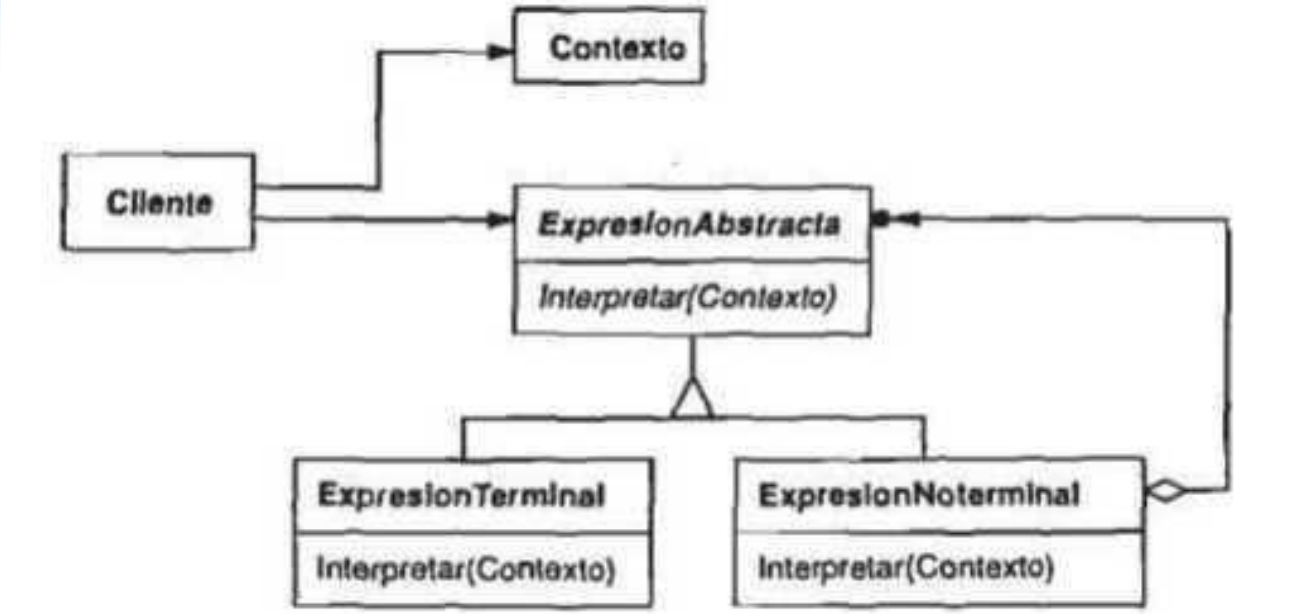
Participantes

- Orden
  - Declara una interfaz para ejecutar una operación.
- OrdenConcreta
  - Define un enlace entre un objeto Receptor y una acción.
  - -implementa Ejecutar invocando la correspondiente operación u operaciones del Receptor.
- Cliente
  - crea un objeto OrdenConcreta y establece su receptor.
- • Invocador
  - le pide a la orden que ejecute la petición.
- Receptor
  - sabe cómo llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede ha- cer actuar como Receptor.

Colaboración

- El cliente crea un objeto OrdenConcreta y especifica su receptor.
- • Un objeto Invocador almacena el objeto OrdenConcreta.
- • El invocador envía una petición llamando a Ejecutar sobre la orden. Cuando las órdenes se pue- den deshacer, OrdenConcreta guarda el estado para deshacer la orden antes de llamar a Ejecu- tar.
- • El objeto OrdenConcreta invoca operaciones de su receptor para llevar a cabo la petición.

Interpreter



Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.

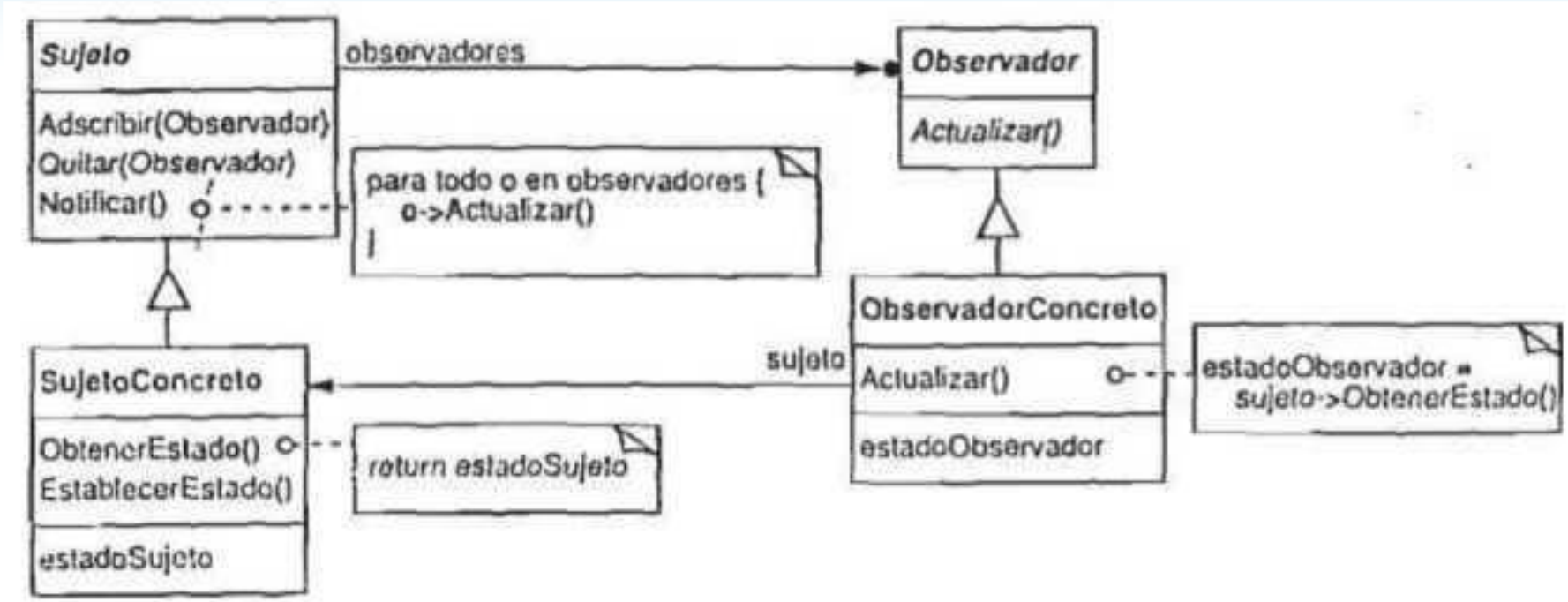
Participantes

- ExpresionAbstracta (ExpresionRegular)
  - declara una operación abstracta Interpretar que es común a todos los nodos del árbol de sin- taxis abstracto.
- ExpresionTerminal (ExpresionLiteral)
  - implementa una operación Interpretar asociada con los símbolos terminales de la gramática.
  - se necesita una instancia de esta clase para cada símbolo terminal de una sentencia.

Participantes

- ExpresionNoTerminal (ExpresionAltemativa, ExpresionRepeticion, ExpresionSecuencia)
  - por cada regla de la gramática R ::= R1, R2... Rn debe haber una de estas clases.
  - mantiene variables de instancia de tipo ExpresionAbstracta para cada uno de los símbolos de Rl a Rn.
  - implementa una operación Interpretar para lo símbolos no terminales de la gramática. Inter- pretar normalmente se llama a sí misma recursivamente sobre las variables que representan de Rl a Rn.
- Contexto
  - contiene información que es global al intérprete.

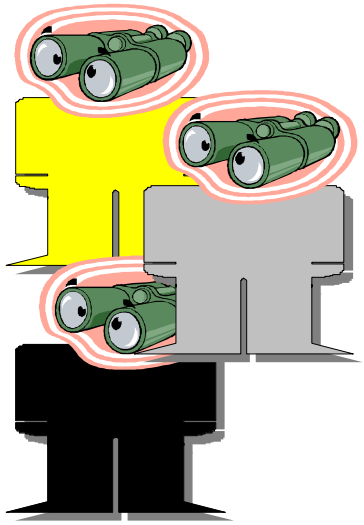
Observer



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

El Problema

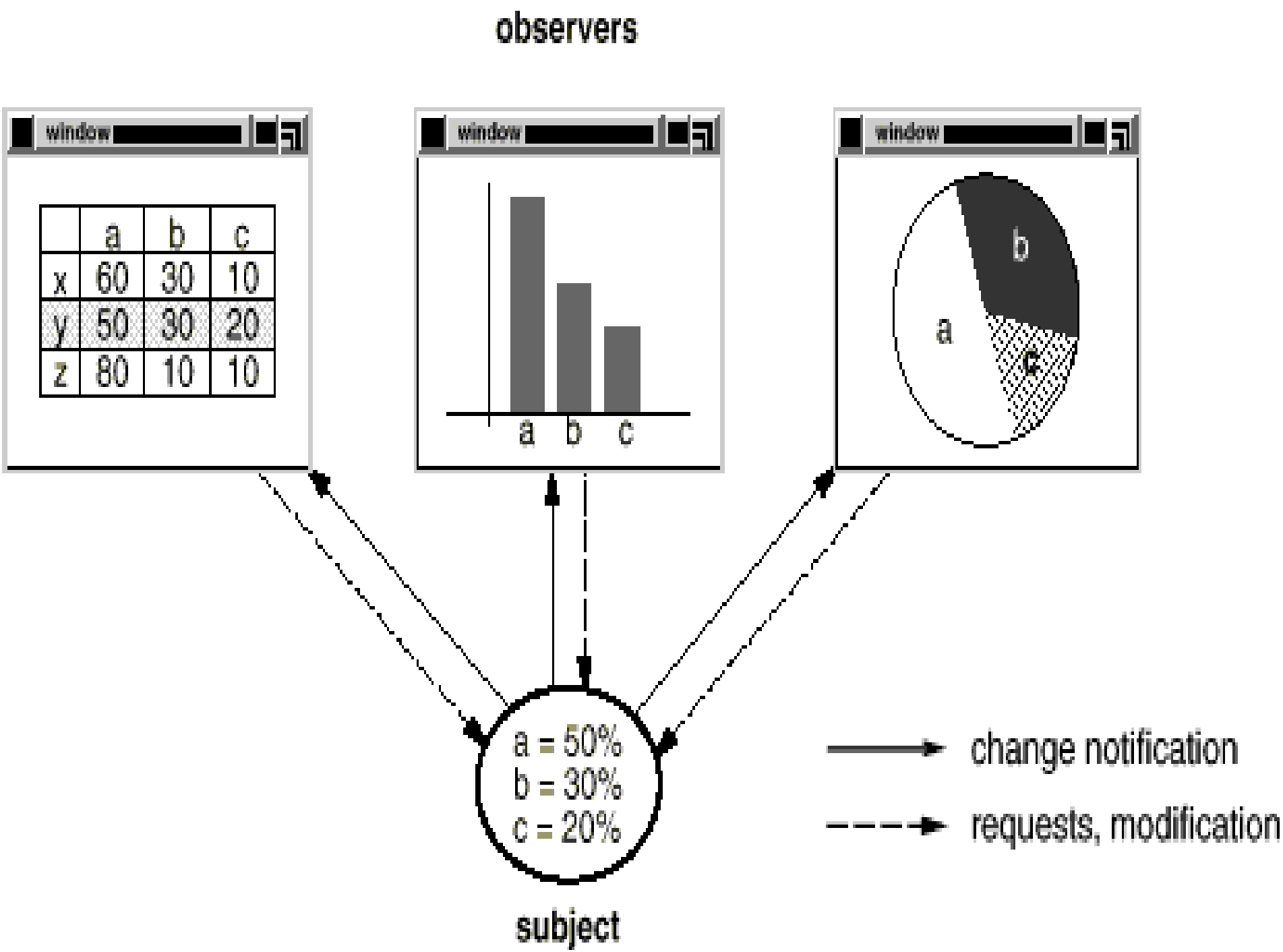
Sujeto



Observadores

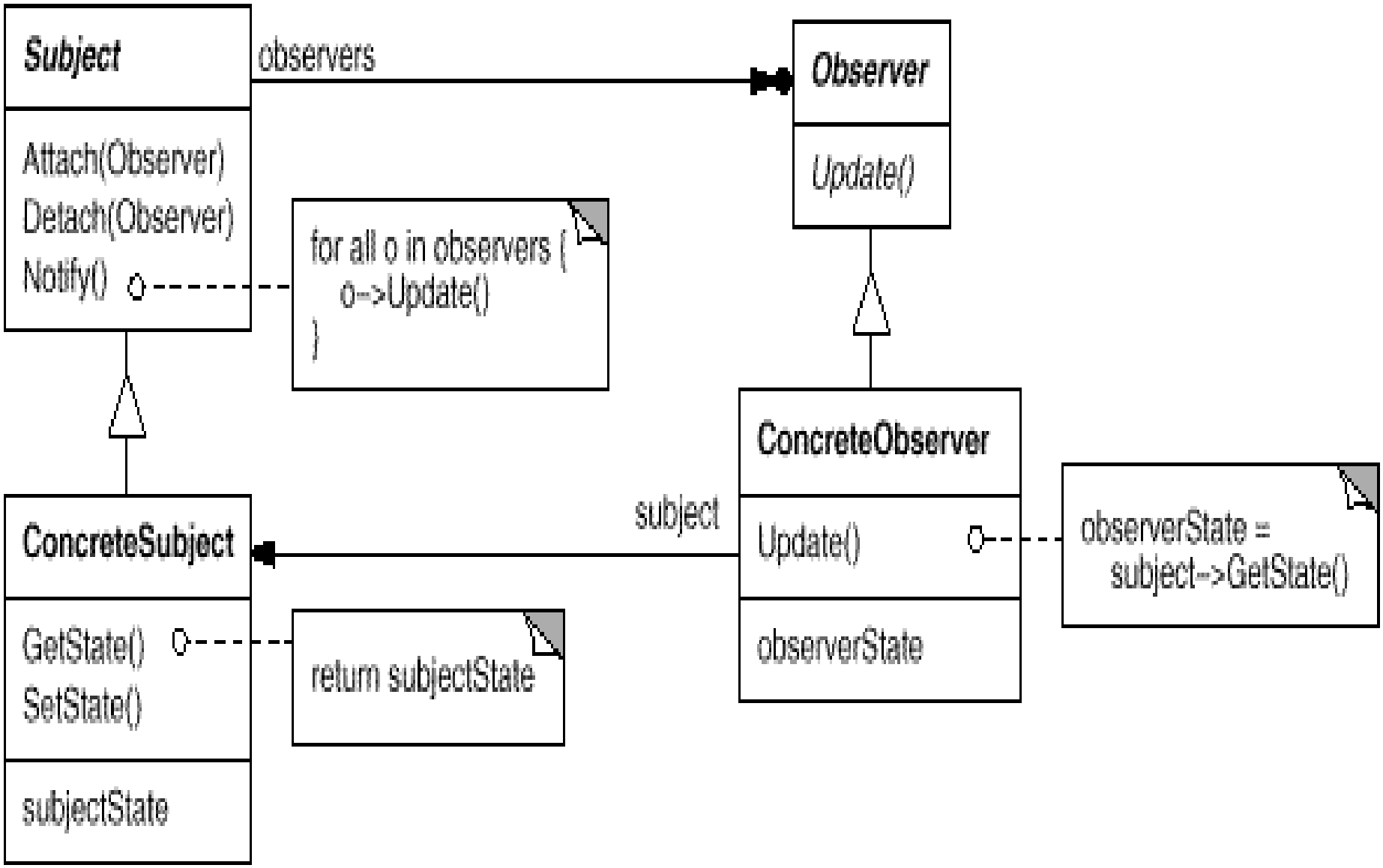
-Diferentes tipos de GUI mostrando los mismos datos
- Diferentes ventanas mostrando diferentes vistas de un mismo modelo.

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

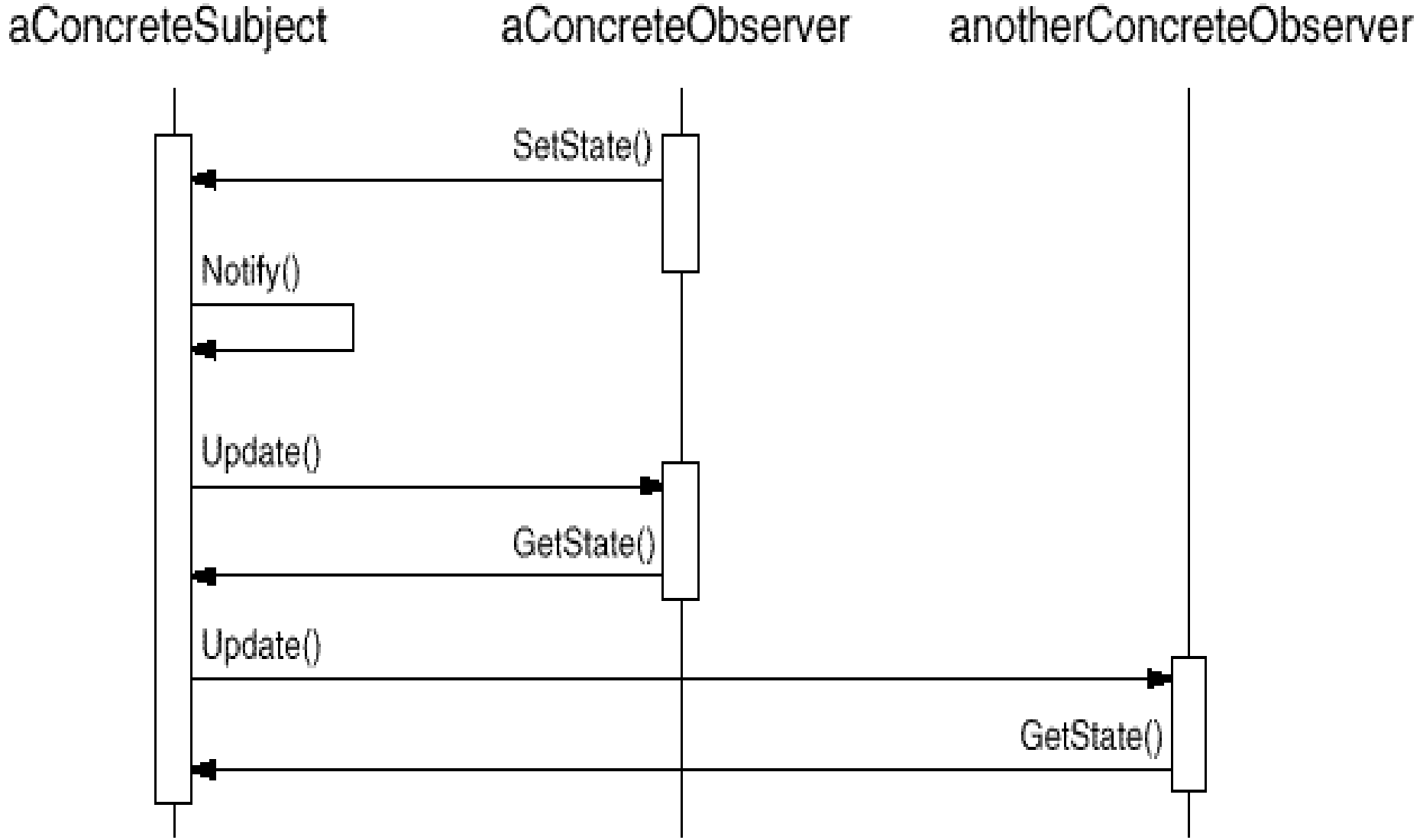
Participantes

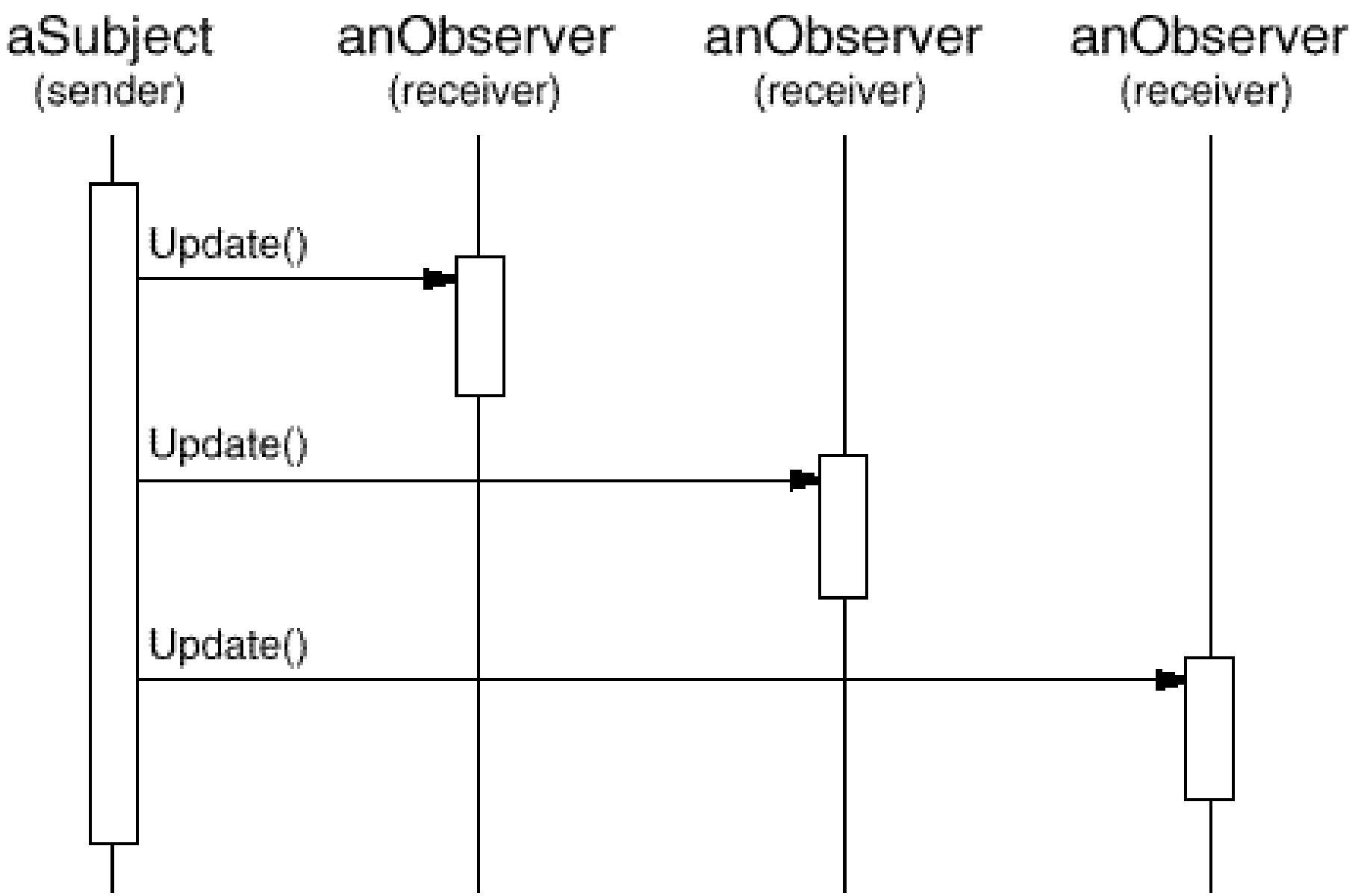
- Sujeto: conoce sus observadores, provee una interfaz para añadir (subscribir) y eliminar (cancelar) observadores y provee un método *notificar* que llama a *actualizar* en todos los observadores
- Observador: provee una interfase *actualizar*
- SujetoConcreto: mantiene el estado relevante para la aplicación, 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()
observerState
return subjectState
subjectState

Colaboración

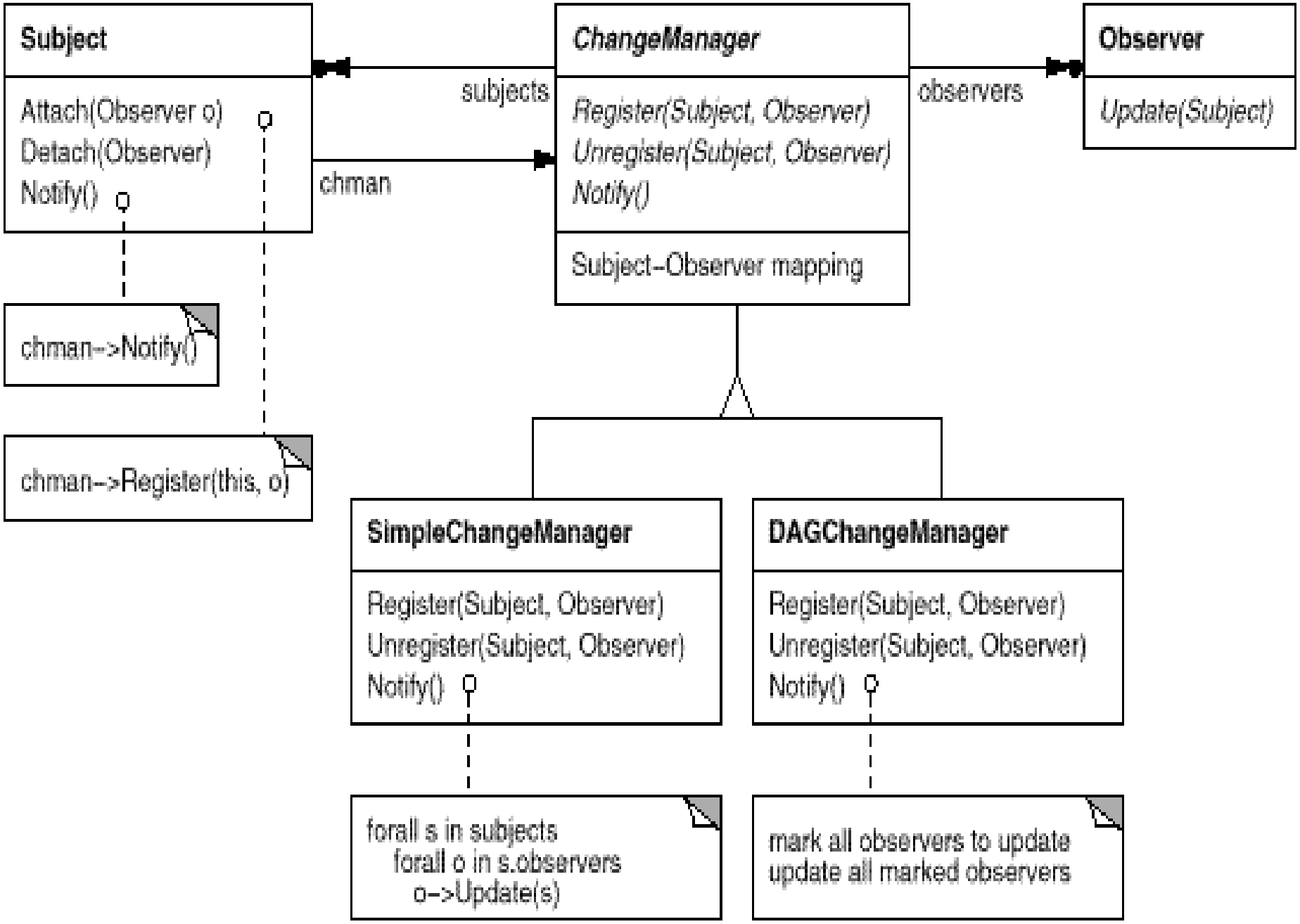
- El SujetoConcreto notifica a sus observadores cada vez que se produce un cambio que pueda volver el estado de los observadores inconsistente.
- Después de ser informados de un cambio, el Observador Concreto puede preguntar al Sujeto acerca de la información concerniente a su estado y entonces reconciliar 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()

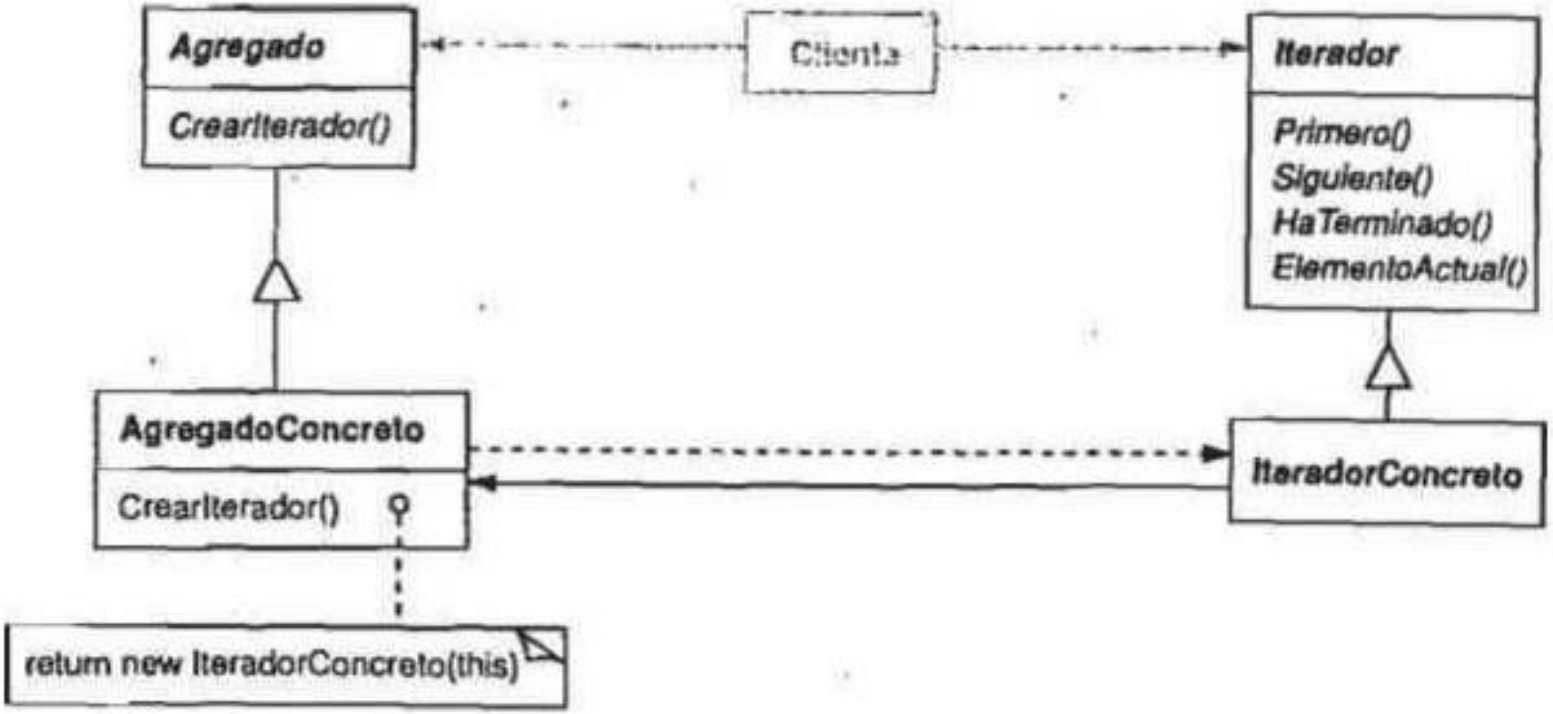

aSubject
(sender)
anObserver
(receiver)
anObserver
(receiver)
anObserver
(receiver)
Update()
Update()
Update()

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.

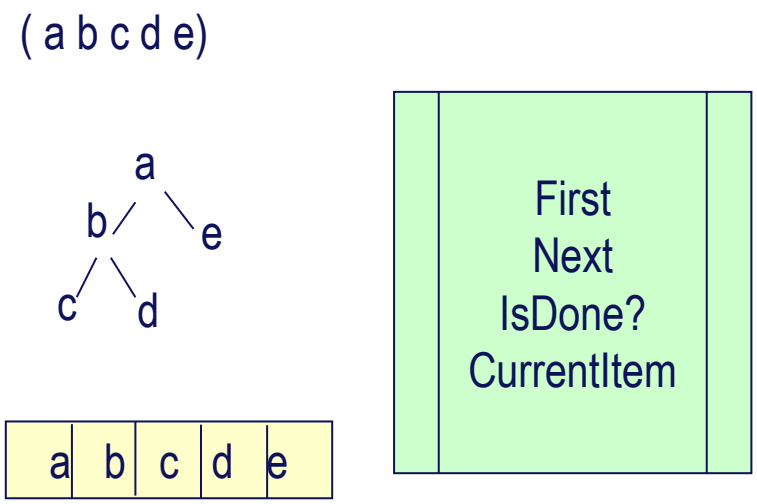

Subject
Attach(Observer o)
Detach(Observer)
Notify()
subjects
chman
ChangeManager
Register(Subject, Observer)
Unregister(Subject, Observer)
Notify()
Subject–Observer mapping
observers
Observer
Update(Subject)
chman->Notify()
chman->Register(this, o)
SimpleChangeManager
Register(Subject, Observer)
Unregister(Subject, 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

Iterator

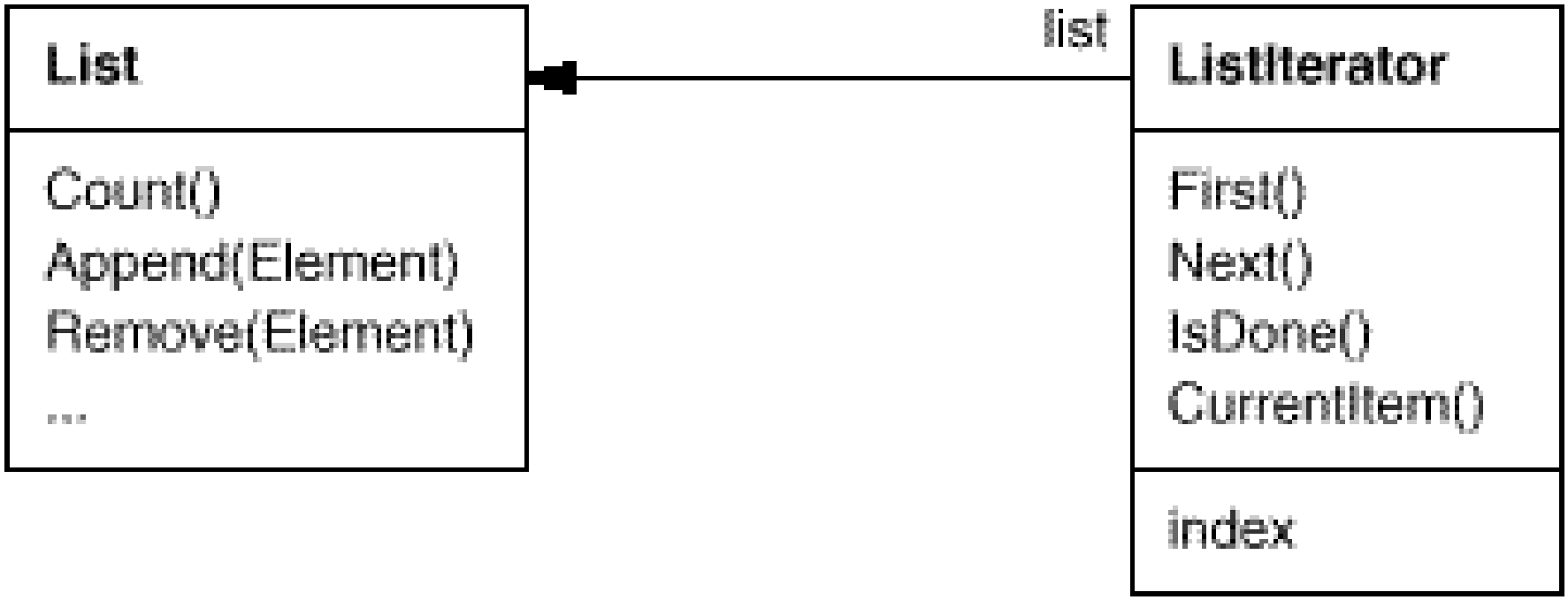


Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representacióninterna.

El Patrón Iterador: El Problema

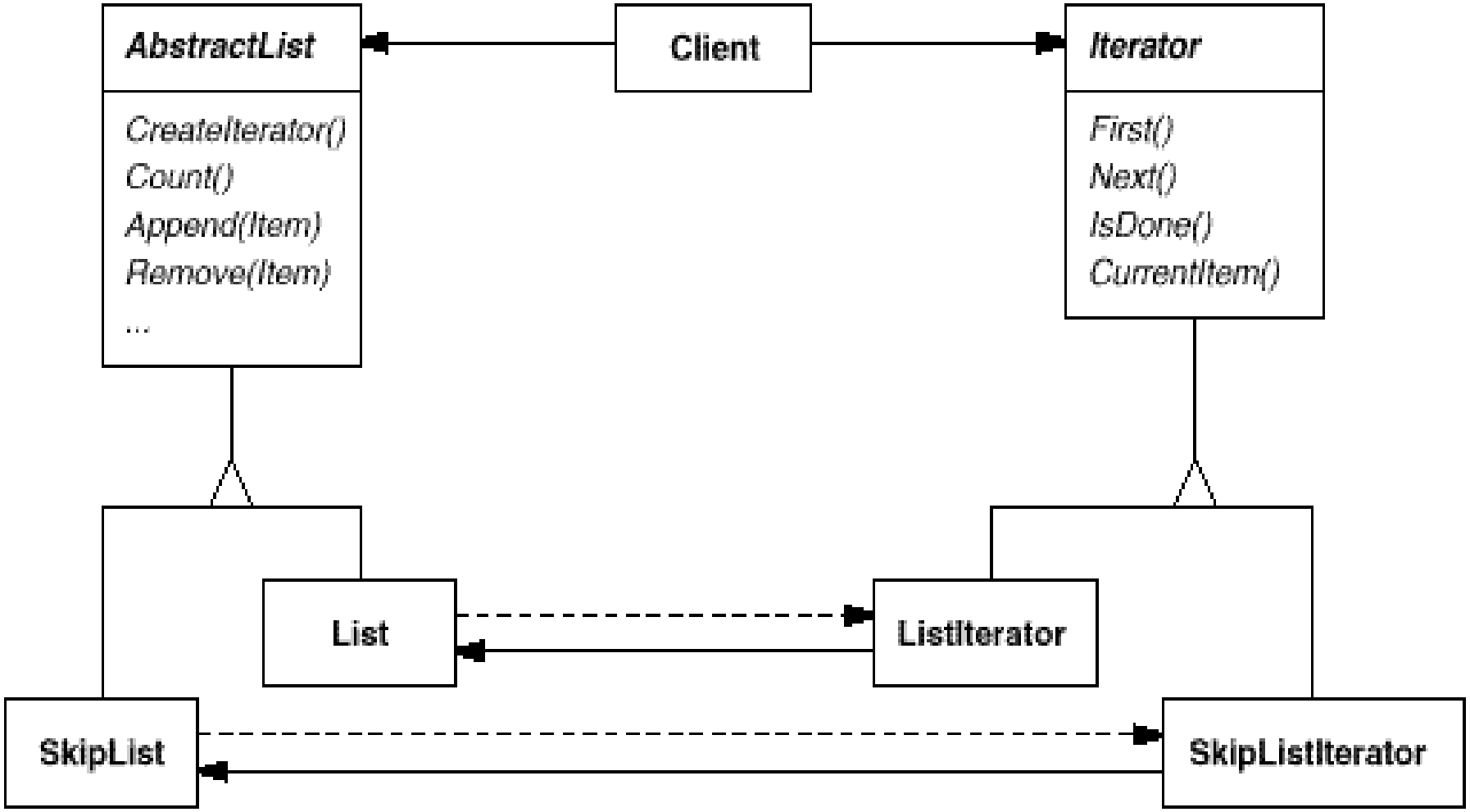


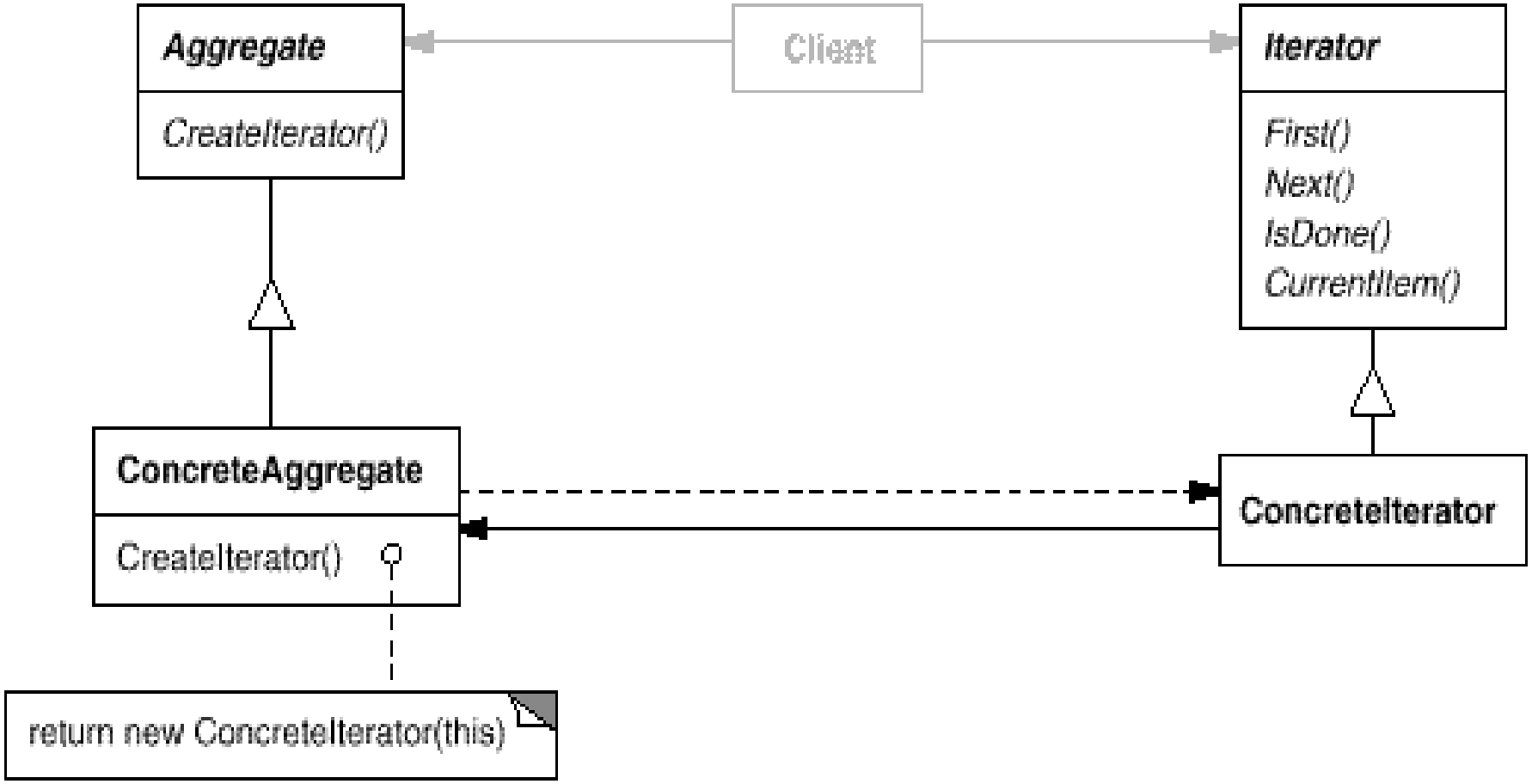
- **Soporta múltiples tipso de recorridos de objetos agregados**
- **Provee una interfaz uniforme para recorrer estructuras agregadas (iteración polimórfica)**


List
Count()
Append(Element)
Remove(Element)
...
list
ListIterator
First()
Next()
IsDone()
CurrentItem()
index

Participantes

- Iterador
  - Define una interfaz para acceder y recorrer elementos
- IteradorConcreto
  - Implementa la interfaz iterador
  - Mantiene la pista de la posición actual en el recorrido del agregado.
- Agregado
  - Define una interfase para definir un objeto Iterador
- AgregadoConcreto
  - Implementa la interfaz de creación de un iterador para retornar una instancia del propio IteradorConcreto
- Un IteradorConcreto mantiene la pista del objeto actual en el agregado y puede calcular el siguiente objeto en el recorrido.

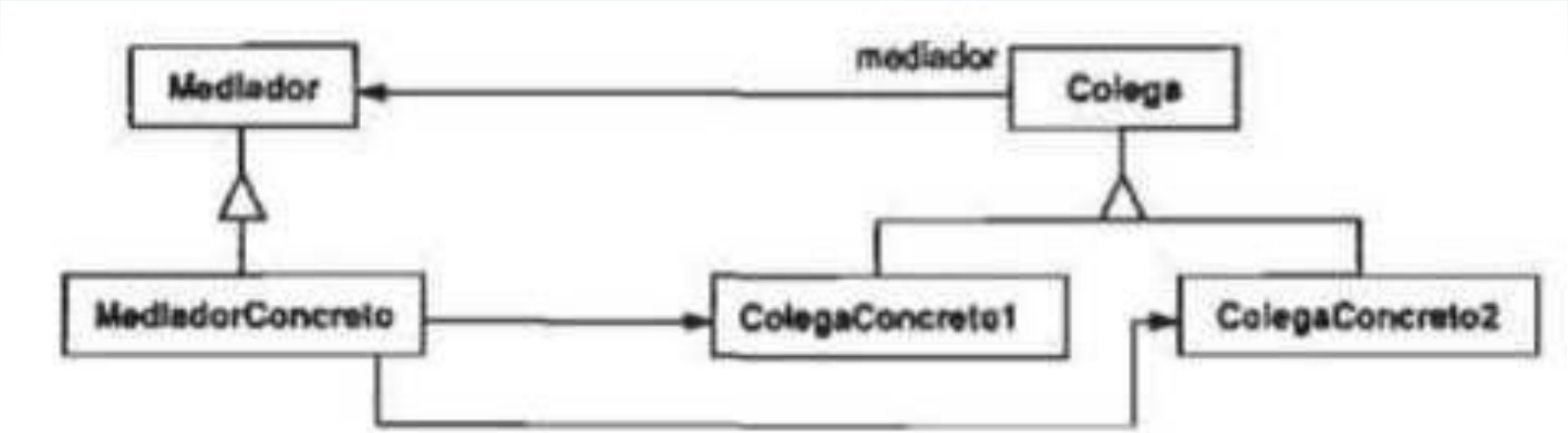

AbstractList
CreateIterator()
Count()
Append(Item)
Remove(Item)
...
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
List
ListIterator
SkipList
SkipListIterator


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Consecuencias

- + Soporta la variación en el recorrido de un agregado: agregados complejos pueden ser recorridos de diferentes maneras, los iteradores hacen fácil cambiar el algoritmo de recorrido solamente usando una instancia diferente de iterador.
- + Simplifica la interfaz del agregado: La interfaz agregado no esta obstaculizada por el soporte de varios tipos de recorrido.
- + Más de un recorrido puede estar pendiente en el mismo agregado: un iterador mantiene registro de su propio estado de recorrido, por lo tanto más de un recorrido puede estar en progreso al mismo tiempo.

Mediator



Define un objeto que encapsula cómo interactúan una serie 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.

Participantes

- Mediador (DirectorDialogo)
  - define una interfaz para comunicarse con sus objetos Colega.
- MediadorConcreto (DirectorDialogoFuente)
  - implemento el comportamiento cooperativo coordinando objetos Colega.
  - conoce a sus Colegas.
- clases Colega (ListaDesplegable, CampoDeEntrada)
  - cada clase Colega conoce a su objeto Mediador.
  - cada Colega se comunica con su mediador cada vez que, de no existir éste, se hubiera comu- nicado con otro Colega.

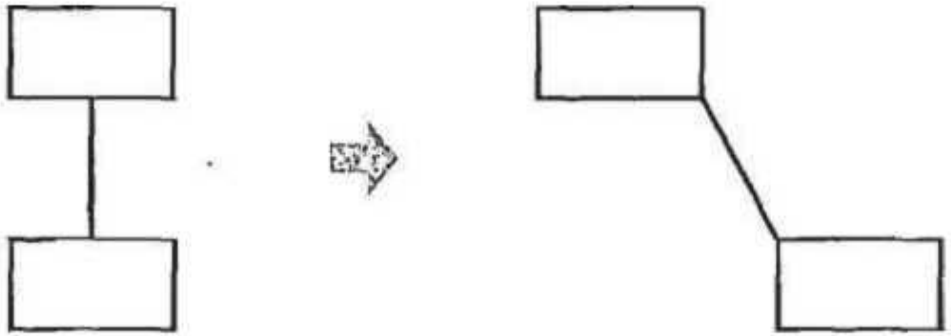
Memento

Representa y cxlcmaliza el estarlo interno de un objeto sin violar la encapsulación, de forma que éste puedo volver a dicho estado más tarde.

Componentes

- Memento (EstadoDeIResolvente)
  - guarda el estado interno del objeto Creador. El memento puede guardar tanta información del estado interno del creador como sea necesario a discreción del creador.
  - protege frente a accesos de otros objetos que no sean el creador. Los mementos tienen real- mente dos interfaces. El Conserje ve una interfaz *reducida* del Memento
    - sólo puede pasar el memento a otros objetos
    - El Creador, por el contrario, ve una interfaz *amplia,* que le permite acceder a todos los datos necesarios para volver a su estado anterior. Idealmente, sólo el creador que produjo el memento estaría autorizado a acceder al estado interno de éste.
- Creador (ResolventeDeRestricciones)
  - crea un memento que contiene una instántanea de su estado interno actual.
  - usa el memento para volver a su estado anterior.
- Conserje (mecanismo de deshacer)
  - es responsable de guardar en lugar seguro el memento.
  - nunca examina los contenidos del memento, ni opera sobre ellos.

Ejemplo

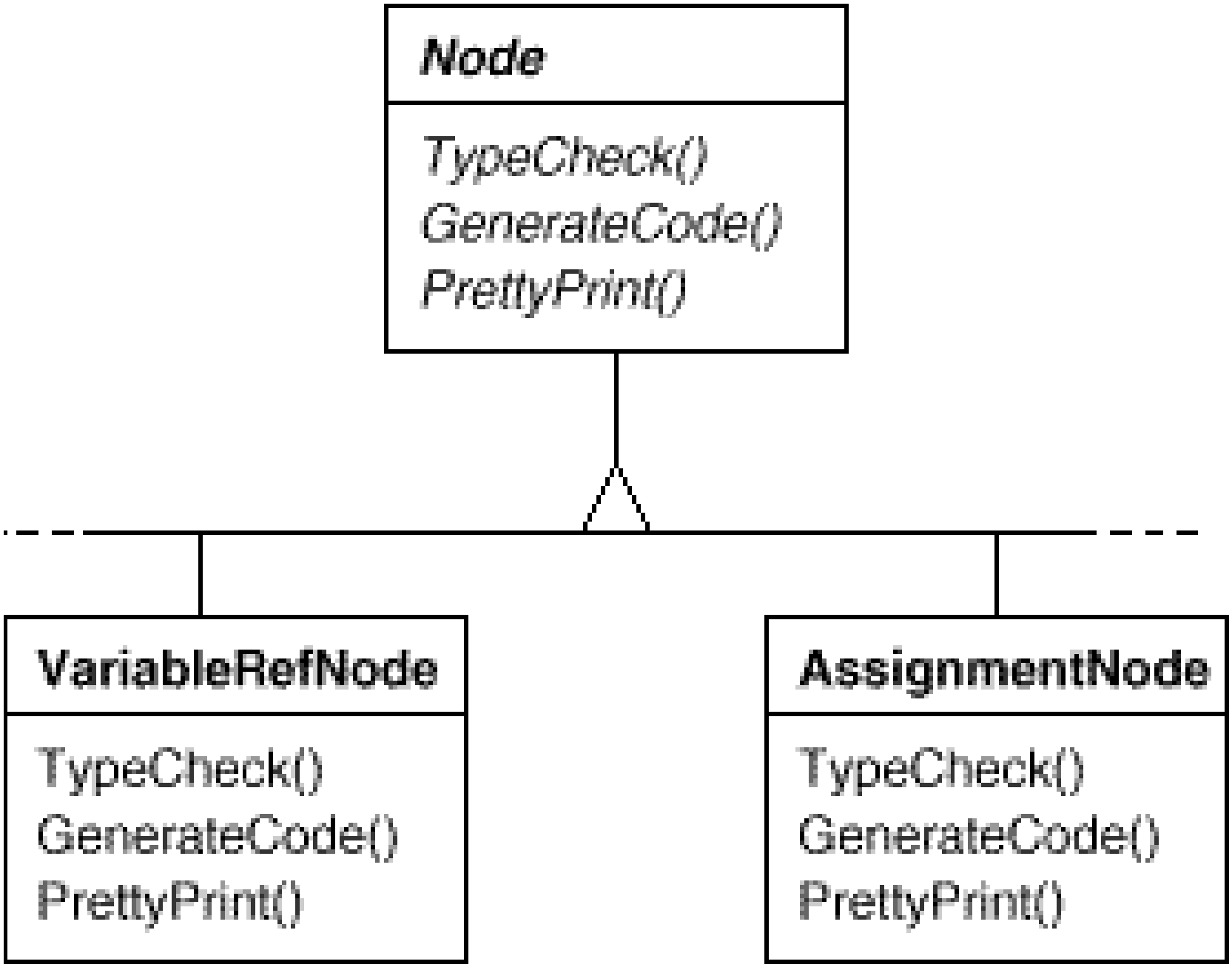


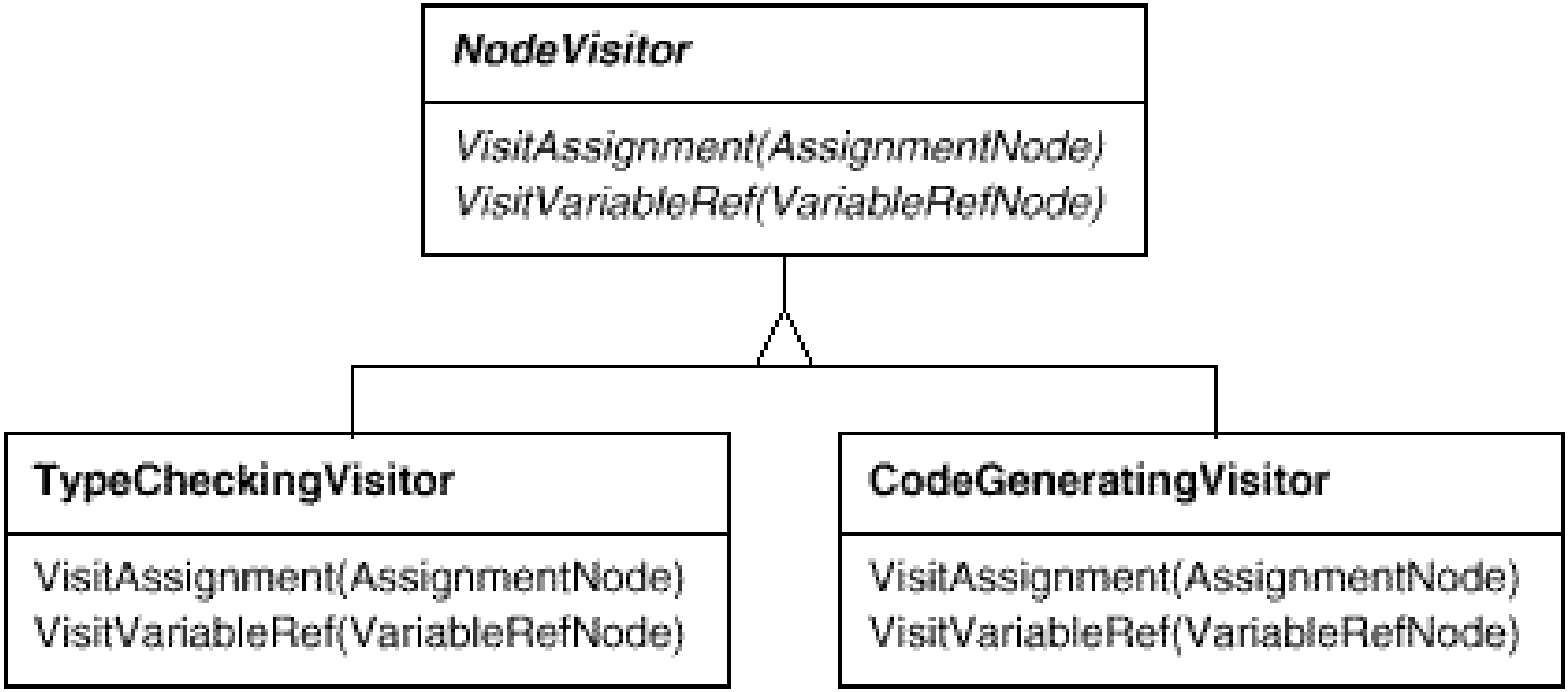
Visitor

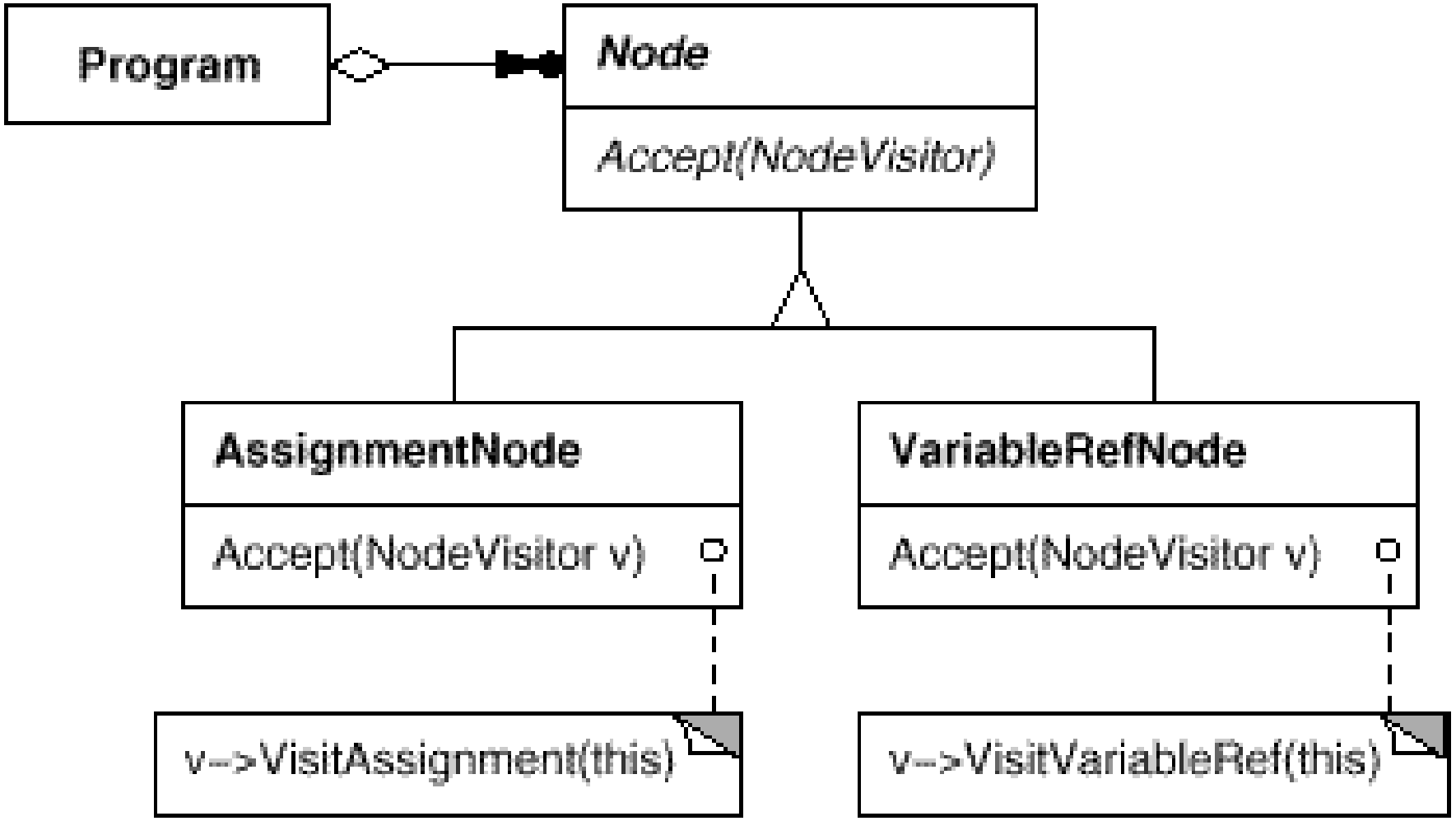
El Problema

Representa una operación a ser llevada a cabo en los elementos de un objeto estructura. Los visitantes permiten definir una nueva operación sin cambiar las clases de los elementos en los cuales opera.

- varias operaciones distintas y sin relación necesitan ser llevadas a cabo en objetos en un objeto estructura y no se quiere “ensuciar” las clases con estas operaciones.
- Las clases que definen el objeto estructura son rara vez cambiadas pero muchas veces se requiere definir nuevas operaciones sobre la estructura.


Node
TypeCheck()
GenerateCode()
PrettyPrint()
VariableRefNode
TypeCheck()
GenerateCode()
PrettyPrint()
AssignmentNode
TypeCheck()
GenerateCode()
PrettyPrint()


NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)


Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
VariableRefNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
v->VisitVariableRef(this)

Participantes

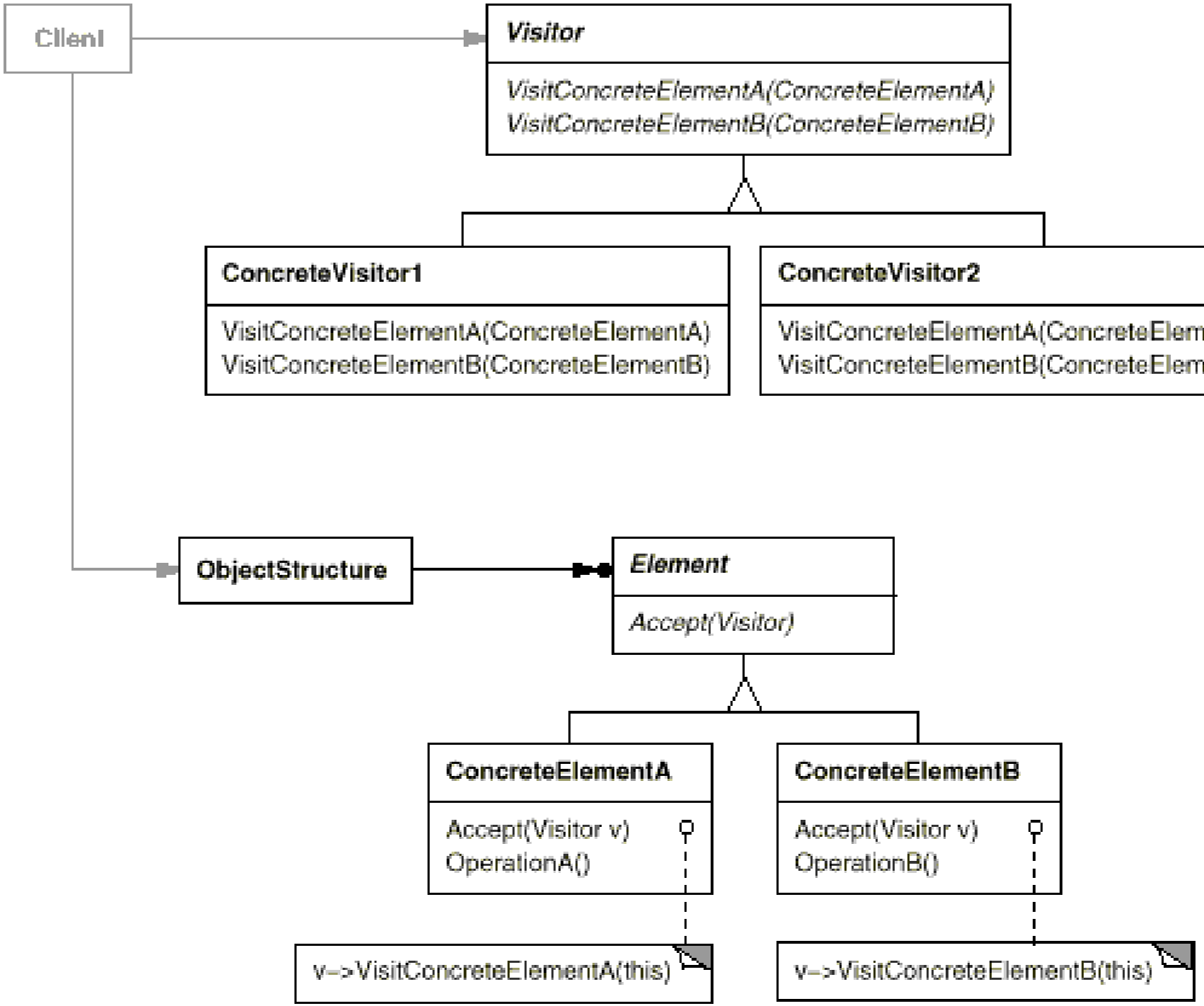
- Contexto
  - Declara una operación visitante para cada clase de ElementoConcreto en el objeto estructura
  - El nombre de la operación y la firma identifica la clase que envía el requerimiento Visitante
- VisitanteConcreto
  - Implementea cada operación declarada por Visitante
  - Cada operación implementa un fragmento del algoritmo para la correspondiente clase del objeto en el objeto estructura
  - Provee el contexto para el algoritmo y almacena su estado (frecuentemente acumulando resultados durante el recorrido)
- Elemento
  - Define una operación de aceptación que toma un visitante como argumento

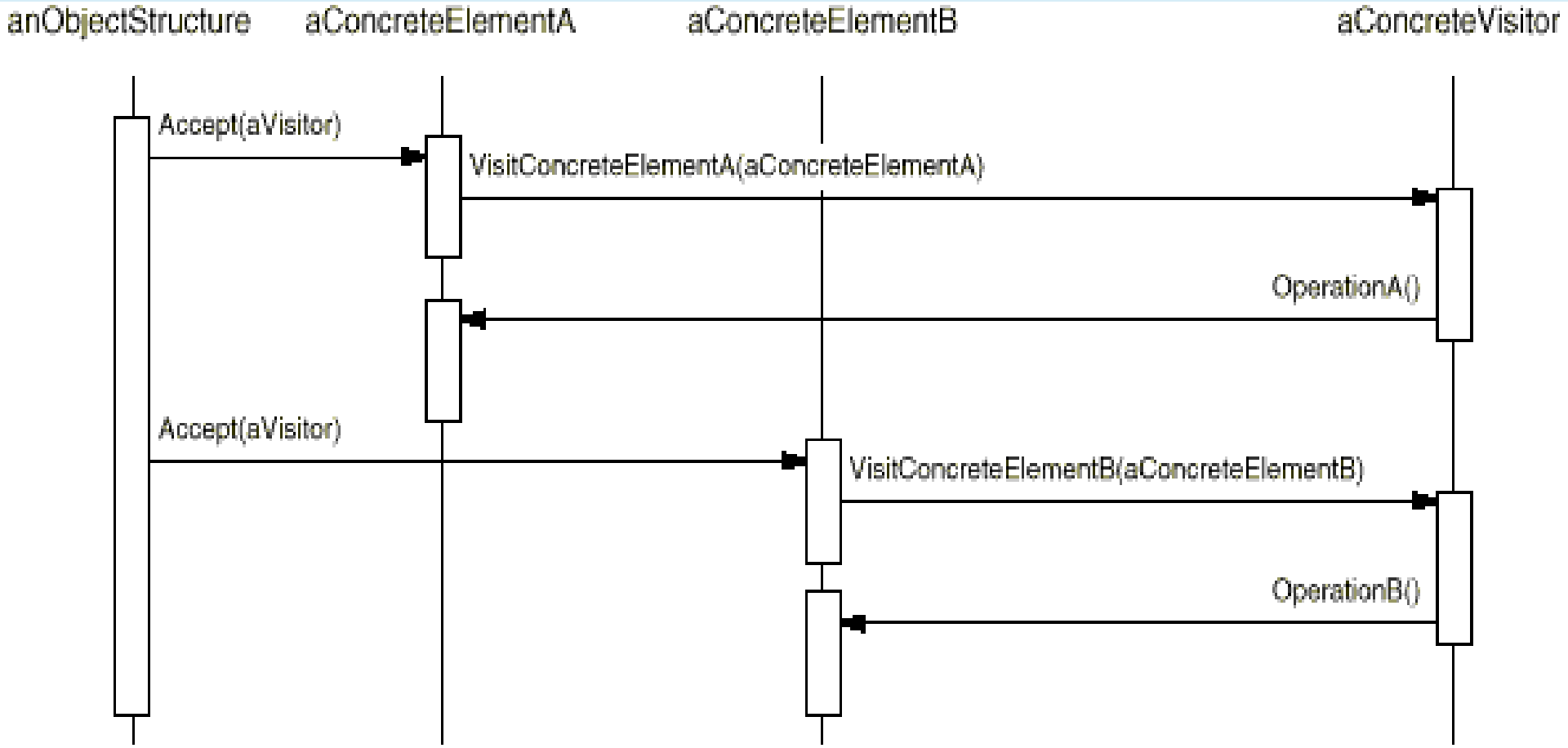
Participantes

- Elemento Concreto
  - Implementa una operación de aceptación que toma un visitante como un argumento
- ObjetoEstructura
  - Puede enumerar los elementos
  - Puede proveer una interfaz de alto nivel para permitir al visitante visitar a sus elementos.
  - Puede ser una composición o una colección tal como una lista o un conjunto.

Colaboración

- Un cliente que usa el patrón visitante debe crear un objeto VisitanteConcreto y recorrer el ObjetoEstructura visitando cada elemento con el Visitante
- Cuando un elemento es visitado, llama a la operación Visitante que corresponde a su clase. El elemento se provee a si mismo como un argumento a esta operación


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)


anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
OperationA()
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()

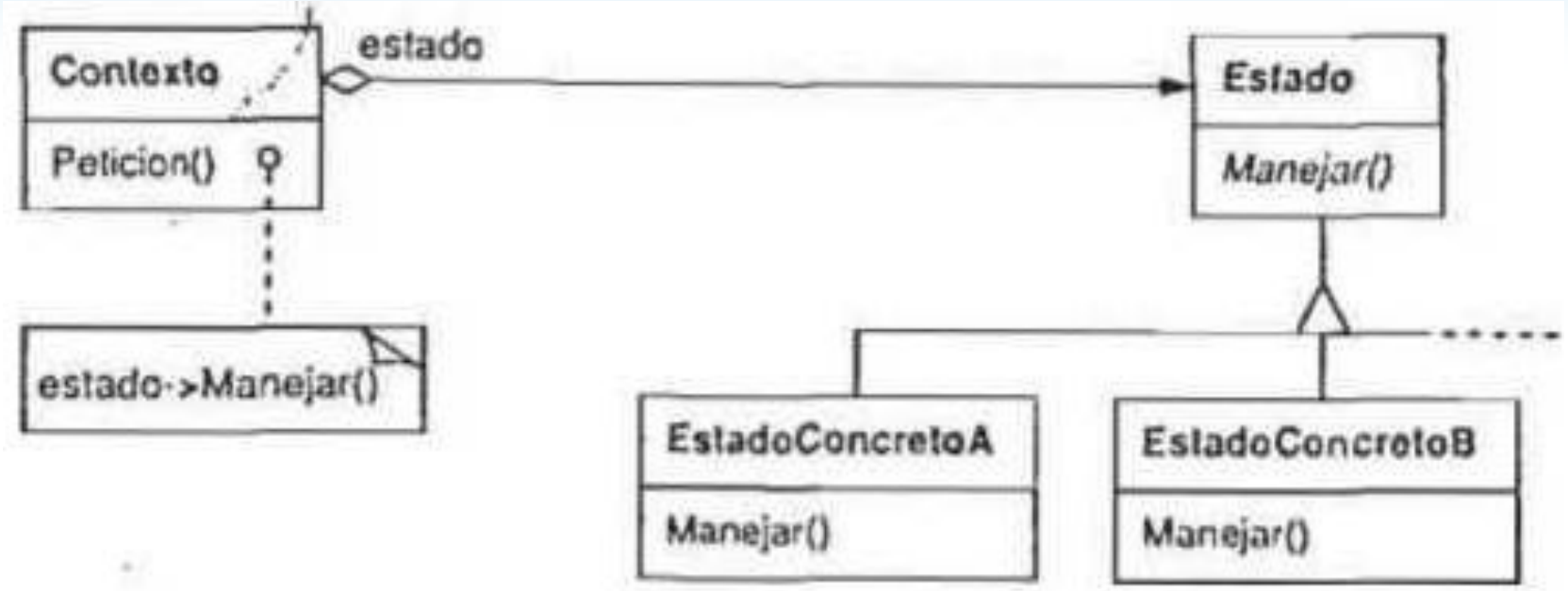
Consecuencias

- + Hace añadir nuevas operaciones fácil: una nueva operación es definida añadiendo un visitante (en contraste, cuando se extiende la funcionalidad sobre varias clases cada clase debe cambiar para definir una nueva operación)
- + Reúne operaciones relacionadas y separa las no relacionadas: comportamiento relacionado se localiza en el visitante y no se dispersa sobre las clases que definen el objeto estructura
- - Añadir nuevos ElementosConcretos es difícil: cada nuevo ElementoConcreto da lugar a una nueva operación abstracta en el Visitante y una correspondiente implementación en cada VisitanteConcreto

Consecuencias

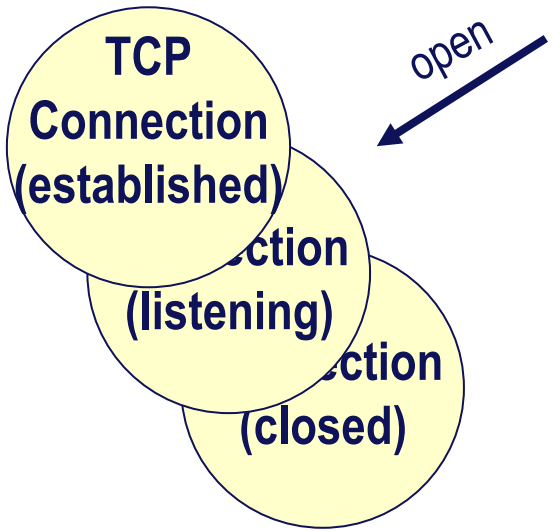
- + Permite visitante a través de jerarquía de clases: un iterador puede también visitar elementos de una estructura de objetos cuando los recorre y llama a las operaciones sobre ellos, pero todos los elementos de la estructura de objetos necesitan tener un padre común. Los Visitantes no tienen esta restricción
- + Acumulación de Estado: El visitante puede acumular estados al proceder con el recorrido. Sin un visitante este estado debe ser pasado como un parámetro extra a ser manejado en variables globales
- - Rompe la encapsulación: El acercamiento del Visitante asume que la interfaz del ElementoConcreto es lo suficientemente poderosa como para permiter al visitante hacer su trabajo. Como resultado el patrón muchas veces fuerza a proveer operaciones públicas para acceder al estado interno del elemento lo cual puede comprometer la encapsulación

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

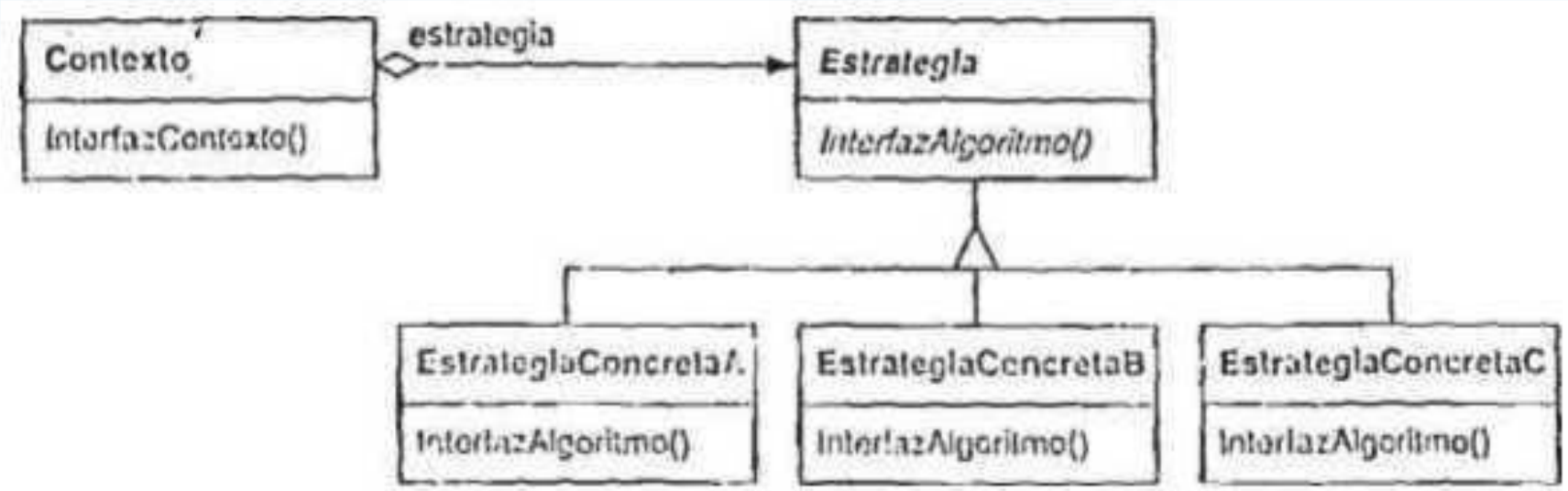
Colaboraciones del Patrón Estado

- 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

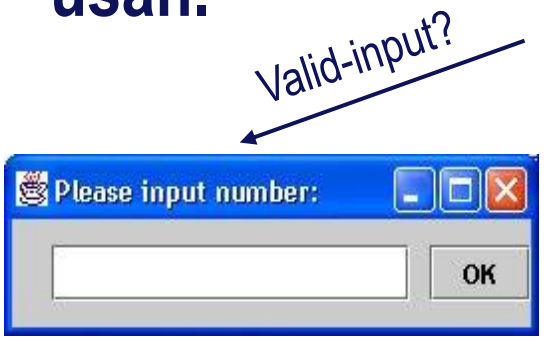
Strategy



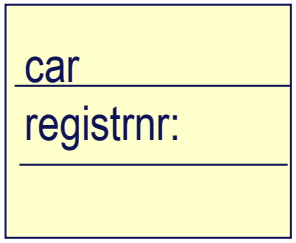
Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan

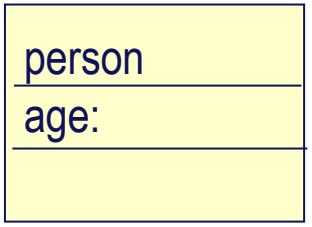
El Problema

Definir una familia de algoritmos, encapsular cada uno, y hacerlos intercambiables. Estrategia permite que el algoritmo varíe independientemente de los clientes que lo usan.



- Diferentes algoritmos de cambio de línea en un editor
- Diferentes tipos de alocación de memoria
- Algoritmos para colecciones de clases
- Diferentes algoritmos para revisar una entrada válida en diferente tipos de cajas de diálogo.





Aplicabilidad

- Varias clases diferentes difieren solamente en su comportamiento; usando Estrategia provee una manera de configurar una clase con uno o más comportamientos.
- Muchas variantes diferentes de un algoritmo son necesarias; usando Estrategia las variantes pueden ser implementadas como una jerarquía de clases de algoritmos.
- Los algoritmos usan datos que los clientes pueden no conocer; usando Estrategia se evita exponer datos complejos específicos del algoritmo.

Participantes

- Estrategia
  - Declara una interfaz común a todos los algoritmos soportados
  - Contexto usa esta interfaz para llamar a los algoritmos definidos por EstrategiaConcreta
- Subclases de EstrategiaConcreta
  - Cada subclase implementa el algoritmo usando la interfase Estrategia
- Contexto
  - Es configurado como un objeto EstrategiConcreta
  - Pude definir una interfaz que permite a Estrategia acceder sus datos

Colaboraciones

- Un contexto envía requerimientos de sus clientes a Estrategia
- Estrategia y Contexto interactúan para implementar el algoritmo escogido
  - Un Contexto puede pasar todo los datos requeridos por el algoritmo a Estrategia cuando se llama al algoritmo.
  - Contexto puede pasarse a si mismo como un argumento a las operaciones de Estrategia de tal menara que estas puedan llamarlo de vuelta si el Contexto es requerido.
- Los clientes usualmente crean y pasan un objeto EstrategiaConcreta al Contexto; de ahí los clientes interactúan con el Contexto

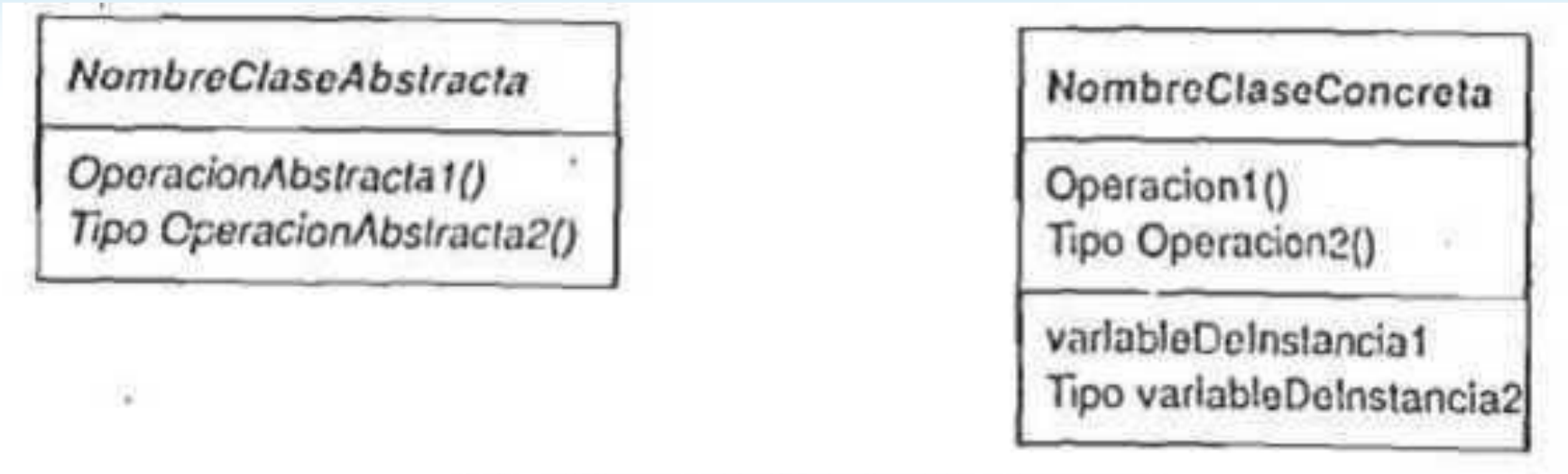
Consecuencias

- + Las Jerarquías de las clases Estrategia definen una familia de algoritmos o comportamientos para reusar. La herencia puede factorizar la funcionalidad común de estos algoritmos.
- + Es alternativo a subclase, Ej. Usar la jerarquía de clases de contexto para implementar las variaciones en el comportamiento. El comportamiento no esta fijo en contesto así que el algoritmo pude variar independientemente de Contexto.
- + Es una alternativa a usar sentencias condicionales para seleccionar el comportamiento deseado
- + Puede ofrecer una elección de implementaciones para el mismo comportamiento (Ej. Compromisos espacio tiempo)

Consecuencias

- - Clientes deben estar consientes de las diferentes Estrategias. Los clientes deben entender las diferencias entre lo que se ofrece y ser capaces de seleccionar el apropiado. Por lo tanto el patrón debería solo ser usado cuando la variación en algoritmo (implementación) es relevante para el cliente.
- - Existe algo de sobrecarga en la comunicación entre la Estrategia y el Contexto. Estrategia define una interfase (general). Muchas de los algoritmos más simples no necesitan toda la información que se envía.
- - El número de objetos en las aplicaciones aumenta Algunas veces la sobrecarga puede ser reducida cuando EstrategiasConcretas puede ser implementado por objetos sin estado que pueden ser compartidos (Ej. El patrón flyweight)

Template Method



Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pa- sos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.