Programación 4

Diseño

Patrones de Diseño

Contenido

- Introducción
- Patrones de Diseño
- Singleton
- Composite
- State
- Observer

Introducción

- Los diseñadores expertos afirman que es casi imposible lograr un diseño flexible y reutilizable en el primer intento
- Los expertos logran buenos diseños mientras que los principiantes se ven abrumados por todas las opciones disponibles
- Se requiere de mucho tiempo para que los principiantes aprendan de qué se trata un buen diseño
- Evidentemente los expertos tienen un conocimiento que los principiantes no

Introducción (2)

- Algo que los expertos saben es que no deben resolver un mismo problema de una forma diferente cada vez
- En cambio, reutilizan soluciones que les dieron buenos resultados en el pasado
- Cuando encuentran una buena solución a un problema la usan una y otra vez cada vez que el mismo problema se les presenta
- Esa experiencia es lo que los convierte en expertos

Introducción (3)

- Al desarrollar un sistema orientado a objetos un diseñador enfrenta una serie de problemas a resolver
- Muchos de ellos aparecerán nuevamente en los siguientes proyectos independientemente del dominio de cada uno de ellos
- Se detectan entonces problemas de naturaleza muy similar entre sí
  - Que aparecen recurrentemente en el diseño de aplicaciones de diversos tipos (patrones)

Introducción (4)

- Problemas recurrentes
  - ¿Cómo acceder a objetos sin acoplarse directamente a ellos? **(Factory)**
  - ¿Cómo acceder a un conjunto de interfaces por medio de un punto de acceso común? **(Facade)**
  - ¿Cómo restringir que una clase tenga una sola instancia y que se tenga visibilidad global hacia ella? **(Singleton)**
  - ¿Cómo estructurar un conjunto de objetos con las mismas operaciones en donde algunos de ellos están compuestos de otros? **(Composite)**

Introducción (5)

- Problemas recurrentes
  - ¿Cómo puedo manejar diferentes estados de comportamiento en un objeto? **(State)**
  - ¿Cómo generar el esqueleto de un algoritmo y permitir variar algunos pasos del mismo? **(Template Method, Strategy)**
  - ¿Cómo controlar o uniformizar el acceso a un objeto? **(Proxy, Adapter)**
  - ¿Cómo reaccionar ante ciertos eventos ocurridos en un objeto? **(Observer)**
  - ¿Cómo acceder a las instancias de una colección? **(Iterator)**

Patrones

Patrones

patrón *sust* **1** modo usual según el cual algo ocurre, se desarrolla o es hecho **2** cosa o forma que representa un ejemplo a copiar

Patrones (2)

- Si fuera posible recordar los detalles de un problema atacado en el pasado y la forma en que este fue solucionado, se podría reutilizar esa experiencia en lugar de redescubrirla
- El propósito de los Patrones de Diseño es registrar esa experiencia para que otros la aprovechen

Patrones de Diseño

- Un Patrón de Diseño sistemáticamente da un nombre, motiva y explica un diseño general que se aplica a un problema de diseño recurrente en sistemas orientados a objetos
- Describe el problema, la solución, cuándo aplicar la solución y sus consecuencias

Patrones de Diseño (2)

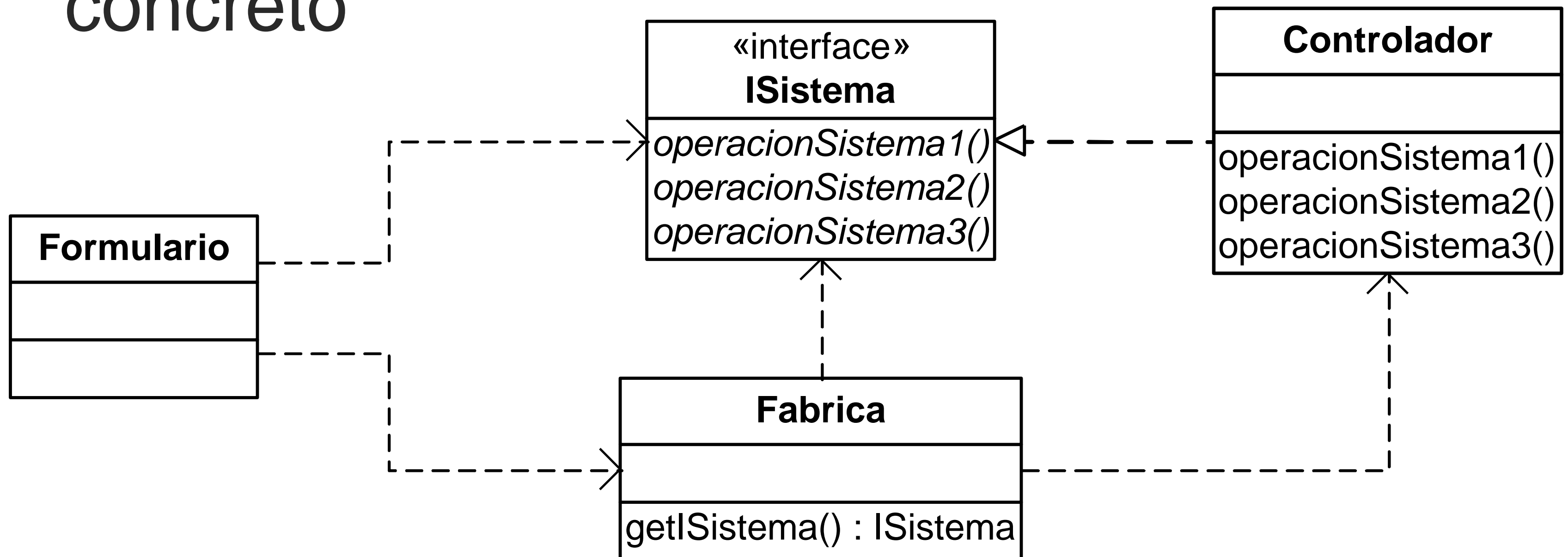
- Además provee guías para su implementación y ejemplos
- La solución es un arreglo general de objetos y clases que solucionan el problema
- La solución concreta es adaptada e implementada a partir de ello para resolver el problema en un contexto particular

Colaboración Abstracta

- La solución propuesta por un Patrón de Diseño es una colaboración abstracta (o paramétrica) que resuelve un problema tipo
- Dicha colaboración es por lo tanto interpretada como una “plantilla de colaboración”
- Si el problema concreto a resolver es compatible con el problema tipo se genera una colaboración concreta a partir de la plantilla
- Dicha colaboración será muy similar a la plantilla y resolverá el problema concreto

Patrones de Diseño
Ejemplo

- El mecanismo de Fábricas utilizado para comunicar el Formulario con el Controlador no es aplicable exclusivamente a este problema concreto

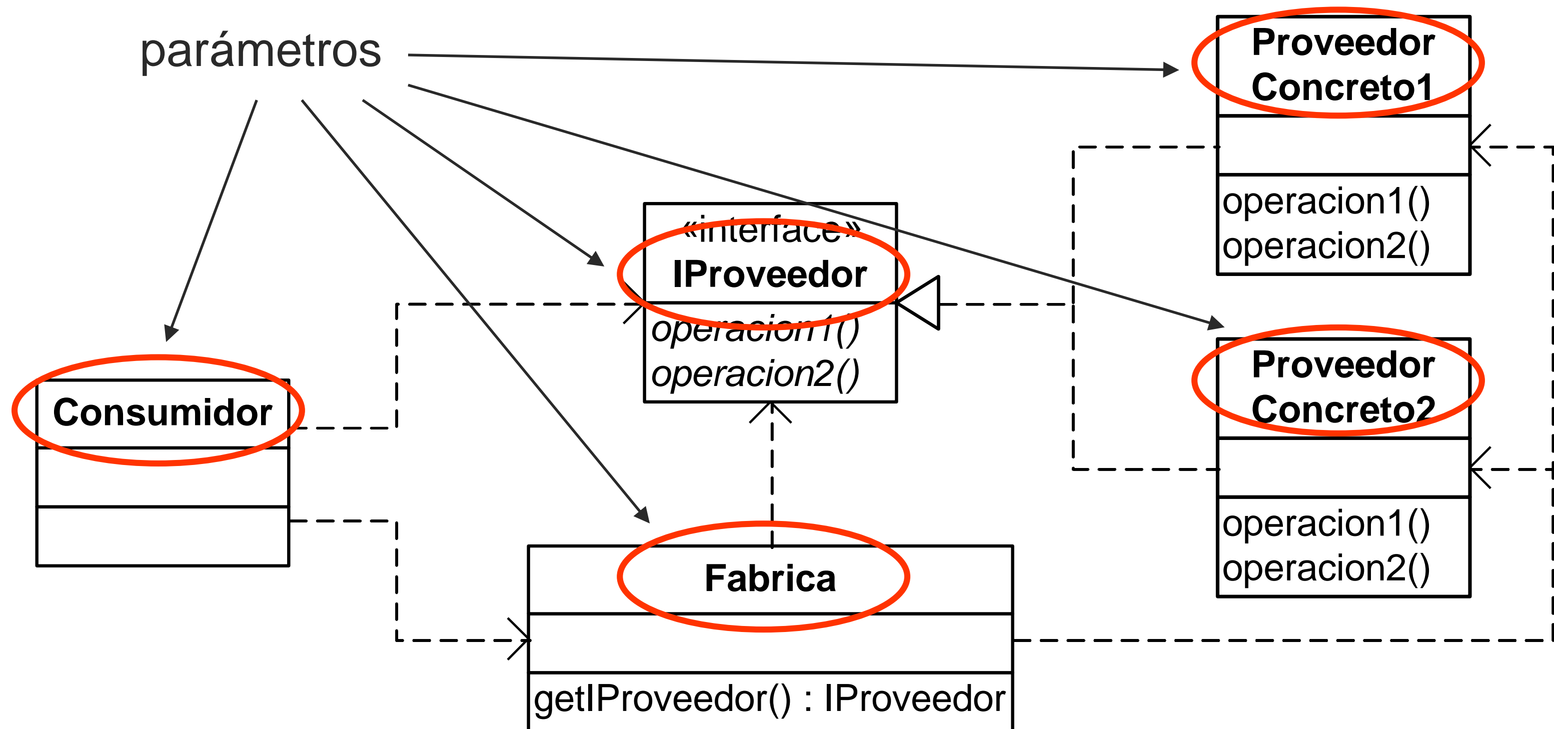


Patrones de Diseño
Ejemplo (2)

- Tampoco es aplicable exclusivamente para "comunicar un elemento de la Capa de Presentación con un Controlador"
- El problema tipo que el mecanismo de Fábricas permite resolver es
  - "Permitir visibilidad desde un consumidor hacia proveedores concretos sin que el consumidor quede acoplado directamente a éstos"

Patrones de Diseño
Ejemplo (3)

- La estructura de una colaboración que solucione el problema tipo puede ser:



Patrones de Diseño
Ejemplo (4)

- Para el problema concreto determinamos las siguientes correspondencias entre clases del diseño y parámetros de la colaboración

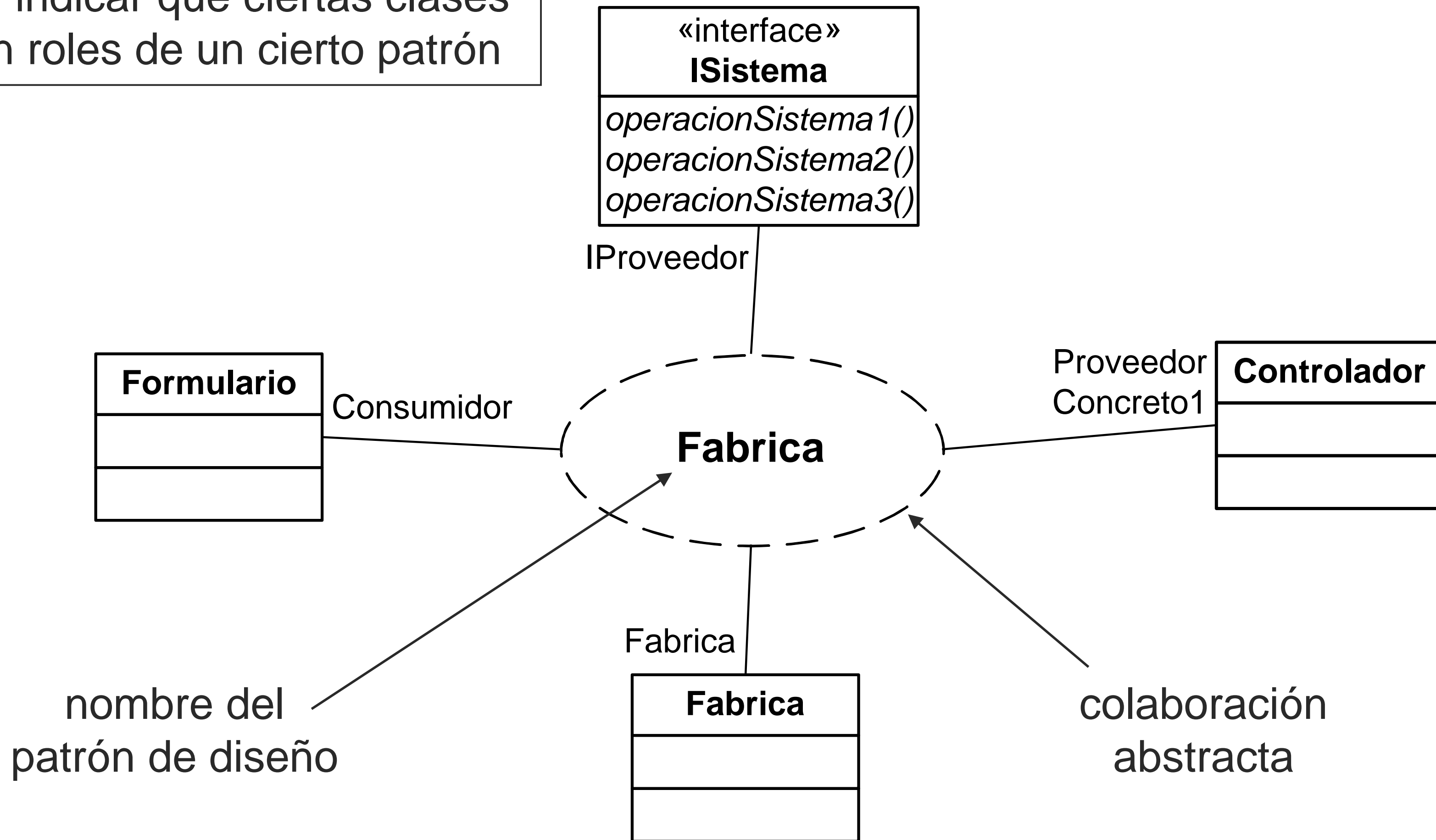
Clase	Parámetro
Formulario	Consumidor
Fabrica	Fabrica
ISistema	IProveedor
Controlador	ProveedorConcreto1

Patrones de Diseño
Ejemplo (5)

- Para explicitar la correspondencia definida es posible embeber la estructura propuesta por el patrón en el diseño de la solución (reemplazando los nombres de parámetros por los de las clases correspondientes)
- Esto conduce a un resultado como el mostrado originalmente
- En general la estructura resultante es idéntica a la del patrón por lo que aumenta la complejidad del diseño final

Patrones de Diseño
Ejemplo (6)

Alternativa: indicar que ciertas clases desempeñan roles de un cierto patrón



Patrones de Diseño
Descripción

- La descripción de un Patrón de Diseño está organizada de la siguiente manera
  - **Nombre:**
    - Se utiliza para referenciar a un problema, su solución y consecuencias en una o dos palabras
    - Asignarle un nombre a un patrón incrementa el vocabulario de diseño permitiendo diseñar en un nivel mayor de abstracción
  - **Problema:** describe cuándo utilizar el patrón, explica el problema tipo y su contexto

Patrones de Diseño
Descripción (2)

- Descripción (cont.)
  - **Estructura:** diagrama de clases que ilustra la estructura de la colaboración abstracta que soluciona al problema tipo
  - **Participantes:** descripción de las clases que forman parte de la estructura y sus responsabilidades
  - **Interacciones:** diagramas de interacción que ilustran el funcionamiento de la colaboración abstracta

Patrones de Diseño
Descripción (3)

- Descripción (cont.)
  - **Consecuencias:** Comentarios, discusiones, sugerencias y advertencias que permitan entender e implementar el patrón

La estructura e interacciones conforman la colaboración abstracta que soluciona el problema tipo

Patrones de Diseño
Sugerencia

- Comprender el detalle de la colaboración propuesta por un patrón es fundamental para su aplicación
- Tan importante como esto es comprender
  - El problema tipo
  - El contexto de aplicabilidad de un patrón
- Es de muy poca utilidad conocer la solución a un problema desconocido o que no sabemos reconocer

Patrones de Diseño
Sugerencia (2)

- Conocer las consecuencias de la aplicación de un patrón es también fundamental
- Aplicar la solución abstracta propuesta a un problema concreto puede no ser siempre adecuado
- Esto depende de las características particulares del problema concreto
  - Por ejemplo: En ocasiones la cantidad de clases puede quedar demasiado grande

Singleton

Singleton :: Motivación

- Controlador de fachada para el cual existirá una única instancia en el sistema
- La fábrica desea acceder fácilmente a dicha instancia

Singleton

- Problema Tipo:

> "Asegurar que una clase tenga una sola instancia y proveer un acceso global a ella"

- Aplicabilidad:
  - Debe existir una única instancia de una clase
  - Dicha instancia debe ser accesible por cualquier cliente de la clase

Singleton (2)

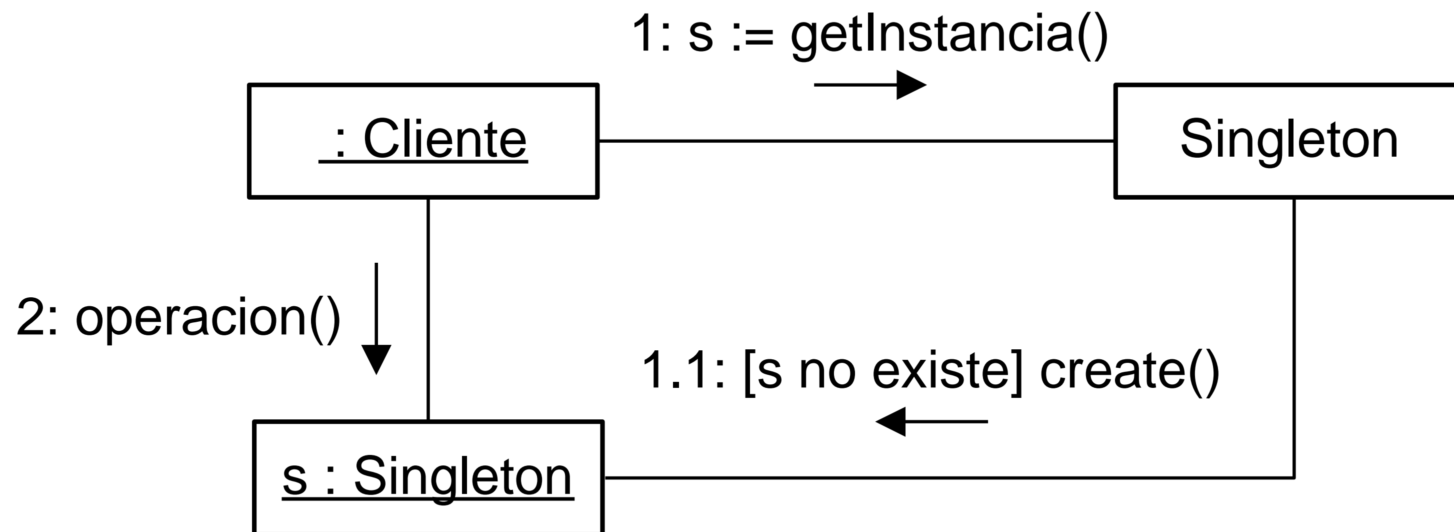
- Estructura

Singleton
-instancia : Singleton
-Singleton() +getInstancia() : Singleton +operacion()

- Participantes
  - **Singleton:** provee una operación de clase (`getInstancia()`) que permite acceder a la única instancia

Singleton (3)

- Interacciones



Singleton (4)

- Consecuencias
  - Se provee acceso controlado a una única instancia
  - Se permiten variantes en las que se varíe el número máximo de instancias
  - Un cliente puede liberar la memoria de la instancia
  - Derivar una clase Singleton resulta complejo
- Aplicaciones
  - Las fábricas y manejadores suelen ser singleton
  - Algunos controladores también

Composite

Composite :: Motivación

- Editor gráfico que permite representar figuras geométricas
- Se desea agruparlas y realizar acciones como un conjunto, por ejemplo moverlas

Composite

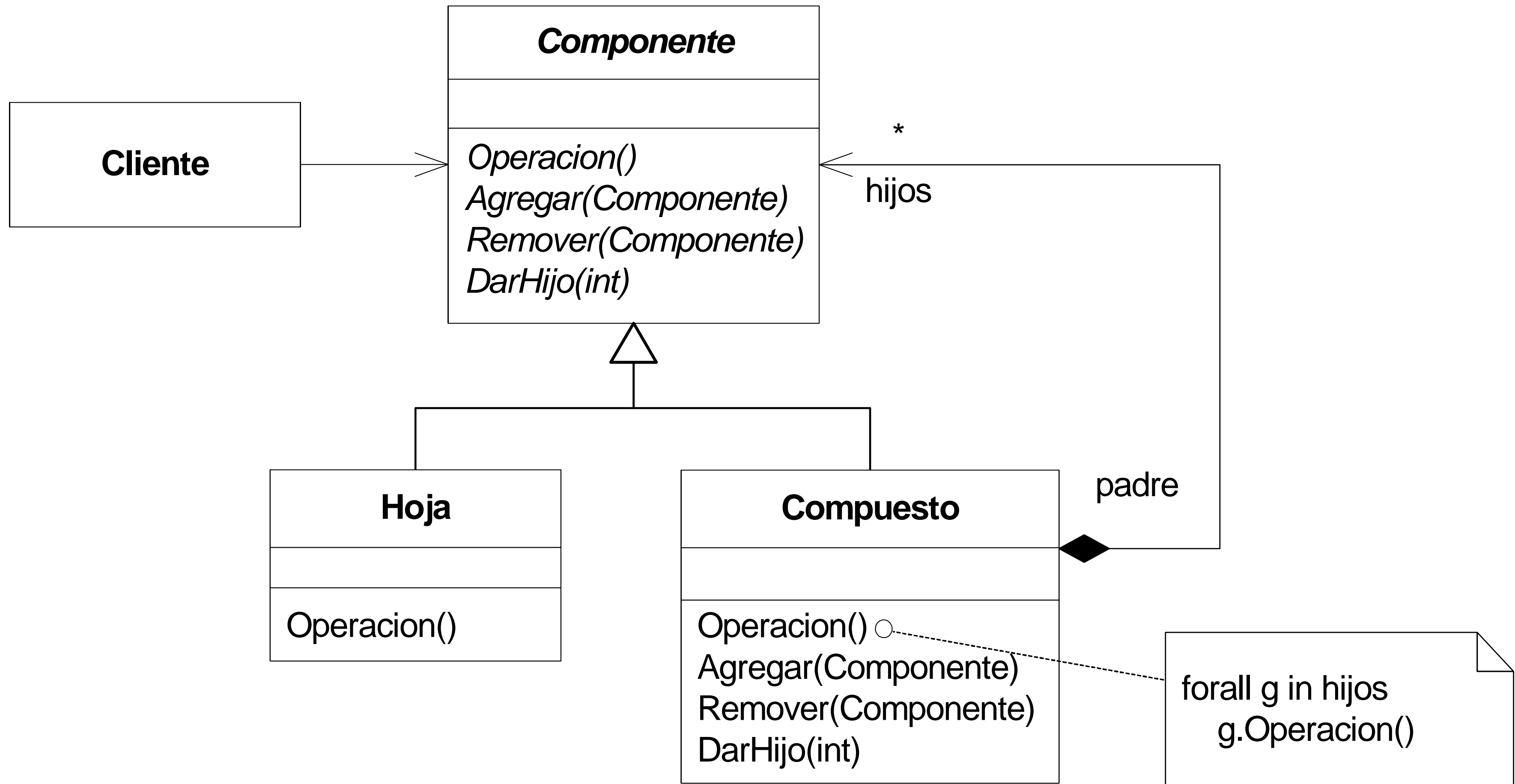
- Problema Tipo:

“Componer objetos en estructuras arborescentes para representar jerarquías de objetos compuestos y tratar uniformemente los mismos.”

- Aplicabilidad:
  - Se debe representar una jerarquía de objetos donde algunos de ellos se componen de otros de la misma jerarquía
  - Se desea que los clientes ignoren las diferencias entre objetos compuestos y objetos primitivos, y los traten uniformemente

Composite (2)

- Estructura



Composite (3)

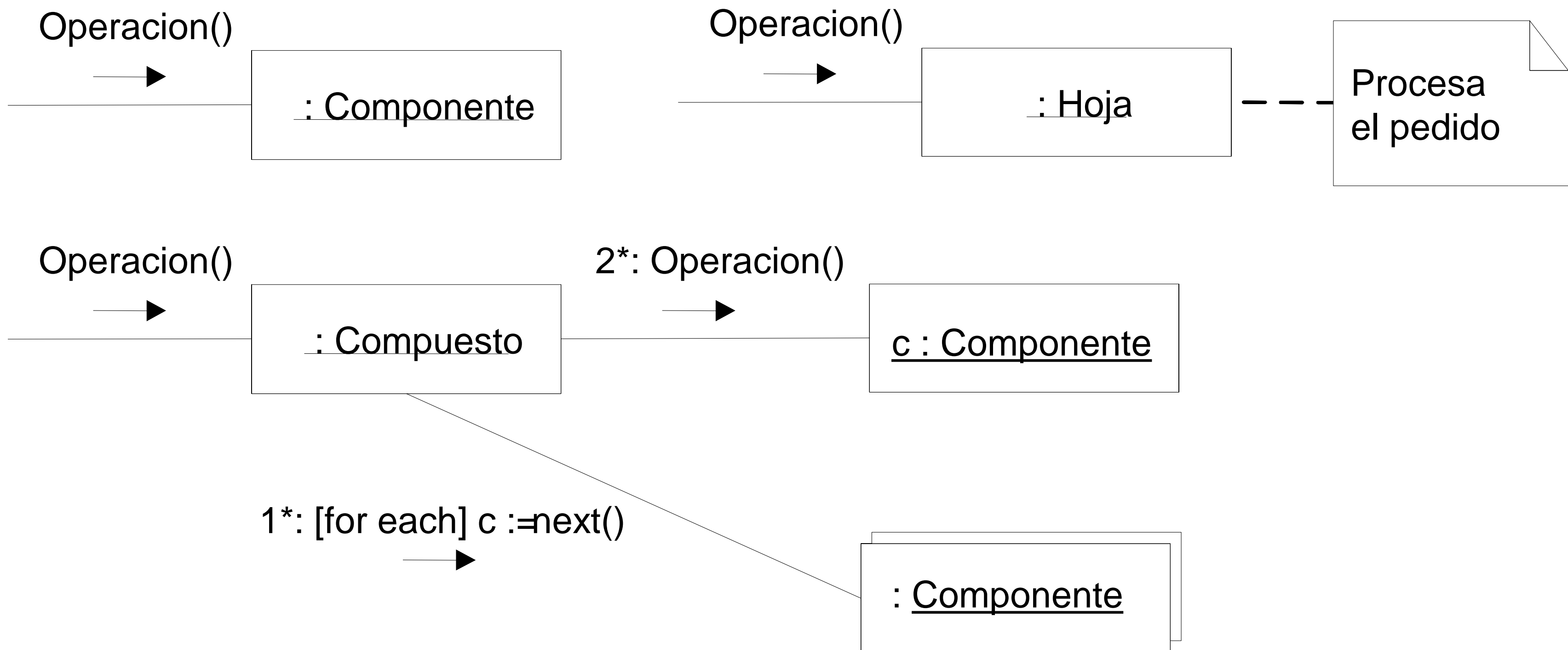
- Participantes
  - **Componente:**
    - Declara el conjunto de operaciones comúnes a los objetos en la jerarquía e implementa el comportamiento común por defecto
    - Permite acceder y manipular los componentes hijo, y opcionalmente los componentes padre
  - **Hoja:**
    - Define el comportamiento de los objetos primitivos en la composición

Composite (3)

- Participantes (cont.)
  - **Compuesto:**
    - Define el comportamiento de los objetos compuestos, delegando a sus hijos el comportamiento relacionado con ellos
    - Mantiene una referencia a sus componentes hijo
  - **Cliente:**
    - Manipula los objetos de la composición a través de una instancia Componente

Composite (4)

- Interacciones



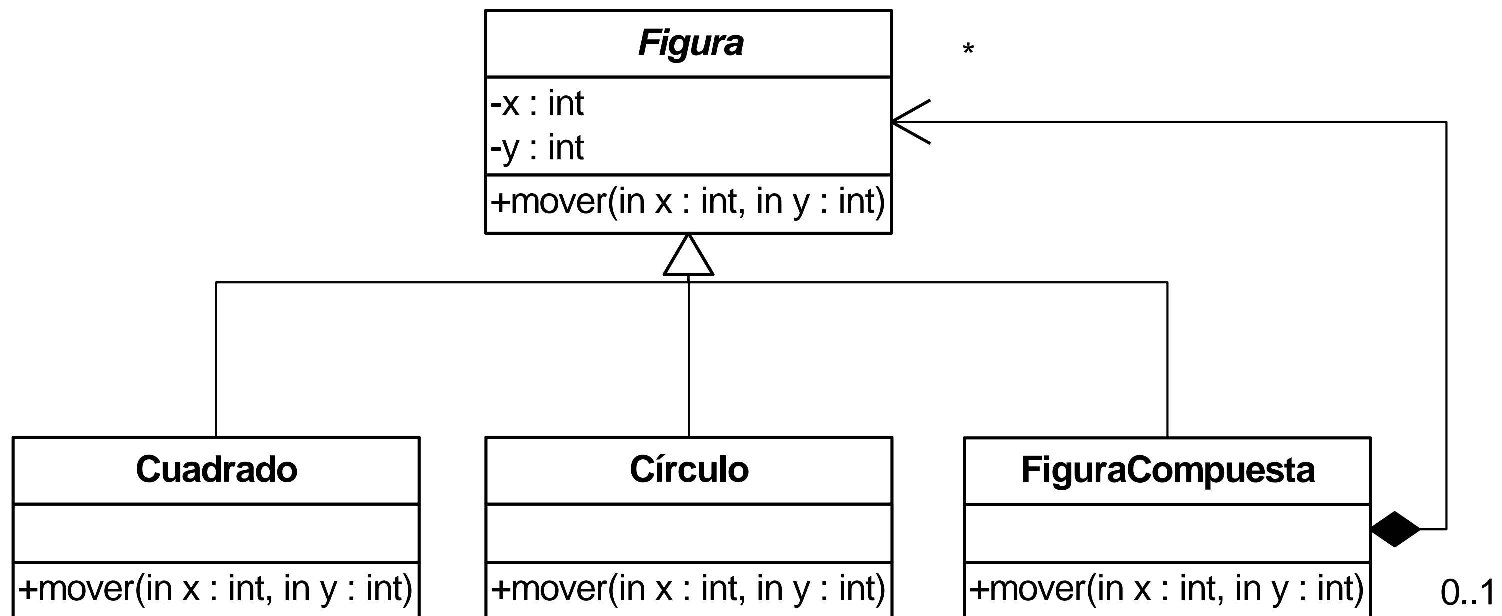
Compuesto puede pre/post procesar información al invocar a sus hijos.
Además, no tiene porqué invocar a todos sus hijos.

Composite (5)

- Consecuencias
  - El cliente manipula objetos de la jerarquía sin discriminar entre objetos primitivos y objetos compuestos (elimina lógica condicional)
  - Es fácil agregar nuevos componentes a la jerarquía sin necesidad de modificar el código del cliente
  - Dificulta restringir el tipo de los objetos hijo de un objeto compuesto

Composite (6)

- Ejemplo
  - Editor de figuras que permite agruparlas y moverlas conjuntamente



State

State :: Motivación

- Existe una puerta automática controlada por control remoto de un solo botón
- Cuando se presiona el botón la puerta se abre o cierra dependiendo de si se encuentra está cerrada o abierta, respectivamente

State

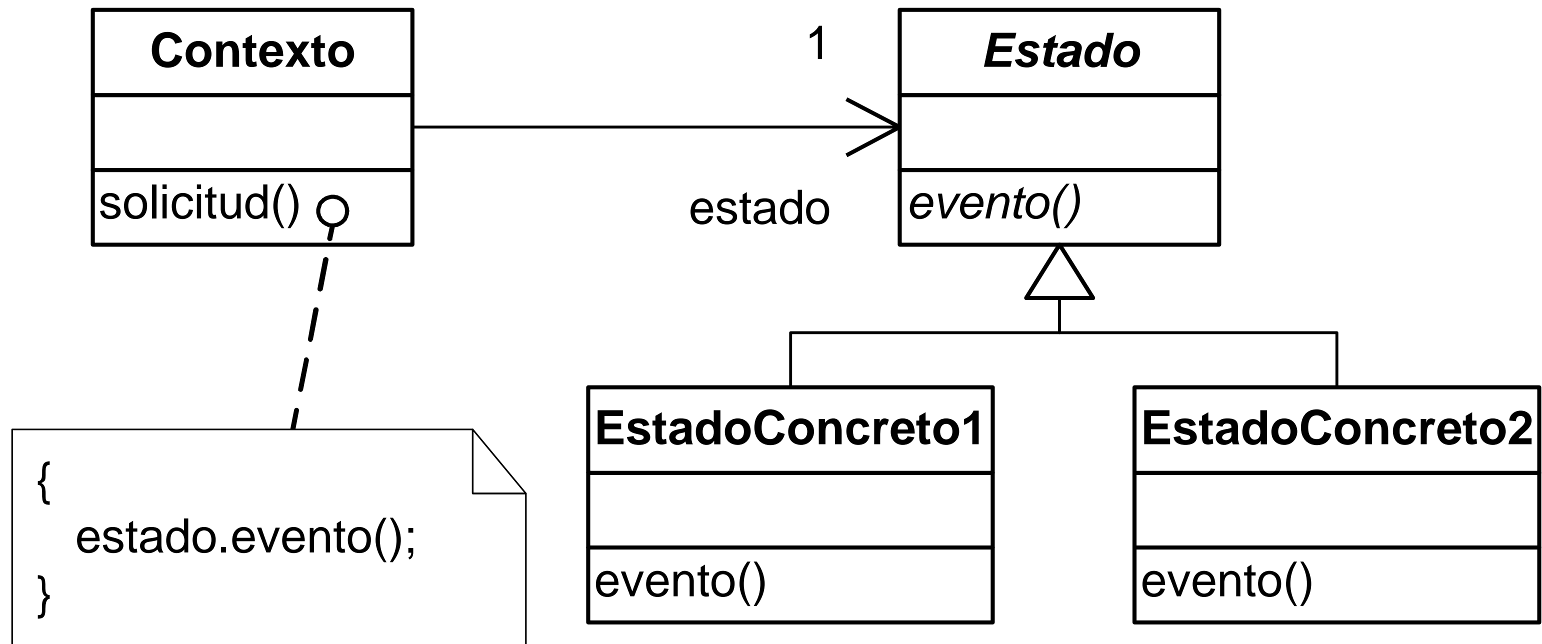
- Problema Tipo:

"Permitir que un objeto varíe su comportamiento cuando su estado interno cambie. El objeto parecerá haber cambiado de clase"

- Aplicabilidad:
  - El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de ese estado
  - Las operaciones de un objeto tienen fragmentos condicionales dependientes de su estado

State (2)

- Estructura



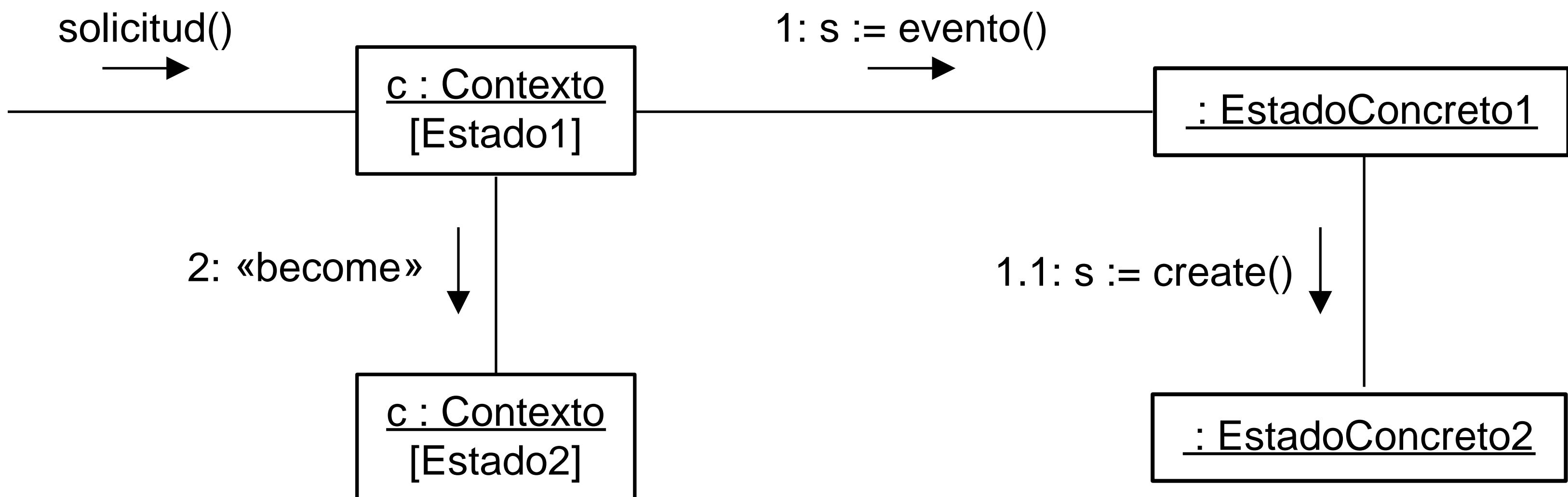
No se incluyen las dependencias

State (3)

- Participantes
  - **Contexto:**
    - Es la clase de objetos cuyo comportamiento varía al cambiar el estado interno
    - Mantiene una referencia a un estado concreto
    - Delega el comportamiento variable al estado actual
  - **Estado:**
    - Generaliza los diferentes estados concretos del Contexto
  - **EstadoConcreto:**
    - Cada una de estas clases implementa un comportamiento particular del Contexto que sea dependiente del estado

State (4)

- Interacciones



El Contexto, estando en Estado1, al recibir una “solicitud” cambia a Estado2

State (5)

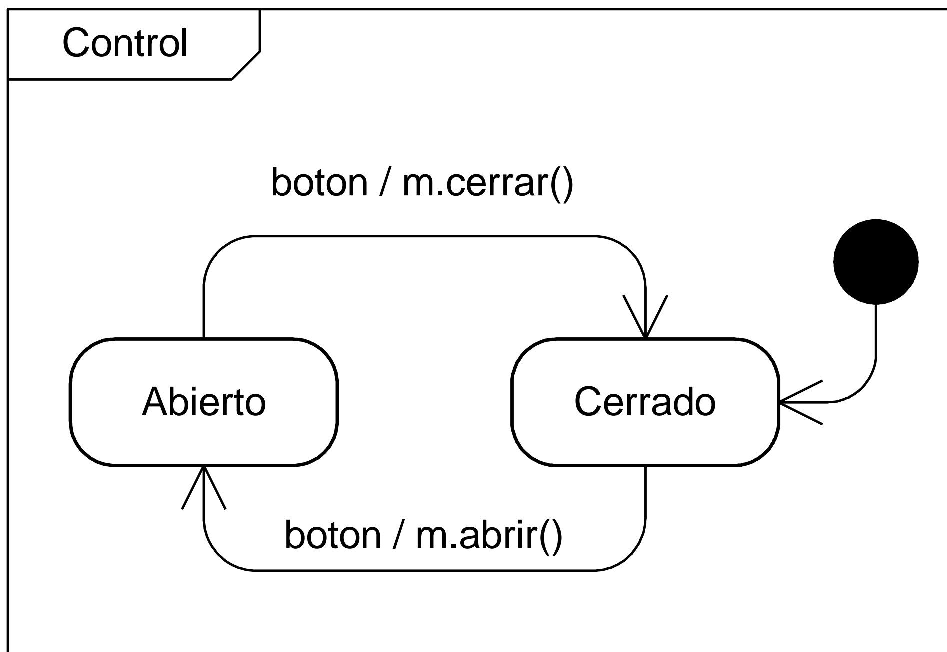
- Consecuencias
  - Sin Garbage Collector es necesario asignar a alguien la responsabilidad de eliminar, luego de una transición, la instancia que representa el estado anterior
  - Los estados concretos pueden tener estado propio
    - Si no lo tienen pueden ser diseñados como singletons
  - En casos en que el Contexto tiene muchos estados la cantidad de clases (a causa de los estados concretos) puede ser muy grande

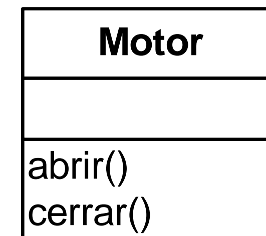
State (6)

- Consecuencias (cont.)
  - Es simple agregar nuevos estados
    - Sin embargo es necesario modificar estados concretos existentes para incluir transiciones al estado nuevo
  - Existen diferentes variantes en la implementación de las transiciones
  - El Contexto puede pasarse como parámetro en los eventos
  - Permite eliminar la lógica condicional del Contexto
  - El estado concreto actual puede usarse para determinar el estado del Contexto

State (7)

- Ejemplo
  - Puerta automática controlada por control remoto de un solo botón





`Motor` maneja el motor que abre y cierra la puerta

Observer

Observer :: Motivación

Un puesto de venta tiene varios productos para vender, de los cuales se necesita controlar su stock. Cuando el stock de alguno de ellos llegue a cero, deberá avisarse al depósito (para que se preparen a recibir los nuevos productos) y a los proveedores (para que provean de más stock).

Observer

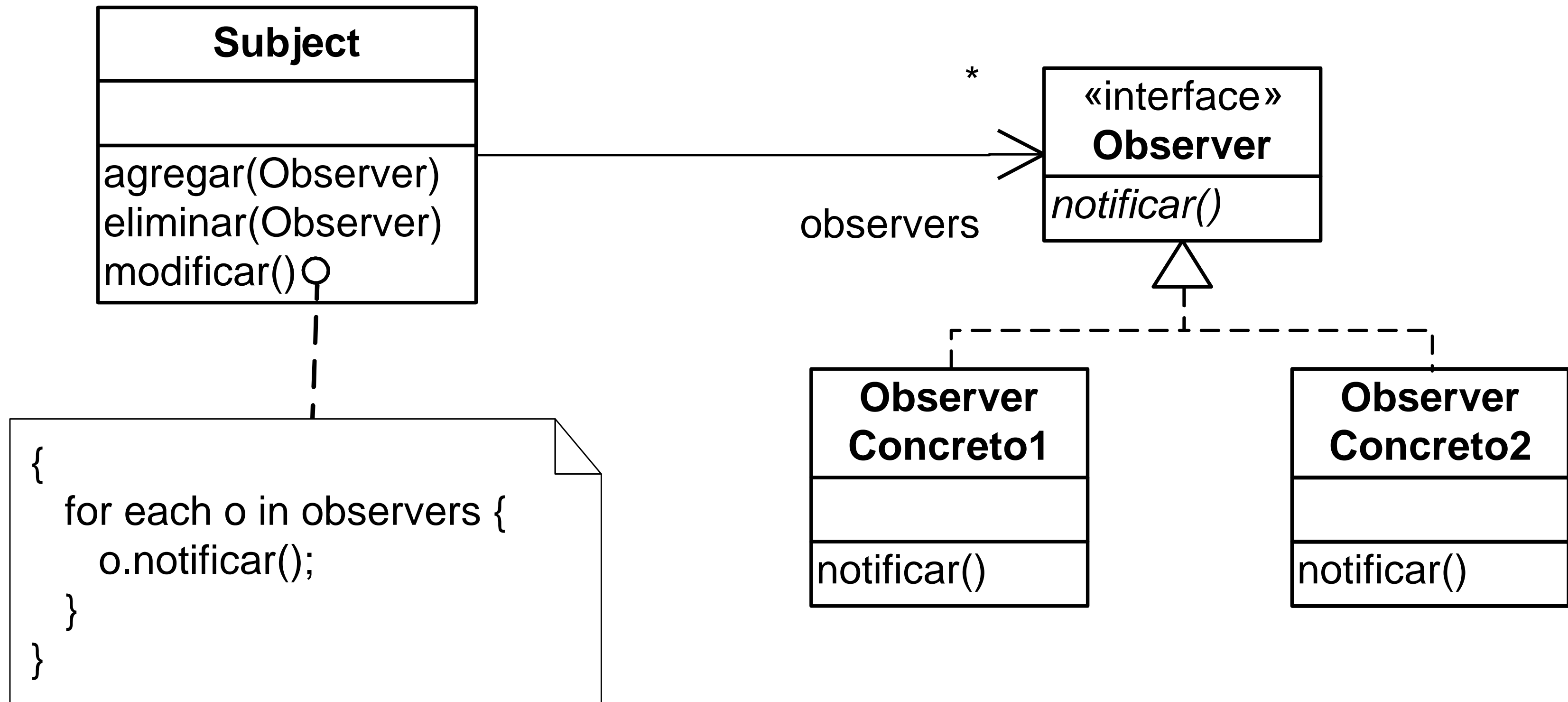
- Problema Tipo

> "Definir una dependencia 1-n entre objetos, de forma que cuando uno cambie de estado todos los dependientes sean notificados"

- Aplicabilidad
  - Cuando un cambio en un objeto requiere cambiar otros y no se desea saber cuales
  - Cuando un objeto debe notificar otros objetos de diferente naturaleza y sin estar acoplado a ellos

Observer (2)

- Estructura

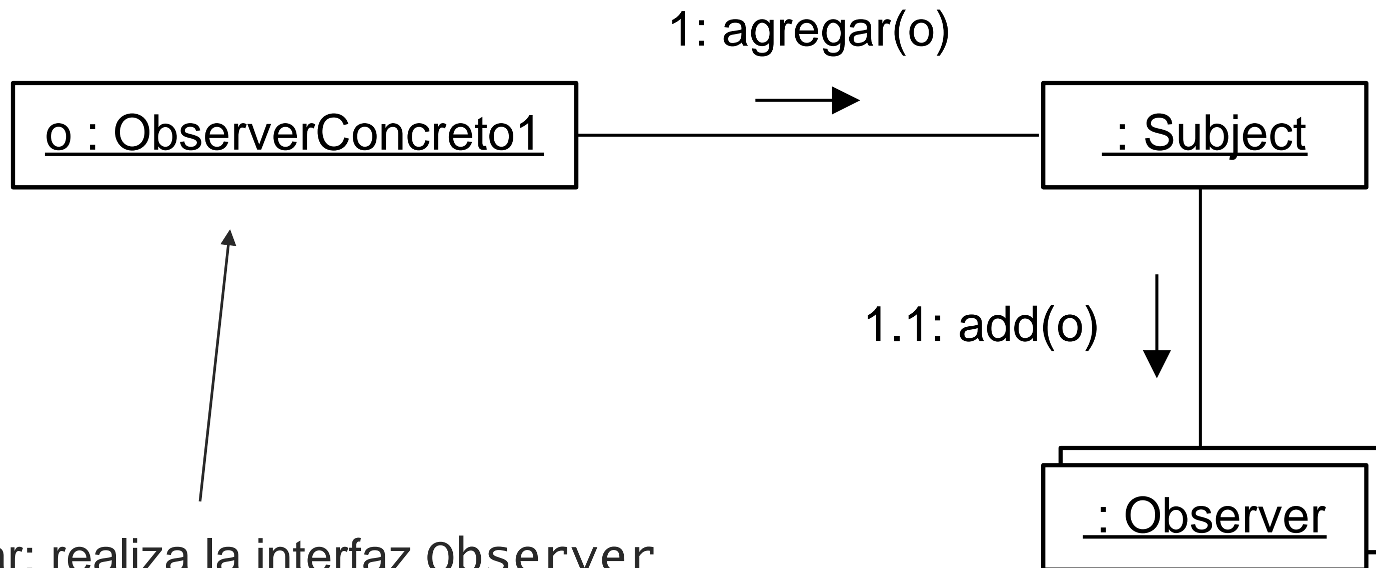


Observer (3)

- Participantes
  - Subject
    - Dispone de información que es de interés para otros objetos
    - Registra un conjunto de objetos interesados y los notifica cuando lo considera necesario
  - Observer: declara la operación por la cual los interesados son notificados
  - Observer Concreto: un interesado en los avisos que el Subject tenga para enviar

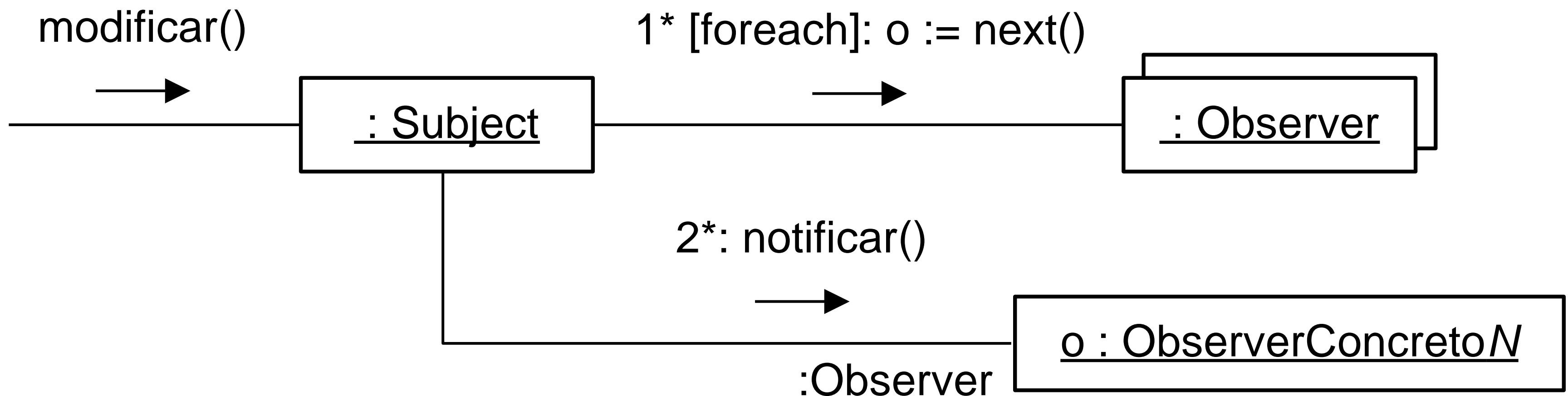
Observer (4)

- Interacciones – `agregar()`



Observer (5)

- Interacciones – `modificar()`



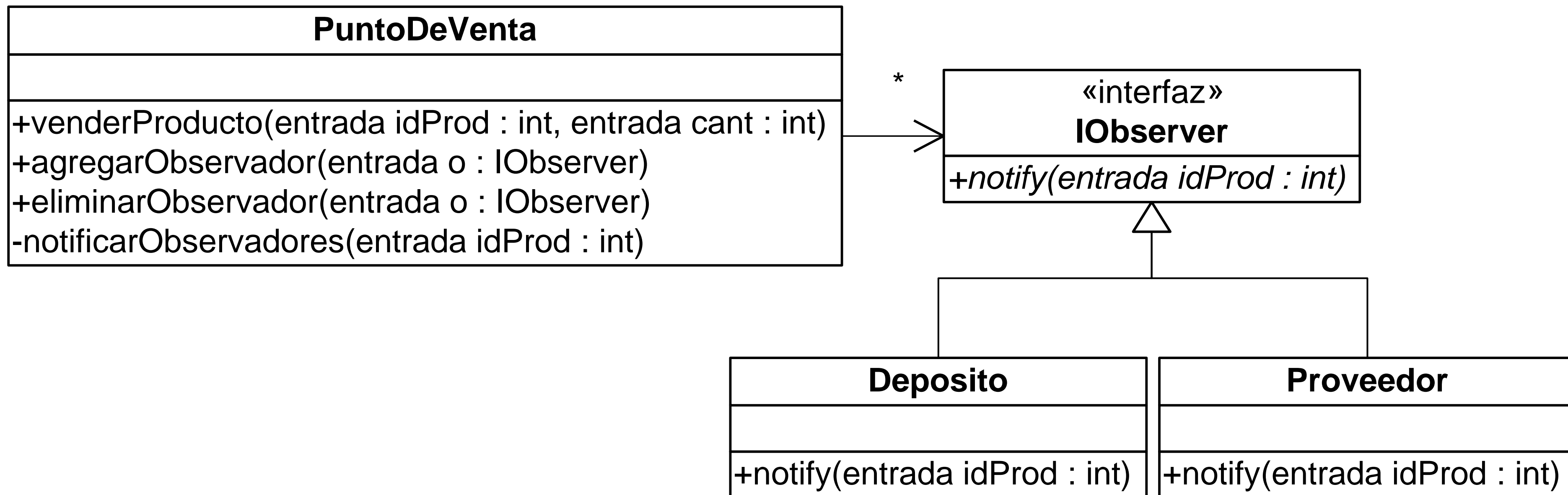
Observer (6)

- Consecuencias
  - No existe acoplamiento entre el Subject y los observadores concretos
  - Este mecanismo es aplicable a la realización de broadcasts
  - Cambios inesperados en el Subject por parte de un observador concreto puede causar notificaciones en cascada hacia los otros observadores concretos

Observer (7)

- Consecuencias (cont.)
  - Por defecto, el orden de notificación es aleatorio
  - La notificación es secuencial por lo que se aguardará a que cada observador procese la notificación
  - La notificación puede llevar parámetros
  - En casos en que un observador observe a mas de un Subject a la vez puede ser necesario que éste se identifique al momento de notificar

Observer (8)



Patrones

Sugerencias

- Pasos a seguir
  - Analizar las características del problema a resolver (Aplicabilidad)
  - Determinar si existe algún patrón de diseño que pueda ser aplicado
  - Aplicar la solución propuesta por el patrón en el diseño existente (Estructura e Interacciones)
- Errores comunes
  - Aplicar un patrón cuando no están dadas las condiciones para ello, porque no se entendió bien el problema