Programación 3

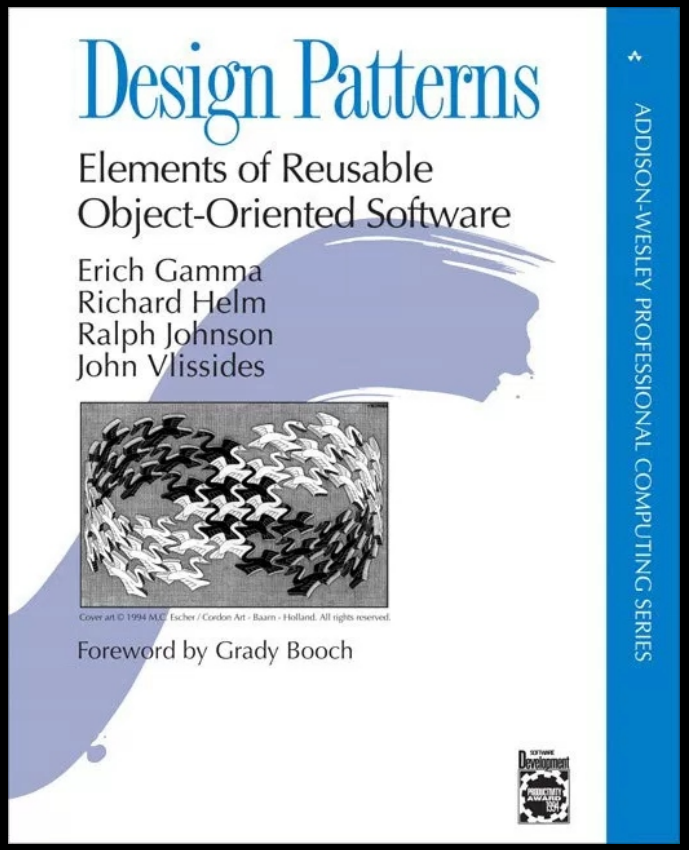
Orientación a objetos con **C#**

7. Patrones de diseño

maurogullino@gmail.com

Design Patterns

- son soluciones reusables a problemas comunes en el diseño de software OOP



- fueron catalogados por E. Gamma y otros en 1994 (GoF)
- aportan nombre, diagramas, análisis y código de ejemplo

Originales del GoF

De creación

Abstract Factory
Builder
Factory Method
Prototype
Singleton

Estructurales

Adapter
Bridge
Composite
Decorator
Façade
Flyweight
Proxy

De comportamiento

Chain of Responsability
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

MVC

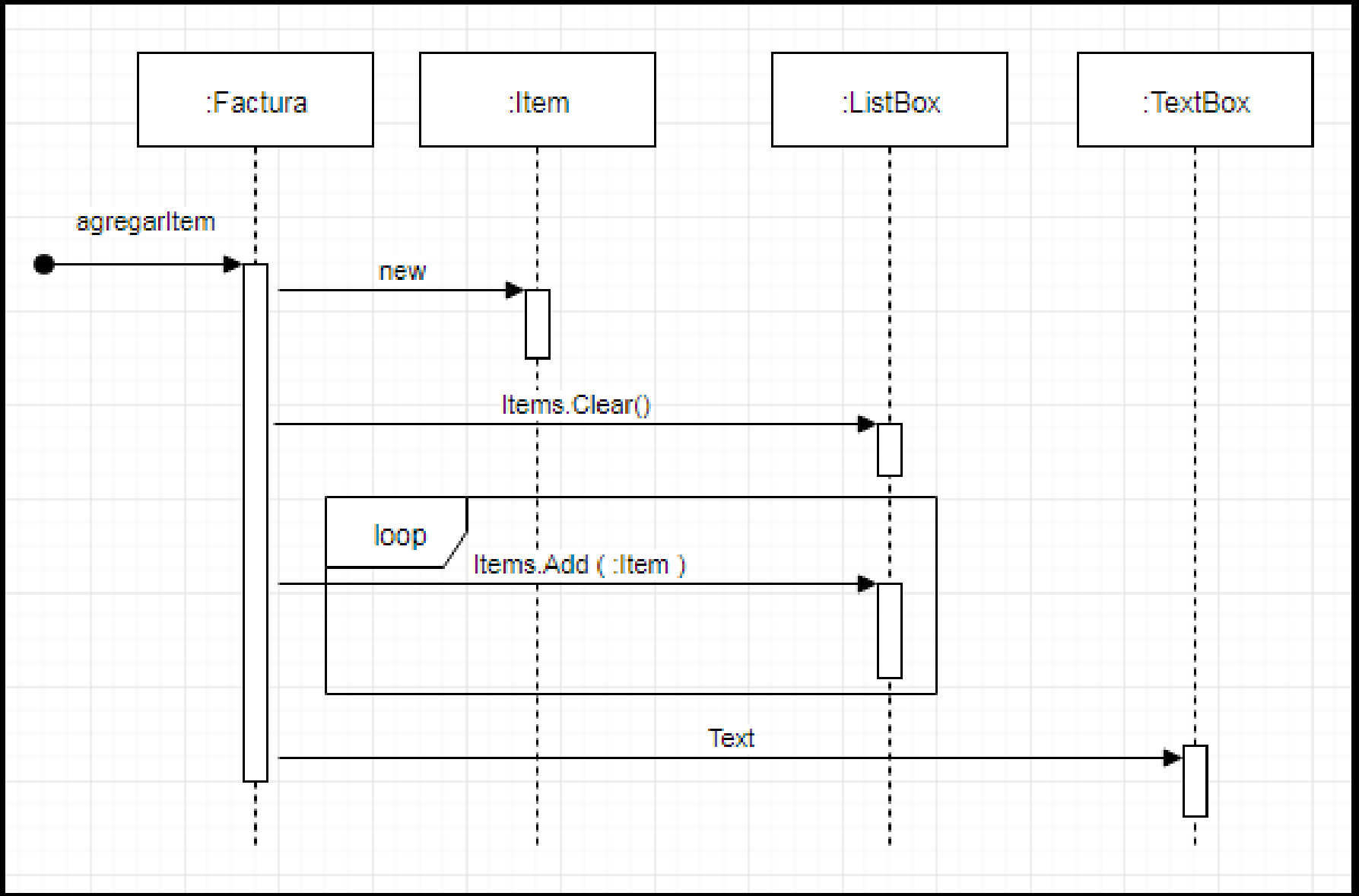
Aspectos deseables

- aportan flexibilidad al software
- están históricamente probados

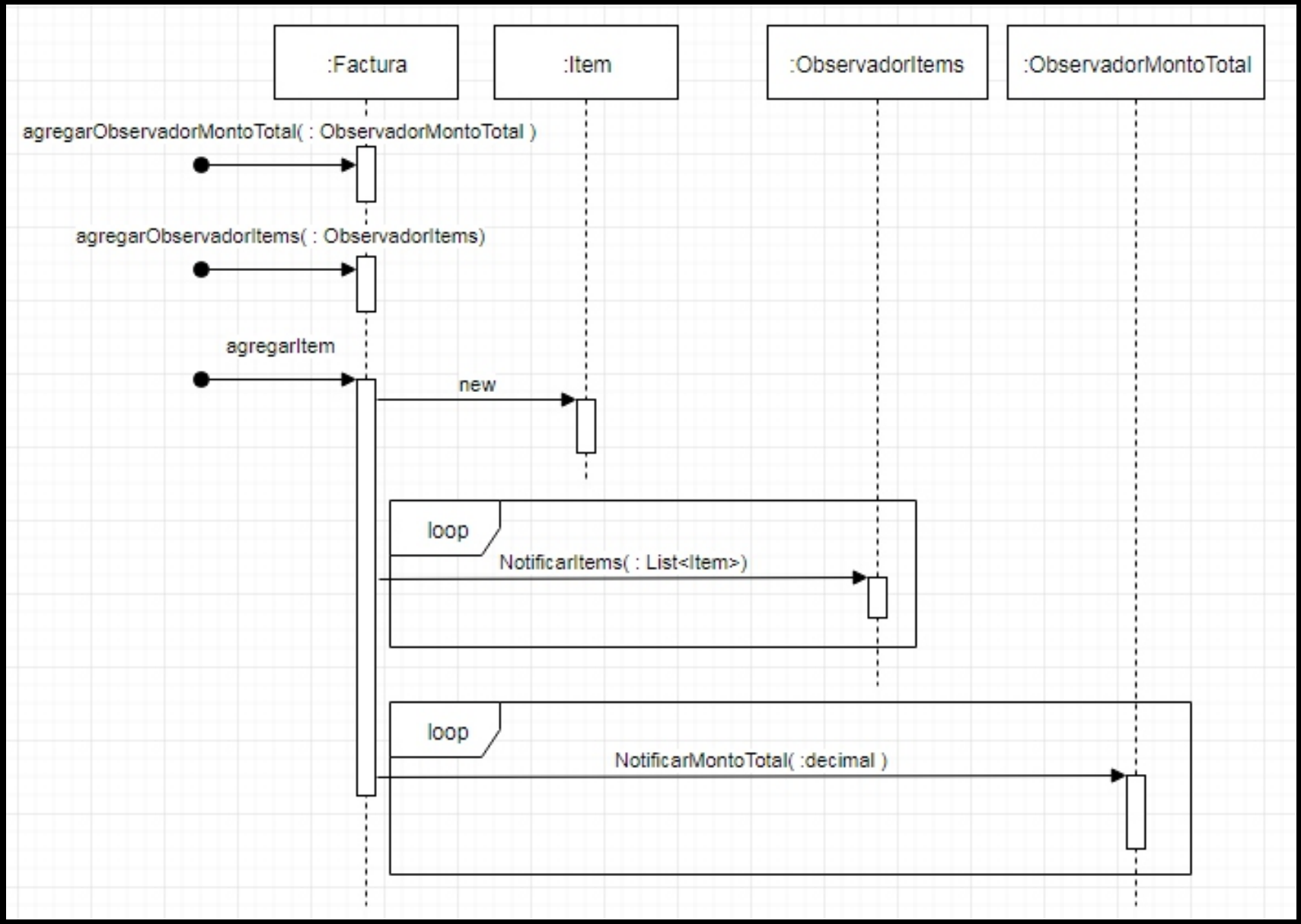
Aspectos no-deseables

- agregan clases al diseño (complejidad)
- son soluciones no obvias
- dificultan la comprensión para quien no los conoce
- esto se traduce en costos

Observer (diseño sin patrón)



Observer (aplicando el patrón)



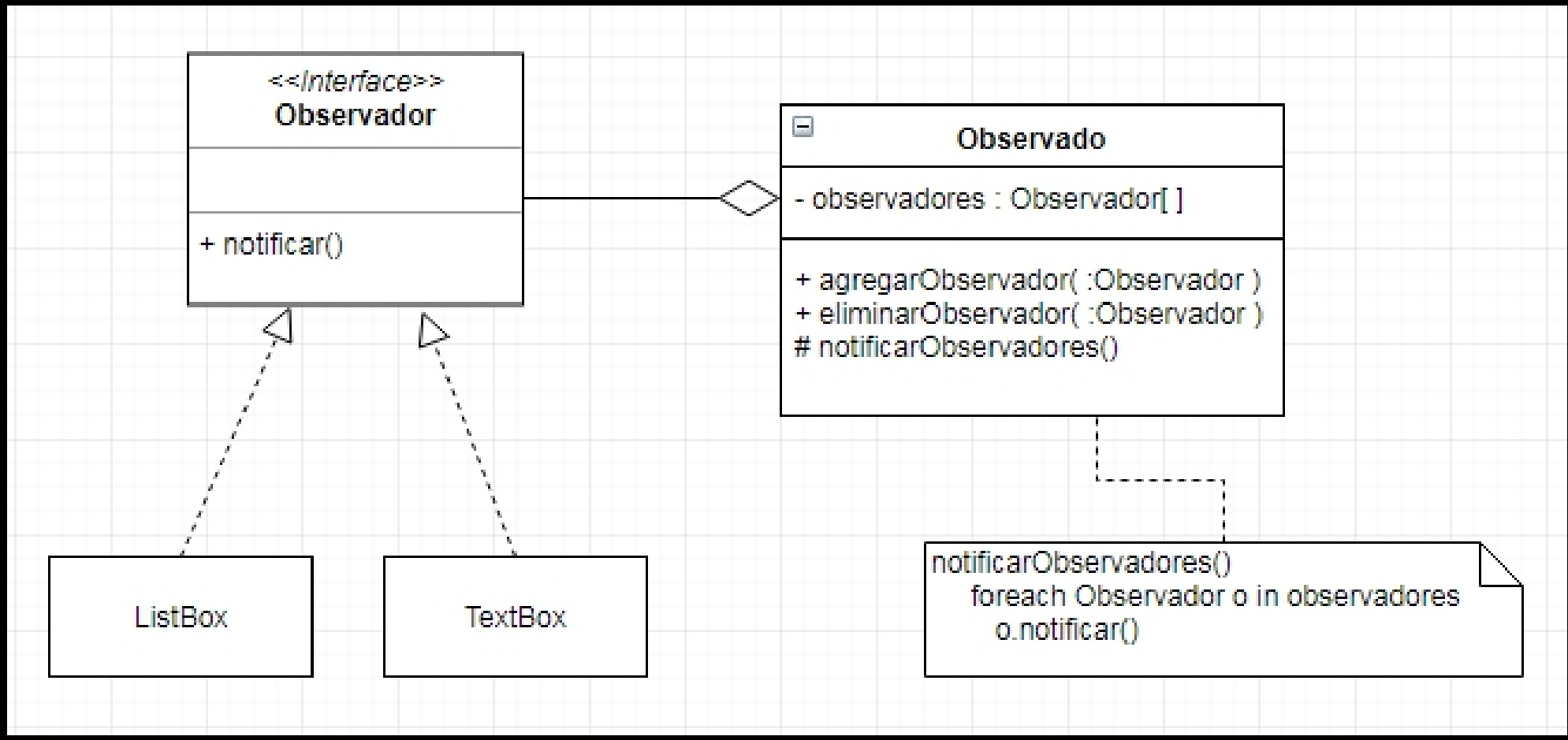
Observer

Permite variar: el número de objetos que dependen de otro y cómo se mantiene actualizado el objeto dependiente

Cuándo

- cuando un cambio en un obj requiere cambiar otros, y no sabemos cuántos son
- cuando un obj debe notificar a otros sin saber quiénes son

Observer (diagrama de clases)



Strategy

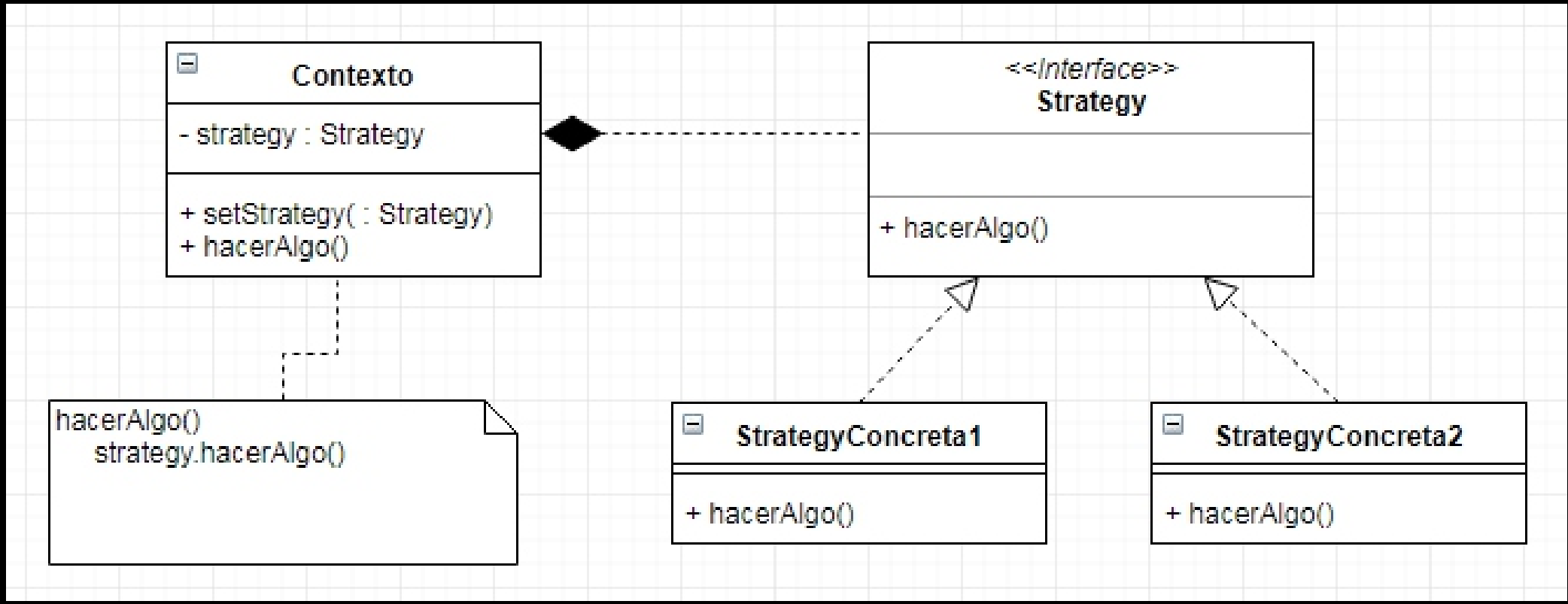
Strategy

Permite variar: un algoritmo

Cuándo

- cuando varias clases relacionadas difieren solo en su comportamiento
- cuando se necesitan distintas variantes de un algoritmo
- cuando un algoritmo utiliza muchos datos que es mejor encapsular (para evitar su conocimiento)

Strategy (diagrama de clases)



Patrones: diseñar para el cambio

· “Cada patrón permite que algún aspecto del sistema varíe independientemente de los otros”

· “... centrarse en encapsular el concepto que puede variar, un tema común a muchos patrones”

State

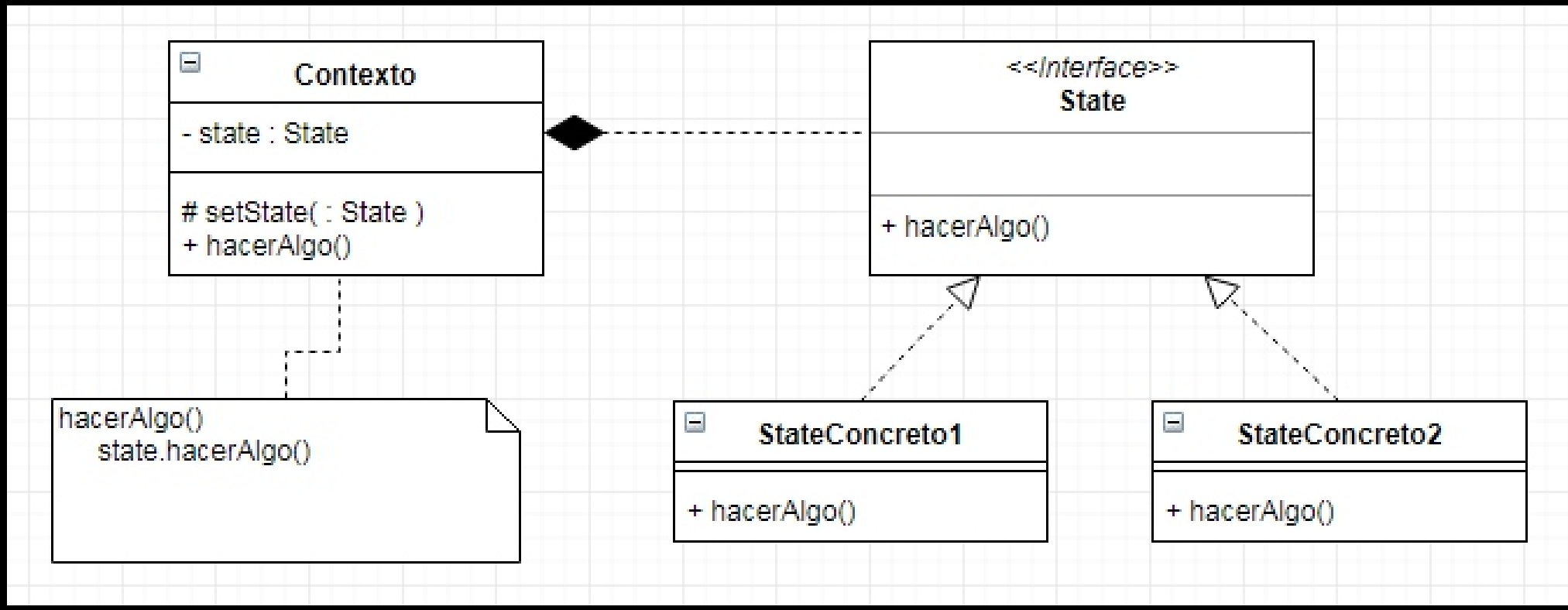
State

Permite variar: el estado interno de un obj

Cuándo

- el comportamiento de un obj depende de su estado, y debe cambiar en tiempo de ejecución
- Cuando las operaciones tienen grandes if con ramas que dependen del estado del obj

State


Contexto
- state : State
setState(: State)
+ hacerAlgo()
<<Interface>>
State
+ hacerAlgo()
hacerAlgo()
state.hacerAlgo()
StateConcreto1
+ hacerAlgo()
StateConcreto2
+ hacerAlgo()

Composite

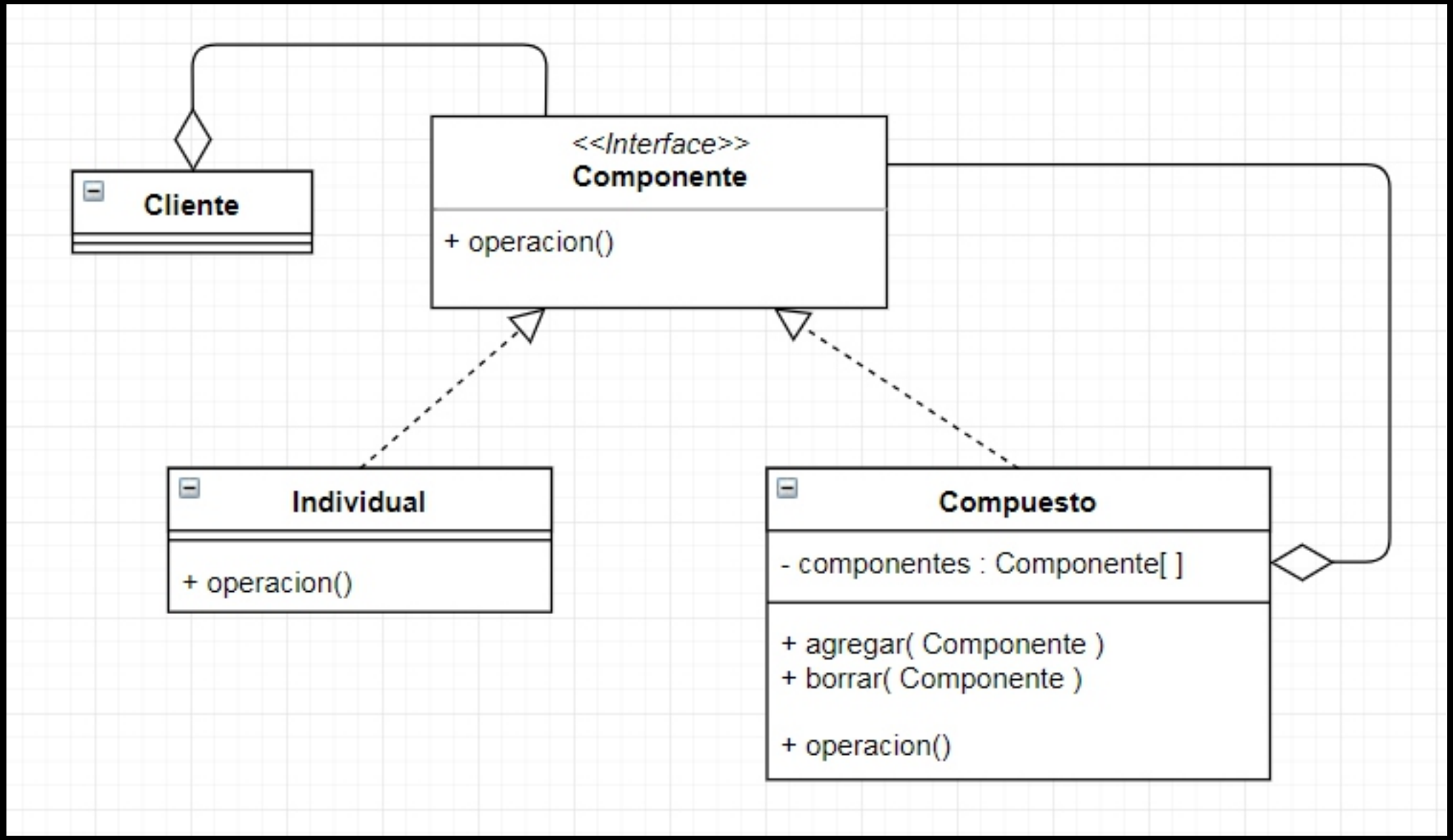
Composite

Permite variar: la estructura y composición de un obj

Cuándo

- para representar jerarquías de obj parte-todo
- cuando los clientes deban trabajar con obj individuales o agrupamientos de manera uniforme

Composite


Cliente
<<Interface>>
Componente
+ operacion()
Individual
+ operacion()
Compuesto
- componentes : Componente[ ]
+ agregar(Componente)
+ borrar(Componente)
+ operacion()

Visitor

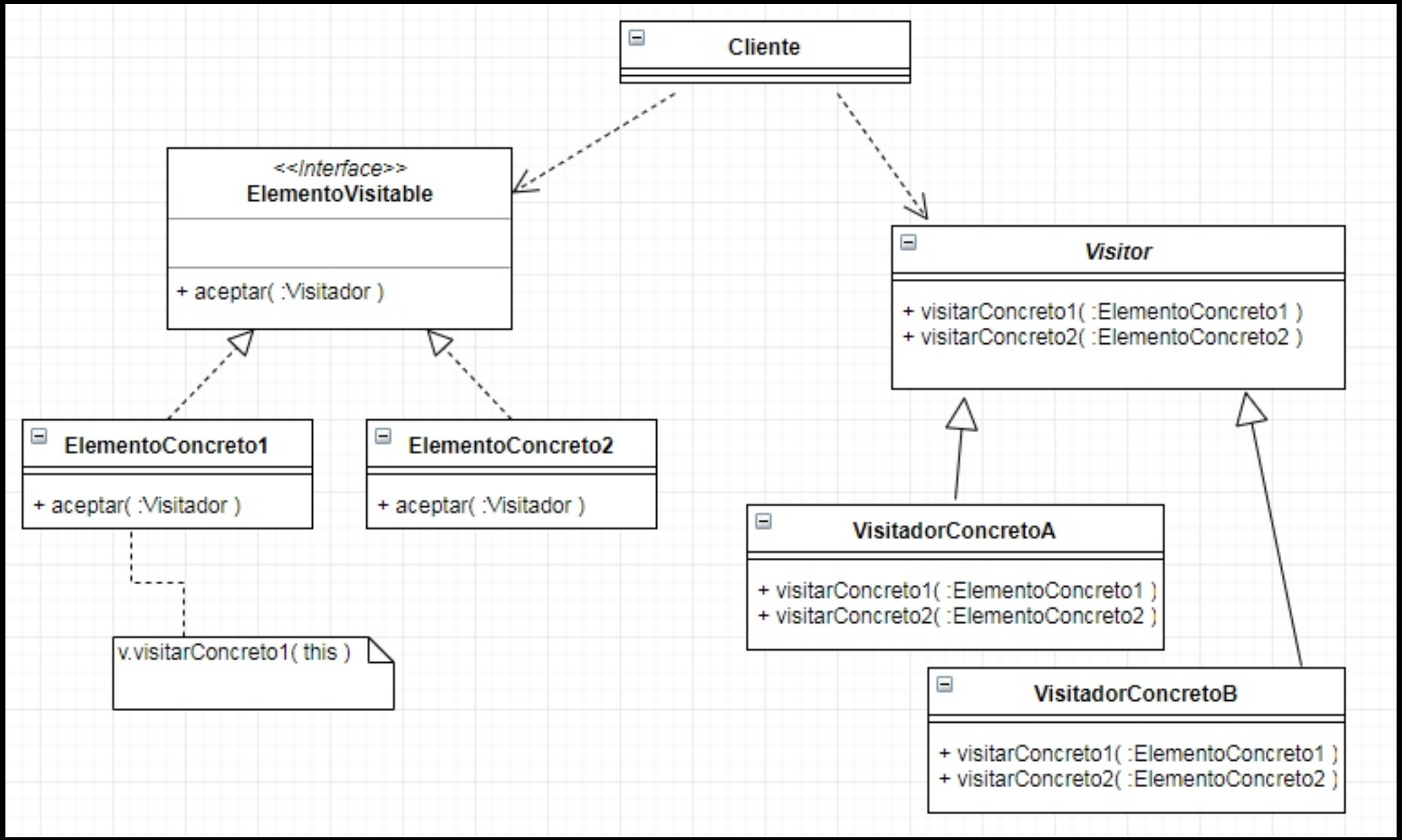
Visitor

Permite variar: las operaciones que se aplican a objs sin cambiar sus clases

Cuándo

- hay muchas clases con distintas interfaces
- hay muchas operaciones distintas y no relacionadas y no queremos “contaminar” la clases con todas esas operaciones
- las clases de la estructura no varían, sólo las operaciones

Visitor


Cliente
<<Interface>>
ElementoVisitable
+ aceptar(:Visitador)
Visitor
+ visitarConcreto1(:ElementoConcreto1)
+ visitarConcreto2(:ElementoConcreto2)
ElementoConcreto1
+ aceptar(:Visitador)
ElementoConcreto2
+ aceptar(:Visitador)
v.visitarConcreto1(this)
VisitadorConcretoA
+ visitarConcreto1(:ElementoConcreto1)
+ visitarConcreto2(:ElementoConcreto2)
VisitadorConcretoB
+ visitarConcreto1(:ElementoConcreto1)
+ visitarConcreto2(:ElementoConcreto2)