IPOO

- Docentes
  - Profesores Adjuntos:
    - Carlos Di Cicco.
    - Diego de la Riva.
  - JTPs:
    - Federico Naso (Junín) .
      Nelson Di Grazia (Pergamino) .
  - Ayudante:
    - Sebastián Sottile (Junín).

Patrones de Diseño

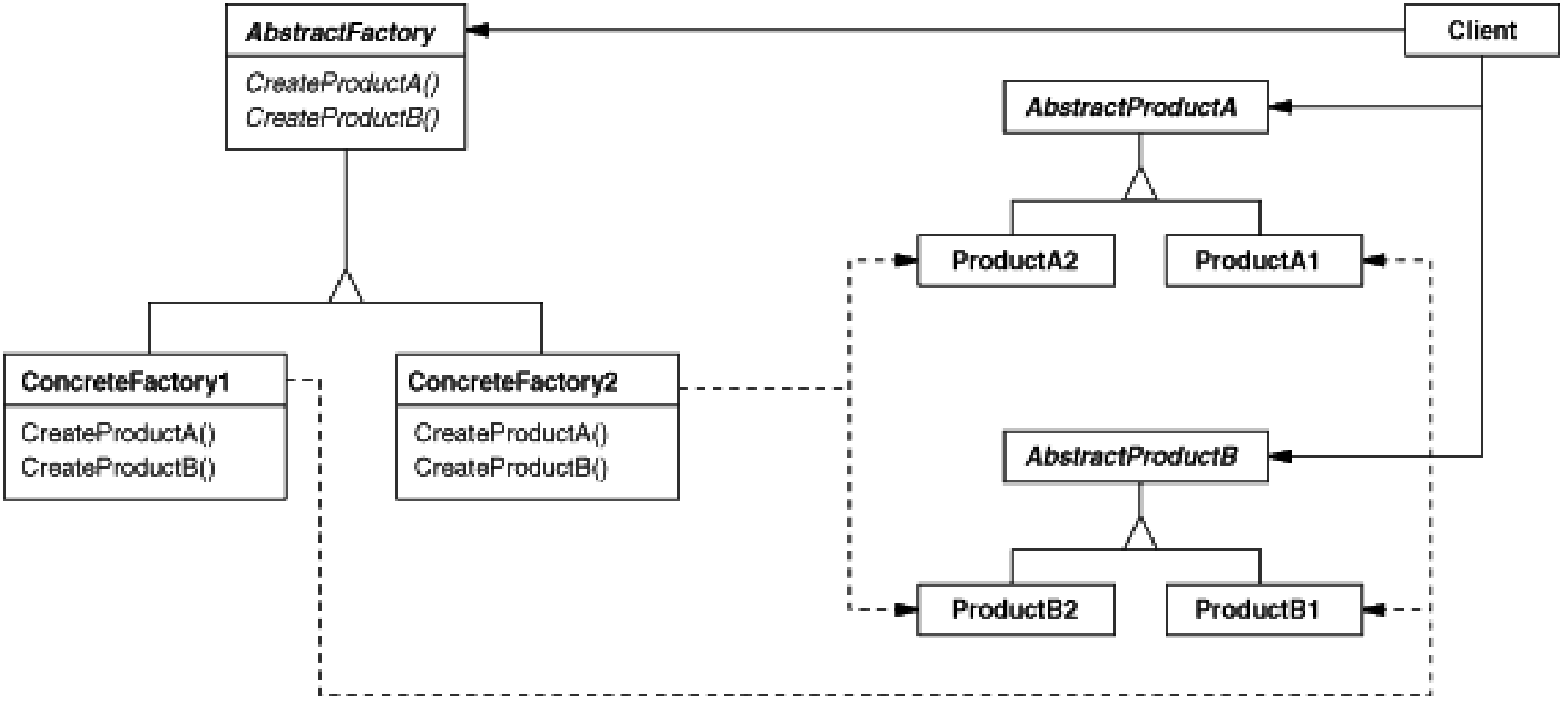
Patrones creacionales

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Patron: Abstract Factory

- Intención: Interfaz para crear familia de objetos relacionados o dependientes sin tener que especificar sus clases concretas.
- Cuando lo tengo que usar?
  - Cuando tengo que usar una o varias familias de productos relacionados. Cuando debo independizar la creación de familias de objetos. Librería.
- Como lo resuelvo?
  - Con una fábrica por familia de productos y con los productos abstractos (en Java con Interfaces o clases abstractas).

Patron: Abstract Factory



Patron: Abstract Factory

- Ejemplo: JDBC
  - Familia de productos relacionados: Connection + Statement + ResultSet.
  - Si creo un OracleConnection voy a crear OracleStatement y OracleResultSet
- Ejemplo: Swing
  - Familia de componentes relacionados: window, panel, botones, scroll, etc.
  - Si creo una window para Mac voy a crear paneles, botones y scroll para mac

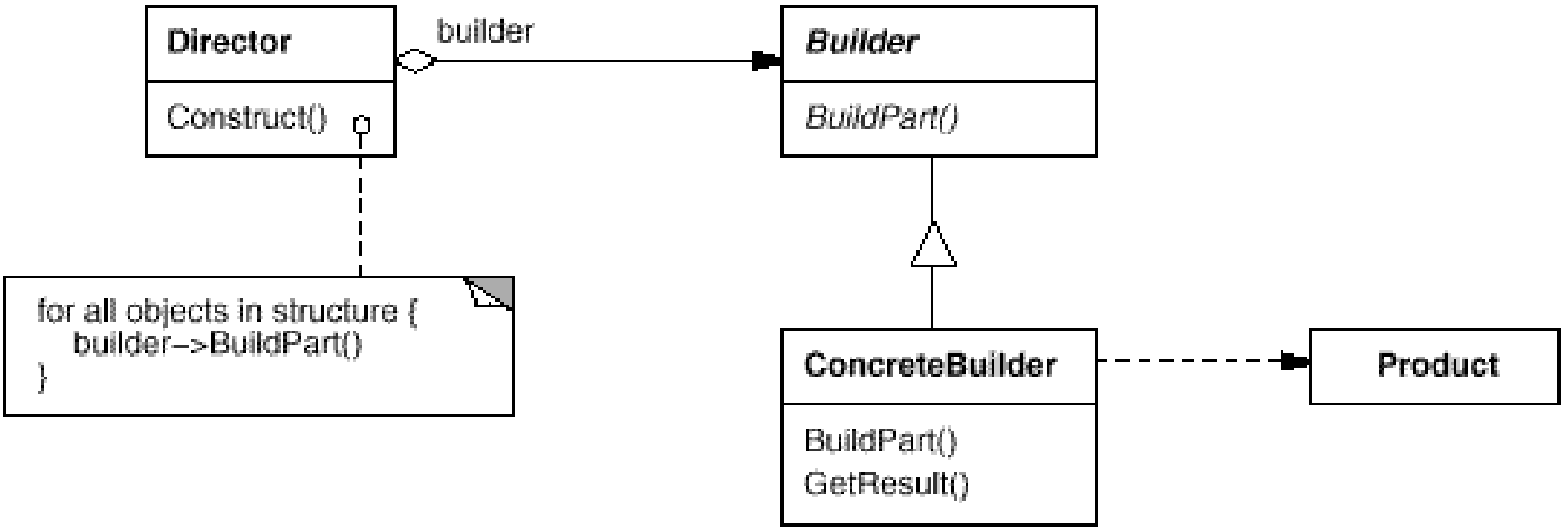
Patron: Builder

- Intención: Separa la construcción de un objeto complejo de su representación tal que el mismo proceso de construcción pueda crear diferentes representaciones.
- Cuando lo tengo que usar?
  - Cuando tengo que construir un objeto de forma independiente de sus partes y de cómo se ensamblan.
  - Cuando el resultado de la construcción puede tener diferentes representaciones

Patron: Builder

- Como lo resuelvo?
  - Con una clase constructor (y subclases concretas) que va creando las diferentes partes del objeto.
  - Lo que varía es el constructor (Builder). Es decir puedo tener un builder para crear un objeto complejo de una forma y otro para crearlo de otra. A diferencia de AbstractFactory el objeto creado es el mismo con diferentes valores

Patron: Builder



Patron: Builder

- Ejemplo: Constructor de paquetes turísticos
  - Una paquete puede tener un destino, traslado, hospedaje y alimento
  - Podemos tener diferentes constructores para paquetes turísticos
- Ejemplo: Constructor de vehículos
  - Un vehiculo tiene motor, chasis y ruedas.
  - Podemos tener diferentes constructores para auto, moto, camioneta, camión.

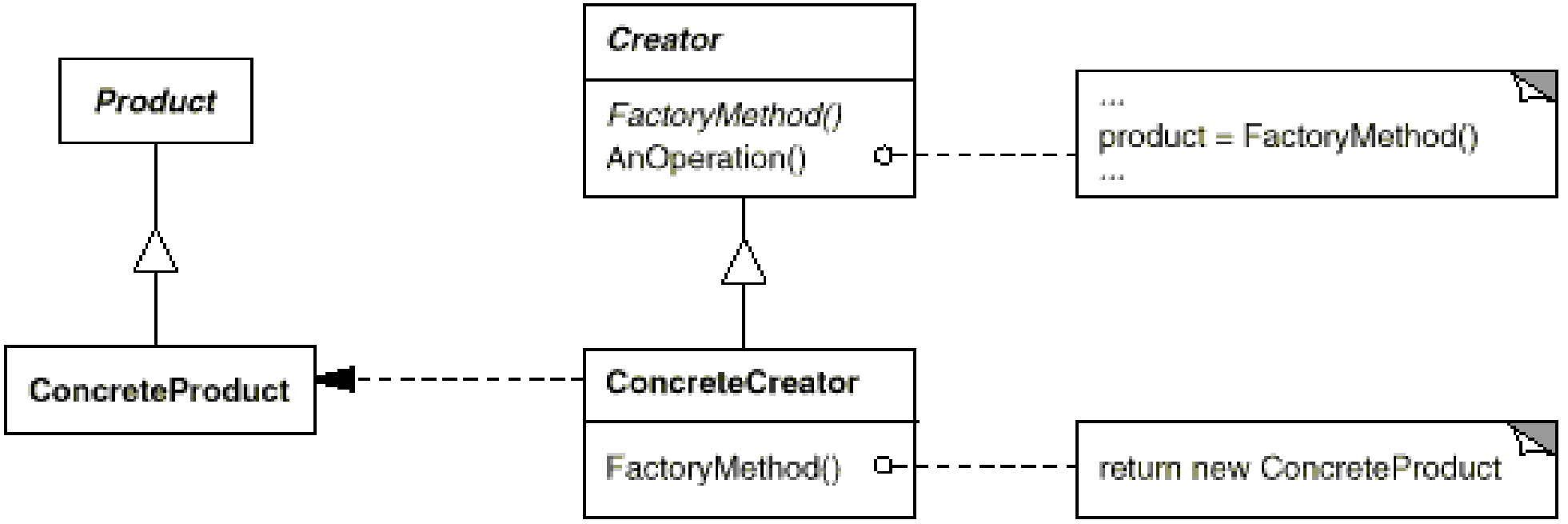
Patron: Factory Method

- Intención: Definir una interfaz para crear un objeto pero dejar la implementación a las subclases. La instanciación la hace la subclase
- Cuando lo tengo que usar?
  - Cuando quiero delegar la creación de objetos en subclases específicas.
  - Cuando quiero variar el tipo de objeto de acuerdo a la clase que lo crea
  - Cuando no puedo determinar el tipo de objeto que se va a crear

Patron: Factory Method

- Como lo resuelvo?
  - Con una clase creador y subclases concretas donde cada subclase crea objetos de diferentes tipos.
  - Lo que varía es el creador y el objeto concreto (Producto).
  - Puedo tener una clase concreta (creador) para instanciar objetos concretos.

Patron: Factory Method



Patron: Factory Method

- Ejemplos:
  - Traductor: puedo tener diferentes objetos traductores y de acuerdo al idioma al que quiero traducir, instancio el traductor adecuado.
  - Juegos: puedo tener diferentes clases de juegos que de instancian de acuerdo a diferentes creadores.

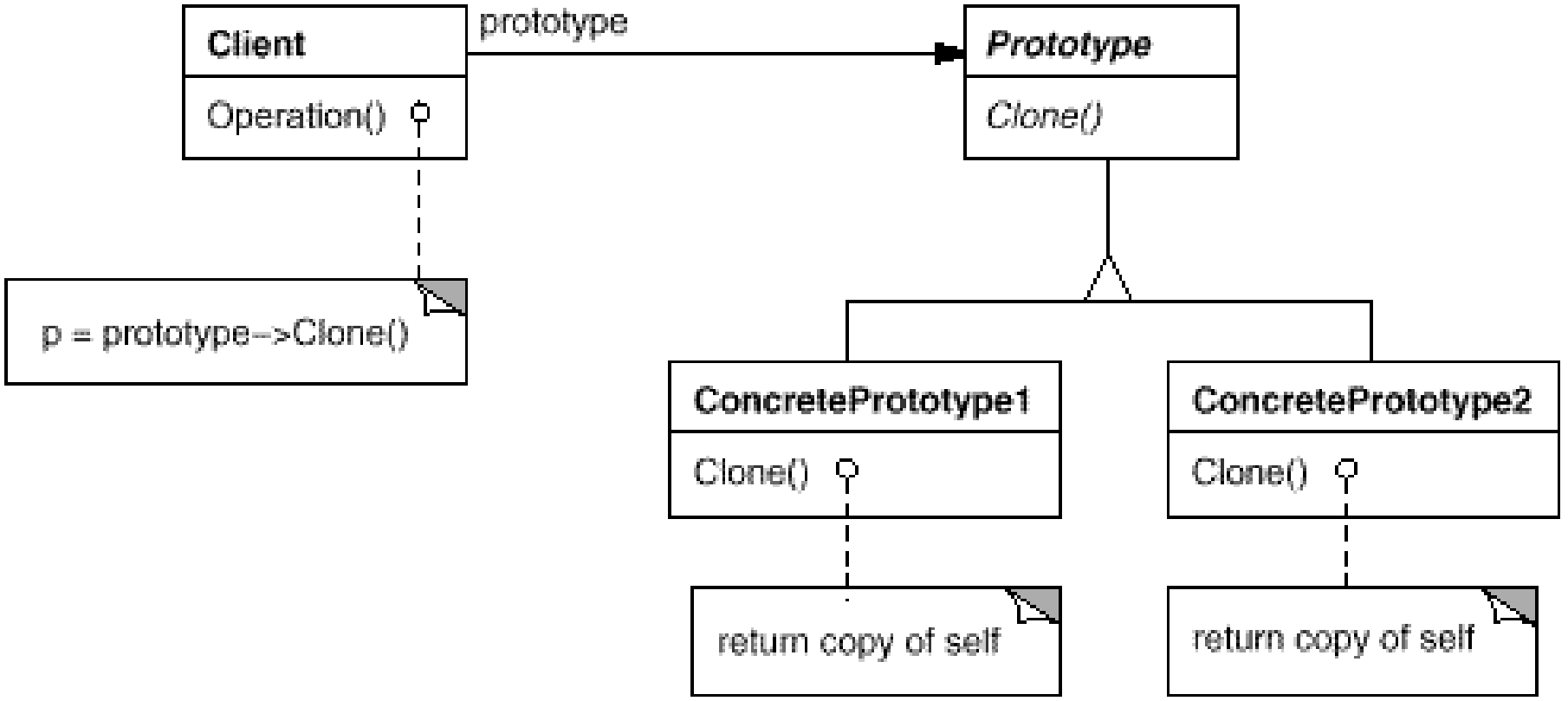
Patron: Prototype

- Intención: Genero una instancia prototipo y creo objetos en base a ese prototipo (clon)
- Cuando lo tengo que usar?
  - Cuando quiero instancias clones de una existente.
  - Cuando el estado interno de un objeto varía poco de una instancia base a otra (pocas combinaciones de estado interno)
  - Para evitar fábricas de objetos similares

Patron: Prototype

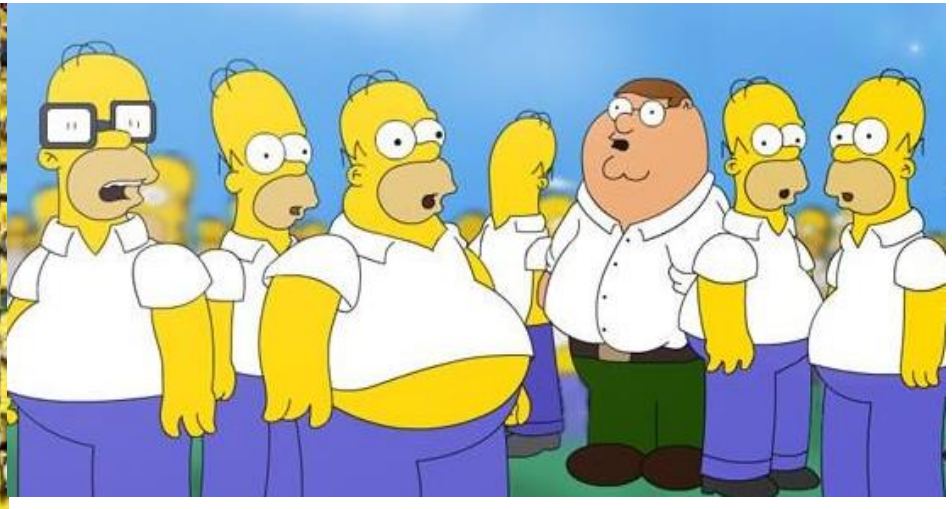
- Como lo resuelvo?
  - Con un método clone() en el objeto y/o en sus subclases, que retorne una copia del objeto (tal cual, o con los valores copiados que quiera).

Patron: Prototype



Patron: Prototype

- Ejemplos:



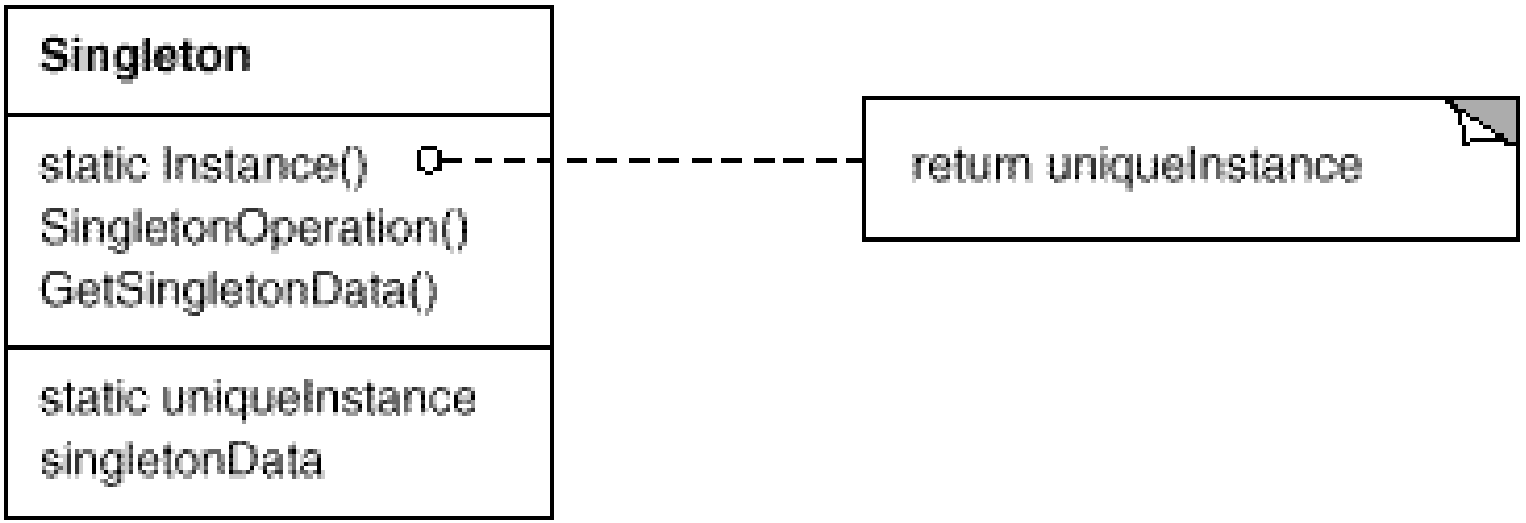
Patron: Singleton

- Intención: Tener una única instancia de una clase y un único punto de acceso a ella
- Cuando lo tengo que usar?
  - Cuando quiero una sola instancia de una clase (o un cupo limitado)
  - Cuando quiero tener una sola instancia de una subclase de la clase mi instancia
  - Cuando necesito funcionalidad específica y no lo puedo resolver con clases abstractas o métodos de clase porque necesito instancias que guarden estado

Patron: Singleton

- Como lo resuelvo?
  - Con un método de clase que devuelva la instancia creada y si no está creada que la cree, la almacene como variable de clase y la devuelva.

Patron: Singleton



Patron: Singleton

- Ejemplos:
  - Pool de impresoras: puedo tener una clase denominada Pool con un método de clase que se llame getPool() y una variable de clase de tipo Pool.

```
public class Pool{
    static Pool pool;
    static Pool getPool(){
        if (pool == null) pool = new Pool(); //constructor privado
        return pool;
    }
    private Pool(){…}
```

Patrones estructurales

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

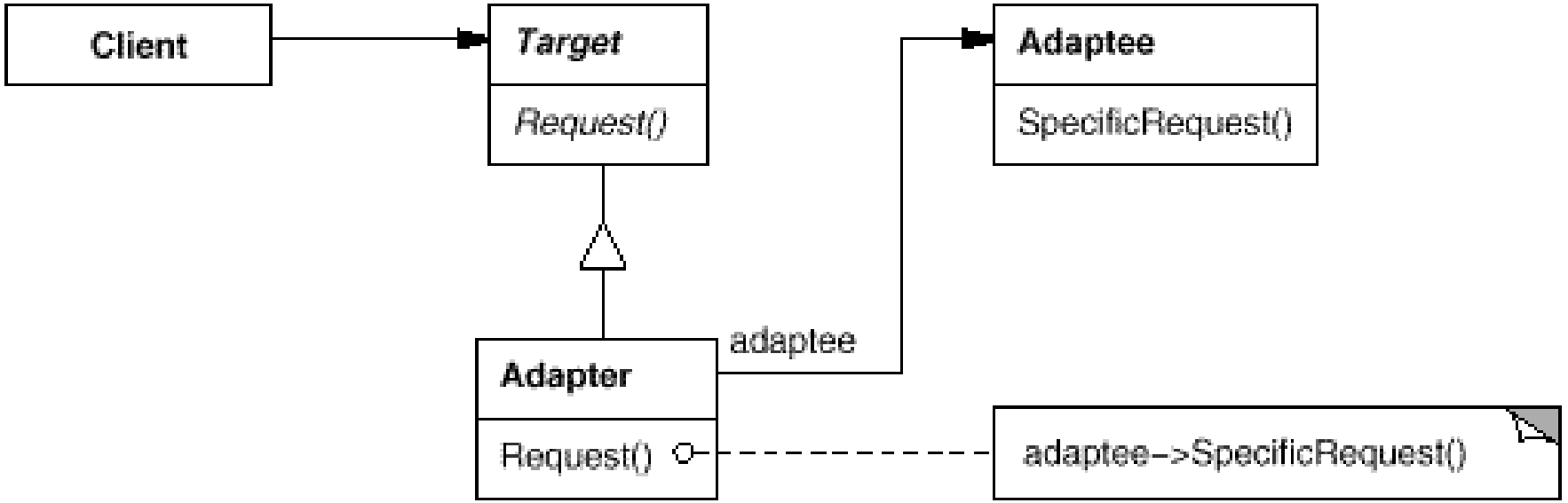
Patron: Adapter

- Intención: Convertir la interfaz de una clase a otra esperada por un cliente.
- Cuando lo tengo que usar?
  - Cuando tengo una clase que no puedo modificar y necesito conectarla con otra que envía mensajes que esta entiende pero de otra forma.

Patron: Adapter

- Como lo resuelvo?
  - Con herencia (“hago subclase del adaptado”) o con composición (“meto el objeto dentro del adaptador”).

Patron: Adapter



Patron: Adapter

- Ejemplos:
  - Login: Utilizar una clase que hace un login contra un LDAP (user, pass, dominio) desde una aplicación que loguea a BBDD (user, pass).
  - Documentos: un software de ofimática que abre documentos ".docx". Si recibe un ".doc" lo adapta para poder abrirlo

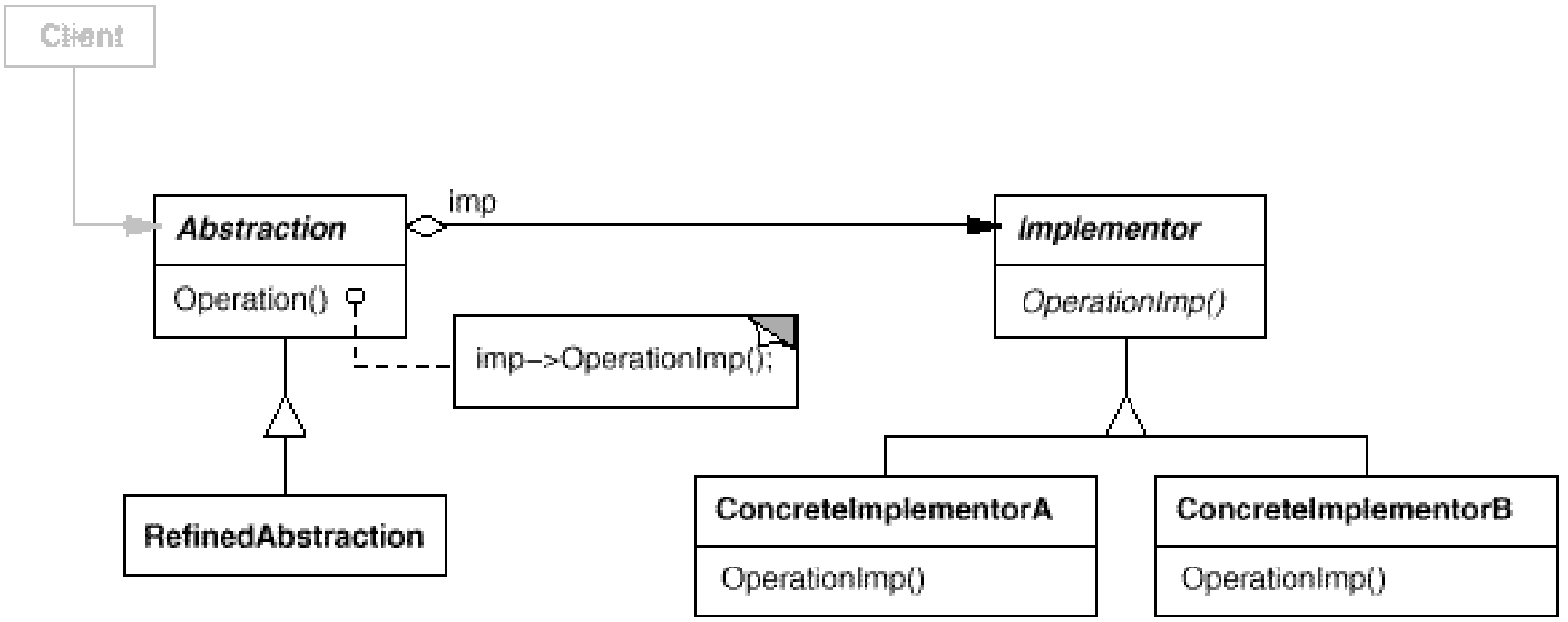
Patron: Bridge

- Intención: Desacoplar una abstracción de su implementación para que ambas puedan variar de forma independiente.
- Cuando lo tengo que usar?
  - Cuando tengo que cambiar las objetos concretos en run-time.
  - Cuando la abstracción y los objetos concretos pueden ser extensibles de forma independiente.
  - Cuando el cambio en los objetos concretos no deben afectar a los clientes

Patron: Bridge

- Como lo resuelvo?
  - Con clases abstractas o interfaces que definan operaciones abstractas que deban ser implementadas por clases concretas

Patron: Bridge



Patron: Bridge

- Ejemplos:
  - JDBC: Define interfaces para conectividad a base de datos (Conexión, Sentencias, Resultados). Cada implementación particular es un bridge. La estructura es Bridge, usa AbstractFactory para crear objetos.
  - Idem JPA
  - Instrumentos musicales: sonar, afinar

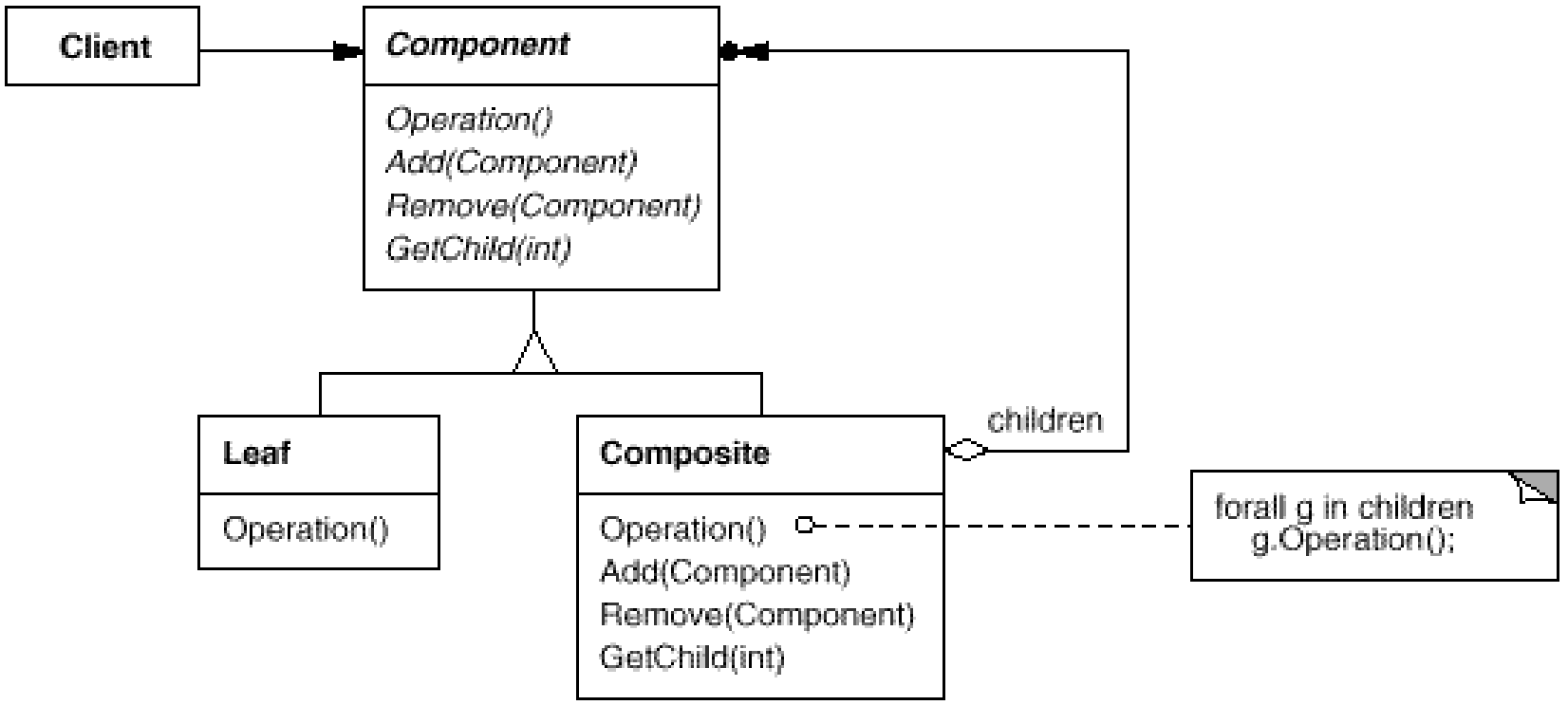
Patron: Composite

- Intención: Componer objetos en estructuras tipo árbol para representar un objeto compuesto. Permite tratar los objetos compuestos como a los simples
- Cuando lo tengo que usar?
  - Cuando tengo jerarquías de objetos compuestos.
  - Cuando quiero tratar los objetos compuestos igual que a los simples

Patron: Composite

- Como lo resuelvo?
  - Con una jerarquía tipo árbol de componentes, con hojas y nodos que referencian a otro componentes

Patron: Composite



Patron: Composite

- Ejemplos:
  - Artículos: puedo tener artículos simples y otros compuestos por varios simples: cartuchera con lápices, caja de herramientas.
  - Combo = hamburguesa + gaseosa + papas.

Patron: Decorator

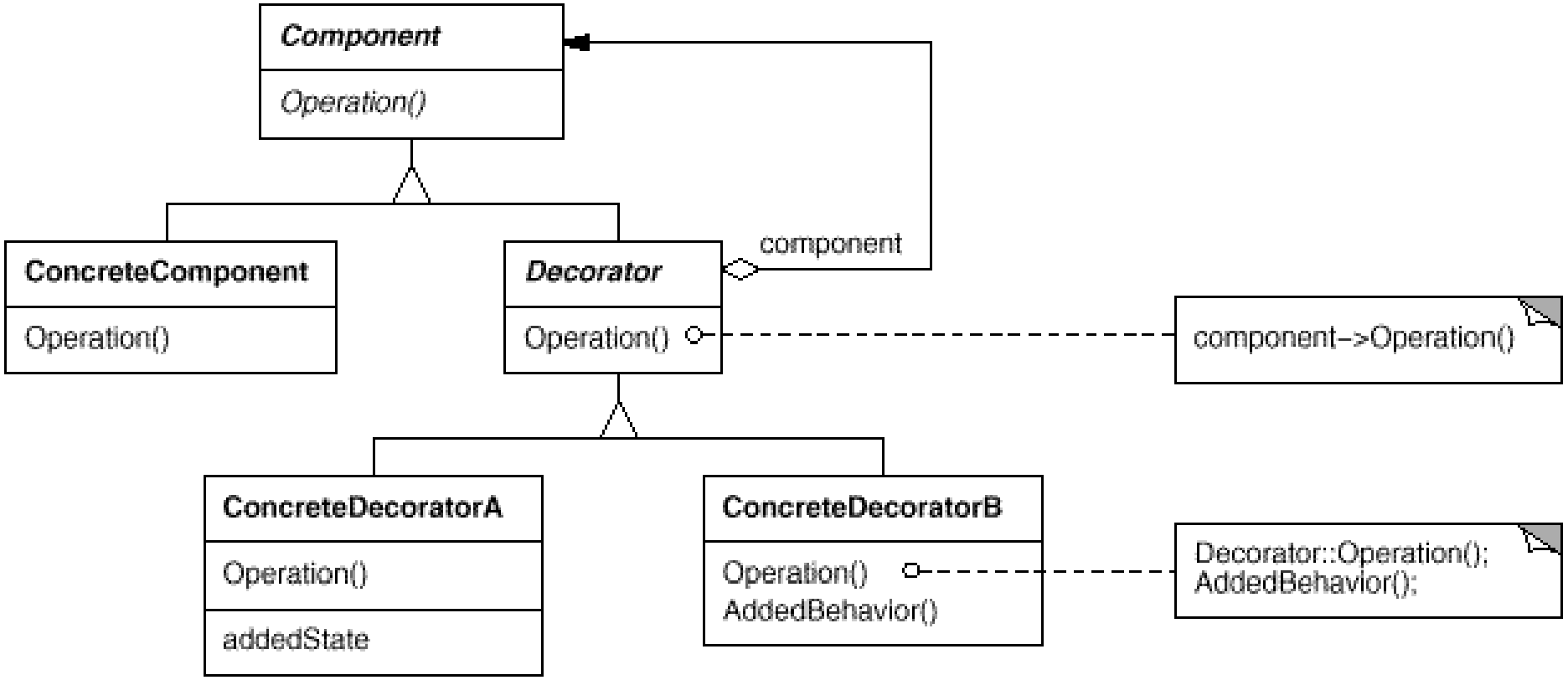
- Intención: Agregar funcionalidad a un objeto de forma dinámica. Alternativa flexible a subclasificar para extender funcionalidad.

- Cuando lo tengo que usar?
  - Cuando tengo que agregar funcionalidad específica a un objeto de forma dinámica, sin afectar a otro objetos
  - Cuando se hace imposible generar una jerarquía por la cantidad de clases que generaría y porque la funcionalidad queda duplicada en varias partes de la jerarquía

Patron: Decorator

- Como lo resuelvo?
  - Con una jerarquía tipo árbol de componentes, que tenga un componente concreto y uno “decorado” que agrega funcionalidad

Patron: Decorator



Patron: Decorator

- Ejemplos:
  - Ventana con scroll, ventana con barra de estado, ventana con scroll y barra de estado.
  - Café con leche, café con crema, café con chocolate, café con leche, crema y chocolate.
  - Auto a nafta, auto a nafta con GNC, auto a nafta con GNC con 2 tubos

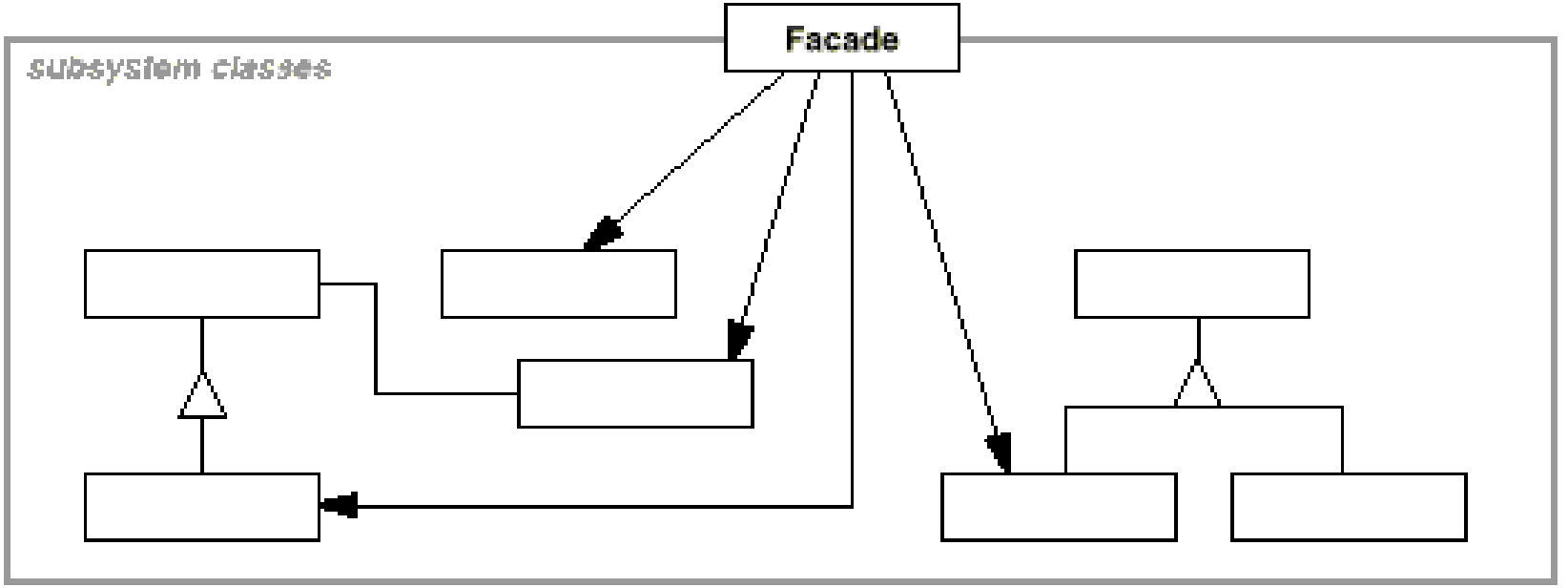
Patron: Facade

- Intención: Interfaz o punto de acceso único a clases y funcionalidad del sistema.
- Cuando lo tengo que usar?
  - Cuando tengo un sistema complejo con muchos objetos y mucha funcionalidad y quiero un punto de acceso único porque no quiero (o no puedo) conocer como está implementado el sistema.
  - Cuando hay mucha dependencia (alto acoplamiento) entre los objetos del sistema y quiero proveer portabilidad.
  - Cuando quiero separar en capas.

Patron: Facade

- Como lo resuelvo?
  - Con una clase que conozca a todas las clases del sistema y provea la interfaz necesaria al cliente.

Patron: Facade



Patron: Facade

- Ejemplos:
  - Préstamo bancario: supongamos que para sacar un préstamo hay que chequear la situación del cliente (cuenta, créditos anteriores) en el banco, la situación del cliente en otros bancos, la situación del banco. Usamos un facade para el préstamo que se encargue de acceder a la funcionalidad para chequear los recursos y la situación del cliente.
  - Mail: involucra conectarse al servidor, autenticarse, invocar al antivirus, al antispan, al antiphishing, chquequear tamaño del mail, destinatario válido, emisor válido, reintento, etc. ¿Qué hace el cliente?

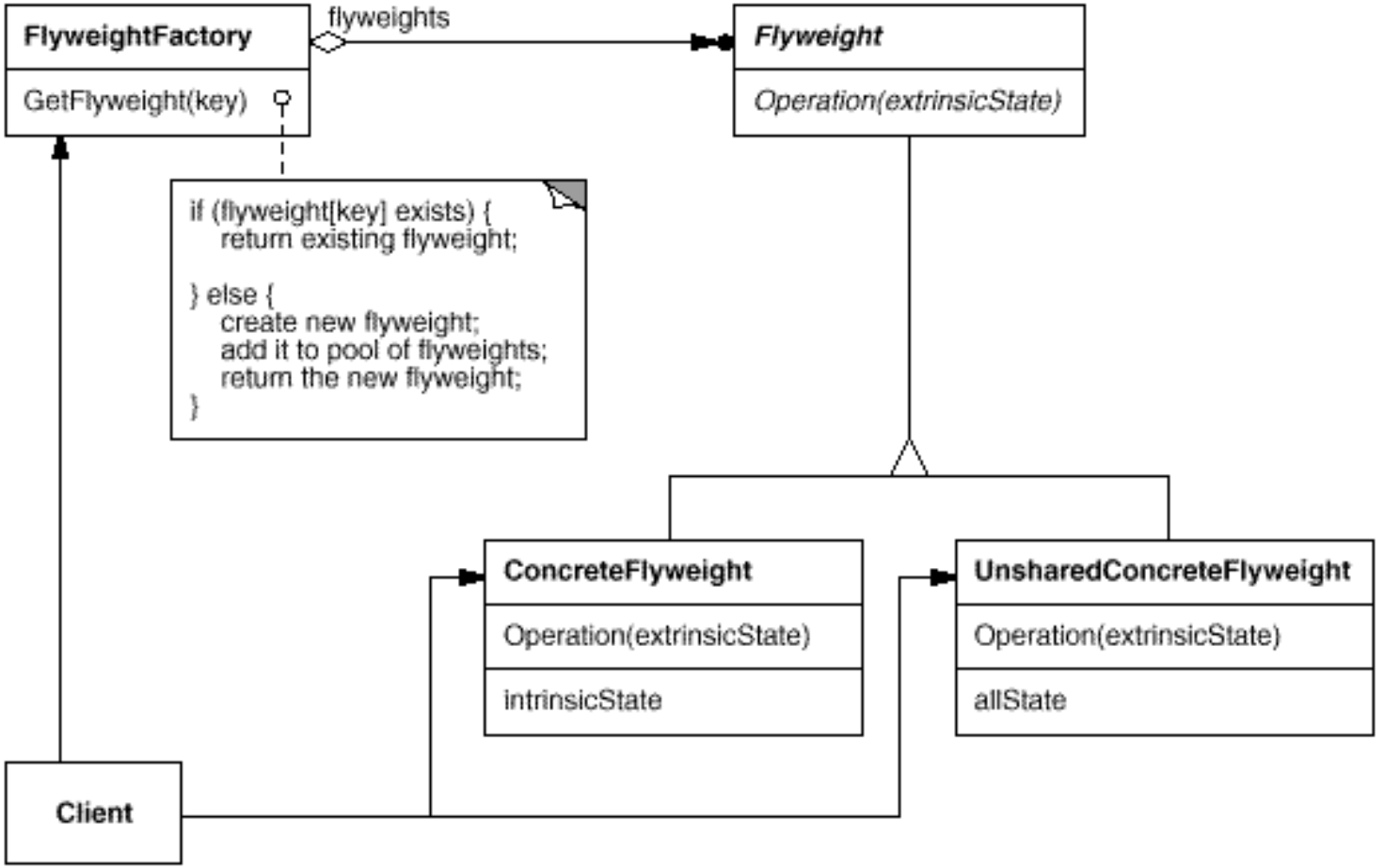
Patron: Flyweight

- Intención: Compartir para usar objetos de forma eficiente.
- Cuando lo tengo que usar?
  - Cuando tengo gran cantidad de objetos que requieren recursos superiores a los que cuento.
  - Cuando puedo representar a un gran conjunto de objetos con un grupo limitado de ellos.
  - Cuando no necesito identificar unívocamente a cada objeto.

Patron: Flyweight

- Como lo resuelvo?
  - Con una fábrica que determine si puede usar un objeto ya existente o si se requiere crear uno nuevo.

Patron: Flyweight



Patron: Flyweight

- Ejemplos:
  - String en java.
  - Pool de conexiones: puedo tener una lista de conexiones a una base de datos previamente creadas. Si quiero acceder a otra base, creo una nueva y la almaceno en el pool.

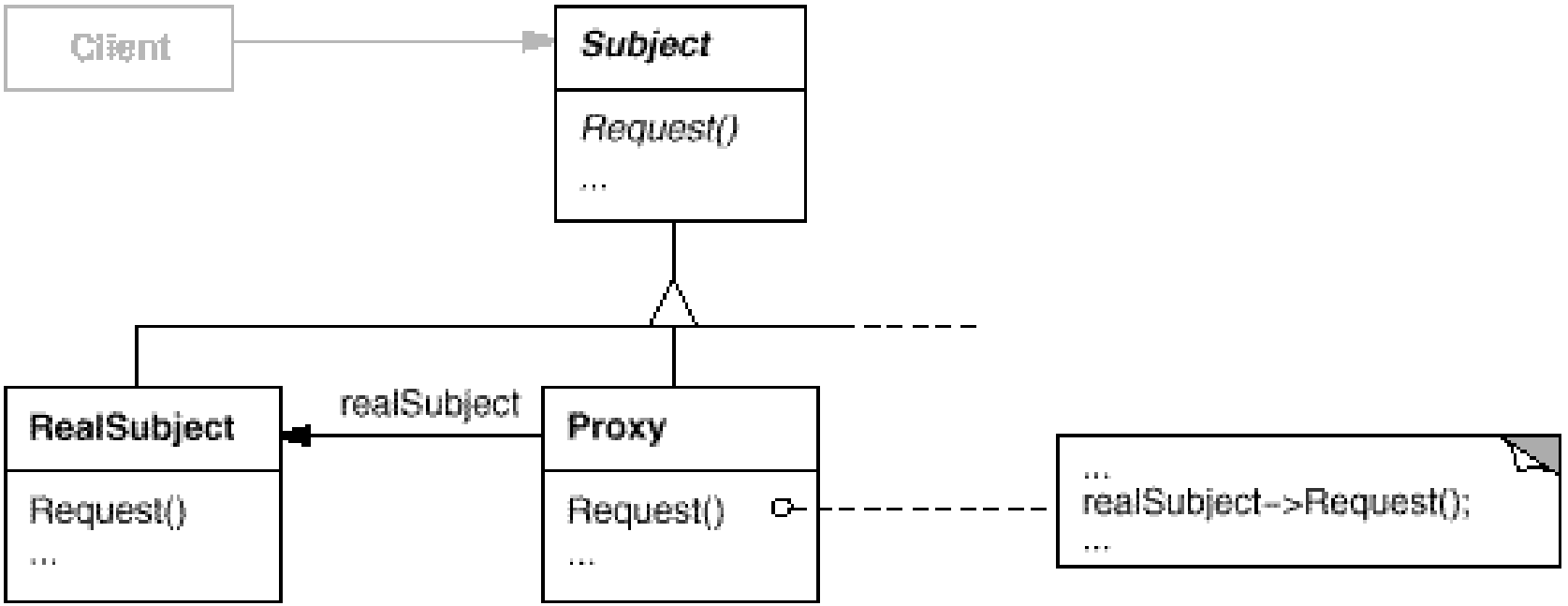
Patron: Proxy

- Intención: Provee un objeto que controla el acceso a otro.
- Cuando lo tengo que usar?
  - Cuando requiero acceso a un objeto a través de otro.
  - Cuando quiero controlar el acceso a un objeto.
  - Cuando quiero acceso inteligente (el proxy resuelve algunas operaciones y el objeto concreto otras).
  - Cuando requiero creación de objetos bajo demanda.

Patron: Proxy

- Como lo resuelvo?
  - Con una clase proxy que reciba los requerimientos y provea el acceso al objeto real

Patron: Proxy



Patron: Proxy

- Ejemplos:
  - Proxy de internet: cachea y permite acceso a una web.
  - Acceso a objetos remotos: tengo un objeto proxy local.
  - Un cheque es un proxy de dinero real

Patrones de comportamiento

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Patron: Chain of responsibility

- Intención: Evitar acoplar el emisor de un requerimiento para dar a otros objetos la posibilidad de manejar antes el requerimiento. Encadenar mensajes y pasarlo a través de una cadena de objetos antes de que llegue al receptor

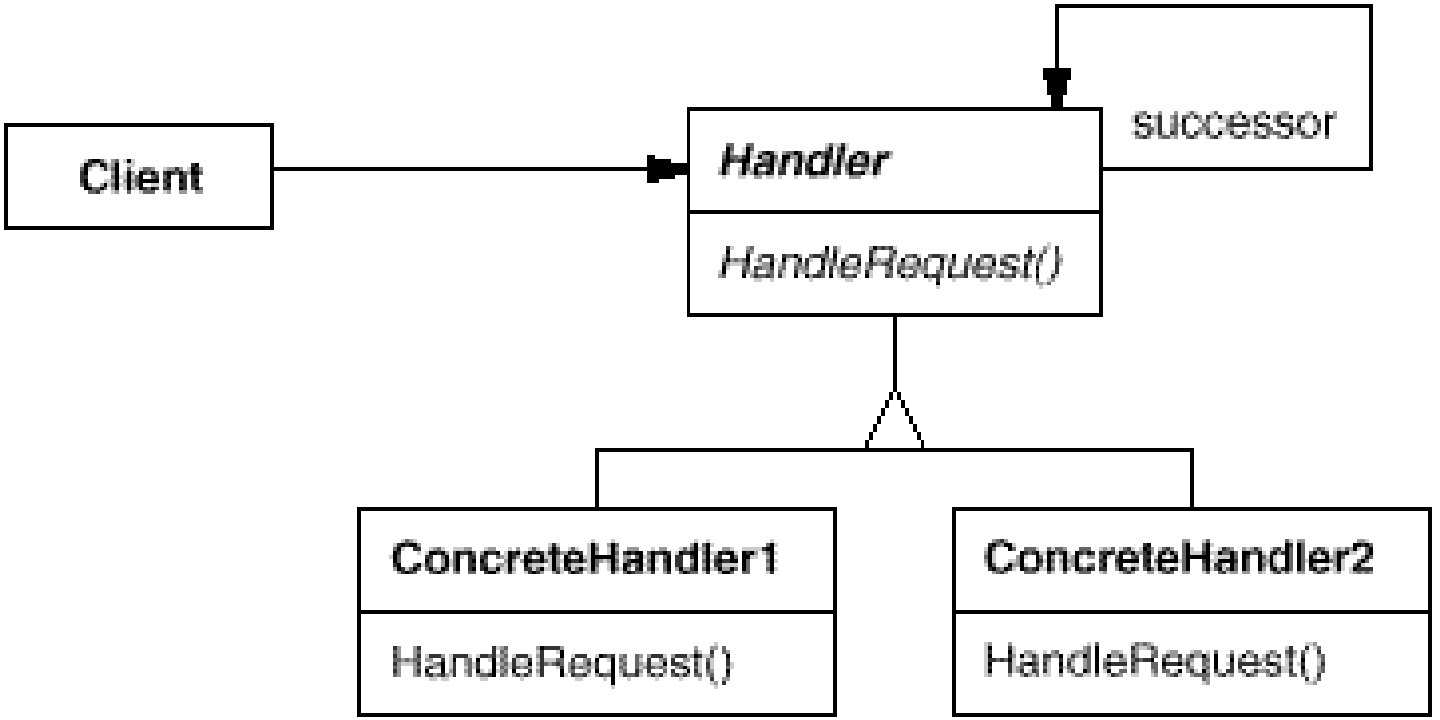
Patron: Chain of responsibility

- Cuando lo tengo que usar?
  - Cuando necesito filtrar requerimientos.
  - Cuando puede haber varios objetos que tengan que cumplimentar el requerimiento y no se conocen de entrada.
  - El receptor puede acoplarse automáticamente y/o dinámicamente

Patron: Chain of responsibility

- Como lo resuelvo?
  - Con una clase que tenga como “sucesor” a otro objeto de la misma clase.

Patron: Chain of responsibility



Patron: Chain of responsibility

- Ejemplos:
  - Servlet Filters: Se encadenan los filtros para permitir acceso al servlet que efectivamente ejecuta el requerimiento.
  - WorkFlow: flujo de trabajo relacionado. Ej. circuito de compras, publicación de contenidos.

Patron: Command

- Intención: Encapsula un requerimiento como un objeto permitiendo parametrizar clientes con diferentes requerimientos, encolar, registrar y deshacer eventos

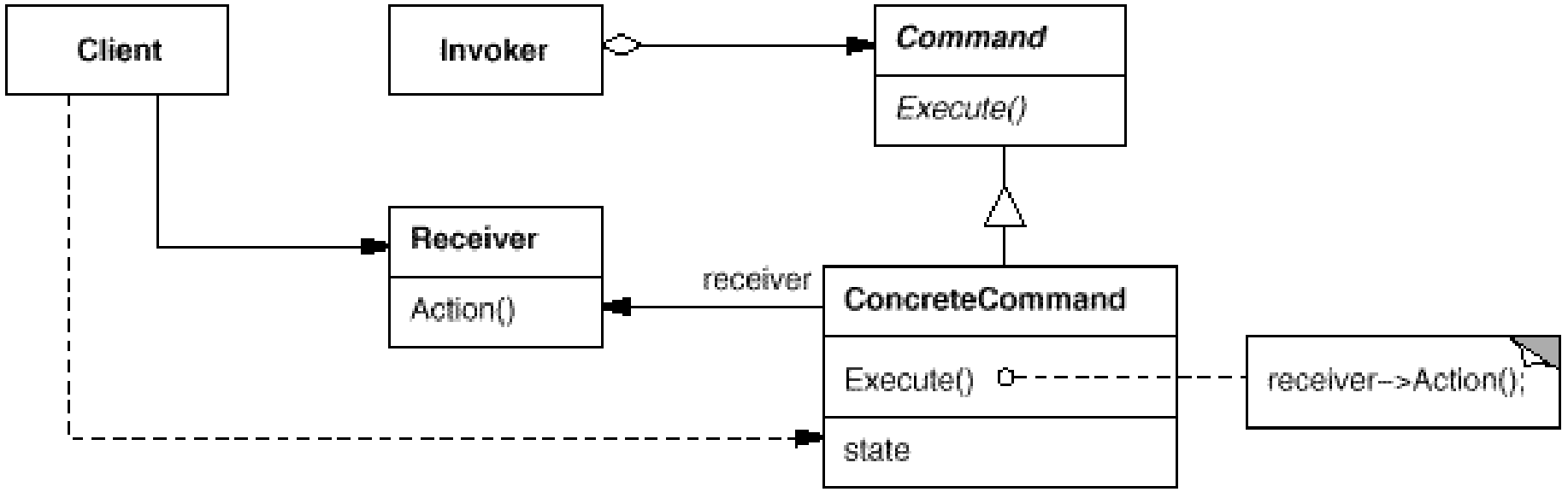
Patron: Command

- Cuando lo tengo que usar?
  - Cuando necesito encolar, loguear o registrar, deshacer eventos, acciones o transacciones.
  - Cuando tengo que estructurar un sistema con operaciones de alto nivel compuestas por operaciones primitivas (transacciones).
  - Cuando tengo que realizar operaciones asincrónicas

Patron: Command

- Como lo resuelvo?
  - Con una jerarquía de clases Command (o interfaz) que invoca a una acción (método) de un objeto receptor, en el momento que es invocado por un invocador (no por el cliente).
  - El cliente es el que crea el comando, el invocador es quien lo ejecuta.

Patron: Command



Patron: Command

- Ejemplos:
  - Transacciones: Varias acciones atómicas componen una transacción. Necesito que todas las acciones atómicas se realicen o que no se realice ninguna.
  - Undo (deshacer ctrl+z): cualquier trabajo con documentos.

Patron: Interpreter

- Intención: Dado un lenguaje, se define una representación para su gramática con un intérprete que usa la representación para interpretar sentencias del lenguaje

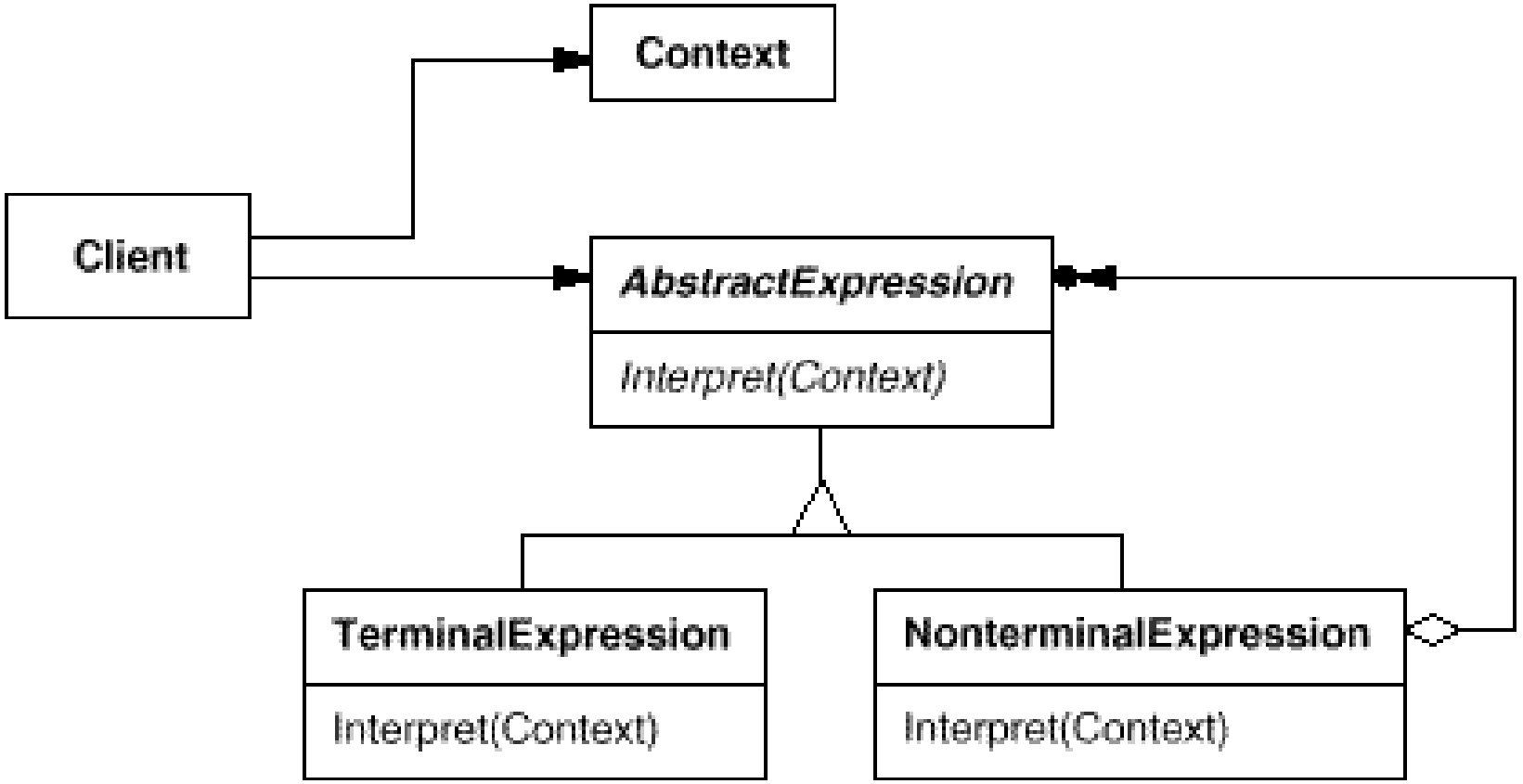
Patron: Interpreter

- Cuando lo tengo que usar?
  - Cuando tengo un lenguaje con una gramática específica que tengo que interpretar

Patron: Interpreter

- Como lo resuelvo?
  - Con un árbol de expresiones con hojas como expresiones terminales y nodos como expresiones que anidan otras.

Patron: Interpreter



Patron: Interpreter

- Ejemplos:
  - Calculadora: pueden pensarse las operaciones aritméticas como un árbol de operaciones y la calculadora las interpreta.
  - Parsers de lenguajes

Patron: Iterator

- Intención: Forma de acceder secuencialmente a los elementos de un objeto compuesto (colección) sin exponer su representación.

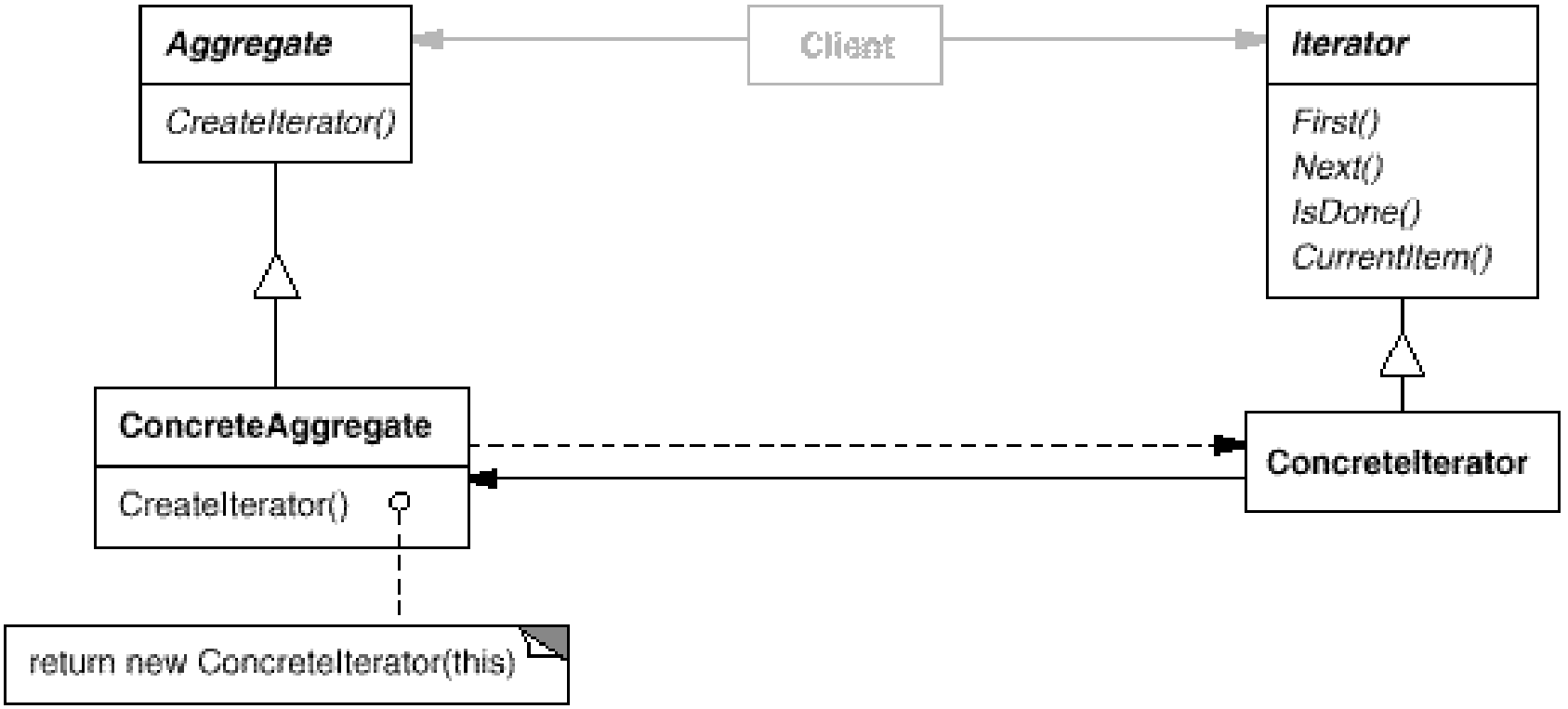
Patron: Iterator

- Cuando lo tengo que usar?
  - Cuando tengo que iterar una colección sin saber como está representada, es decir, sin importar que tipo de colección es.
  - Para lograr la misma forma de iteración independiente de la colección (polimorfismo)
  - Cuando necesito varias formas de recorrer colecciones

Patron: Iterator

- Como lo resuelvo?
  - Con un iterador (interface o superclase) genérico e iteradores concretos para cada tipo de colección.

Patron: Iterator



Patron: Iterator

- Ejemplos:
  - Colecciones en Java usan iteradores. Si uso collection.iterator() me devuelve el iterador concreto de acuerdo al tipo de colección y lo uso siempre de la misma forma (hasNext(), next())
  - Control remoto

Patron: Mediator

- Intención: Define un objeto que encapsula la forma de interacción de otros objetos. Promueve el bajo acoplamiento evitando la referencia explícita entre objetos y permite variar la interacción de forma dinámica.

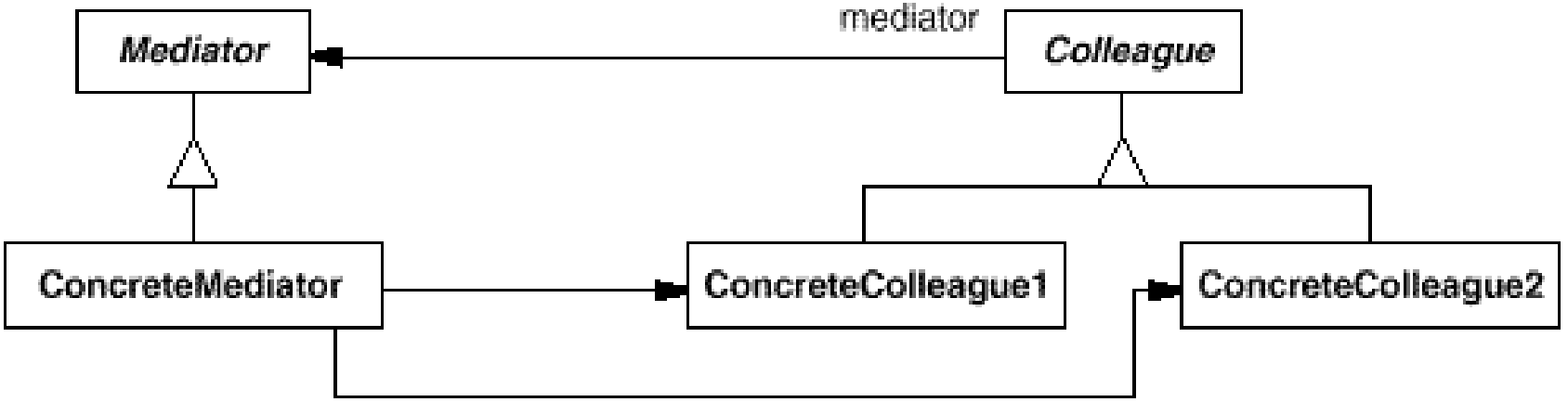
Patron: Mediator

- Cuando lo tengo que usar?
  - Cuando la interacción entre objetos en compleja, desestructurada y difícil de entender
  - Cuando la reutilización de un objeto es complicada porque se comunica con muchos otros objetos
  - Cuando debo personalizar comportamiento entre objetos

Patron: Mediator

- Como lo resuelvo?
  - Con una clase mediador (y subclases concretas) que se encarga de la comunicación entre los objetos concretos.

Patron: Mediator



Patron: Mediator

- Ejemplos:
  - Juego: el juego es un mediador entre los jugadores
  - Chat: mediador entre usuarios
  - Switch

Patron: Memento

- Intención: capturar el estado interno de un objeto (frizar) para que este pueda ser restaurado a este estado mas tarde.

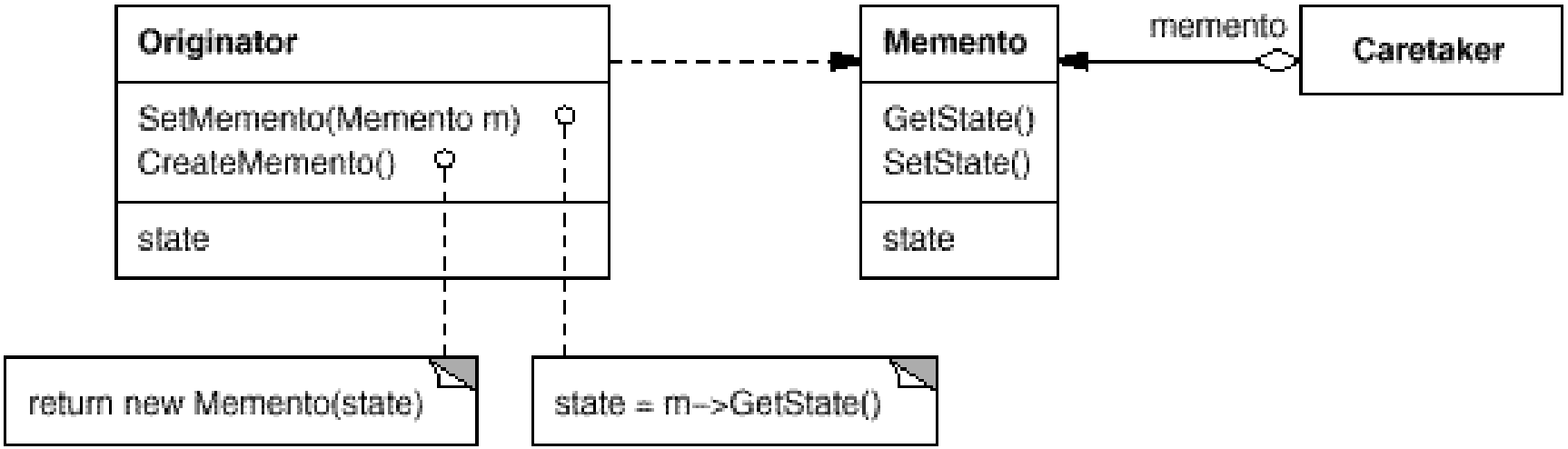
Patron: Memento

- Cuando lo tengo que usar?
  - Snapshot: pantallazo del estado de un objeto debe ser almacenado y posiblemente restaurado
  - Cuando la exposición de los atributos podría violar el encapsulamiento

Patron: Memento

- Como lo resuelvo?
  - Con una clase memento que solicita el estado al objeto concreto y lo almacena, pudiendo también restaurar el estado de este objeto. Los terceros objetos acceden al memento, no al estado del objeto.

Patron: Memento



Patron: Memento

- Ejemplos:
  - Pausa: si quiero pausar un juego para continuarlo luego.
  - Versionado: quiero guardar el estado de un objeto (versión) y seguir trabajando creando una rama. Luego puedo querer volver al estado anterior (por error o para tener una versión diferente).
  - Frizado

Patron: Observer

- Intención: Define relaciones de dependencia (uno a muchos) entre objetos, dado que si el observado cambia de estado, los dependientes se actualizan de forma automática

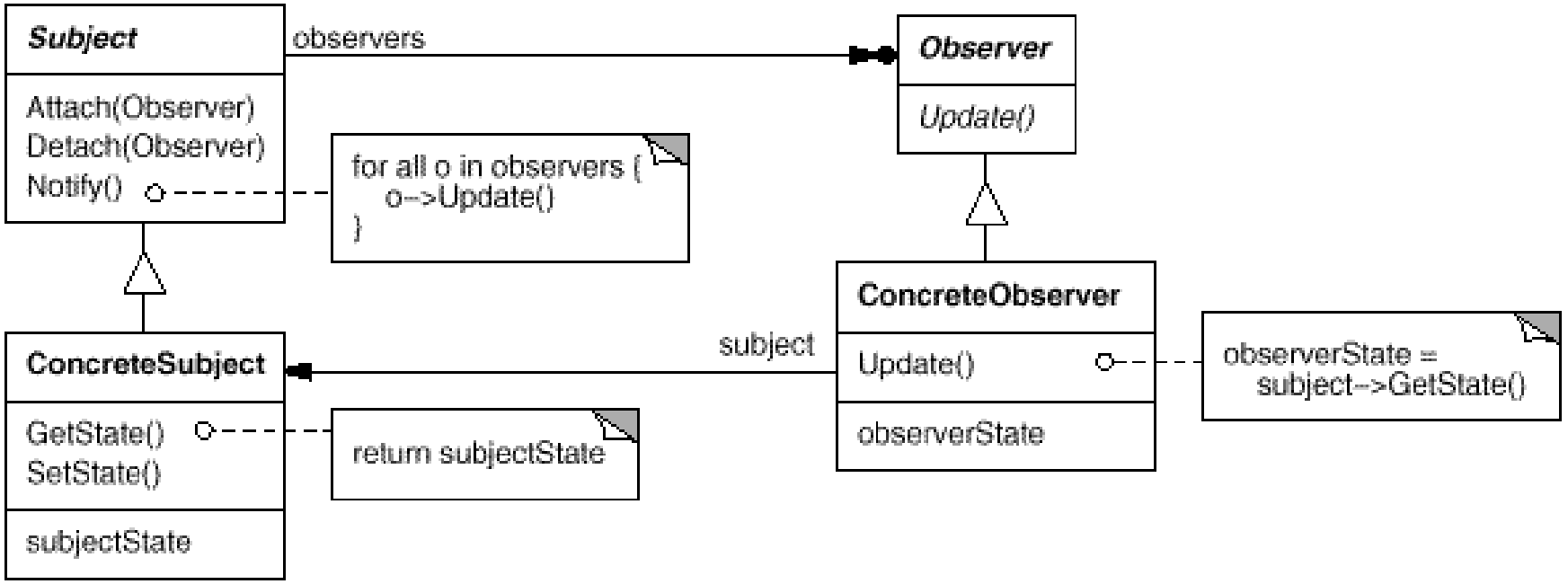
Patron: Observer

- Cuando lo tengo que usar?:
  - Cuando una abstracción tiene mas de un aspecto y son independientes. Encapsular esos aspectos en objetos separados, permite intercambiarlos y reusarlos independientemente.
  - Cuando el cambio de un objeto obliga a la modificación de otros (dependencia) y no se cual son esos otros.
  - Cuando necesito notificar a otros objetos de que algo ha sucedido sin saber a quienes debo notificar

Patron: Observer

- Como lo resuelvo?:
  - Con una colección de objetos observadores que reciben la notificación que el objeto ha cambiado

Patron: Observer



Patron: Observer

- Ejemplos
  - MVC: cuando cambia el estado de un objeto del modelo, las vistas deben cambiar de forma automática. En una aplicación C/S es simple, lista de observadores, ahora ¿Qué pasa en aplicaciones Web?
  - Sección crítica (notify)
  - Algo mas complejo: como recibir notificaciones de un web-services: http://dsv.su.se/soa/implmonster/communication/ObserverForWebServices.pdf

Patron: State

- Intención: cambiar el comportamiento de un objeto cuando cambia su estado interno.

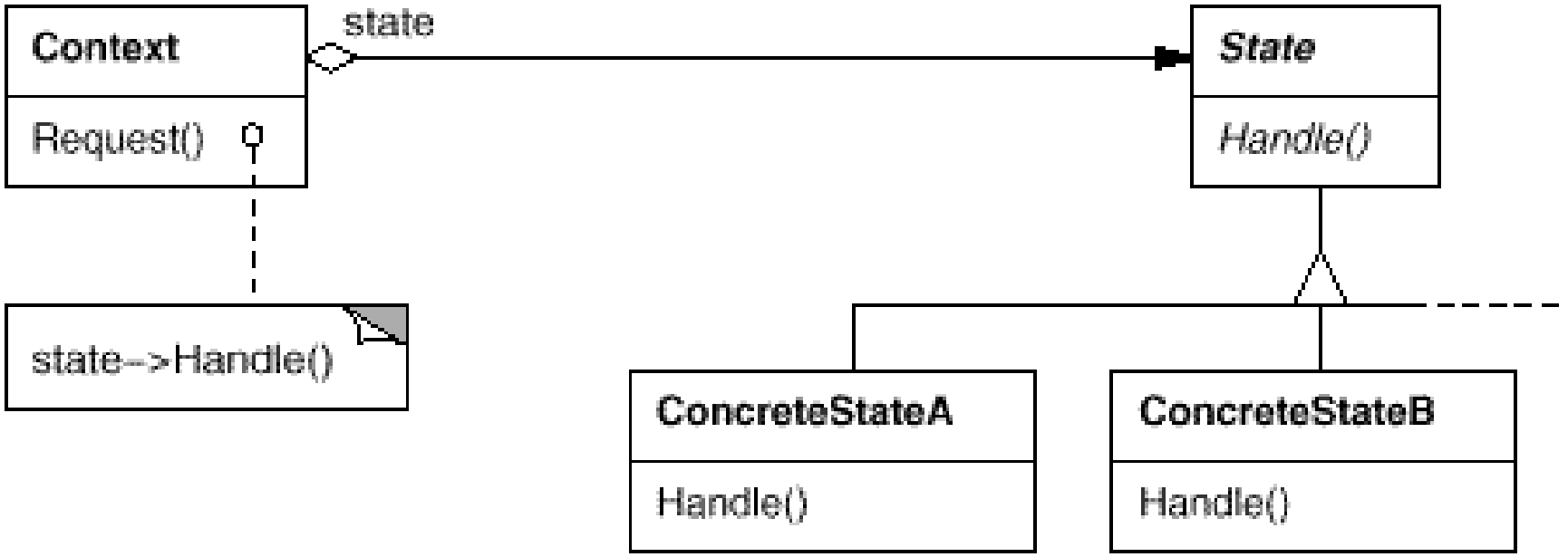
Patron: State

- Cuando lo tengo que usar?
  - Cuando el comportamiento del objeto depende de su estado y debo cambiar ese comportamiento en ejecución
  - Cuando quiero evitar los if anidados o switchs porque pueden ser ilegibles o pueden aparecer nuevas condiciones

Patron: State

- Como lo resuelvo?
  - Con una jerarquía de estados que manejan el requerimiento que se le hace al objeto. De acuerdo al estado es el comportamiento que toma el objeto. El objeto puede pasarse como parámetro al estado para ejecutar los métodos específicos del mismo (doble dispaching).

Patron: State



Patron: State

- Ejemplos:
  - Usuarios de chat: según el estado puede actuar ante el envío de un mensaje
  - Máquina de café: puede tener diferentes estados: lista, sirviendo, apagada, no disponible (sin café, sin leche o sin vasito) y actuar distinto ante diferentes eventos como ingreso de dinero o solicitud de lágrima

```
public class NoDisponible extends EstadoMaquina {
  public void ingresoDinero(Maquina maquina, float
    monto){
       maquina.tragarMoneda();
     maquina.mostrarMensaje("jaja");
  }
}
```

Patron: Strategy

- Intención: familia de algoritmos encapsulados en objetos haciéndolos intercambiables. Permite variar el algoritmo de forma independiente del cliente que lo usa

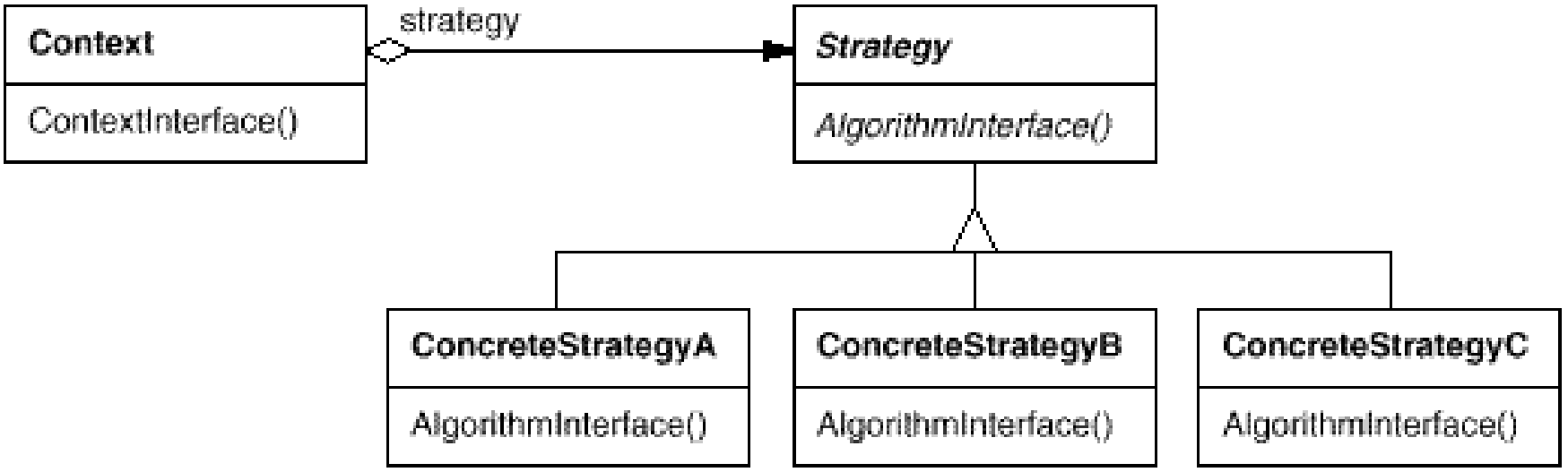
Patron: Strategy

- Cuando lo tengo que usar?
  - Cuando tengo muchas clases similares que varían solo en algún comportamiento
  - Cuando necesito variantes de un algoritmo
  - Para evitar if anidados o switchs complejos

Patron: Strategy

- Como lo resuelvo?
  - Con una jerarquía de algoritmos (o estrategias) donde el objeto instancia la estrategia concreta en el momento indicado

Patron: Strategy



Patron: Strategy

- Ejemplos:
  - Login: puedo tener una aplicación que se loguee con diferentes algoritmos (base de datos, archivo de texto, LDAP)
  - Forma de pago: hago diferentes cosas de acuerdo a la forma de pago (efectivo, tarjeta, cuenta corriente)

Patron: Template method

- Intención: define el esqueleto o estructura de un algoritmo y con subclases específicas se implementa pasos del algoritmo sin cambiar la estructura.

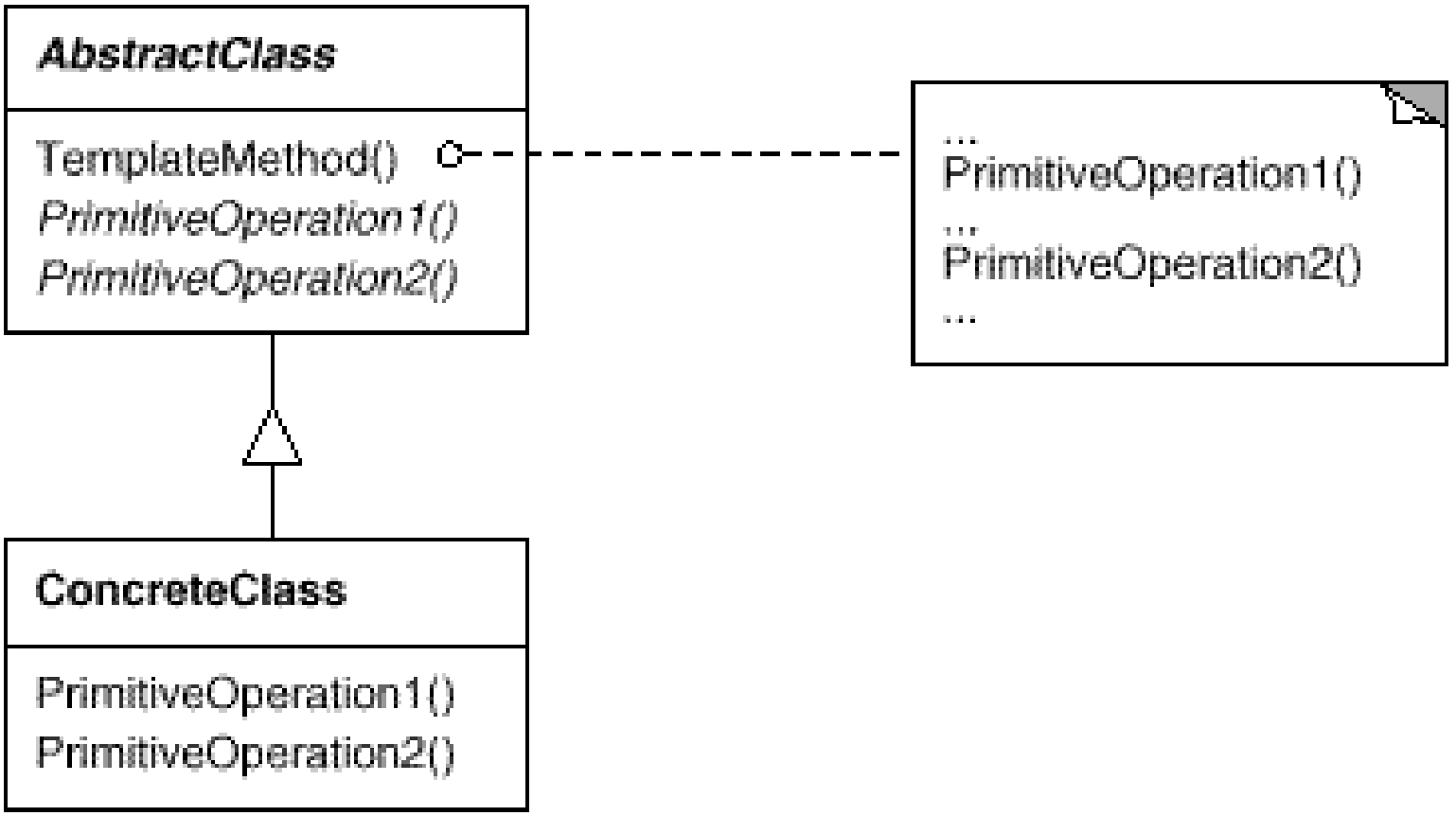
Patron: Template method

- Cuando lo tengo que usar?
  - Cuando tengo partes invariantes y partes dinámicas de un algoritmo
  - Para evitar duplicar código permitiendo tener las partes específicas en subclases
  - Para controlar la subclasificación permitiendo extender solo partes de una clase (final, abstract)

Patron: Template method

- Como lo resuelvo?
  - Con una jerarquía de clases que define operaciones (probablemente abstractas) que pueden implementar las subclases

Patron: Template method



Patron: Template method

- Ejemplos:
  - Login: en Strategy es todo el algoritmo, ¿qué pasa si quiero variantes?: usuario en bd, pass en LDAP, usuario en txt, pass en bd, usuario en LDAP, pass en txt
  - Construir o modelar una casa:

    crearPiso();
    crearParedes();
    crearAberturas();
    crearTecho();

    Chalet, Quincho

Patron: Visitor

- Intención: representa una operación que debe ser ejecutada sobre una estructura de objetos. Posibilita definir una nueva operación sin cambiar las clases de los elementos sobre los que se ejecuta

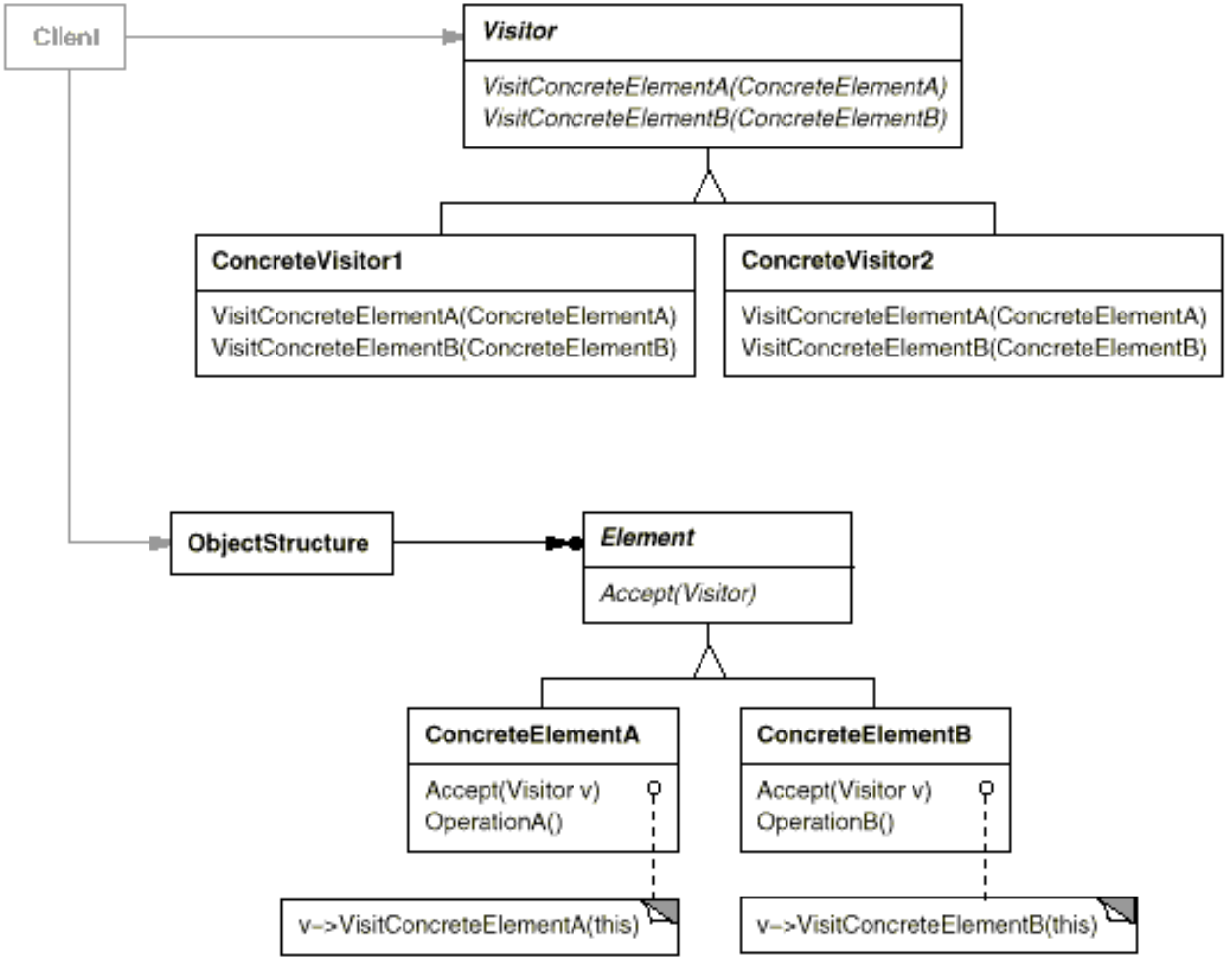
Patron: Visitor

- Cuando lo tengo que usar?
  - Cuando la estructura de objetos contiene muchas clases de objetos con diferentes interfaces y quiero ejecutar operaciones sobre esos objetos
  - Cuando las operaciones sobre la estructura son diferentes

Patron: Visitor

- Como lo resuelvo?
  - Con una jerarquía de visitadores que ejecutan operaciones específicas sobre una jerarquía de clases
  - El objeto a ser visitado debe aceptar la visita del visitor

Patron: Visitor



Patron: Visitor

- Ejemplos:
  - Vigía: Controla diferentes partes del auto. Tengo un control específico para bomba de nafta, nivel de refrigerante, carga de batería, etc.
  - Ídem control de una PC