Índice.

Contenido.

1 Patrones de Diseño

Los patrones de diseño son una técnica para resolver problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Tienen su origen en un campo distinto al de la informática: La Arquitectura.
En 1979 el arquitecto Christopher Alexander aportó al mundo de la arquitectura el libro The Timeless Way of Building; en él proponía el aprendizaje y uso de una serie de patrones para la construcción de edificios de una mayor calidad. Dentro de las soluciones de Christopher Alexander se encuentran cómo se deben diseñar ciudades y dónde deben ir las perillas de las puertas.
En 1987, Ward Cunningham y Kent Beck, para resolver problemas que causaban los nuevos programadores de objetos, usaron varias ideas de Alexander para desarrollar cinco patrones de interacción hombre-ordenador (HCI) y publicaron un artículo en OOPSLA-87 titulado Using Pattern Languages for OO Programs.
Pero recién en la década 1990 los patrones de diseño tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro Design Patterns escrito por el grupo Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones de diseño comunes.

> "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma."
>
> **Christopher Alexander**

Patrón: <**problema, contexto, solución**>
+<recurrencia, enseña, nombre>

1.1 ¿Qué es un Design Pattern?

Un Design Pattern:

- "es una regla que expresa la relación entre un contexto, un problema y una solución" (Christopher Alexander, creador de Patterns para la Ingeniería Civil)
- "Una solución (probada) a un problema en un determinado contexto" (Erich Gamma)
- "A Design Pattern names, asbtracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design." (nuevamente, Erich Gamma dixit)

Partiendo de estas definiciones, definimos qué debe contener un pattern:

Un nombre que describe el problema.
Esto nos permite:

- Tener un vocabulario de diseño común con otras personas. Un pattern se transforma en una herramienta de comunicación con gran poder de simplificación
- Por otra parte, se logra un nivel de abstracción mucho mayor. De la misma manera que una lista doblemente enlazada define cómo se estructura el tipo de dato y qué operaciones podemos pedirle, el pattern trabaja una capa más arriba: define un conjunto de objetos/clases y cómo se relacionarán entre sí (resumiéndolo, en una palabra).
- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente

El problema define cuándo aplicar el pattern, siempre que el contexto lo haga relevante. La solución contiene un template genérico de los elementos que componen el diseño, sus responsabilidades, relaciones y colaboraciones. Un pattern no es instanciable per se, la abstracción representada en el problema debe aplicarse a nuestro dominio.

En las consecuencias se analiza el impacto de aplicar un pattern en la solución, tanto a favor como en contra

1.2 ¿Qué no es un Design Pattern?

- No es garantía de un sistema bien diseñado. Tengo las respuestas, pero me falta saber si hice la pregunta correcta.

- Es un buen punto de partida para pensar una solución, no la solución. Al ser una herramienta, no puede reemplazar al diseñador, que es quien maneja la herramienta.

- De la misma manera que los estudiantes de psicología encuentran rasgos de neurosis obsesiva entre sus parientes, amigos e incluso en sí mismos, el estudiante de patterns suele querer "descubrir" patrones en su solución aun cuando no siempre se justifique. Entonces aparecen signos de sobre diseño: abuso por prever todo tipo de escenarios.

- Un pattern no es instanciable, y no es dependiente de un dominio. Es el bosquejo de una idea que ha servido en otras ocasiones, para una problemática similar. Representa una unidad de abstracción mayor a simples líneas de código.

Cómo no usarlos (Consejo de Gamma)

Cada vez que decidimos flexibilizar una parte de un sistema, se agrega complejidad al diseño y esto redunda en un mayor costo de implementación.
Aparecen fuerzas contrapuestas:

Los Design Patterns no deben ser usados indiscriminadamente. Si bien se logra mayor flexibilidad, también se agregan niveles de indirección que pueden complicar el diseño y/o bajar la performance. Un patrón de diseño sólo debería usarse cuando la flexibilidad pague su costo.

1.3 Características

Solucionar un problema: los patrones capturan soluciones, no sólo principios o estrategias abstractas
Ser un concepto probado: capturan soluciones demostradas, no teorías o especulaciones
La solución no es obvia: los mejores patrones generan una solución a un problema de forma indirecta
Describe participantes y relaciones entre ellos: describen módulos, estructuras del sistema y mecanismos complejos
El patrón tiene un componente humano significativo: todo software proporciona a los seres humanos confort y calidad de vida (estética y utilidad)

1.4 Clases de Patrones

Patrones de arquitectura: se expresa una organización o esquema estructural fundamental para sistemas software. Proporciona un conjunto de subsistemas predefinidos y sus responsabilidades.

Patrones de diseño: proporciona esquemas para refinar subsistemas o componentes de un sistema

Patrones de programación: Describe la implementación de aspectos de componentes

Patrones de análisis: prácticas que aseguran la consecución de un buen modelo de un problema y su solución

Patrones organizacionales: describen la estructura y prácticas de las organizaciones humanas

1.5 Elementos Característicos

Un patrón de diseño tiene cuatro elementos característicos:

- El **nombre** del **patrón**, describe el problema de diseño, su solución, y consecuencias en una o dos palabras. Tener un vocabulario de patrones nos permite hablar sobre ellos.
- El **problema** describe cuando aplicar el patrón. Se explica el problema y su **contexto**. Puede describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible. Se incluye una lista de condiciones.
- La **solución** describe los elementos que forma el diseño, sus relaciones, responsabilidades y colaboraciones. No se describe un diseño particular. Un patrón es una plantilla.
- Las **consecuencias** son los resultados de aplicar el patrón.

1.6 Estructura de un Patrón

Para describir un patrón se usan plantillas más o menos estandarizadas, de forma que se expresen uniformemente y puedan constituir efectivamente un medio de comunicación uniforme entre diseñadores. Varios autores eminentes en esta área han propuesto plantillas ligeramente distintas, si bien la mayoría definen los mismos conceptos básicos.

La plantilla más común es la utilizada precisamente por el GoF y consta de los siguientes apartados:

- **Nombre del patrón**: nombre estándar del patrón por el cual será reconocido en la comunidad (normalmente se expresan en inglés).
- **Clasificación del patrón**: creacional, estructural o de comportamiento.
- **Intención**: ¿Qué problema pretende resolver el patrón?
- **También conocido como**: Otros nombres de uso común para el patrón.
- **Motivación**: Escenario de ejemplo para la aplicación del patrón.
- **Aplicabilidad**: Usos comunes y criterios de aplicabilidad del patrón.
- **Estructura**: Diagramas de clases oportunos para describir las clases que intervienen en el patrón.
- **Participantes**: Enumeración y descripción de las entidades abstractas (y sus roles) que participan en el patrón.
- **Colaboraciones**: Explicación de las interrelaciones que se dan entre los participantes.
- **Consecuencias**: Consecuencias positivas y negativas en el diseño derivadas de la aplicación del patrón.
- **Implementación**: Técnicas o comentarios oportunos de cara a la implementación del patrón.
- **Código de ejemplo**: Código fuente ejemplo de implementación del patrón.
- **Usos conocidos**: Ejemplos de sistemas reales que usan el patrón.
- **Patrones relacionados**: Referencias cruzadas con otros patrones

1.7 Tipos de Patrones

Los patrones se agrupan en tres grandes categorías:

- **Creacionales: abstraen el proceso de instanciación, Muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán resueltas dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades**.

Abstract Factory: Proporciona una interfaz para **crear familias** de objetos relacionados o dependientes sin especificar su clase concreta

Builder: Permite a un objeto **construir** un objeto complejo especificando sólo su tipo y contenido
Factory Method: Define una interfaz para **crear** un objeto, dejando a las subclases decidir el tipo específico
Prototype: Permite a un objeto **crear** objetos personalizados sin conocer su clase exacta o los detalles de cómo crearlos
Singleton: Garantiza que solamente se **crea** una instancia de la clase y provee un punto de acceso global a él

- **Estructurales: se ocupan de generar estructuras entre clases y objetos, se estudian con los diagramas de clases/objetos y Describen la forma en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros**

Adapter: **convierte la interfaz** que ofrece una clase en otra esperada por los clientes
Bridge: **desacopla** una abstracción de su implementación y les permite variar independientemente
Composite: permite **construir** objetos complejos mediante **composición** recursiva de objetos similares
Decorator: **extiende la funcionalidad** de un objeto dinámicamente de tal modo que es transparente a sus clientes
Facade: **simplifica los accesos** a un conjunto de objetos relacionados proporcionando un objeto de comunicación
Flyweight: usa la **compartición** para dar soporte a un gran número de objetos de grano fino de forma eficiente
Proxy: **proporciona un objeto** con el que controlamos el acceso a otro objeto

- **Comportamiento: se encargan de la asignación de responsabilidades entre objetos y cómo se comunican entre sí, se estudian con diagramas de secuencia/colaboración. Se utilizan para organizar, manejar y combinar comportamientos.**

Chain of Responsability: evita el acoplamiento entre quien envía una petición y el receptor de la misma
Command: **encapsula una petición** de un comando como un objeto
Interpreter: dado un lenguaje define una representación para su gramática y permite **interpretar sus sentencias**
Iterator: acceso secuencial a los elementos de una colección
Mediator: define una **comunicación simplificada** entre clases
Memento: **captura y restaura** un estado interno de un objeto.
Observer: una forma de **notificar cambios** a diferentes clases dependientes
State: **modifica el comportamiento** de un objeto cuando su estado interno cambia
Strategy: define una **familia de algoritmos**, encapsula cada uno y los hace intercambiables
Template Method: define un esqueleto de algoritmo y **delega partes** concretas de un algoritmo a las subclases
Visitor: representa una operación que será realizada sobre los elementos de una estructura de objetos, permitiendo **definir nuevas operaciones** sin cambiar las clases de los elementos
sobre los que opera.

1.7.1 Creacionales

1.7.1.1 Abstract Fatory

AbstractFactory: Declara una interfaz de operaciones que crean productos abstractos
ConcreteFactory: Implementa las operaciones que crean los objetos producto
AbstractProduct: Declara una interfaz para un tipo de objeto producto
Product: Define un objeto producto que será creado por el correspondiente ConcreteFactory
Client: Usa las interfaces

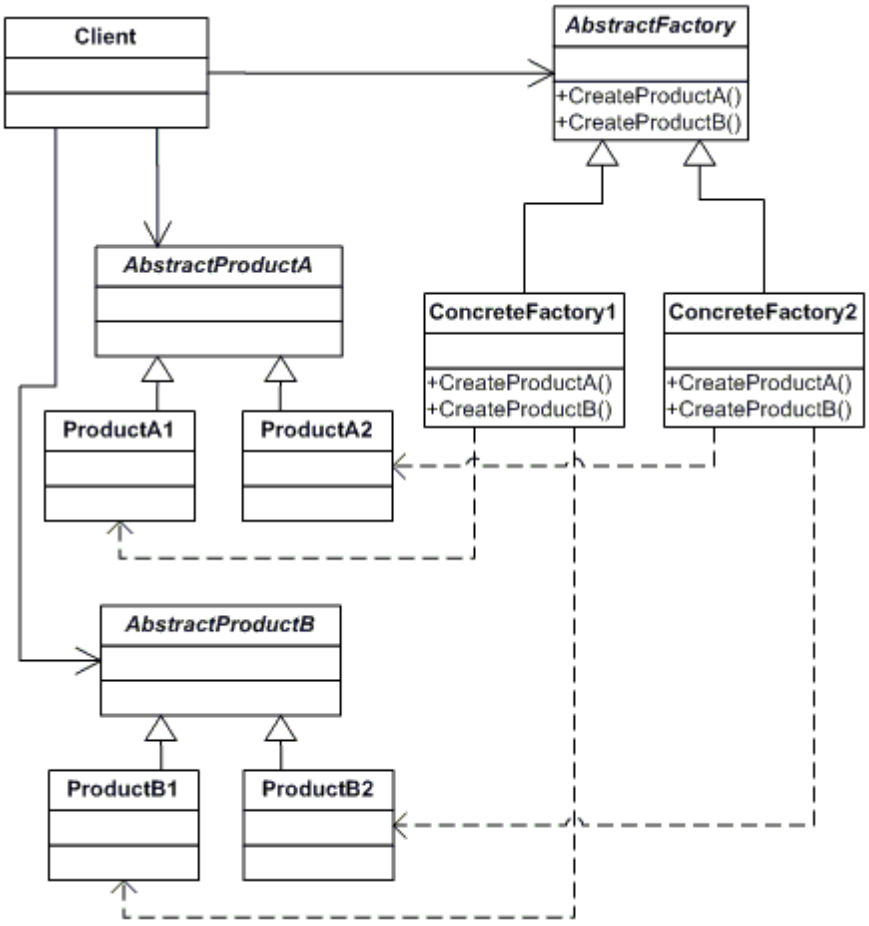

Ilustración 1 Patrón: Abstract Factory

1.7.1.2 Builder

Builder: Define una interfaz para la creación de partes de un objeto complejo
ConcreteBuilder: Implementa la interfaz de Builder, mantiene referencia del producto creado y proporciona una interfaz que retorna el producto. Ensambla las piezas constituyentes
Director: Construye un objeto usando la interfaz proporcionada por Builder
Product: Define las partes constituyentes del producto y representa un objeto complejo bajo construcción

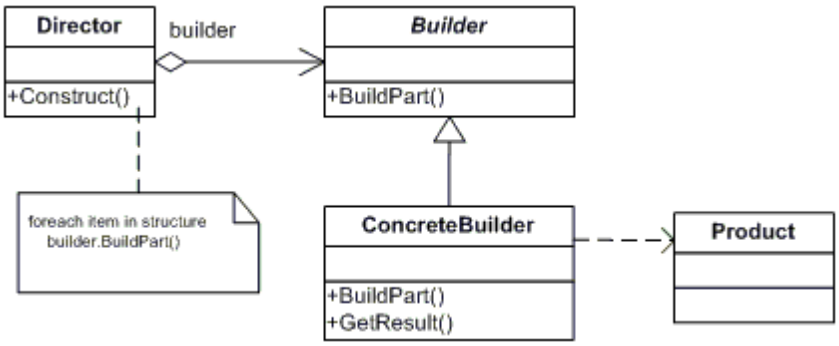

Ilustración 2 Patrón: Builder

1.7.1.3 FactoryMethod

Creator: Declara el método FactoryMethod(). Se puede definir una implementación por defecto para él
ConcreteCreator: Sobrescribe el método FactoryMethod(),devolviendo una instancia de ConcreteProduct
Product: Define la interfaz de objetos que crea el método FactoryMethod
ConcreteProduct: Implementa la interfaz del producto

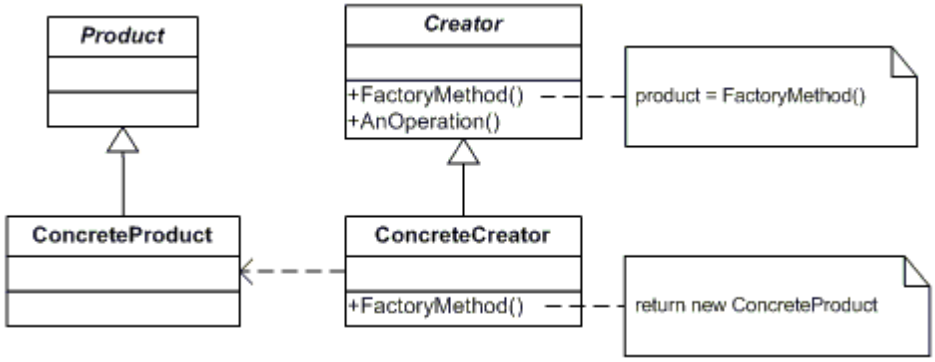

Ilustración 3 Patrón: FactoryMethod

1.7.1.4 Prototype

Prototype: Declara una interfaz definiendo el método clone()
ConcretePrototype: implementa el método clone()
Return: (Prototype) this
Client: Crea un nuevo objeto pidiendo a un prototipo que se clone

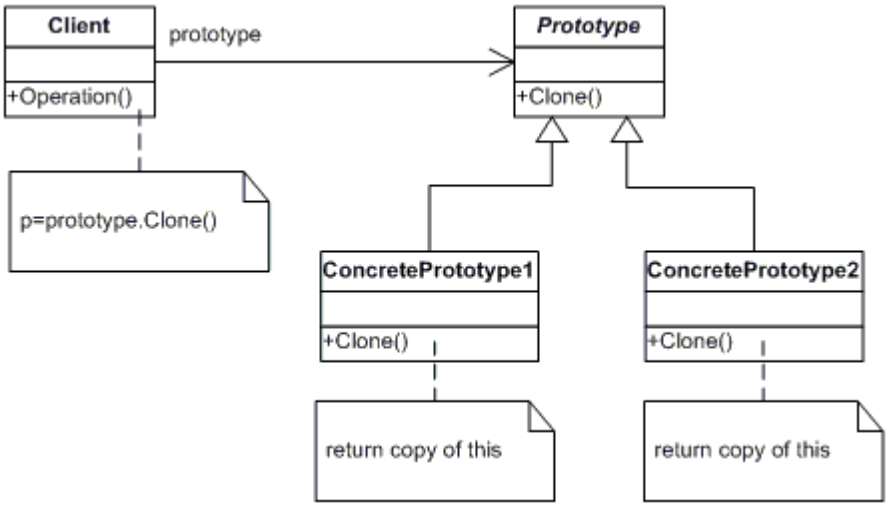

Ilustración 4 Patrón: Prototype

1.7.1.5 Singleton

Singleton: Define un método Instance() que permite al cliente el acceso a su única instancia. El método Instance() es estático. Esta misma clase es la responsable de la creación y mantenimiento de su propia instancia

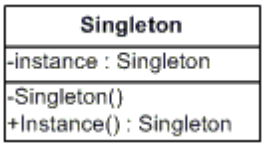

Ilustración 5 Patrón: Singleton

1.7.2 Estructurales

1.7.2.1 Adapter

Target: Define una interfaz de dominio específico que el cliente utiliza
Adapter: Adapta interfaces de Adaptee y Target
Adaptee: Define una interfaz que existiendo necesita adaptación
Client: utiliza objetos ajustándose a la interfaz ofrecida por Target

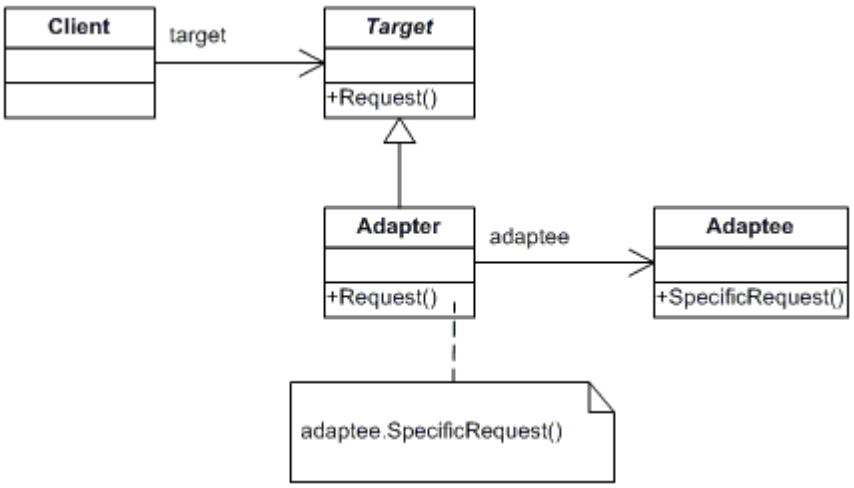

Ilustración 6 Patrón: Adapter

1.7.2.2 Bridge

Abstraction: define la interfaz de la abstracción
RefinedAbstraction: Extiende el interfaz definido por la abstracción
Implementor: define la interfaz de las clases de implementación, no tiene por qué corresponderse con el interfaz de Abstraction. Aquí se suelen proporcionar primitivas, en Abstraction operaciones de más alto nivel
ConcreteImplementor: Implementa la interfaz de Implementor

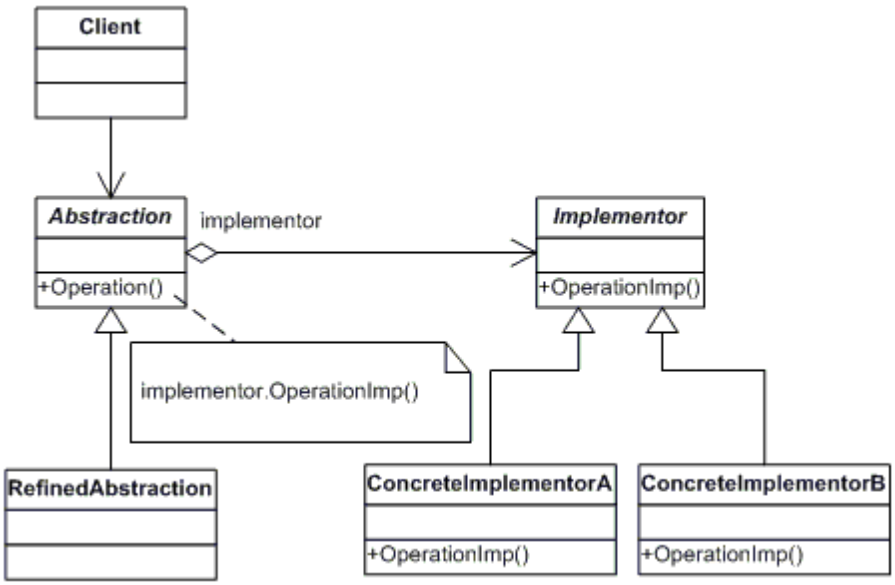

Ilustración 7 Patrón: Bridge

1.7.2.3 Composite

Component: declara la interfaz de los objetos involucrados en la composición, declara un interfaz para acceder y manejar sus hijos
Leaf: representa objetos hoja en la composición.
Composite: define el comportamiento de los componentes que tienen hijos, almacena los componentes hijo, implementa operaciones relacionadas con los hijos
Client: manipula los objetos de la composición a través de Component

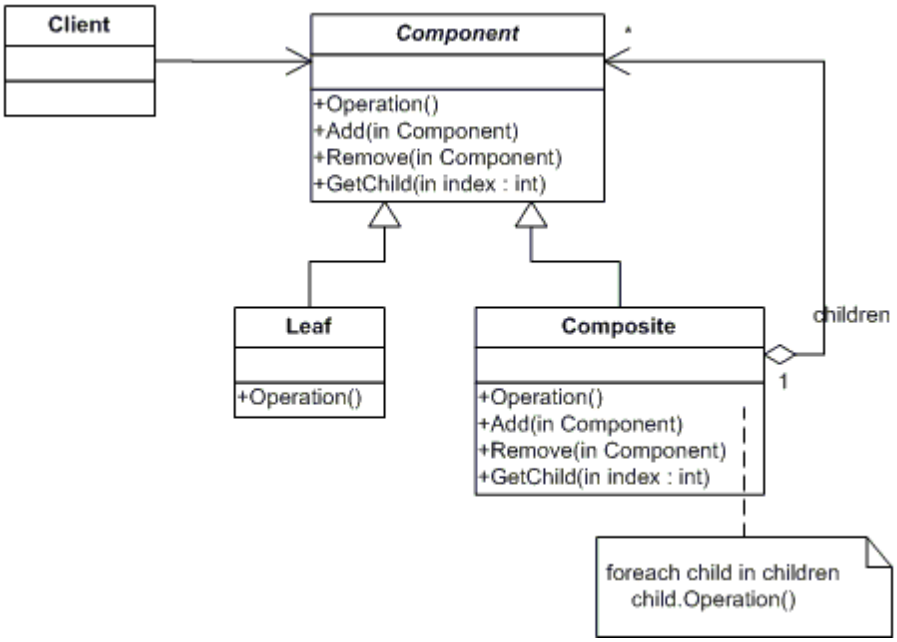

Ilustración 8 Patrón: Composite

1.7.2.4 Decorator

Component: Define la interfaz que ofrecen los objetos que poseen responsabilidades añadidas dinámicamente
ConcreteComponent: Define un objeto al que responsabilidades adicionales pueden serle añadidas
Decorator: Mantiene referencia a un objeto Component y define una interfaz que conforma el interfaz del Component
ConcreteDecorator: Añade las responsabilidades al Component

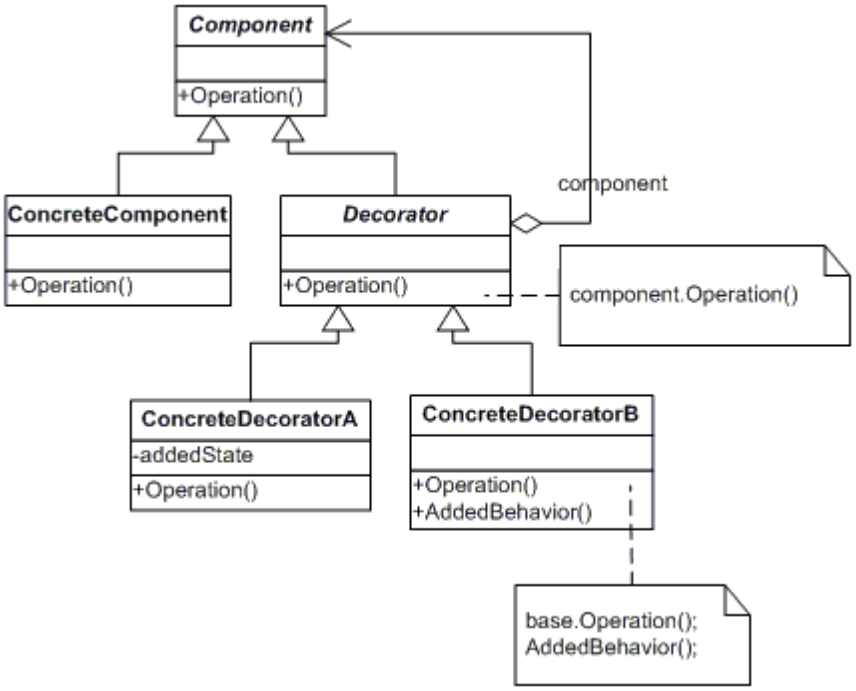

Ilustración 9 Patrón: Decorator

1.7.2.5 Facade

Facade: Conoce que clases subsistema son responsables de que peticiones y así, delega las peticiones a los objetos apropiados

Subsystem: Implementa la funcionalidad del subsistema, realiza el trabajo solicitado por el objeto Facade y no conoce, ni mantiene referencia alguna del objeto Facade

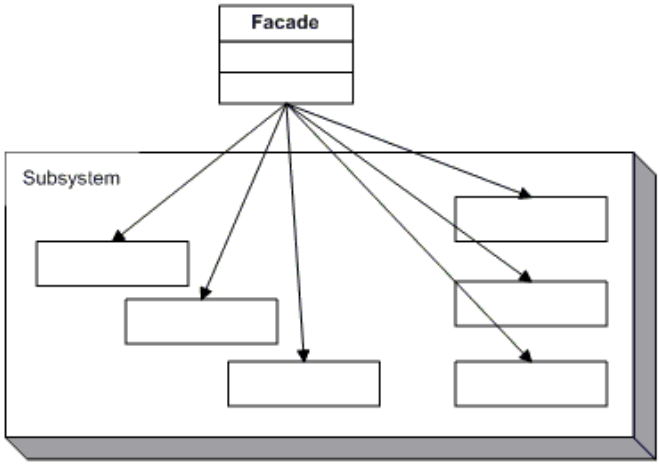

Ilustración 10 Patrón: Facade

1.7.2.6 Flyweight

Flyweight: declaran una interface a través de la que flyweights pueden recibir y actuar sobre estados externos

ConcreteFlyweight: implementa la interfaz del flyweight y añade almacenamiento para el estado interno
UnsharedConcreteFlyweight: no todas las subclases de Flyweight necesitan ser compartidas
FlyweightFactory: crea y gestiona los objetos Flyweight.
Client: mantiene una referencia a objetos flyweights.

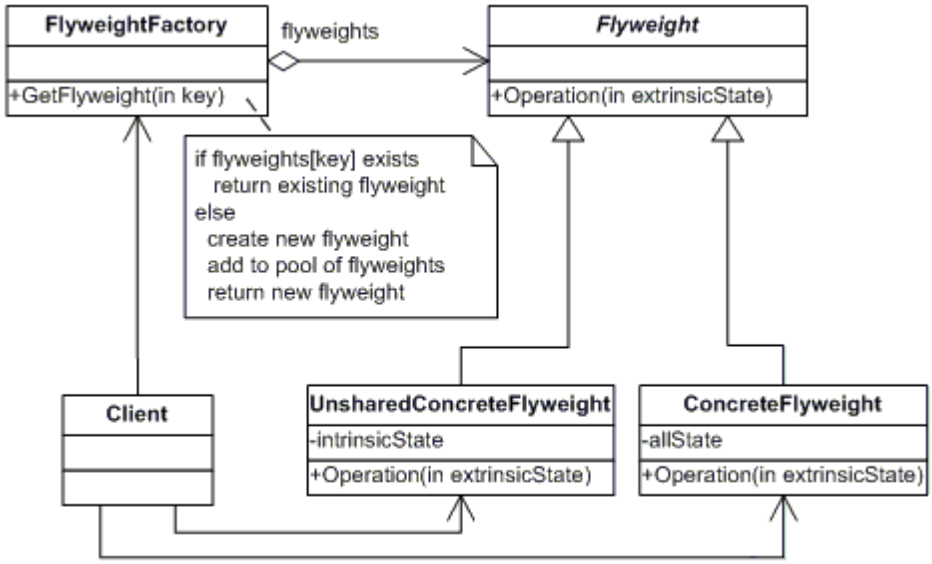

Ilustración 11 Patrón: Flyweight

1.7.2.7 Proxy

Proxy: tiene referencia al objeto RealSubject, tiene una interfaz idéntica a la de Subject así un proxy puede sustituirse por un RealSubject, y controla el acceso al RealSubject y puede ser el responsable de su creación y borrado
Subject: define una interfaz común para RealSubject y Proxy
RealSubject: define el objeto real que Proxy representa

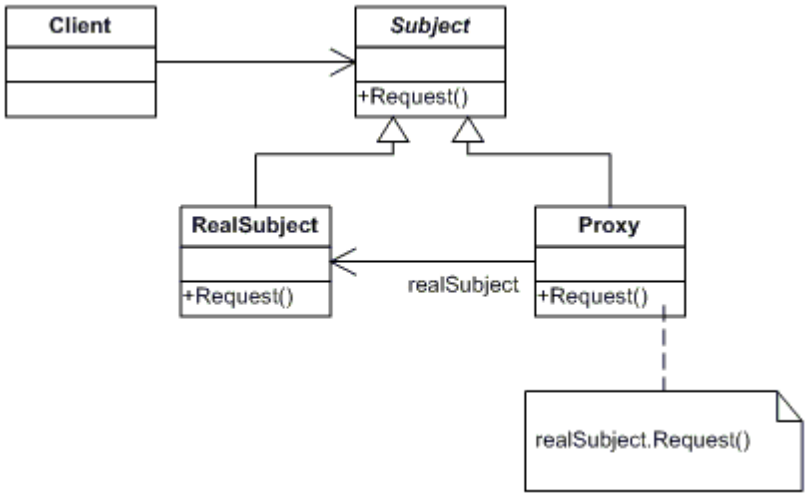

Ilustración 12 Patrón: Proxy

1.7.3 Comportamiento

1.7.3.1 Chain of Responsability

Handler: define una interfaz para manejar las peticiones, opcionalmente implementa enlaces de sucesión de atención de tales peticiones
ConcreteHandler: recibe las peticiones y las atiende si es posible, de otra forma pasa la petición a su sucesor
Client: hace las peticiones a los objetos ConcreteHandler pertenecientes a la cadena

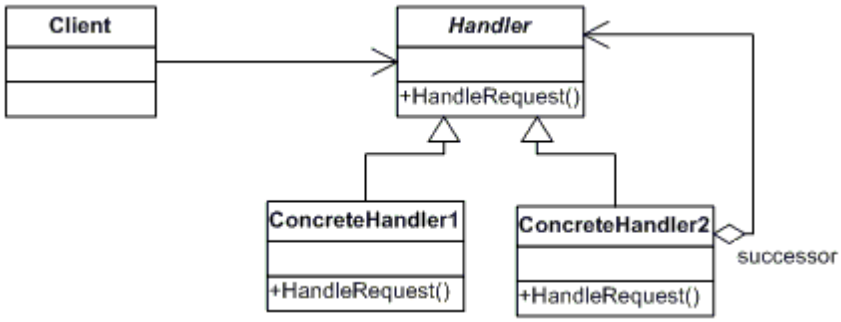

Ilustración 13 Patrón: Chain of Responsability

1.7.3.2 Command

Command: declara una interface para la ejecución de una operación
ConcreteCommand: define un vínculo entre un objeto Receiver y una acción, implementa Execute() invocando al método correspondiente de Receiver
Cliente: Crea un objeto ConcreteCommand y establece su receptor
Invoker: pide a Command que lleve a cabo su petición
Receiver: sabe cómo realizar las operaciones asociadas con la puesta en marcha de la petición

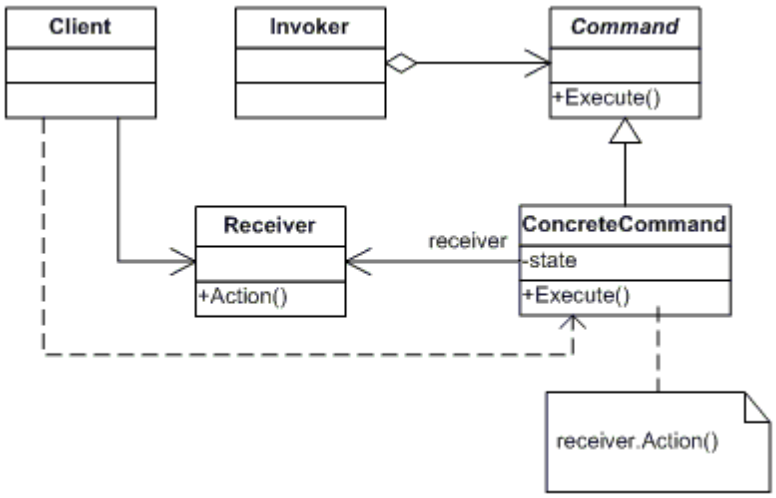

Ilustración 14 Patrón: Command

1.7.3.3 Interpreter

AbstractExpression: declara un interfaz para la ejecución de una operación
TerminalExpression: Implementa una operación de interpretación asociada con símbolos terminales asociados con la gramática

NonterminalExpression: implementa una operación de interpretación asociada con los símbolos no terminales asociados con la gramática
Context: contiene información global para el interprete
Client: construye un árbol sintáctico abstracto que representa una sentencia particular en el lenguaje que la gramática define

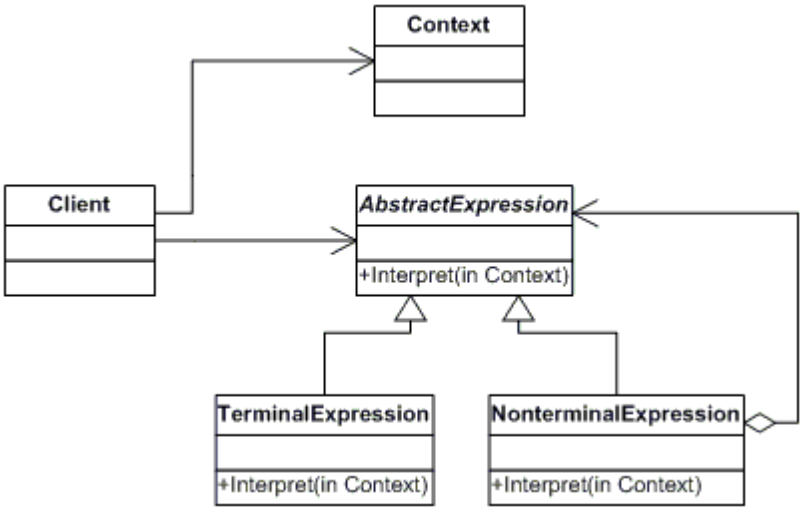

Ilustración 15 Patrón: Interpreter

1.7.3.4 Iterator

Iterator: define una interfaz para atravesar y acceder a los elementos
ConcreteIterator: implementa la interfaz del iterator, mantiene un puntero al conjunto de elementos
Aggregate:define un interfaz para crear un objeto iterator
ConcreteAggregator: Implementa la interfaz de creación del Iterator devolviendo un objeto de ConcreteIterator apropiado.

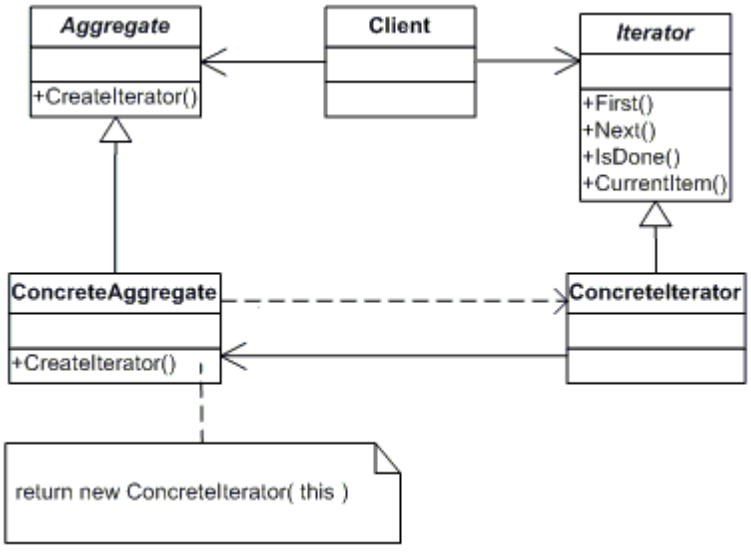

Ilustración 16 Patrón: Iterator

1.7.3.5 Mediator

Mediator: define una interfaz para la comunicación entre objetos Colleagues
ConcreteMediator: Implementa un comportamiento cooperativo coordinando objetos Colleagues
Colleague classes: cada objeto Colleague conoce su objeto Mediator, cada colleague comunica con su mediador cuando tiene que comunicar con otro colleague.

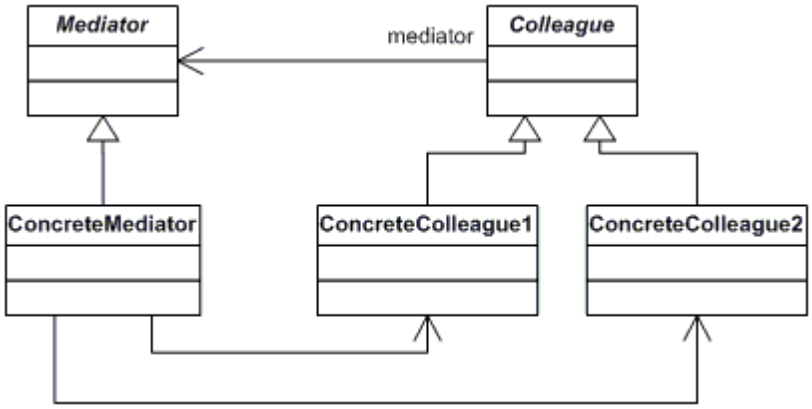

Ilustración 17 Patrón: Mediator

1.7.3.6 Memento

Memento: Almacena el estado interno del objeto Originator, protege del acceso por parte de objetos distintos al Originator
Originator: crea un objeto memento, que contiene una foto fija de su estado interno
Caretaker: es el responsable de mantener la seguridad del objeto memento, no opera o examina el contenido de memento

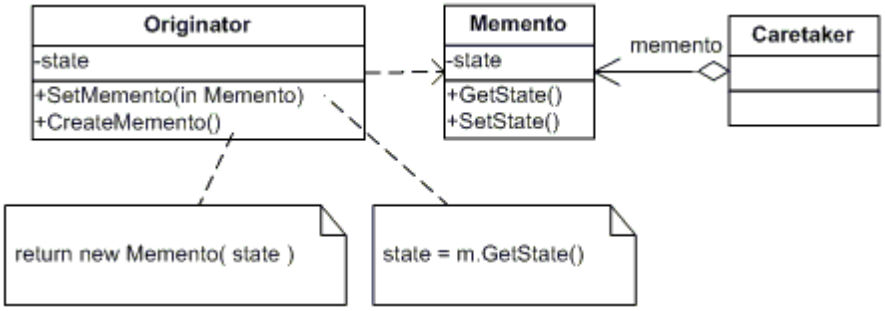

Ilustración 18 Patrón: Memento

1.7.3.7 Observer

Subject: conoce sus observadores y proporciona una interfaz para gestionarlos
ConcreteSubject: almacena el estado de interés y envía una notificación a sus observadores cuando su estado cambia
Observer: define una interfaz para los objetos a los que debe notificarse el cambio de estado en ConcreteSubject
ConcretObserver: mantiene una referencia a un objeto ConcreteSubject, mantiene información consistente relacionada con el estado implementando el método Update()

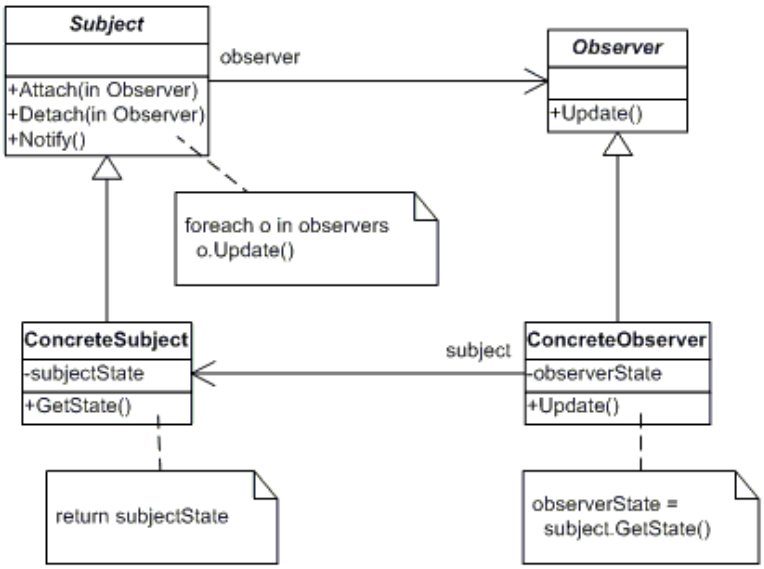

Ilustración 19 Patrón: Observer

1.7.3.8 State

Context: define la interfaz de interés a clientes
State: define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto
ConcreteState: cada subclase implementa un comportamiento asociado con un estado del Context

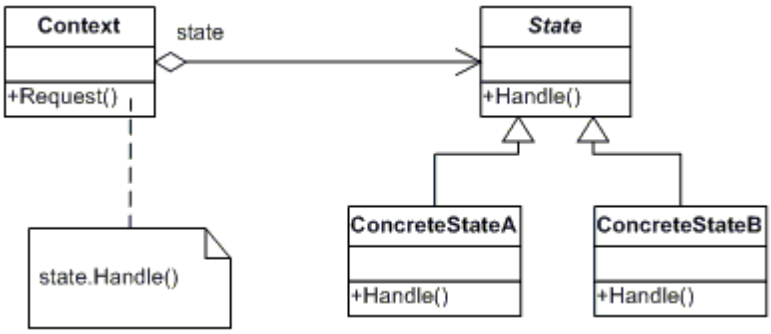

Ilustración 20 Patrón: State

1.7.3.9 Strategy

Strategy: declara un interfaz común para dar soporte a todos los algoritmos. Context utiliza esta interfaz para llamar al algoritmo definido por un ConcreteStrategy
ConcreteStrategy: Implementa el algoritmo usando la interfaz Strategy
Context: se configura con un objeto ConcreteStrategy, sobre el que mantiene una referencia, puede definir una interfaz que permita a Strategy acceder a sus datos.

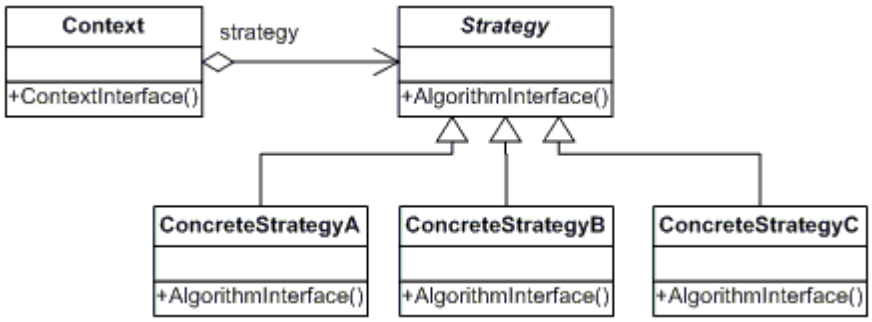

Ilustración 21 Patrón: Strategy

1.7.3.10 Template Method

AbstractClass: define operaciones primitivas abstractas cuya concreción se delega a las subclases, estas primitivas se utilizan en el cuerpo de un algoritmo esqueleto
ConcreteClass: implementa las operaciones primitivas abstractas anteriores

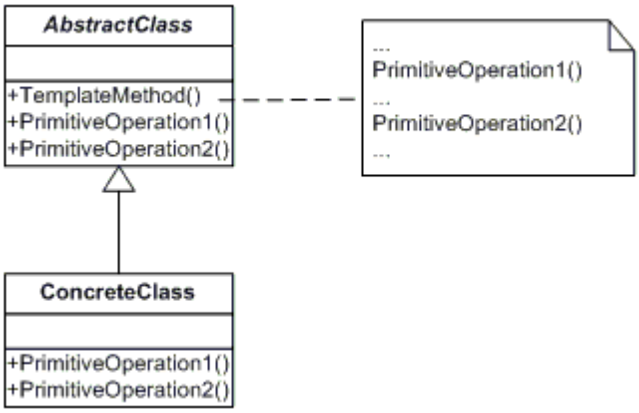

Ilustración 22 Patrón: Template Method

1.7.3.11 Visitor

Visitor: Establece las operaciones a realizar
ConcreteVisitor: implementa dichas operaciones
Element: define un método Accept() que toma un Visitor como argumento
ConcreteElement: implementa el método Accept()**ObjectStructure**:

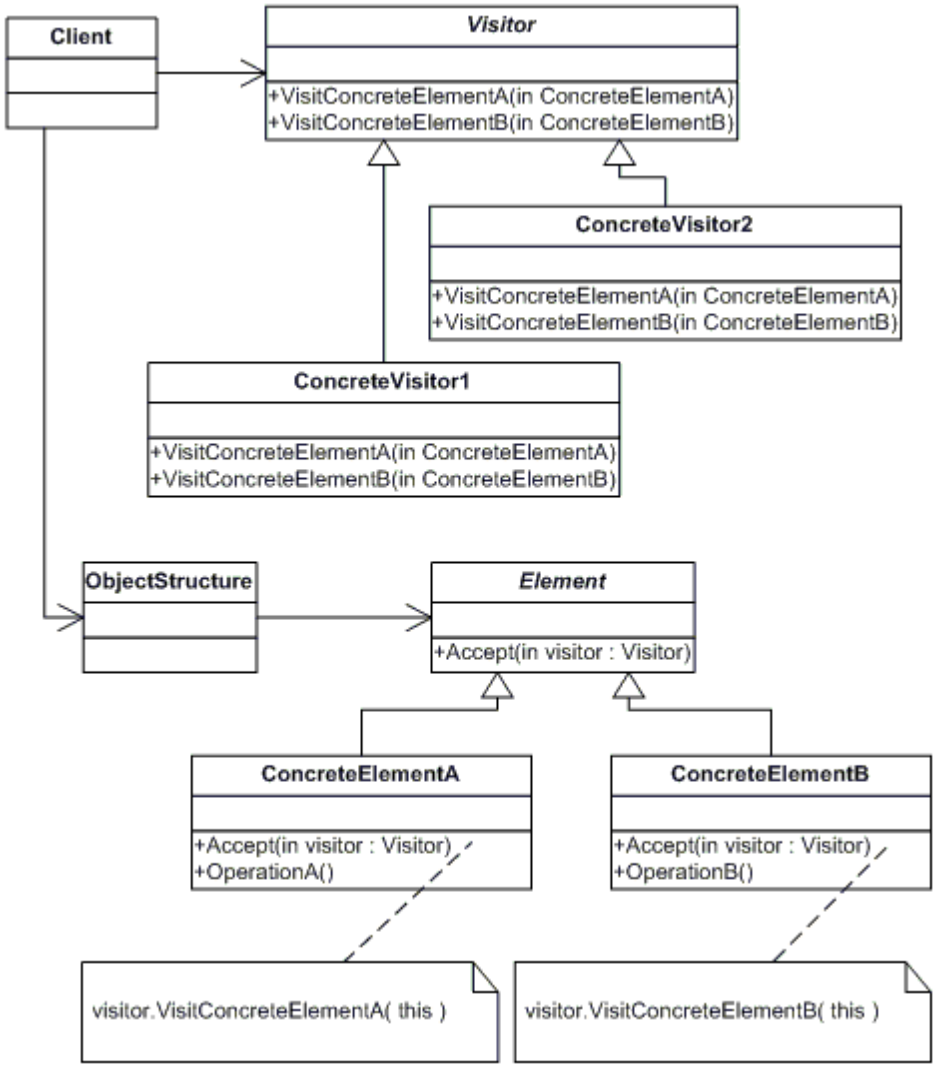

Ilustración 23 Patrón: Visitor