Patrones de Diseño

Ezequiel Postan

1 Libro e índice

Gamma, E., Helm, R., Johnson, R., Vlissides, J., Patrones de diseño, Addison-Wesley, 2003.

- Páginas 2-69: Introducción. Composite. Strategy. Decorator. Abstract Factory. Bridge. Command. Iterator. Visitor
- Páginas 79-87: Abstract Factory
- Páginas 141-169: Bridge, Composite (en la 151) , Decorator (en la 161)
- Páginas 215-223: Command
- Páginas 237-249: Iterator
- Páginas 289-297: Strategy
- Páginas 305-316: Visitor

1.1 Clasificación de Patrones

- **Patrones de Creación:** Abstract Factory.
- **Patrones Estructurales:** Bridge, Composite, Decorator.
- **Patrones de Comportamiento:** Command, Iterator, Strategy, Visitor.

2 Introducción

2.1 ¿Por qué usar patrones de diseño?

Algunas breves ventajas de utilizar patrones de diseño

- Los patrones de diseeño son soluciones estudiadas que se han aplicado cn éxito en sistemas diversos y que han respondido con buenos resultados a las exigencias que plantearon resolver.
- Permiten reutilizar con mayor facilidad buenos diseños y arquitecturas
- Facilitan la documentación de los sistemas

2.2 ¿Qué es un patrón de diseño?

Un patrón representa una solución a un problema en término de objetos e interfaces. En general consta de cuatro partes

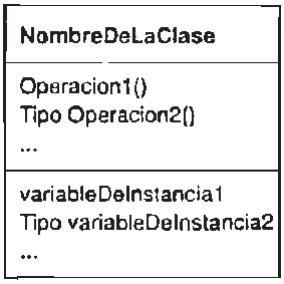
1. **Nombre:** Permite generar un vocabulario de diseño, esto trae ventajas de comunicación dentro de un grupo de trabajo y acorta la documentación refiriendo a los conceptos representados por el patrón.
2. **Problema:** Describe las características y contexto en que aplicar el patrón puede traer buenos resultados.
3. **Solución:** Describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones. No describe una implementación concreta. Es una descripción abstracta de la solución a un conjuntos de problemas con características comunes.
4. **Consecuencias:** Resultados, ventajas e inconvenientes al aplicar el patrón. Impacto sobre la flexibilidad, extensibilidad y portabilidad de un sistema.

2.3 Cómo resuelven los patrones los problemas de diseño

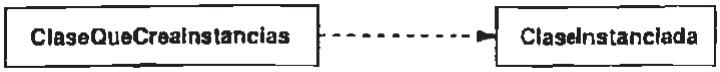
- Encontrar los módulos apropiados: Sugieren la consideración de módulos que muchas veces no aparecen intuitivamente. El uso de una mala metodología como intentar crear objetos que sólo representen el contexto en que se trabaja sugiere modelos que ignoran la metodología de Parnas. Dejando de lado la consideración de crear módulos para ocultar cambios en algoritmos o estados que muchas veces no se reflejan en un mundo real.
- Determinan la granularidad de los módulos: Establecen el nivel de abstracción adecuado para representar componentes del sistema.
- Especificar las interfaces de los módulos: Establecen relaciones de herencia, composición y signatura de métodos que aparecen en la interfaz de los módulos.

2.4 Notación

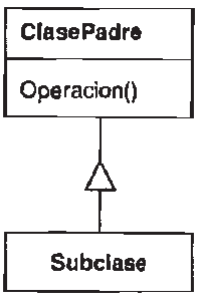
Caja: Representa un módulo.



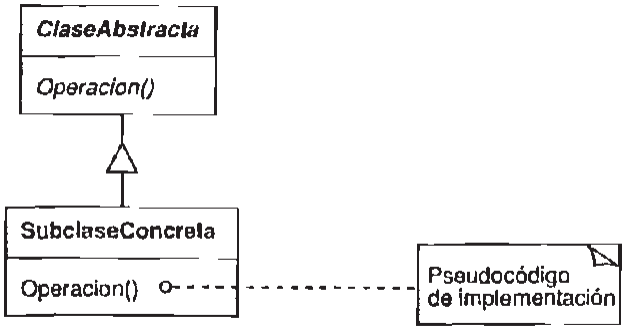
Flecha puntuada: Un módulos crea instancias de otro.



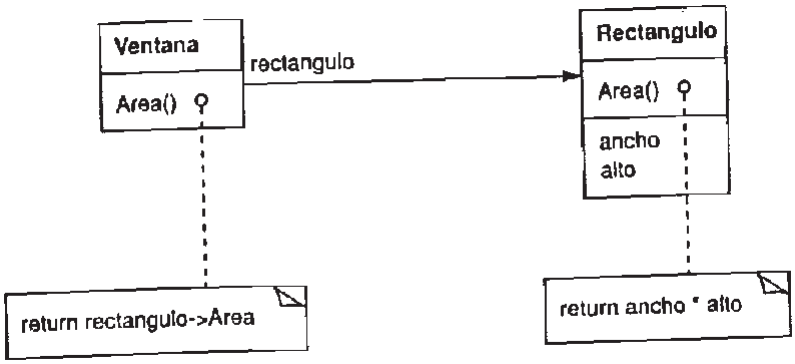
Línea vertical con triángulo: Herencia de clases



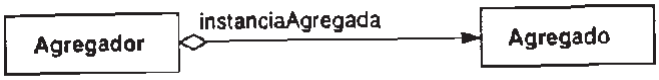
Caja con nombre en cursiva: Módulo abstracto (lo que llamamos supertipo para usar herencia de interfaz). No implementa su interfaz. Sus herederos pueden tener asociados un cuadro con la esquina doblada unido con una linea de puntos con pseudocódigo de los métodos que implementa



Flecha lisa: Un módulo tiene referencia a una instancia de otro módulo (asociación). Es opcional colocar un nombre a la referencia



Flecha lisa con un rombo: El autor lo llama agregación, es lo que nosotros definimos como composición (forma parte del estado y suele delegar operaciones)



2.5 ¿Cómo seleccionar un patrón de diseño?

- Considerar los problemas que resuelve cada patrón
- Prestar atención al propósito de cada patrón y a toda su documentación.
- Estudiar como se interrelacionan los patrones
- Analizar patrones de propósito general
- Examine los item de cambio
- Piense en qué aspectos querrá modificar el diseño a futuro (ampliaciones/reducciones del sistema)

3 Abstract Factory (Kit)

Es un patrón de creación, estos patrones aseguran que el sistema se escriba en términos de interfaces y no de implementación.

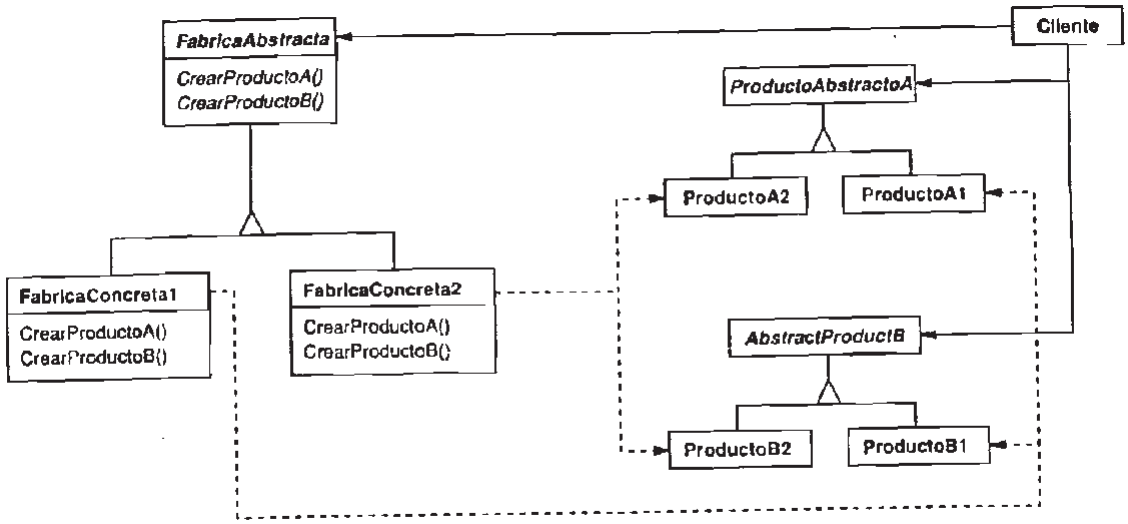
3.1 Objetivo

Brindar interfaz para crear familias de objetos relacionados entre sí

3.2 Aplicaciones

- Cuando el sistema debe ser independiente de como se crean, componen y representan sus productos
- Cuando el sistema debe ser configurado con una familia de producto entre varias
- Cuando es necesario hacer cumplir una restricción de que una familia de objetos sea usado conjuntamente

3.3 Estructura



3.4 Participantes

- Fábrica Abstracta (FabricaDeUtiles): Declara una interfaz para operaciones que crean objetos producto abstracto
- Fábrica Concreta: Implementa las operaciones para crear productos de objetos concretos
- Producto Abstracto: Declara la interfaz para un tipo de objeto producto
- Producto Concreto: Define un objeto producto que será creado por una fábrica concreta correspondiente. Implementa interfaz de producto abstracto
- Cliente: Sólo usa interfaces declaradas por las clases Fábrica Abstracta y Producto Abstracto.

3.5 Colaboraciones

- Normalmente sólo se crea una única instancia de una fábrica concreta en tiempo de ejecución. Esta fábrica concreta crea objetos producto que tienen una determinada implementación. Para crear distintos objetos producto, los clientes deberán usar una fábrica concreta diferente.
- Fábrica Abstracta delega la creación de objetos producto en su subclase Fábrica Concreta

3.6 Consecuencias

Aísla clases concretas. Los nombres de las clases producto concreto nunca aparecen en el código del cliente

Facilita intercambio de familias de productos: Sólo basta con cambiar de fábrica concreta y volver a crear los objetos producto

Promueve la consistencia entre productos

Es dificil dar lugar a nuevos tipos de productos

3.7 Patrones Relacionados

- Fábrica Abstracta suele implementarse con Factory Method o en ocasiones con prótotype (cuando se usan varias familias de productos)
- Una Fábrica Concreta suele ser un Singleton

4 Bridge (Handle/Body)

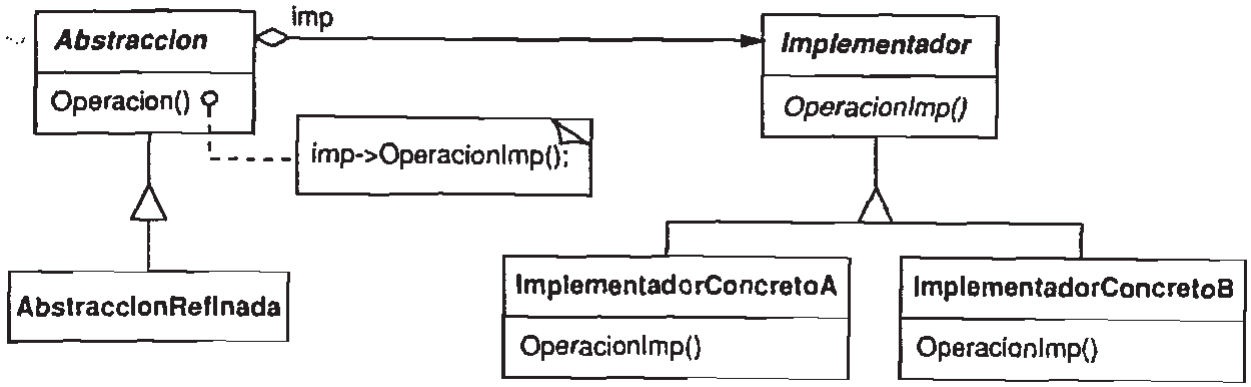
4.1 Objetivo

Desacopla una abstracción de una implementación de manera tal que a,bas puedan variar de forma independiente

4.2 Aplicaciones

- Evitar un enlace permanente entre una abstracción y su implementación. Permite variar ambos en tiempo de ejecución.
- Evitar que cambios de la implementación de una abstracción impacte en el código del cliente
- Compartir una implementación entre varios objetos
- Cuando uno tiene un sistema en el que variarán ciertos dispositivos que brindan primitivas básicas para los requerimientos funcionales de otros módulos (ejemplo: al portar un sistema a otra plataforma, otro SO, etc). La idea es encapsular los requerimientos funcionales en una abstracción y por cada sistema crear una abstracción refinada que se compondrá con un implementador característico del sistema que brindará las primitivas para la abstracción. Se espera que todas las abstracciones refinadas cumplan aproximadamente las mismas funciones

4.3 Estructura



4.4 Participantes

- Abstracción: Define la interfaz de la abstracción. Mantiene una referencia a un objeto de tipo implementador
- Abstracción Refinada: Extiende la interfaz definida por Abstracción
- Implementador: Define la interfaz de las clases de implementación. Esta interfaz no tiene por qué corresponderse exactamente con la de Abstracción; de hecho ambas interfaces podrían ser muy distintas. Normalmente la interfaz implementador sólo proporciona operaciones primitivas, y Abstracción define operaciones de más alto nivel basadas en dichas primitivas.
- Implementador Concreto: Implementa la interfaz de implementador y define su implementación concreta

4.5 Colaboraciones

Abstracción redirige las peticiones del cliente a su objeto implementador.

4.6 Consecuencias

- Desacopla la interfaz de la implementación. Permite cambios dinámicos de implementación lo que evita tener que recompilar para reconfigurar un sistema
- Mejora la extensibilidad. Podemos extender las gerarquías de Abstracción y de Implementador de forma independiente
- Oculta detalles de implementación a los clientes

4.7 Patrones Relacionados

El patrón Abstract Factory puede crear y configurar el Bridge

5 Composite

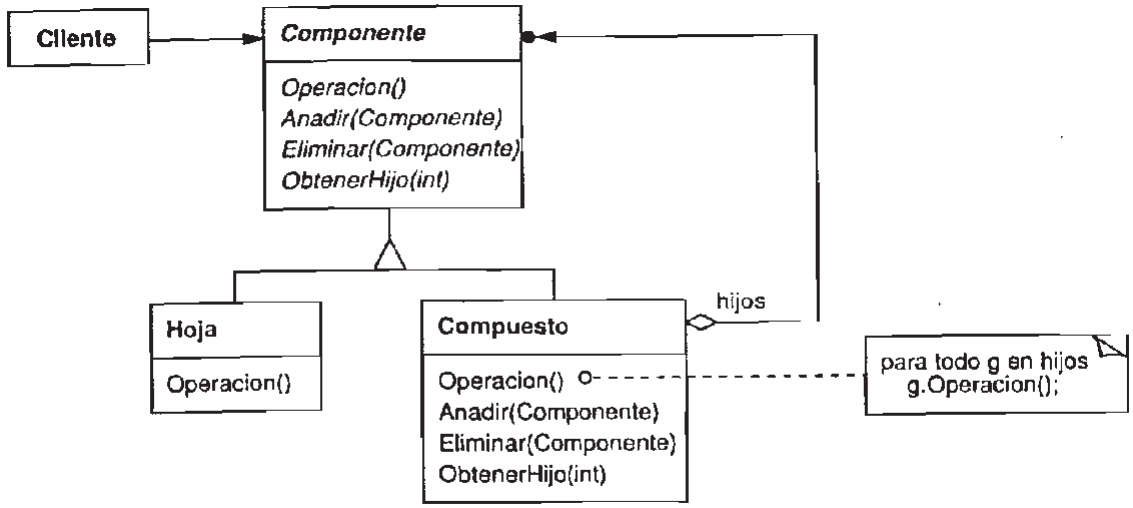
5.1 Objetivo

Compone objetos en estructura de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos

5.2 Aplicaciones

- Representar gerarquías de objetos parte-todo
- Permitir que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales

5.3 Estructura



5.4 Participantes

- Componente: Declara la interfaz de los objetos de la composición. Declara una interfaz para acceder a sus componentes hijos y gestionarlos. Opcionalmente también define la interfaz para acceder al padre de un componente en la estructura recursiva.
- Hoja: Representa objetos hojas en la composición, es decir los que no tienen hijos. Define el comportamiento de los objetos hijos de la composición.
- Compuesto: Define el comportamiento de los componentes que tienen hijos. Almacena componentes hijos. Implementa las operaciones de la interefaz Componente relacionadas con los hijos.
- Cliente: Manipula objetos en la composición a través de la interfaz de Componente

5.5 Colaboraciones

En general el cliente manipula objetos en la composición a traves de la interfaz de Componente. Si el objeto es una hoja simplemente ejecuta su acción. Si es un compuesto suele redirigir la operación las peticiones a sus componentes hijos realizando operaciones adicionales antes o después.

5.6 Consecuencias

- Define gerarquías de clases formadas por componentes primitivos y compuestos
- Simplifica el código del cliente ya que le permite a este tratar indistintamente a los objetos primitivos y compuestos.
- Facilita añadir nuevos tipos de componentes
- Puede hacer que el diseño sea demasiado general en el sentido de que no creamos restricciones para establecer cuáles objetos se componen con cuáles otras.

5.7 Patrones Relacionados

- Decorator es normalmente usado junto con Composite. Decoradores y compuestos suelen tener una clase padre común. Por tanto los decoradores tendrán que admitir la interfaz Componente con operaciones como Añadir, Eliminar y ObetenerHijo.
- Iterator para recorrer las estructuras definidas por el patrón Composite
- Visitor localiza operaciones y comportamiento que de otro modo estaría distribuído en varias clases Compuesto y Hoja
- Flyweight permite compartir componentes
- Chain of responsability suele usarse cuando se implementa referencia al padre de cada componente

6 Decorator (Wrapper)

Evitar explosión de clases.

6.1 Objetivo

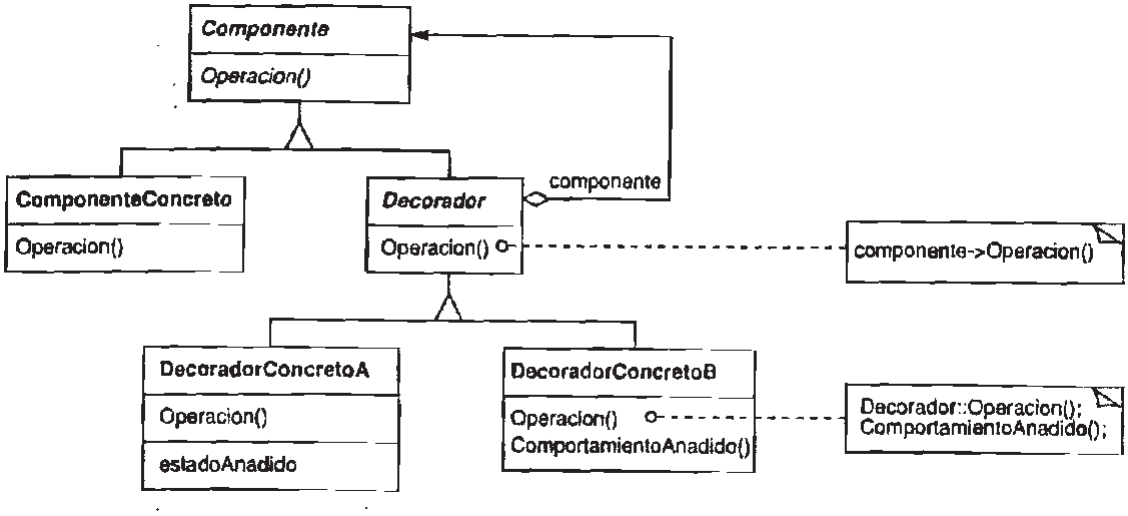
Asigna responsabilidades a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

6.2 Aplicaciones

Añadir responsabilidades que pueden ser retiradas

Cuando la extensión mediante la herencia no es viable. A veces es posible tener un gran número de extenciones independientes, produciéndose una explosión de subclases para permitir todas las combinaciones. O puede ser que una definición de una clase esté oculta o que no esté disponible para ser heredada

6.3 Estructura



6.4 Participantes

- Componente: Define la interfaz para objetos a los que se puede añadir responsabilidades adicionales.
- Componente Concreto: Define un objeto al que se pueden añadir responsabilidades adicionales.
- Decorador: Mantiene una referencia a un objeto Componente y define una interfaz que se ajuste a la interfaz del Componente.
- Decorador Concreto: Añade responsabilidades al Componente.

6.5 Colaboraciones

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

6.6 Consecuencias

- Brinda más flexibilidad que la herencia estática.
- Evita clases cargadas de funciones en la parte superior de la jerarquía.
- Deriva en diseño formado por muchos componentes pequeños
- Puede generar diseños difíciles de entender o depurar

6.7 Patrones Relacionados

- Composite
- Adapter: La diferencia de un Decorador con un Adaptador es que el Decorador sólo cambia las responsabilidades de un objeto, no modifica su interfaz. Mientras que un Adaptador le da a un objeto una interfaz completamente nueva.
- Strategy: Un Decorador permite cambiar el exterior de un objeto. Una estrategia permite cambiar sus "tripas". Son dos alternativas de modificar un objeto

7 Command (Action, Transaction)

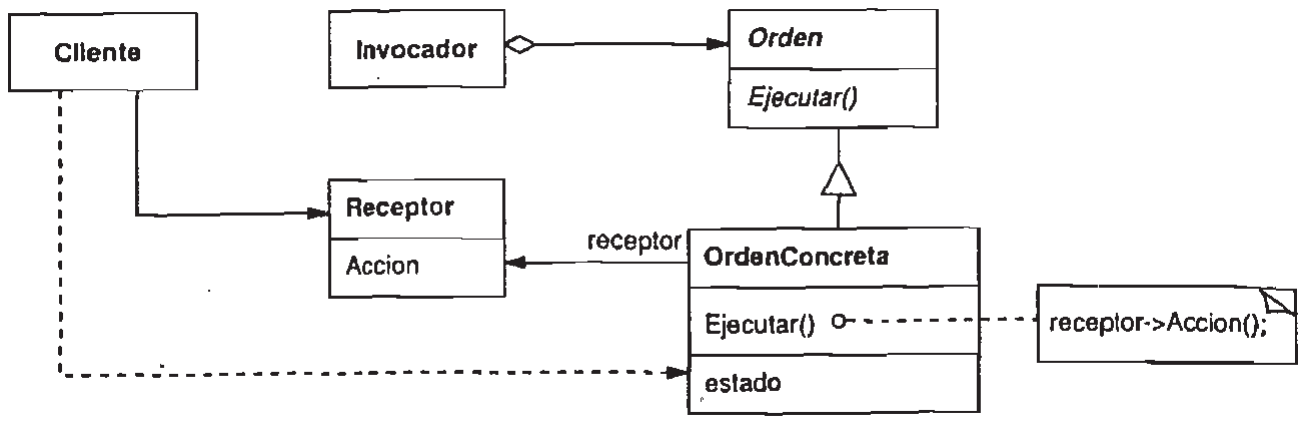
7.1 Objetivo

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.

7.2 Aplicaciones

- Parametrizar objetos con una acción a realizar
- Especificar, poner en cola y ejecutar peticiones en diferentes instantes de tiempo
- Permitir "Deshacer": La operación Ejecutar de la Orden puede guardar el estado que anule los efectos que la orden causa. Debe añadirse a la interfaz Orden la operación Deshacer que anule los efeectos de la llamada a Ejecutar. Las órdenes ejecutadas se guardan en una lista que funciona como historial. Se pueden lograr niveles ilimitados de deshacer y repetir recorriendo dicha lista hacia atrás y hacia delante llamando respectivamente a Deshacer y Ejecutar.
- Permitir regjstrar los cambios que permiten volver a ejecutar las acciones en caso de caída del sistema. Agregando operaciones de guardar y cargar se puede hacer un registro persistente de los cambios.

7.3 Estructura



7.4 Participantes

- Orden: Declara una interfaz para ejecutar una operación
- Orden Concreta: Define un enlace entre un objeto Receptor y su acción. Implementa Ejecutar invocando la correspondiente operación u operaciones del Receptor
- Cliente: Crea un objeto Orden Concreta y establece su receptor
- Invocador: Le pide a la orden que ejecute su Acción. Ejemplo botones, sensores, etc.
- Receptor: Sabe cómo llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede hacer actuar como Receptor. (Actuadores)

7.5 Colaboraciones

- El Cliente creo un objeto Orden Concreta y especifica su Receptor
- Un objeto Invocador almacena el objeto Orden Concreta
- El Invocador envía una perición llamando a Ejecutar sobre la orden. Cuando las órdemes se pueden deshacer, Orden Concreta guarda el estado para deshacer la orden antes de llamar a Ejecutar.
- El objeto Orden Concreta invoca operaciones de su Receptor para llevar a cabo la petición.

7.6 Consecuencias

- Orden desacopla el objeto que invoca la operación de aquel que sabe cómo realizarla
- La órdenes son objetos de primera clase. Pueden ser manipulados y extendidos como cualquier otro objeto
- Se pueden ensamblar órdenes en una orden compuesta. En general las órdenes compuestas con una instancia del patrón Composite
- Es fácil agregar nuevas órdenes ya que no hay que cambiar las clases existentes

7.7 Patrones Relacionados

- Composite para implementar órdenes compuestas
- Memento para mantener el estado que necesitan las órdenes para anular (deshacer) sus efectos
- Una orden que puede ser copiada antes de ser guardada en el historial funciona como un Prototype

8 Iterator (Cursor)

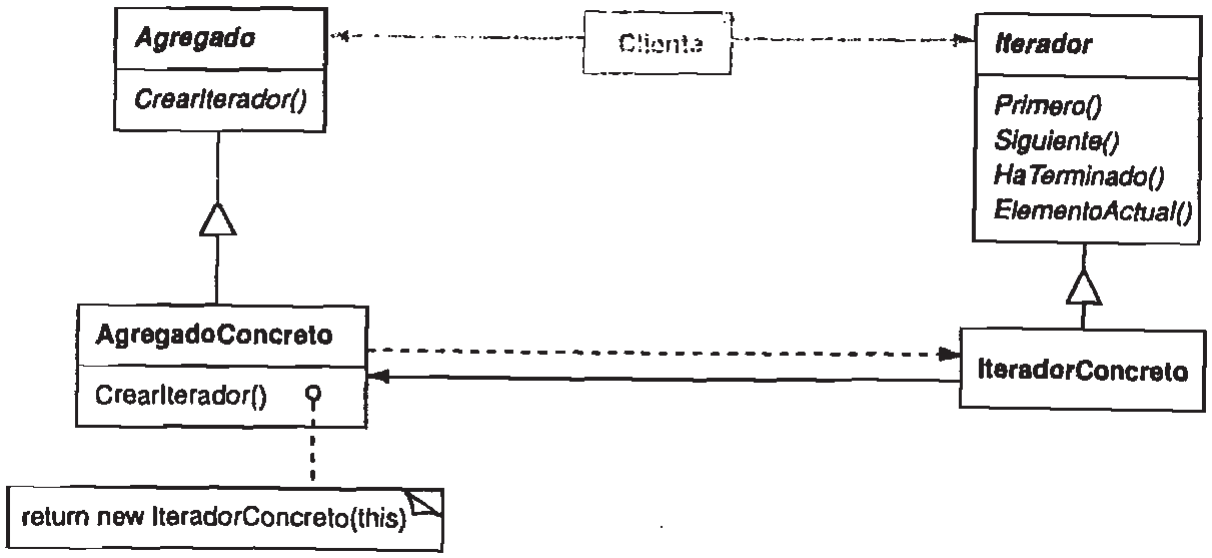
8.1 Objetivo

Proporsiona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

8.2 Aplicaciones

- Permitir varios recorridos sobre objetos agregados
- Proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (permitir iteración polimórfica)

8.3 Estructura



8.4 Participantes

- Iterador: Define una interfaz para recorrer los elementos y acceder a ellos
- Iterador Concreto: Implementa la interfaz de Iteradoe. Mantiene la posición actual en el recorrido de agregado
- Agregado: Define la interfaz para crar un objeto Iterador
- Agregado Concreto: Implementa la interfaz de creación de Iterador para devolver una instancia del Iterador Concreto apropiado

8.5 Colaboraciones

Un Iterador Concreto sabe cuál es el objeto actual del Agregado y puede calcular el objeto siguiente en el recorrido.

8.6 Consecuencias

- Permite variaciones en el recorrido de un agregado
- Lo iteradores simplifican la interfaz de los Agregados ya que el recorrido de los mismos se define en una interfaz separada
- Se puede hacer más de un recorrido a la vez sobre un mismo agregado ya que cada iterador mantiene el estado de su propio recorrido.

8.7 Patrones Relacionados

- Composite: Los iteradores suelen aplicarse a estructuras recursivas como los Compuestos
- Factory Method: Iteradores polimórficos se basan en métodos de fabricación para crear las instancias de las subclases apropiadas del Iterador
- Memento: Para representar el estado de una iteración. El Iterador almacena el Memento internamente

9 Strategy (Policy)

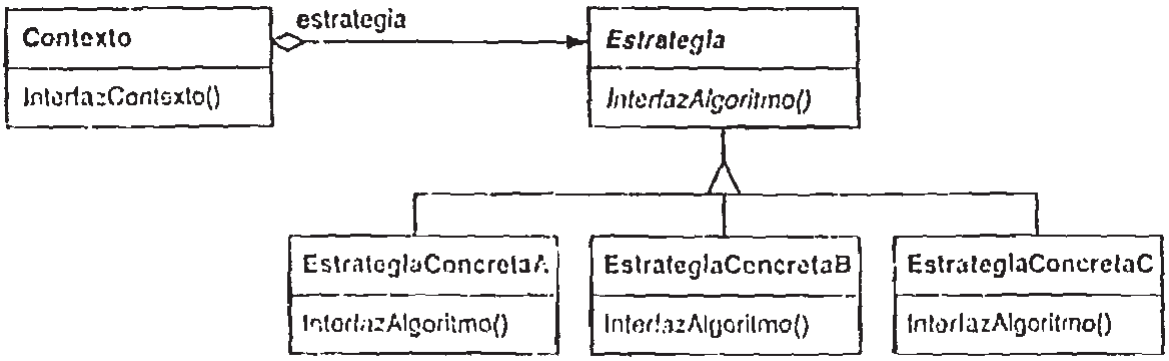
9.1 Objetivo

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 usen.

9.2 Aplicaciones

- Muchas clases relacionadas difieren sólo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
- Se necesitan distintas variantes de un algoritmo
- Un algoritmo usa datos que los clientes no deberían conocer. Este patrón evita exponer estructuras de datos complejas y dependientes del algortimo.
- Cuando una clase define muchos comportamientos podemos evitar la inclusión de condicionales trasladando cada comportamiento a su propia clase estrategia-

9.3 Estructura



9.4 Participantes

- Estrategia: Declara una interfaz común a todos los algoritmos permitidos. El Contexto usa esta interfaz para llamar al algoritmo definido por una Estrategia Concreta
- Estrategia Concreta: Implementa el algoritmo
- Contexto: Se configura con un objeto Estrategia Concreta. Mantiene una referencia a un objeto Estrategia. Puede definir una interfaz que permita a la Estrategia acceder a sus datos

9.5 Colaboraciones

- Estrategia y Contexto interactúan para implementar el algoritmo elegido. Un Contexto puede pasar a la Estrategia todos los datos requeridos por el algortimo cada vez que se llama a éste. Otra alternativa es que el Contexto se pase a sí mismo como argumento de las operaciones Estrategia. Esto permite a la Estrategia hacer llamadas al contexto cuando sean necesario.
- Un Contexto redirige peticiones de los clientes a su Estrategia. Los clientes normalmente crean un objeto Estrategia Concreta, el cual pasan al Contexto. Por tanto, los clientes interactúan exclusivamente con el Contexto.
- Suele haber una familia de clases Estrategia Concreta a elegir por el cliente.

9.6 Consecuencias

- Se brinda una alternativa a la herencia. No se crea un contexto por cada algoritmo.
- Se eliminan sentencias condicionales para elegir algoritmos.
- Permite variar las implementaciones de un mismo algoritmo para explotar eficiencia
- Los clientes deben conocer las diferentes estrategias
- Costo de comunicación entre Contexto y Estrategia
- Incremento del número de objetos

9.7 Patrones Relacionados

- Flyweight: Los objetos Estrategia suelen ser buenos pesos ligeros

10 Visitor

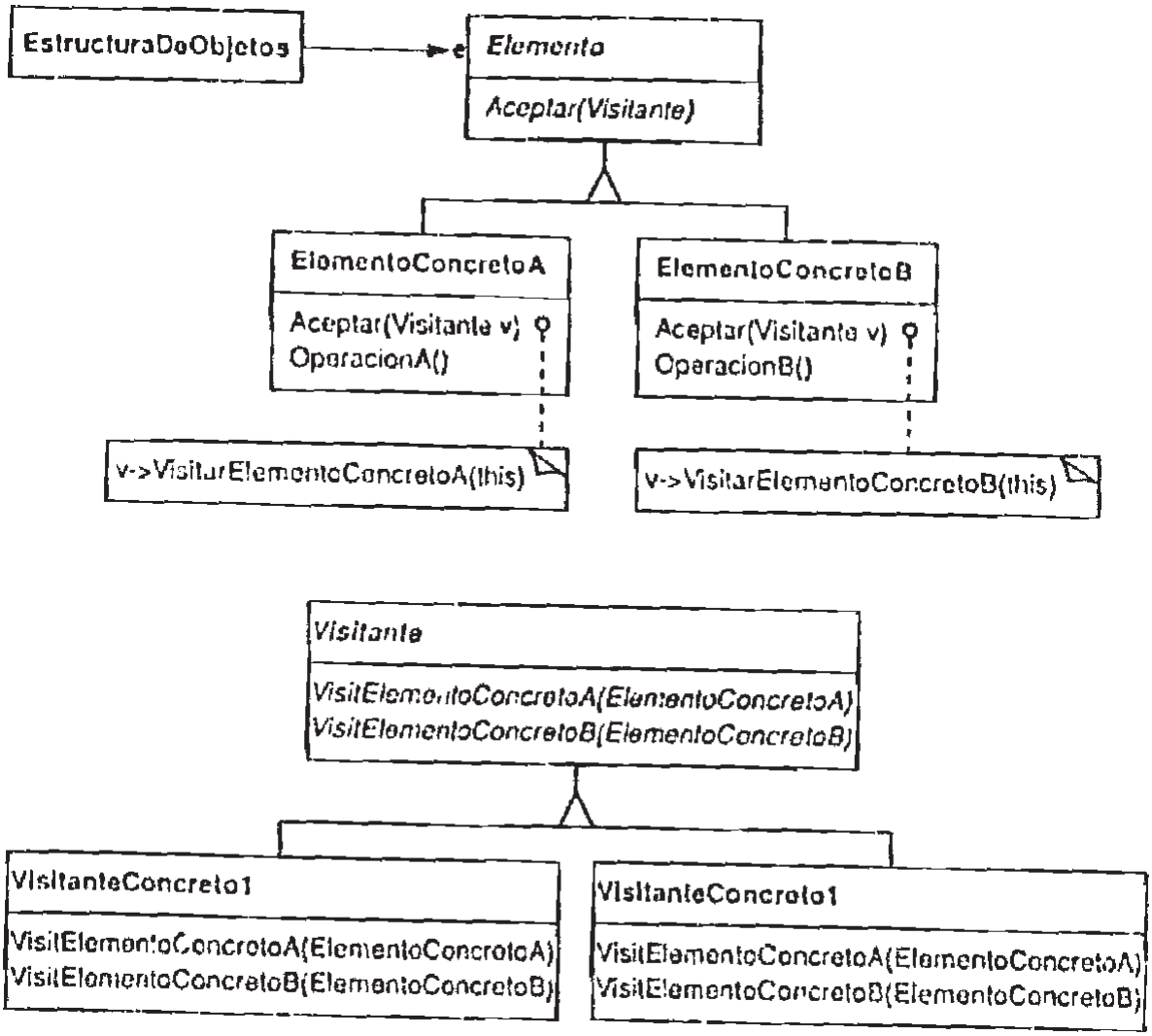
10.1 Objetivo

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

10.2 Aplicaciones

- Cuando una estructura de objetos contiene muchas clases de objetos con diferentes interfaces y queremos realizar operaciones sobre esos elementos que dependen de su clase concreta
- Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos y queremosevitar contaminar sus clases con dichas operaciones
- Las clases que definen la estructura de objetos rara vez cambian, pero queremos definir nuevas operaciones sobre la estructura

10.3 Estructura



10.4 Participantes

- Visitante: Declara una operación Visitar para cada clase de operación Elemento Concreto de la Estructura de Objetos. El nobre y signatura de la operación identifican a la clase que envían la petición Visitar al Visitante. Esto permite al Visitante determinar la clase concreta de Elemento que está siendo visitada. A continueación el Visitante puede acceder al elemento directamente a través de su interfaz particular.
- Visitante Concreto: Implementa cada operación declarada por Visitante. Cada operación implementa un fragmento del algoritmo definido para la clase correspondiente de la estructura. Visitante Concreto proporciona el contexto para el algoritmo y guarda su estado local. Muchas veces este estado acumula resultados durante el recorrido de la estructura.

- Elemento: Define una operación Aceptar que toma un Visitante como argumento
- Elemento Concreto: Implementa una operación Aceptar que toma un visitante como argumento.
- Estructura de Objetos: Puede enumerar sus elementos. Puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos. Puede ser un compuesto del patrón Composite o una colección como una lista o un conjunto

10.5 Colaboraciones

- Un cliente que usa el patrón Visitor debe crear un objeto Visitante Concreto y a continuación recorrer la estructura, visitando cada objeto con el visitante
- Cada vez que se visita a un Elemento, este llama a la operación del Visitane que se corresponde con su clase. El elemento se pasa a sí mismo como argumento de la operación para permitir al visitante acceder a su estado en caso de que sea necesario.

10.6 Consecuencias

- El Visitante facilita añadir nuevas operaciones
- Un Visitante agrupa operaciones relacionadas y aísla las que no lo esan
- Es dificil añadir nuevas clases de Elementos Concretos
- A diferencia de un Iterador, el patrón Visitor permite visitar varias jerarquías de clases y no restringe a que todos los Elementos Concretos sean herederos de una misma clase.
- Los Visitantes pueden acumular un estado a medida que van visitando cada elemento de la Estructura de Objetos
- Rotura de la encapsulación: el enfoque del patrón Visitor asume que la interfaz de Elemento Concreto es lo bastante potente como para que los Visitantes hagan su trabajo. Como resultado, el patrón suele obligarnos a proporcionar operaciones públicas que acceden al estado interno de un elemento, lo que puede comprometer su encapsulación.

10.7 Patrones Relacionados

- Composite: Los Visitantes pueden usarse para aplicar una operación sobre una estructura de objetos definida por el patrón Composite
- Interpreter: Se puede aplicar el patrón Visitor para llevar a cabo la interpretación.

11 Esquema de resolución de ejercicios

- Describir los módulos 2MIL aplicando el patrón pedido
- Escribir la Guía de Módulos
- Dibujar la Estructura de Uso del patrón
- Completar el Pattern

Pattern	[Nombre del Ejercicio]
Based on	[Nombre del Patrón]
Because	[Motivos de uso del Patrón]
Where	[Nombre del Módulo 1] is [Correspondencia en la Estructura de Uso]
	⋮
	[Nombre del Módulo N] is [Correspondencia en la Estructura de Uso]