Patrones de diseño

los patrones de diseño son soluciones habituales a problemas comunes
los usamos para tener diseños más flexibles, extensibles y reusables, describe un problema recurrente y su solución de manera de poder ser re-utilizada muchas veces
cada uno es como un plano que se pueden personalizar para resolver un problema. NO es código si no un concepto.
según Alexander

> *un patrón describe un problema que ocurre repetidas veces*

CLASIFICACIÓN

- según actividad: de software, de testing, etc.
- según nivel de abstracción: de análisis, de arquitectura, de diseño
- según dominio: financiero, comercio, salud, etc.
- según paradigma: web, modelo de datos, XML, etc.
- según propósito: estructurales, creacionales, de comportamiento, etc.

ELEMENTOS DE UN PATRÓN

- su nombre, que en una o dos palabras describe el problema y su solución
- descripción de cuándo aplicarlo y qué hace
- elementos que lo componen
- consecuencias de los resultados

PROTOCOLO DE CLASE= conjunto de mensajes que entienden sus instancias

PATRONES ARQUITECTURALES
los patrones arquitecturales son formas de organizar aplicaciones
Hay dos opciones

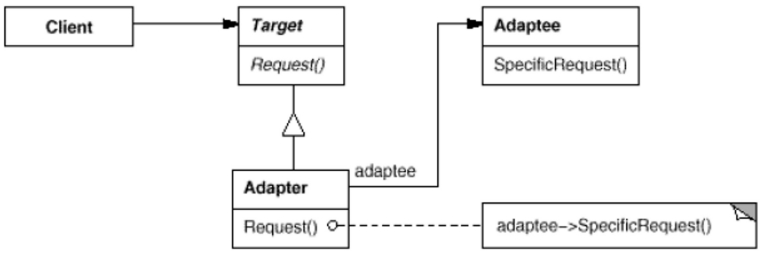
1) arquitectura en capas: aplicación dividida en capas de software. Las capas más superiores son las que más cerca están del usuario
2) arquitectura microkernel: incluye un centro del sistema y alrededor tiene el resto de la funcionalidad

PATRONES ESTRUCTURALES
los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura

adapter pattern

el adapter pattern permite que ciertas clases trabajen en conjunto cuando no podrían porque tienen interfaces incompatibles, entonces usaré adapters cuando quiera usar una clase existe y su interfaz no es compatible con la que necesite

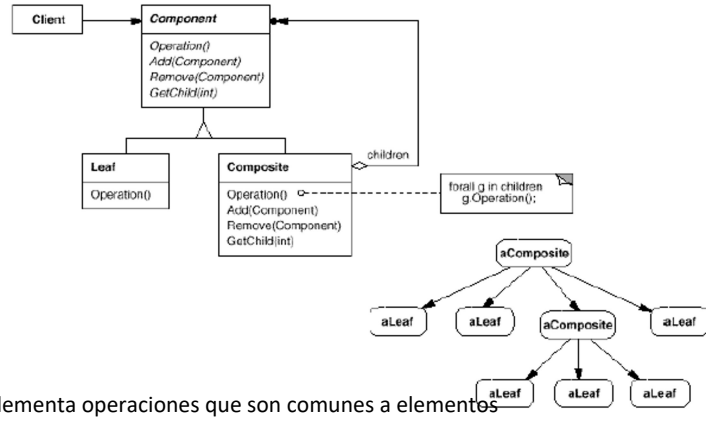
1) Client contiene la lógica de negocio existente del programa
2) Target define la interface de dominio específico que el cliente usa
3) Adaptee es alguna clase útil. El cliente no puede usar directamente esta clase porque tiene una interfaz incompatible
4) Adapter implementa la interfaz con el target mientras envuelve el objeto de la clase adaptee



composite pattern

el composite pattern es usado para representar jerarquías. Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales. Sirve si puedo representar el problema en un árbol

ejemplo: tengo un pedido que contiene cajas que a su vez contiene productos. La solución es trabajar con productos y cajas con una interfaz común que declara un método para calcular el precio total. Este método devolvería el precio si es un producto y si es una caja, recorre lo que contiene, pregunta precios y devuelve el total por caja
La ventaja es que no me preocuparía por la complejidad de la caja



1) Component declara la interfaz para los objetos de la composición e implementa operaciones que son comunes a elementos simples y complejos del árbol.
2) Leaf representa hojas, define el comportamiento de objetos primitivos en la composición. Es el elemento básico de un arbol y no tiene subelementos. Generalmente acaban realizando la mayoría del trabajo porque no tienen a quien delegárselo

3) Composite define el comportamiento para componentes con "hijos", contiene las referencias a los hijos e implementa operaciones para manejar "hijos"

wrapper/ decorator pattern

el composite pattern permite agregar funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores que contienen estas funcionalidades. La idea es como una cabeza, donde el cerebro hace sus cosas y las capas que lo protegen no se entera y así en viceversa

Tiene como objetivo agregar comportamiento a un objeto dinámicamente y en forma transparente

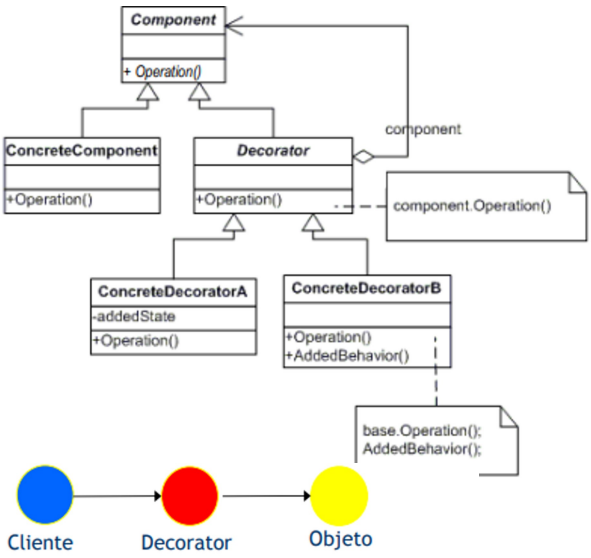
aplicabilidad: agregar responsabilidades a objetos individualmente y de forma transparente, quitar responsabilidades, ya que es impráctico subclasificar

consecuencias: mayor flexibilidad que herencia, agrego funcionalidad

implementación: misma interface entre componente y decorator

decorator vs adapter: ambos patrones "decoran" el objeto para cambiarlo, el decorator preserva la interface del objeto para el cliente, adapter convierte la interface del objeto para el cliente

decorator vs stategy: ambos permiten que un objeto cambie su funcionalidad dinámicamente. Su diferencia es que el stategy cambia el algoritmo por dentro y el decorator lo hace por fuera



permite tener mayor flexibilidad en tiempo de ejecución que una solución con herencia

se usa cuando subclasificar es impráctico

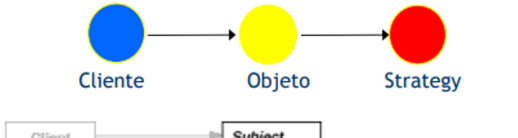
proxy pattern

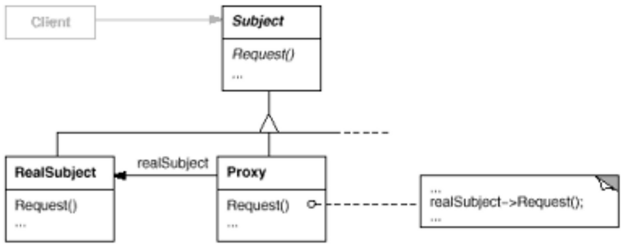
proxy pattern permite proporcionar un sustituto o marcador de posición para otro objeto. Controla el acceso al objeto original permitiendo hacer algo antes o después de que la solicitud llegue al objeto original

1) subject declara la interfaz de los servicios, el proxy debe seguir esta interfaz para poder camuflarse como objeto de servicio
2) proxy tiene un campo de referencia que apunta a un objeto de servicio. Cuando el proxy finaliza su procesamiento, pasa la solicitud del objeto al servicio
3) servicio proporciona una lógica de negocio útil

aplicaciones

- virtual proxy: demorar la construcción de un objeto hasta que sea necesario
- protection proxy: restringir el acceso a un objeto por seguridad (puede verse como un decorator)
- remote proxy: presentar un objeto remoto en el espacio de memoria local. Es la forma de implementar objetos distribuidos. Se encarga de comunicar el objeto remoto y de serializar/deserializar los mensajes y resultados





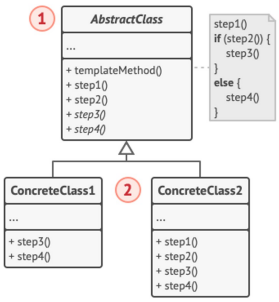
PATRONES DE COMPORTAMIENTO

tratan con algoritmos y la asignación de responsabilidades entre objetos

template method

el template method es un patrón que define el esqueleto de un algoritmo en la super clase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

es usado para aplicar variantes de un algoritmo una vez y dejar que las subclases implementen los aspectos que varían



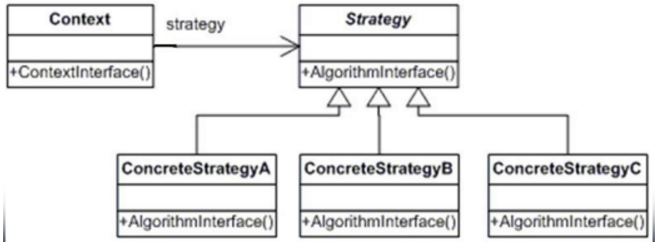
strategy pattern

el strategy method permite definir una familia de algoritmos, colocar cada uno en una clase separada y hacer sus objetos intercambiables

su intent es desacoplar un algoritmo del objeto que lo usa, cambiar el algoritmo que un objeto usa en forma dinámica, brindar flexibilidad para agregar nuevos algoritmos que lleven a cabo una función determinada

- strategy es común a todas las estrategias concretas. Declara un método que la clase contexto usa para ejecutar una estrategia
- concreteStrategy implementan distintas variaciones de un algoritmo que la clase contexto usa

como consecuencia tengo una alternativa para subclasificar contexto, desacoplar el contexto de los detalles de implementación de estrategias y eliminar condicionales



state pattern

el state pattern permite a un objeto alterar su comportamiento cuando su estado interno cambia.

sugiere que implementes nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos del estado para colocarlos en esas clases

aplicado cuando el comportamiento de un objeto depende del estado en el que se encuentre y los métodos tienen sentencias condicionales que dependen del estado

- state declara los métodos específicos del estado, deben tener sentido para todos los estados correctos porque no queremos que uno de los estados tengan métodos inútiles
- concrete state proporciona implementaciones para los métodos del estado, puedo implementar clases para evitar repartición de código

entonces, localiza el comportamiento relacionado con cada estado o usado para transiciones de estados implícitas