Resumen

Metodologías de Programación

Patrones de Diseño

Definición:

> **Un patrón de diseño describe un problema que ocurre una y otra vez, así como también su solución. De modo tal que se pueda aplicar esa solución un millón de veces sin hacer lo mismo dos veces.**

Todo patrón de diseño tiene:

. Un nombre

. Describe brevemente el problema y la solución

. El problema que intenta resolver

. Cuando aplicar el patrón. Describe el problema y su contexto.

. La solución al problema

. Elementos que constituyen la solución. Describe las clases, las relaciones entre ellas y la responsabilidad de cada una.

. Consecuencias

. Resultados de aplicar el patrón. Describe las ventajas y consecuencias de utilizarlo.

Ventajas de utilizar un patrón de diseño

. Tenemos la solución a un montón de problemas

. Aplicamos en el desarrollo del software un diseño que es conocido por la comunidad y que es fácil de mantener y de extender su funcionalidad

. Permiten ampliar el lenguaje entre los diseñadores y programadores de sistemas

. Son independientes del lenguaje de programación.

Clasificación de Patrones

Los patrones se dividen en tres categorías

. **Creacionales**: Determinan como es el proceso de creación de las instancias en un problema determinado

Estos patrones abstraen el proceso de creación de instancias. Facilitan el diseño ya que el proceso de creación es independiente al resto del sistema

Patrones de creación
Factory method
Abstract factory Builder Prototype Singleton

. **Estructurales**: Definen como es la estructura y composición de clases

Estos patrones se ocupan de como se combinan las clases para formar estructuras de objetos más grandes. Estos patrones, describen formas de componer objetos para obtener nueva funcionalidad.

Patrones estructurales
Adapter
Adapter Bridge Composite Decorator Facade Flyweight Proxy

. **De Comportamiento**: Definen el modo en que interactuan los objetos y como se reparte la responsabilidad.

Estos patrones estan relacionados con los algoritmos y las resposabilidades que tiene cada objeto. Estos patrones, tambien describen la comunicación entre los diferentes objetos.

Patrones de comportamiento
Interpreter Template method
Chain of responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Patrones

Patron de Comportamiento Strategy

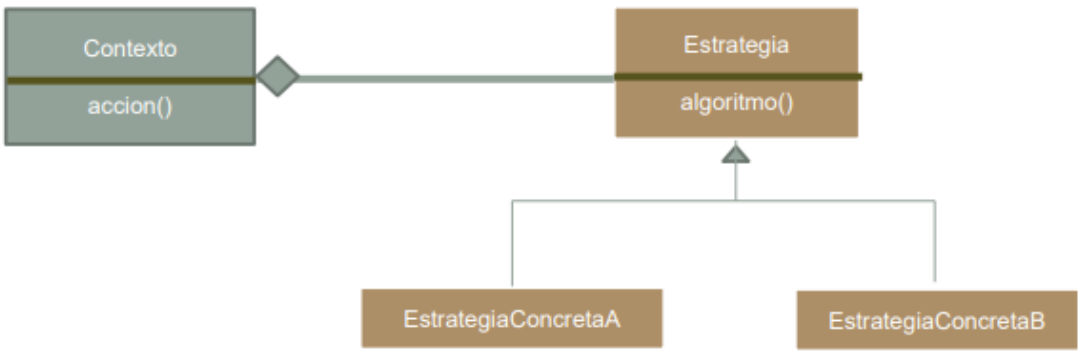
Se busca un diseño que permita agregar, eliminar e intercambiar politicas, según diferentes situaciones.

Proposito: Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varie independientemente de los clientes que lo usan.

Aplicabilidad: Se utiliza cuando

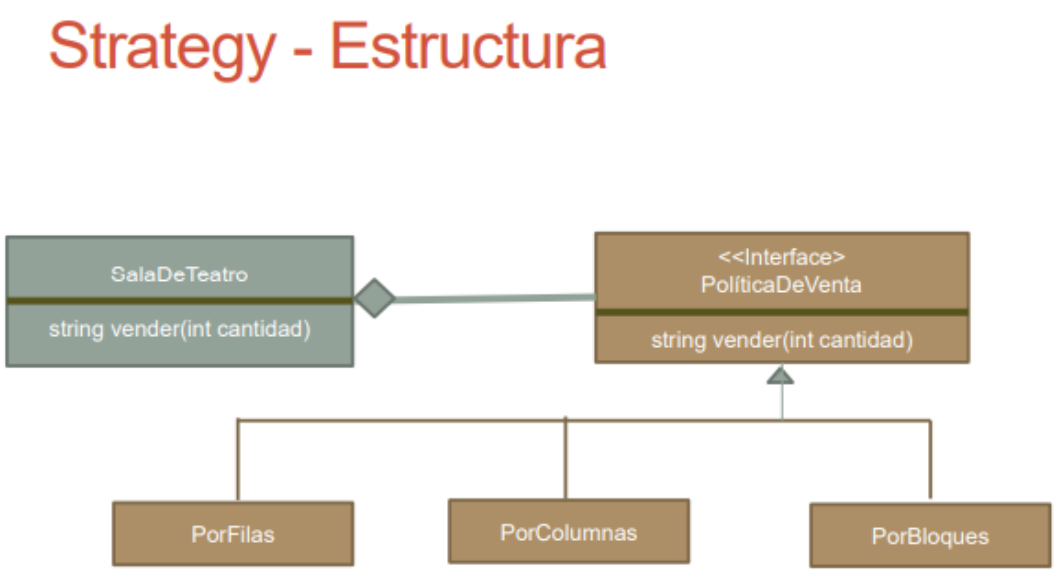
- Muchas clases relacionadas difieren solo en su comportamiento
- Se desee configurar una clase con un determinado comportamiento entre muchos posibles.
- Se necesitan muchas variantes de un mismo algoritmo

Strategy - Estructura



El contexto tiene una referencia de tipo estrategia, y la estrategia es padre de las estrategias concretas las cuales heredan de la estrategia padre. Para crear otra estrategia es tan facil como crear otra estrategia concreta hacerla heredar de estrategia. Y instanciarla en la referencia del contexto, donde en se delegan los metodos a la estrategia.

Ejemplo:



Ventajas:

. Familia de algoritmos relacionados. La herencia puede ayudar a factorizar estos algoritmos

. Las estrategias eliminan las sentencias condicionales

. Las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento

. Resulta muy simple agregar nuevas estrategias

Patron de comportamiento Iterator

Propósito: Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representacion interna.

Se busca un diseño que permita que una impresora pueda "recorrer" todas las paginas de un documento independientemente de como esten almacenadas.

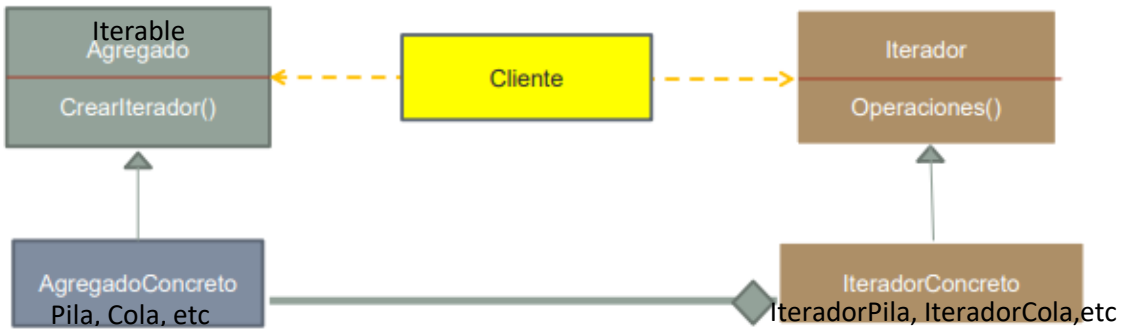
Aplicabilidad: Se utiliza cuando

. Necesite acceder al contenido de un objeto agregado sin exponer su representacion interna

. Necesite establecer varios recorridos sobre objetos agregados

. Necesite proporcionar una interface uniforme para recorrer diferentes estructuras agregadas

Iterator - Estructura



Ventajas:

. Permite variaciones en el recorrido de un agregado. Los agregados complejos pueden recorrerse de diferentes maneras.

. Los iteradores simplifican la interfaz del agregado. La interfaz de recorrido elimina la necesidad de tener una interfaz similar en el agregado.

. Es posible hacer más de un recorrido a la vez. Como el estado del recorrido lo tiene el iterador, es posible tener más de un iterador sobre la misma colección al mismo tiempo.

Patrón creacional Factory Method

Buscamos un mecanismo donde se encapsule la creación de los objetos, de modo tal que el cliente solo pida la creación de ellos, sin especificar su clase.

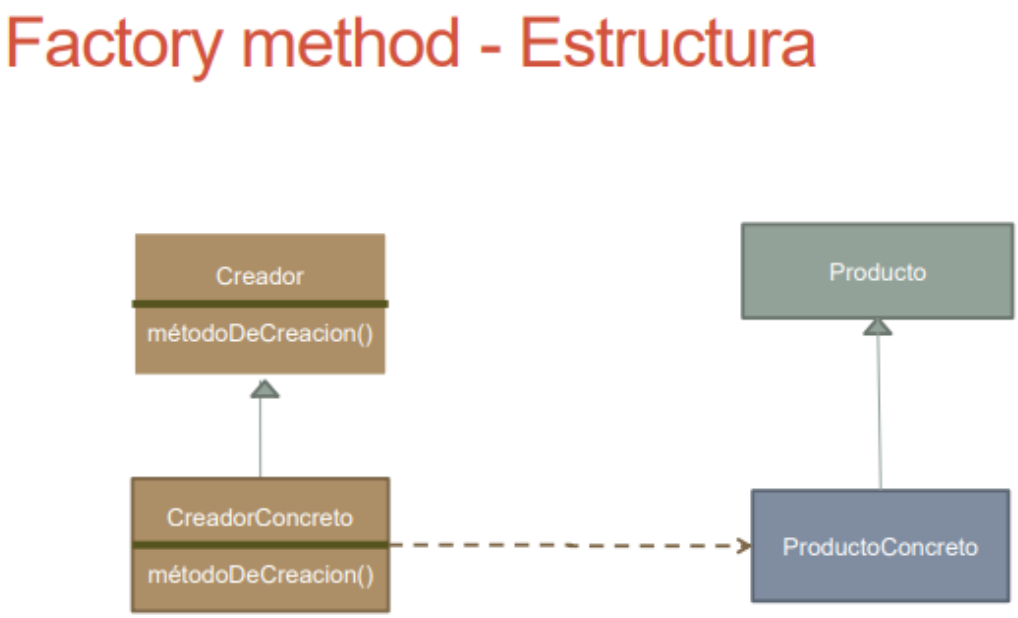
Propósito: Define una interfaz para crear un objeto, pero dejan que sea las subclases quienes decida qué clase de objetos instanciar.

Aplicabilidad: Se utiliza cuando

. Una clase no puede prever la clase de objetos que debe crear

. Una clase quiere que sean sus subclases quienes especifiquen los objetos que esta crea

. Las clases delegan la responsabilidad en una de entre varias clases auxiliares y queremos localizar que subclase auxiliar concreta es en la que delega.



Ventajas:

. Este patrón elimina la necesidad de ligar clases específicas de la aplicación a nuestro código

. El código solo trata con la interfaz del Producto

. Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente

Patrón de comportamiento Observer

Sería interesante contar con un mecanismo donde un objeto "anuncie" que un cliente llego y todos los interesados "escuchen" de la llegada del cliente, sin necesidad que la terminal le avise a cada uno de los interesados.

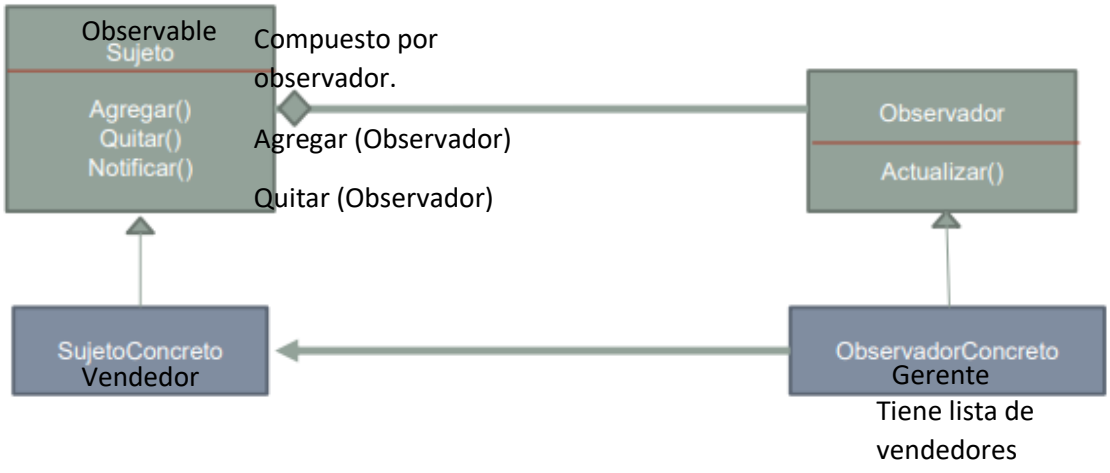
Propósito: Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependan de él.

Aplicabilidad: Se utiliza cuando

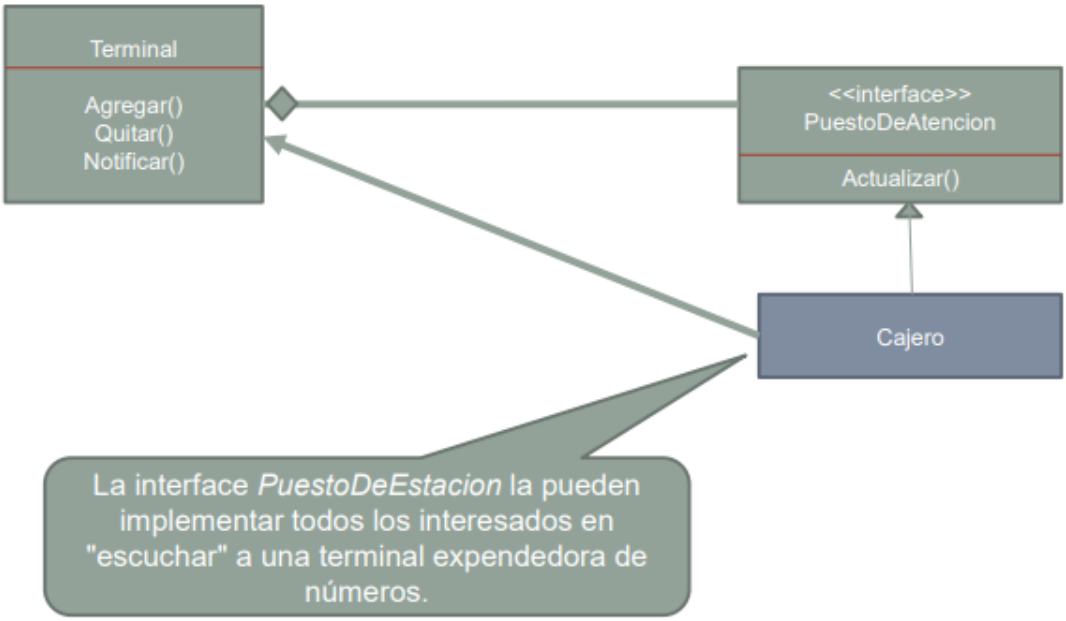
. Una abstracción tiene dos aspectos y uno depende del otro

. Un cambio en objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse

. Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quienes son dichos objetos



Observer - Estructura



Ventajas:

. Permite modificar los sujetos y los observadores de forma independiente. Permite añadir observadores sin modificar el sujeto

. Acoplamiento abstracto entre sujeto y observador. El sujeto no conoce ninguna clase concreta de observador

. Al sujeto no le interesa saber cuántos observadores hay

Patrón estructural Adapter

¿Es posible contar con un mecanismo que permita usar más de una interface como si fuera la misma?

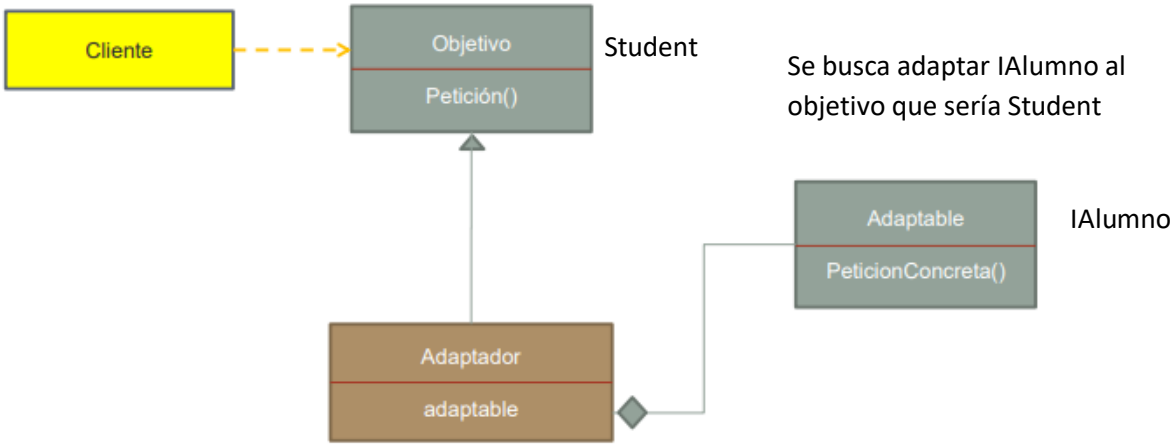
Propósito: Convierte la interface de una clase en otra interface que es la que esperan los clientes. Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles.

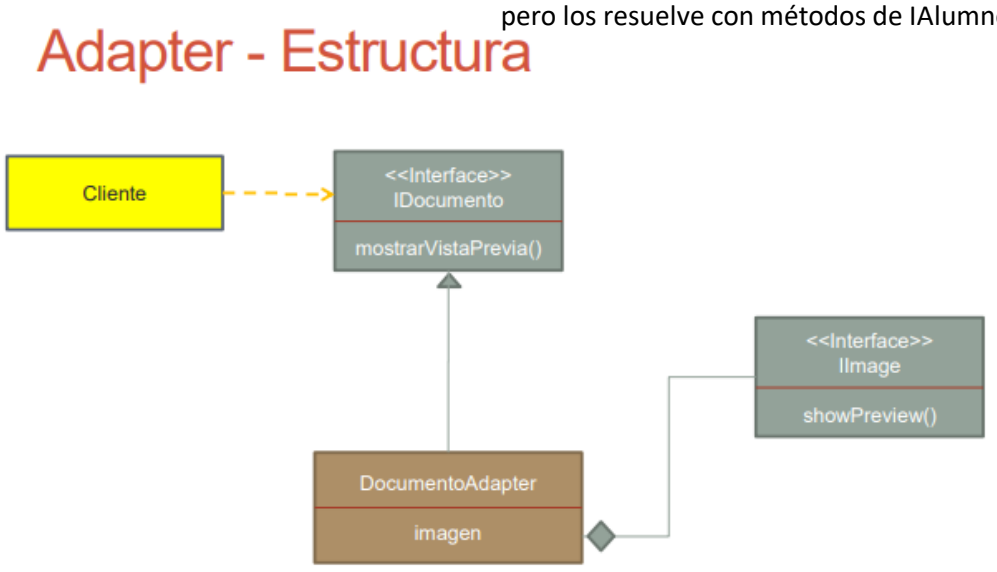
Aplicabilidad:

. Se quiere usar una clase existente y su interface no concuerda con la que necesita

. Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido provistas

. Es necesario usar varias subclases existentes, pero no resulta practico adaptar su interface heredando de cada una de ellas.





Ventajas:

. Adopta una clase Adaptable a Objetivo, pero se refiere únicamente a una clase Adaptable Concreta

. Permite que Adaptador redefina parte del comportamiento de Adaptable, por ser Adaptador una subclase Adaptable

. Introduce un solo objeto para obtener el objeto adaptado

. Permite que un Adaptador funcione con muchos adaptables (si este es una superclase)

Patrón estructural Decorator

Buscamos un diseño que permita agregar cualquier número de características o “decorados” a un objeto, sin alterar el propio objeto y que sea transparente para el cliente y que pueda tratar con el objeto simple o “decorado” de manera independiente.

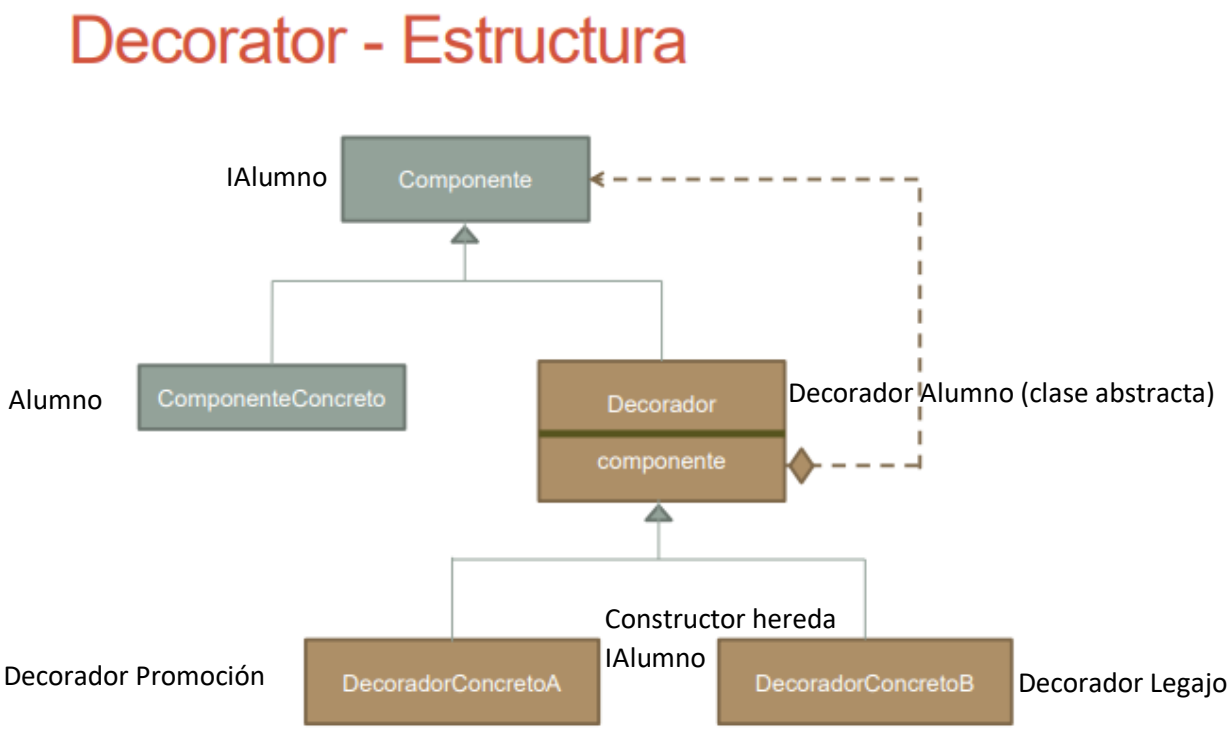
Propósito: Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

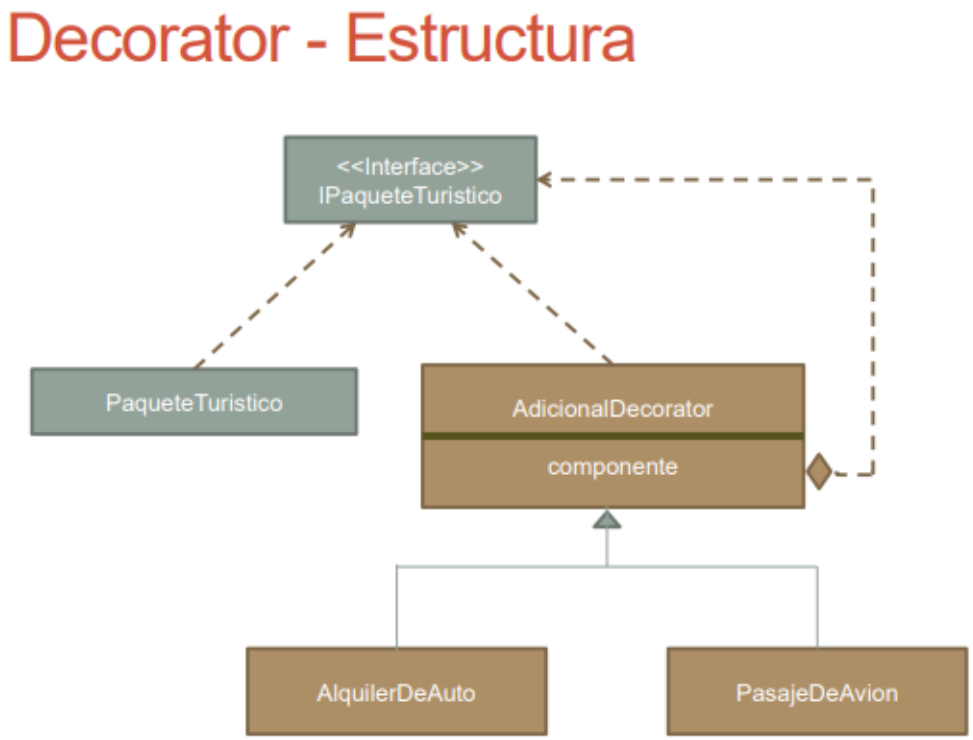
Aplicabilidad: Se utiliza cuando

. Quiera añadir responsabilidades individuales de forma dinámica y transparente.

. Sea necesario retirar responsabilidades

. La extensión por herencia no es viable





Ventajas:

. Más flexibilidad que la herencia estática. El patrón Decorator proporciona una manera más flexible de añadir responsabilidades a los objetos. Es posible añadir y quitar responsabilidades en tiempo de ejecución.

. Evita clases cargadas de funciones en las partes superiores de las jerarquías

. Muchos objetos pequeños. El uso del patrón Decorator da como resultado sistemas formados por muchos objetos pequeños muy parecidos entre sí.