PUCMM

Pontificia Universidad Católica
Madre y Maestra

Pontificia Universidad Católica Madre y Maestra Campus Santo Tomás de Aquino

Facultad de Ciencias e Ingeniería

CSD-1770-1843 - Programación I

Alicia Santos

Asignación final

Dayán Paniagua Robles 2019-5025

Freddy Antonio Cruz Valerio 2019-5033

Miércoles 26 de julio de 2021

Patrón Decorador.

El patrón decorador permite modificar un objeto de manera dinámica. Este se puede usar cuando queremos agregarle funcionalidades a una clase que normalmente lo haríamos con herencia agregando subclases, pero estas funcionalidades las quieres agregar al momento de ejecución del programa. Lo que hace a este patrón una mejor opción que las herencias a la hora de agregar funcionalidades a un objeto, es que este patrón lo hace de manera flexible. Por ejemplo, si tenemos un objeto personaje y queremos agregar cosas como ropa, gorra, zapatos, entre otros; pero, no queremos que a todos se les agregue estos accesorios. Tendríamos que hacer herencia para cada combinación de un personaje y al final terminaríamos con una gran cantidad de subclases. En cambio, con este patrón puedes agregar estas funcionalidades usando clases simples y en vez de reescribir tu antiguo código, lo que haces es extenderlo.

Decorador es un patrón de tipo estructural, los cuales son patrones que se ocupan de cómo las clases y objetos se pueden componer, para formas largas estructuras de datos. Los patrones estructurales simplifican la estructuración de los objetos identificando las relaciones que esto tienen. En este caso, el patrón decorador hace más simple la estructura al eliminar el uso de herencia. Permitiéndonos agregarle funcionalidades a una instancia de un objeto sin necesidad de incluir esta funcionalidad en dicho objeto.

La estructura de este patrón está formada por el componente, los componentes concretos, decorador y los decoradores concretos. El componente es una clase abstracta y base de la estructura, que define la interfaz para los objetos los cuales tendrán habilidades agregadas de manera dinámica. Luego, los componentes concretos que son los objetos a los cuales las habilidades mencionadas anteriormente son agregadas. Después, el decorador que mantiene una referencia al componente y define una interfaz que será la que conformará al componente. Por último, los objetos decoradores concretos que son las clases las cuales agregarán las habilidades a las clases de componentes concretos.

En conclusión, este patrón de diseño lo podemos usar para agregar comportamiento a objetos sin utilizar herencia o sin agregar estos cambios en el objeto en sí. Una aplicación de este patrón podría ser en una plataforma por suscripción como Platzi, para agregar comportamiento a las cuentas. Es decir, si estos ofrecen un período de prueba de la plataforma, con el patrón decorador podemos agregar esta habilidad a la cuenta del usuario. No utilizaríamos herencia en estos casos, pues el período de prueba gratis no es tipo de cuenta, es una simple funcionalidad.

Entonces, una vez se acabe el período de prueba de la cuenta y el usuario decida no quedar en la plataforma, solo tendríamos que remover el decorador de prueba de la cuenta.

Patrón Compuesto.

El patrón de diseño compuesto nos permite acceder a un atributo de un objeto y tratar a este como un objeto. Es decir, el patrón compuesto nos permite tratar objetos específicos de un todo, sin dañar la estructura general. Por ejemplo, podemos pensar en una tienda de discos musicales los cuales están organizados por géneros musicales, pero dentro de estos generales musicales tenemos los distintos discos de ese género que cuenta con diferentes atributos. Entonces, nuestro objeto principal en este caso es la tienda de disco, los atributos serían cada género musical, ya uno vez dentro del género tenemos los discos y estos discos cuenta con atributos como el nombre y la banda que lo creó. Con el patrón compuesto, podemos acceder a las canciones como un objeto, pero estas canciones siguen perteneciendo a un todo que es la tienda de discos.

Cuando nos adentramos en la estructura del patrón compuesto, este está formado por un cliente, el componente que es manejado por el cliente, un compuesto y una hoja. El componente es una clase abstracta y es la base de la jerarquía, este tiene algunas operaciones generales como añadir hijos a la composición o eliminarlos. Luego tenemos el compuesto, que es el que implementa las operaciones de manipulación de los hijos y este relaciona con el objeto componente. Es importante resaltar, que un elemento del objeto compuesto puede ser otro objeto o una hoja. Por último, tenemos la hoja que un elemento el cual no se relaciona con el objeto componente de manera que pueda ser este un componente u hoja a la vez; sino más bien, cuando caemos en el objeto hoja se termina el árbol.

Nos referimos a árbol, puesto que este patrón esta diseñado como tal. Ya que, el árbol vendría siendo el componente (clase base y abstracta) y a este árbol se le puede agregar ramas u hojas. Las ramas vendrían siendo el objeto compuesto que se relaciona con el árbol de forma que una rama puede tener diferentes hojas u otra rama. A esto nos referíamos, cuando dijimos anteriormente que un compuesto puede ser una hoja o un compuesto. Al final, tendremos un objeto que es el conjunto o el todo, este vendría siendo el árbol y podemos tratar las diferentes ramas como objetos también.

El patrón compuesto podemos aplicarlo para muchos sistemas utilizados hoy en día. Como lo puede ser un programa para un

restaurante, el cual tiene un objeto con todos los menús disponibles. El objeto todos los menús podemos agregarle o eliminarle menús. Dentro de los menús de este objeto, podemos tener el menú para el desayuno, comida y cena. Pero dentro de estos menús, podemos tener menús adicionales como el menú de entrada o el menú de postres, o podemos ser que un menú este formado solo por platos directamente sin necesidad de tener otro menú. Entonces, en este caso el componente sería el objeto de todos los menús, el compuesto serían un menú y la hoja un plato, pues en la hoja ya no derivaríamos más de esta.

Así pues, podemos clasificar el patrón de diseño compuestos de tipo estructural. Pues, la base de este son las clases y objetos que va haciendo particiones de estos. Podemos aplicar este patrón para cualquiera jerarquía parte todo, las cuales son aquellas jerarquías que podemos dividirlas en pequeños componentes. Esto quiere decir, que poder estructurar los datos de tal manera que represente los distintos componentes de un conjunto, como un todo.

Patrón método de factoría.

El patrón método de factoría es un arquetipo creacional que define una interfaz para originar un objeto, sin embargo, no especifica qué objetos instanciarán las implementaciones unipolares de esa interfaz. Lo que esto significa es que, al utilizar este patrón, se puede definir ciertos métodos y propiedades de objeto que serán comunes a todos los objetos creados usando el Método de Fábrica, pero deje que los Métodos de Fábrica individuales definan qué asuntos específicos instanciarán.

Ahora que ya sabemos lo que es el patrón de diseño método de factoría, pasemos a ver de qué manera este funciona y las utilidades que se le puede proporcionar a este. Entonces para entender el funcionamiento de este patrón de diseño debemos de dirigirnos a su diagrama de clase (el cual estará en el apartado de anexos). Bueno, luego de visto este nos percatamos de que, en primera instancia, el Producto define las interfaces de los objetos que creará el método de fábrica.

Luego de esto, los objetos producto concreto implementan la interfaz Producto. Seguido de esto, el creador declara el método de fábrica, que devuelve un objeto de tipo producto. El creador también puede definir una implementación predeterminada del método de fábrica. Los objetos CreadorConcreto anula el método de fábrica para devolver una instancia de un producto concreto.

Como ejemplo para este patrón de diseño utilizaremos el escenario del desarrollo de un video juego, como en todo videojuego los enemigos son una parte muy importante, ya que le da dinamismo y variedad, o incluso ir aumentando la dificultad de los enemigos a medida que se avanza en el juego, por lo tanto, en nuestro juego acabamos teniendo un conjunto de clases (enemigos), los cueles son ***Boo***, ***Koopa*** y ***Goomba***, todas estas clases heredarían de una misma clase padre llamada entidad. Estás clase entidad definiría todo aquello que tenga logia e interacción en el juego. Esto nos ayudará a aplicar polimorfismo y de esta manera si en algún momento queremos agregar más entidades lo que haremos es heredar a la clase sucesora llamada entidad.

Siguiendo con el desarrollo de nuestro juego, queremos que el jugador no se memorice los sitios en los que aparecen cada tipo de enemigo, debido a que esto terminaría siendo algo monótono y predecible. Para poder sorprender al jugador e instanciar enemigos de manera aleatoria implementamos una función. Pero el problema surge cuando los usuarios quieren que los niveles sean más difíciles, quieren que el juego se un mayor reto. Tenemos la opción de modificar nuestro código y aumentar la probabilidad de que aparezcan enemigos más fuertes, pero, de esta manera se verían afectados todos los jugadores por igual y no queremos esto. Para este caso en concreto funcionaria perfecto nuestro patrón de diseño método de factoría. En primera instancia crearíamos un método base llamado fábrica de enemigos, a los cuales le podemos instanciar un creador de enemigos aleatorio y un creador de enemigos de mayor dificultad. De esta manera para crear un nuevo patrón de creación de enemigos bastaría con heredar el método ya creado creador de enemigos. En conclusión, el tipo de enemigo que el usuario desee o la dificultad si se ve de esta manera, lo podrá tener solamente pasando los parámetros de qué tipo de fábrica quiere y lo más importante es que todo es en tiempo de ejecución.

Patrón observador.

El patrón de diseño observador es un patrón de diseño que define una sujeción del tipo uno a muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, este comunica el cambio a todos los demás dependientes de este.

El objetivo principal que posee este patrón de diseño es el de poder tener constancia de los estados que poseen los diferentes objetos dentro de un constructor, que a su vez esta albergado dentro de una clase. Siendo así sus mayores utilidades en aplicaciones de mensajería que necesiten tener constantemente el estado de los diferentes chats, ya que de esta manera se podrá dar una alerta al usuario cuando se presente cualquier

tipo de cambio. Ya sea que se reciba un mensa o llamada, todo tipo de alteración del estado será vista y notificada al usuario, si este así lo desease.

Para entender más a profundidad el funcionamiento de este patrón de diseño pongamos dos ejemplos, el primero tratara el problema que se le presenta a alguien que posee una canal de YouTube. Este usuario no está conforme con las visitas que está teniendo su contenido y dice que esto se debe a que los usuarios que se suscriben a su canal no reciben ningún tipo de aviso o alerta.

Para solucionar dicha problemática el dueño del canal de YouTube se dispone a crear una aplicación de alerta mediante las cuales en el momento en el que usuario se suscriba recibirá notificaciones personalizadas cada vez que este canal publique un nuevo video o esté haciendo algún tipo de streaming o actividad alternativa.

Dicho lo cual se procede a implementar el patrón de diseño observador el cual trabajara con los parámetros pasados por el programador los cuales serán la feed de videos actuales y le dirá a su programa que una vez sea posteado un video nuevo diferente a los ya publicados que genere una cadena de strig que diga textualmente ¨Se ha publicado un nuevo video del canal …¨, Luego de ya tener el mensaje almacenado en una cadena de caracteres procederemos a enviarle dicho mensaje a todo los usuarios que estén dentro de la lista de los suscriptores de nuestro canal de YouTube.

Para el siguiente ejemplo imaginemos que tenemos una aplicación como Gmail que se le está presentando el problema de que los usuarios no saben cuándo envían un correo de manera efectiva. Por lo cual, procedemos a implementar nuestro patrón de diseño observador para solucionarnos este problema. Lo primero que haremos será crear un apartado para los correos enviados. Pero, dado que no todos los usuarios se percatarán siempre de este apartado dentro del mismo implementaremos el patrón de diseño observador, el cual cada vez que la feed de nuestra bandeja de correos enviados se actualizada, es decir, haya un correo nuevo enviando, pues esta nos envíe una réplica a nuestro correo diciéndonos que nuestro mensaje se envió de manera éxito.

Patrón Iterador.

Este último patrón de diseño es de tipo conductual, es decir, que se encarga de algoritmos y asignar responsabilidades entre los objetos. Este patrón es usado como una forma para acceder a conjuntos de elementos de un objeto de manera secuencial, pero

sin acceder al objeto completo en sí. Por ejemplo, tenemos un objeto que puede ser un juego y este tiene diferentes métodos, entonces con este patrón podemos iterar (recorrer) el conjunto de enemigos, pero sin afectar el comportamiento del objeto juego.

Entonces, la intención de este patrón es proporcionar un objeto el cual recorre un conjunto de una estructura, separando estos de la implementación de la estructura en su objeto. Entonces, el conjunto recorrido pasa a un objeto diferente del objeto donde se recorrió este objeto, el objeto al que pasan es llamada iterador. El objeto iterador guarda todos los detalles de los elementos recorridos del conjunto, como la posición y cuántos elementos faltan hasta acabar el recorrido del conjunto.

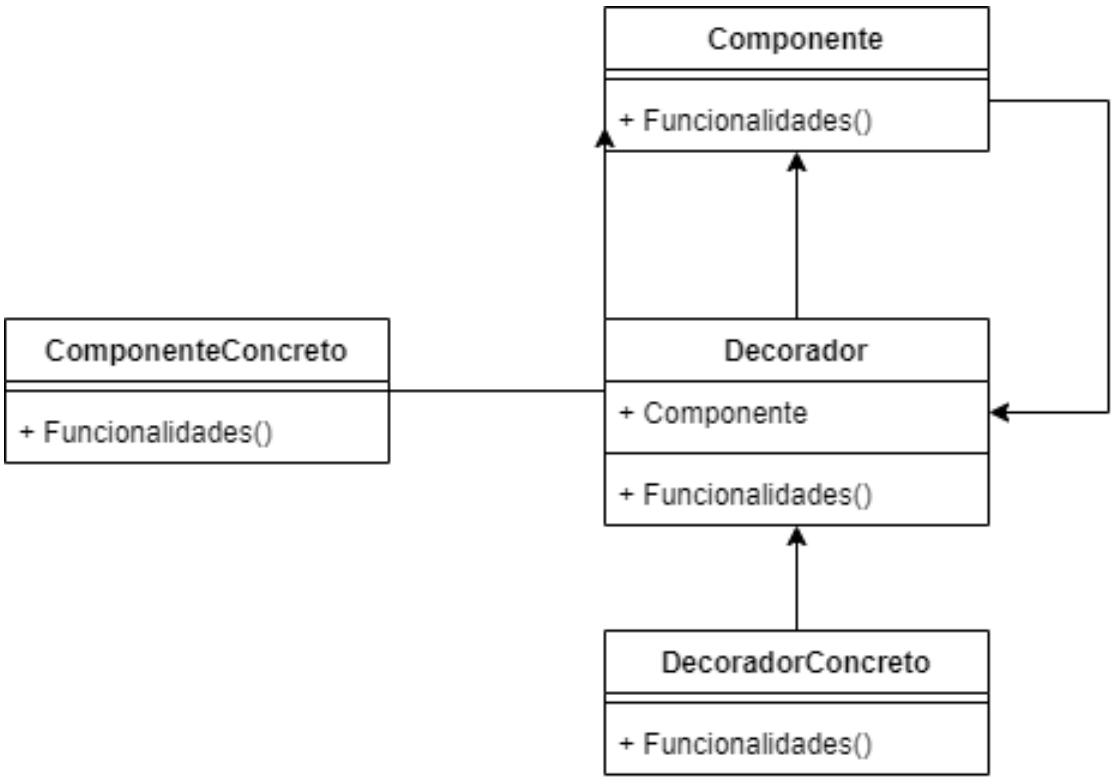
La estructura de este patrón está formada por el cliente, iterador, el iterador concreto, colección y colección concreta. El iterador es un objeto abstracto que declara una interfaz de ciertas operaciones necesarias para recorrer un conjunto, como los es el obtener el siguiente elemente, la posición actual y reiniciar la iteración. Luego, está el iterador concreto que implementa algoritmos específicos para recorrer un conjunto y debe de mantenerse dándole seguimiento a la posición actual de su recorrido dentro del conjunto. Después, la colección o conjunto declara métodos para obtener iteradores que sean compatibles con el conjunto que se recorrerá. El tipo de datos que retornarán estos métodos deben ser declarados en la interfaz del iterador, de esta manera se puede retornar diferentes tipos de iteradores para conjunto concretos. Por último, la colección o conjunto concretos implementa la interfaz de la creación del iterador y devuelve un iterador concreto para ese conjunto concreto.

Dentro de las ventajas del patrón iterador, la más importante es que la permite al cliente usar métodos simples para recorrer un conjunto, estos métodos de navegación pueden ser como siguiente, anterior y último. El cliente navega de manera simple, sin necesidad de conocer la estructura de datos de ese objeto ni ningunas otras cosas que pueden resultar complejas para este. Esto hace que estructuras de datos bastantes complejas sean accesibles para los clientes.

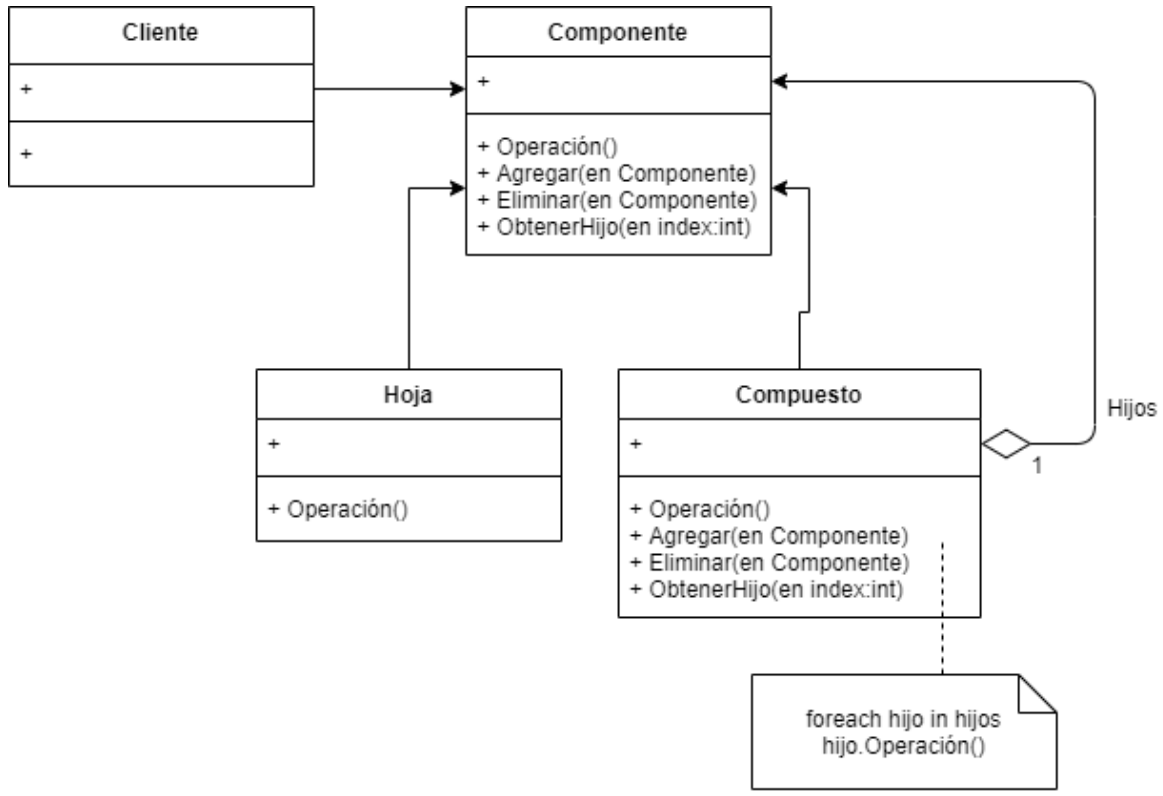
En conclusión, el patrón de diseño iterador es aplicado para aquellas estructuras de datos complejas que se requiere recorrer un conjunto de datos. Una de estas puede ser los perfiles de las redes sociales, que podamos recorrerlos con facilidad. De igual forma, las notificaciones que nos permita recorrerlas sin adentrarnos en su estructura. También, podríamos usarlo en algo más simple como una aplicación de lista de cosas que hacer.

Anexos

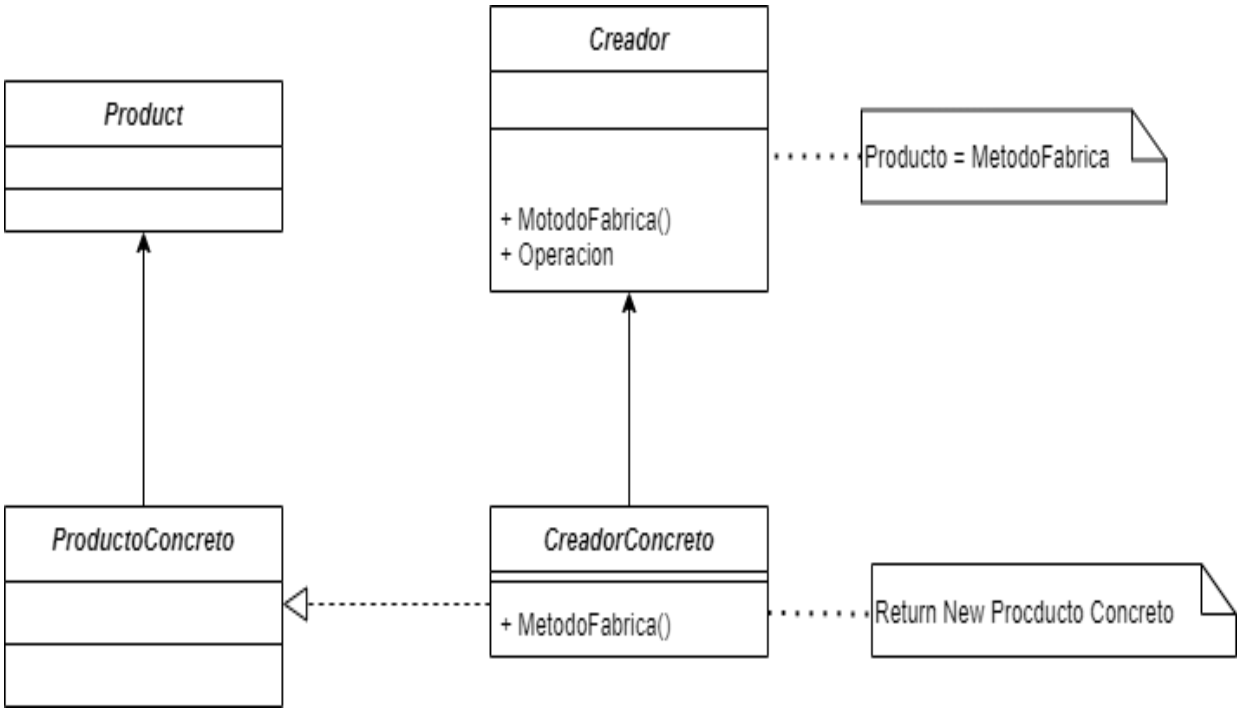
UML patrón decorador:



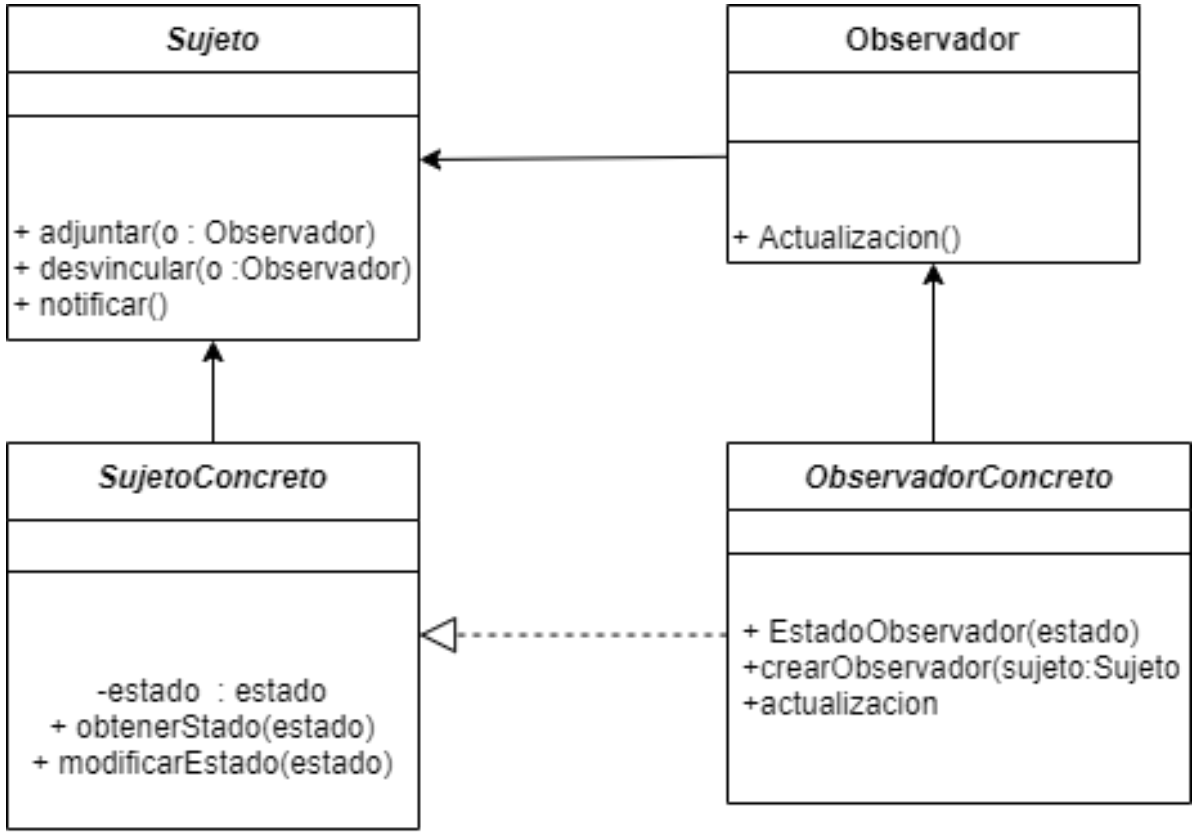
UML patrón compuesto:



UML patrón método de factoría:



UML patrón observador:



UML patrón iterador: