

PATRONES

CREACIONALES

ABSTRACT ① ✓

BUILDER ② ✓

FACTORY METHOD ③ ✓

PROTOTYPE ④ ✓

SINGLETON ⑤ ✓

ESTRUCTURALES

ADAPTER ① ✓

BRIDGE ② ✓

COMPOSITE ③ ✓

DECORATOR ④ ✓

FAÇADE ⑤ ✓

Flyweight ⑥ ✓

Proxy ⑦ ✓

DE COMPORTAMIENTO

CHAIN OF RESPONSABILITY ①

COMMAND ②

INTERPRETER ③

ITERATOR ④ ✓

MEDIATOR ⑤

MEMENTO ⑥

OBSERVER ⑦ ✓

STATE ⑧ ✓

STRATEGY ⑨ ✓

TEMPLATE METHOD ⑩

VISITOR ⑪

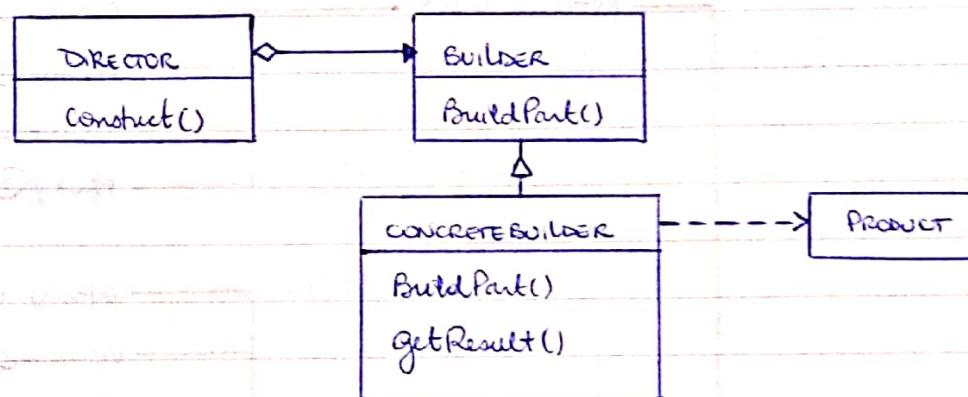
Patrón (según Christopher Alexander)

Un patrón de diseño describe un problema que ocurre una y otra vez en un contexto, y describe el núcleo de la solución a ese problema, tal que puede ser utilizada infinitamente sin tener necesariamente lo mismo que de lo mismo manera.

Patrones Creadionales ⇒ corresponden a patrones de diseño de software que solucionan problemas de creación de instancias. Nos ayudan a encapsular y abstractar dicha creación.

② **Builder** ⇒ es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto). El objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas e interfaces comunes de la clase *Abstract Builder*.

- ESTRUCTURA



- **Builder** → clase abstracta para crear productos
- **Concrete Builder** → implementación del Builder
→ construye y reúne las partes para constituir productos
- **Director** → construye un objeto usando Builder
- **Producto** → objeto complejo bajo construcción

• INTENCIÓN : separa la construcción de un objeto complejo, de su representación; tal que el mismo proceso de construcción puede crear diferentes representaciones.

• MOTIVACIÓN : se utiliza cuando tiene que construir un objeto en forma independiente de sus partes y de cómo se ensamblan. Cuando el resultado de la construcción puede tener diferentes representaciones.

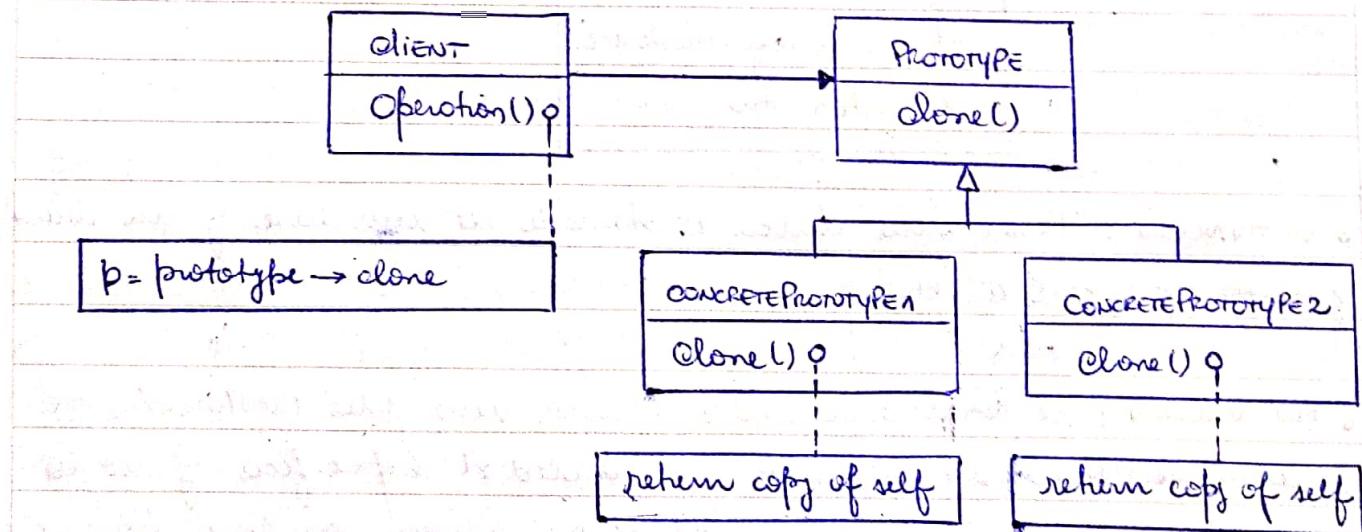
NOTA

Se resuelve con uno o más constructores (y subclases concretas) que se encargan de las diferentes partes del objeto.

- Ejemplos: Constructor de paquetes turísticos, en el que un paquete puede tener un destino, hotel, hospedaje y alimento. Podemos tener diferentes constructores para paquetes turísticos.

④ Prototype → tiene como finalidad crear nuevos objetos clonando una instancia creada previamente. Este patrón especifica la clase de objetos a crear mediante la clonación de un prototipo que es una instancia ya creada. La clase de los objetos que serán de prototipo, deberán incluir en su interfaz la función de solicitar una copia, que será desarrollada luego por las clases concretas de prototipos.

• ESTRUCTURA



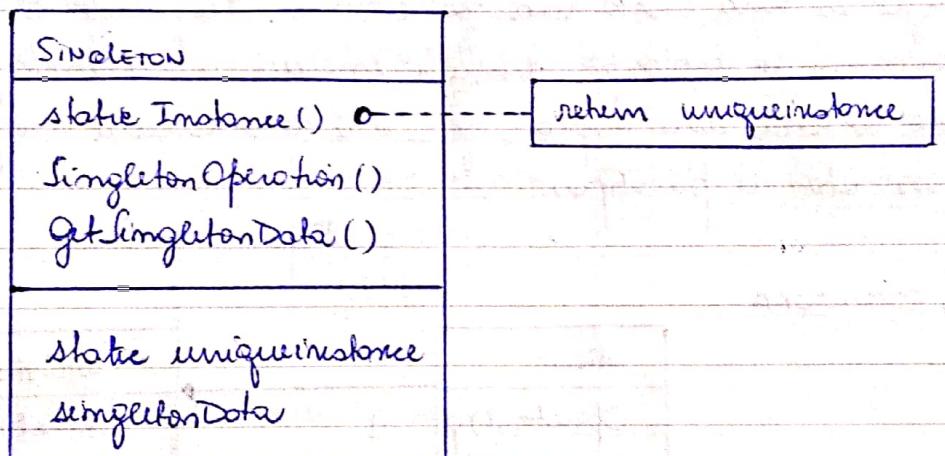
- Cliente → es el encargado de solicitar la ejecución de los nuevos objetos a partir de los prototipos.
- Prototipo concreto → posee características concretas que serán reproducidas para nuevos objetos e implementa una operación para clonarse.
- Prototipo → define una interfaz para donarse, a lo que accede el cliente.

• INTENCIÓN: Genero una instancia prototípico y sus objetos en base a ese prototípico (clon)

• MOTIVACIÓN: se utiliza cuando queremos instancias clones de un objeto existente. Para escribir fábricas de objetos similares, se resuelve con un método clone() en el objeto y/o en sus subclases, que retorne una copia del objeto (tal cual, o con los valores copiados que quiera).

⑤ SINGLETTON: permite restringir la creación de objetos pertenecientes a una clase o rollo de un tipo a un único objeto.

• ESTRUCTURA



• INTENCIÓN: Tener una única instancia de una clase y un único punto de acceso a ella.

• MOTIVACIÓN: se utiliza cuando queremos una sola instancia de una clase. Cuando necesito funcionalidad específica y no lo puedo resolver con clase abstractas o métodos de clase porque necesito instancias que guarden estado.

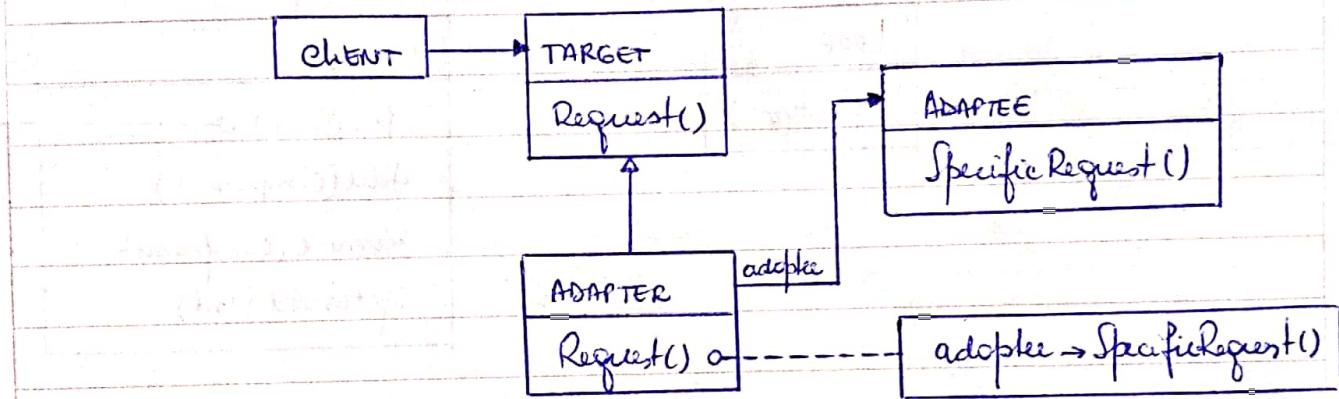
Se resuelve con un método que devuelva la instancia creada, y si no está creada, que la cree, la almacene como variable de clase y la devuelva.

NOTA

Patrones Estructurales ⇒ son los patrones de diseño de software que solucionan problemas de composición (agregación) de clases y objetos

① ADAPTER ⇒ el patrón adaptador se utiliza para transformar una interfaz en otra, de tal modo que una clase que no puede utilizar la primera haga uso de ella a través de la 2^a.

• ESTRUCTURA



• INTENCIÓN: convertir la interfaz de una clase, en otra interfaz que el cliente espera.

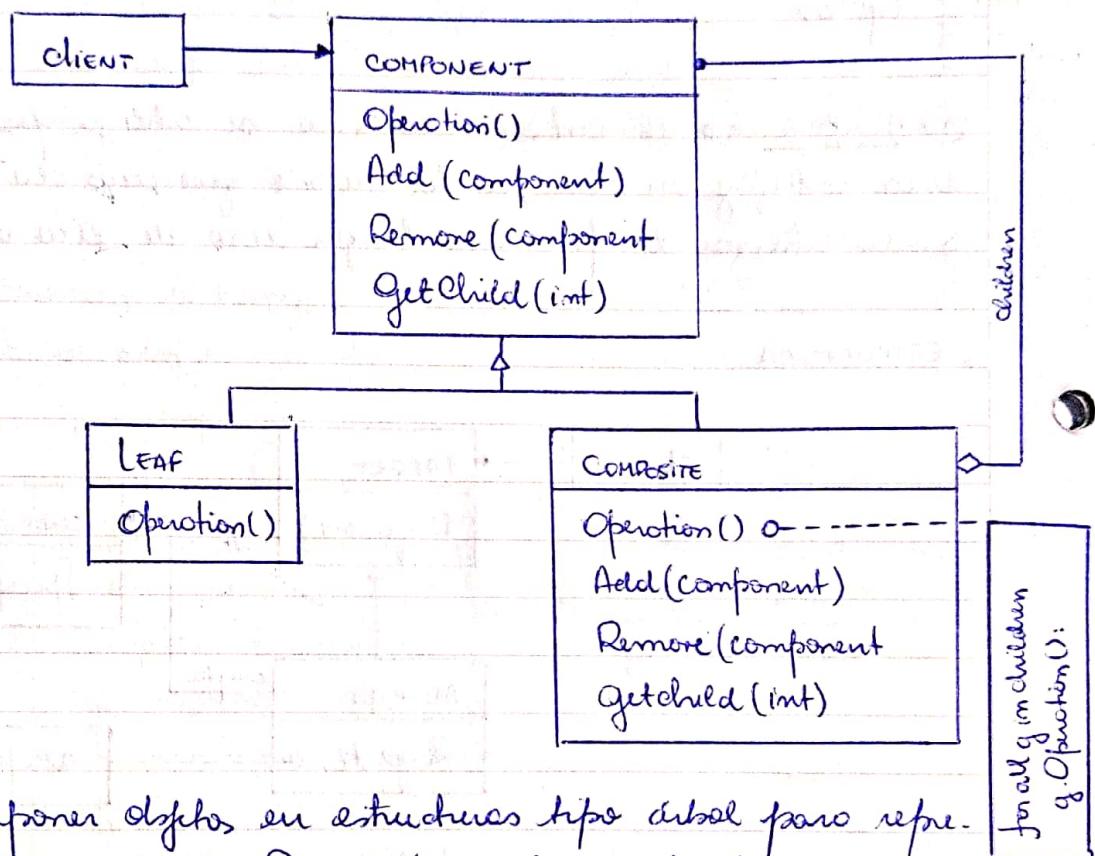
• MOTIVACIÓN: se utiliza cuando una clase que no puedes modificar y necesitas conectarla con otra que envía mensajes que está ente de periodo de otra forma.

Se resuelve con una herencia (hago subclase del adaptado) o con composición (meto el objeto dentro del adaptador).

• Ejemplo: un software de informática que abre documentos ".docs". Si recibe un "doc" lo adopta para poder abrirla.

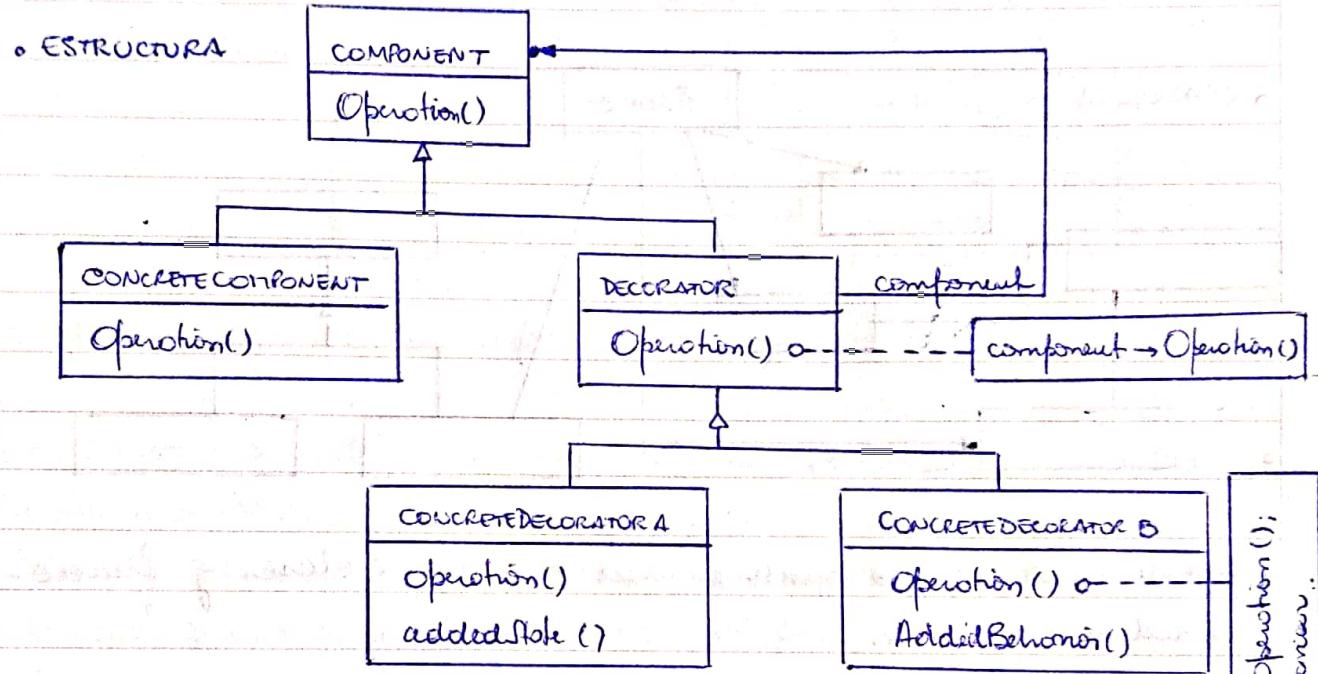
③ COMPOSITE ⇒ el patrón composite sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol.

- ESTRUCTURA :



- INTENCIÓN : componer objetos en estructuras tipo árbol para representar un objeto compuesto. Permite tratar los compuestos como a los simples.
- MOTIVACIÓN : se utiliza cuando tengo jerarquías de objetos compuestos. Cuando quiero tratar los objetos compuestos igual que a los simples, lo resuelvo con una jerarquía tipo árbol de componentes, con hojas y nodos que refieren a otros componentes.
- EJEMPLO : combo almuerzo = hamburguesa + gaseosa + papas

④ DECORATOR: este patrón responde a la necesidad de añadir dinámicamente funcionalidad a un objeto. Esto nos permite no tener que crear sucesivas clases que hereden de la 1^a implementando nueva funcionalidad, sino otras que la implementan y se asocian a la primera.

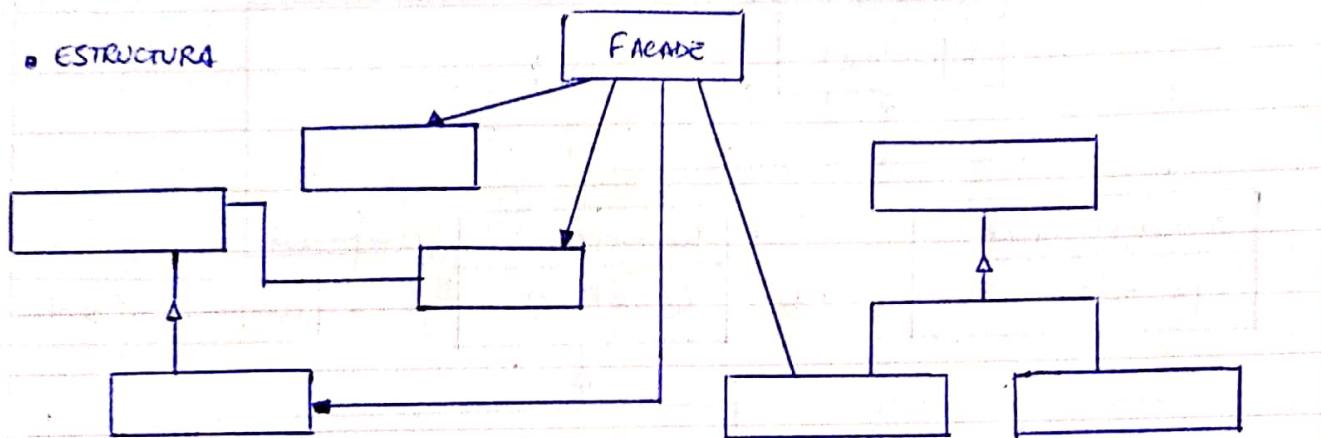


- INTENCIÓN: se utiliza cuando se permite agregar funcionalidad a un objeto de forma dinámica. Alternativa flexible a subclase para extender funcionalidad.
- MOTIVACIÓN: se utiliza cuando tengo que agregar funcionalidad específica a un objeto de forma dinámica, sin afectar a otros objetos. Cuando se hace imposible generar una jerarquía por lo contidod de clases que generaría y porque la funcionalidad quedaría duplicada en varias partes de la jerarquía.
Se resuelve con uno jerarquía tipo árbol de componentes, que tiene un componente concreto y uno "decorado" que agrega funcionalidad.

- Ejemplos: sentando con scroll, sentando con botón de estado, sentando con scroll y botón de estado.

⑤ Facade ⇒ es un tipo de patrón de diseño estructural. Nace motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencia entre éstos.

- ESTRUCTURA



- INTENCIÓN: interfaz o punto de acceso común a clases y funcionalidad del sistema.

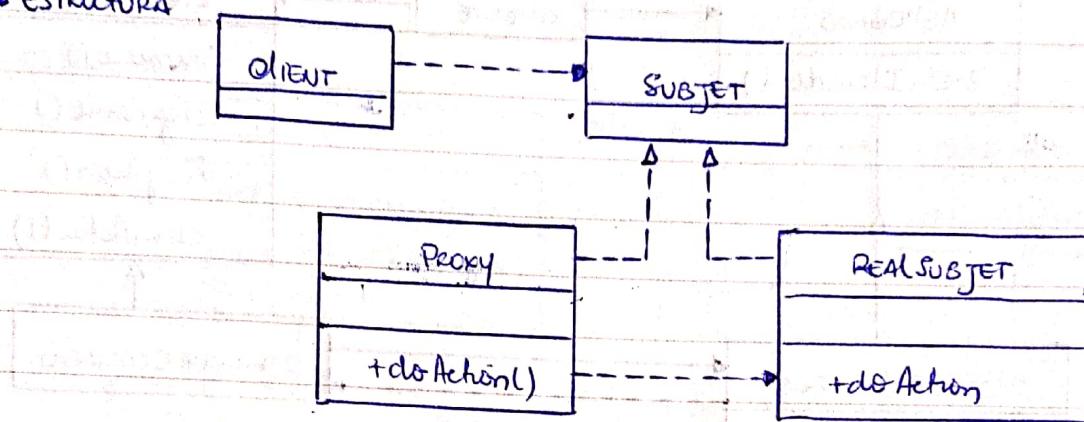
- MOTIVACIÓN: se utiliza cuando tengo un sistema complejo con muchos objetos y mucha funcionalidad y quiero un punto de acceso único; porque no puedo / quiero conocer como están implementados el sistema.

Cuando queremos separar en capas. Cuando hay mucha dependencia (alto acoplamiento) entre los objetos del sistema y queremos proveer portabilidad.

- Ejemplo: Préstamo bancario, que para obtenerlo hay que chequear la situación del cliente (cuenta, créditos anteriores, etc.) en el banco, en otros bancos, etc. Usamos un facade para el préstamo que se encargue de acceder a la funcionalidad para chequear los recursos y la situación del cliente.

⑦ Proxy \Rightarrow es un patrón estructural que tiene como propósito proporcionar un intermediario de un objeto para controlar su acceso.

- ESTRUCTURA

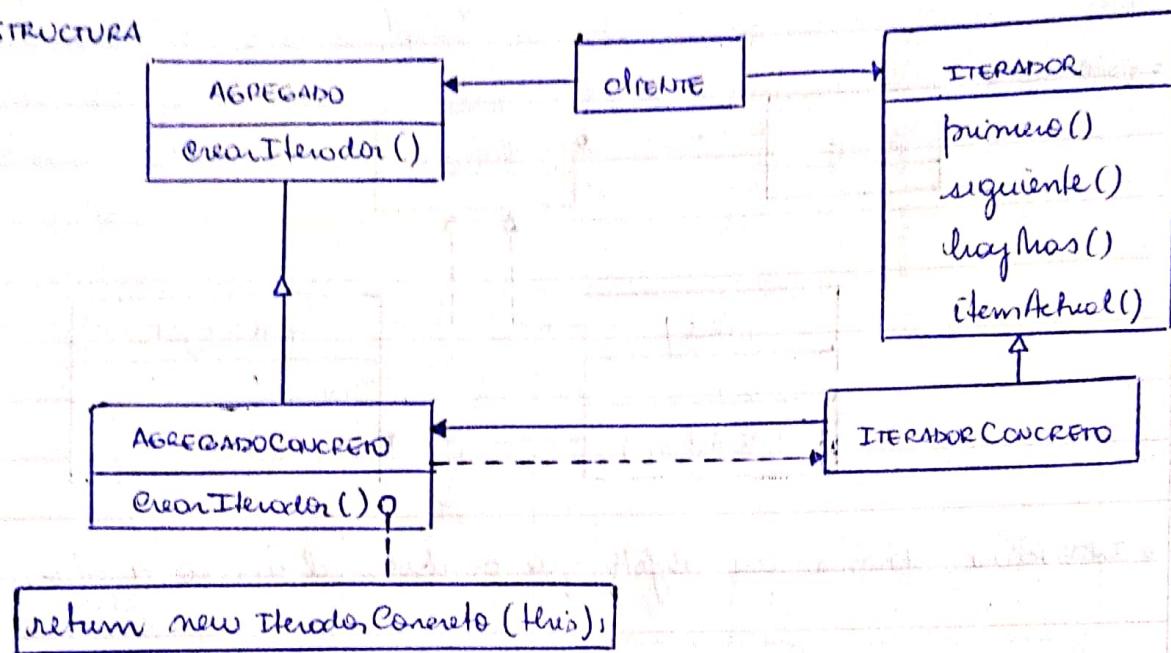


- INTENCIÓN: proveer un objeto que controle el acceso a otro.
- MOTIVACIÓN: se utiliza cuando requiero un acceso a un objeto a través de otro.
Cuando quiero controlar el acceso a un objeto.
Cuando quiero acceso inteligente (el proxy resuelve algunas operaciones y el objeto concreto otras).
Lo resuelvo con una clase proxy que reciba los requerimientos y provea el acceso al objeto real.
- EJEMPLO: Un proxy de internet: cachea y permite acceso a una web.
Un ejemplo es un proxy de medios real.

Patrones de Comportamiento \Rightarrow Son los patrones de diseño de software, que se relacionan con la asignación de responsabilidades entre clases. Enfocaron la colaboración entre objetos.

④ ITERATOR → define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección.

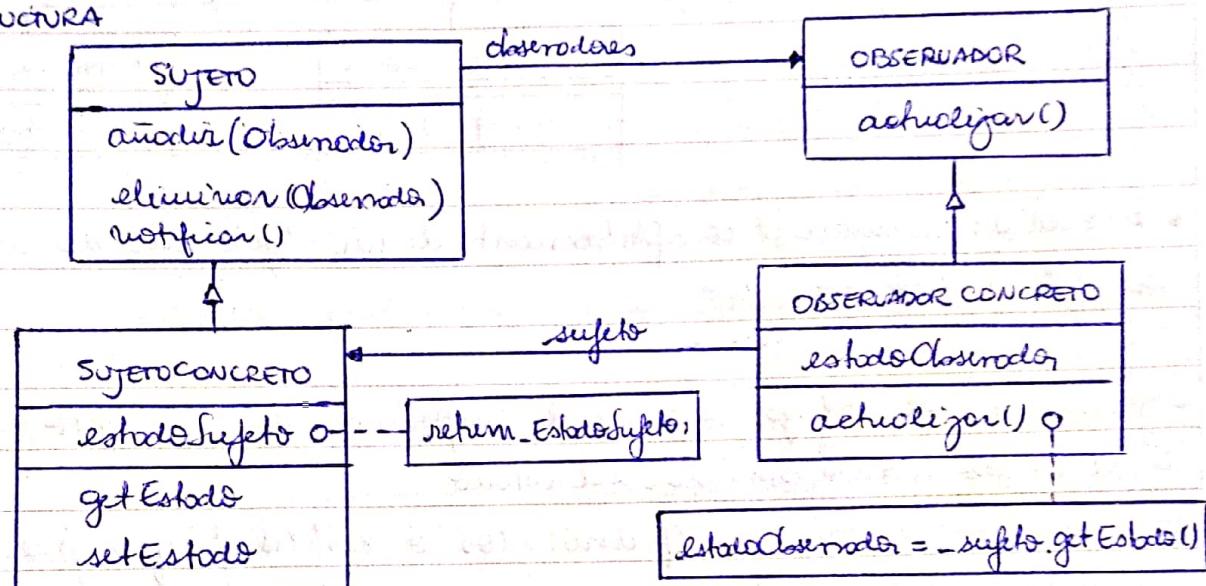
- ESTRUCTURA



- INTENCIÓN: forma de acceder secuencialmente a los elementos de un objeto compuesto (colección) sin exponer su representación.
- MOTIVACIÓN: se utiliza cuando tengo que iterar una colección sin saber como está representada, es decir, sin importar que tipo de colección es.
Para lograr la misma forma de iteración me pongo independiente de la colección (polimorfismo).
Se resuelve con un iterador (interface o superclase) genérico e iteradores concretos para cada tipo de colección.
- EJEMPLO: las colecciones en forma son iteradores. Si uso collection, iterator() me devuelva el iterador concreto de acuerdo al tipo de colección y lo uso siempre de la misma forma (hasNext(), next()). El control remoto es otro ejemplo.

⑦ OBSERVER ⇒ este patrón permite reaccionar a ciertas clases llamadas observadores, sobre un evento determinado. Es usado para monitorear el estado de un objeto en un programa.

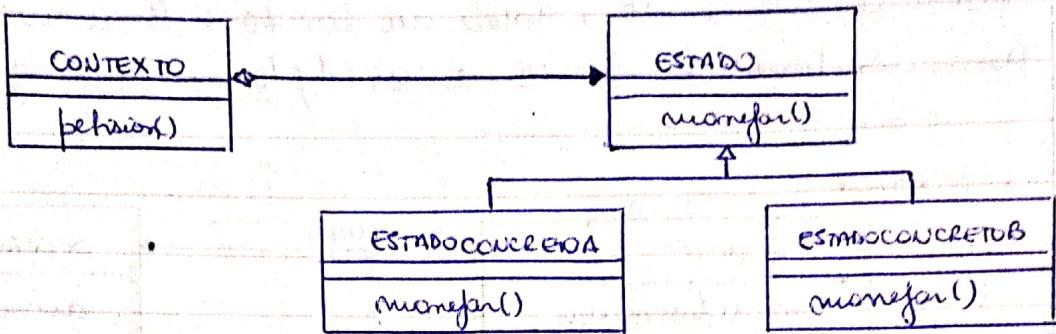
- ESTRUCTURA



- INTENCIÓN: define relaciones de dependencia (uno a muchos) entre objetos, dado que si el observado cambia de estado, los dependientes se actualizan de forma automática.
- MOTIVACIÓN: se utiliza cuando una abstracción tiene más de un aspecto y son independientes. Encapsular esos aspectos en objetos separados, permite intercambiarlos y reusarlos independientemente. Cubriendo el cambio de un objeto obliga a la modificación de otros y no sé cuáles son (esos otros). Cuando necesito notificar a otros objetos de que algo ha sucedido sin saber a quienes debo notificar lo resuelvo con una colección de objetos observadores que reciben la notificación que el objeto ha cambiado.
- Ejemplo: MVC: cuando cambia el estado de un objeto del modelo, las vistas deben cambiar de forma automática.

- ⑧ STATE \Rightarrow permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

- ESTRUCTURA

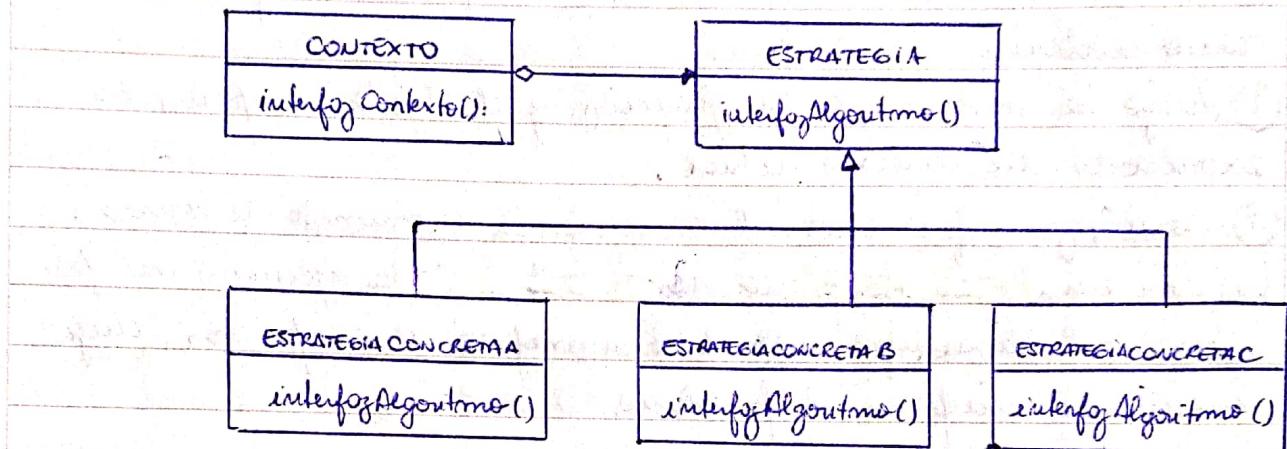


- INTENCIÓN: combina el comportamiento de un objeto cuando cambia su estado interno
- MOTIVACIÓN: si utiliza cuando el comportamiento del objeto depende de su estado y debes combinar ese cuando quieras escribir los if anidados o switchs porque pueden ser ilegibles o pueden aparecer muchas condiciones. Se resuelve con una jerarquía de estados que manejan el siguiente que se le hace al objeto. De acuerdo al estado es el comportamiento que tiene el objeto. El objeto puede pasarse como parámetro al estado para ejecutar los métodos específicos del mismo (doble dispatching).

- EJEMPLO: usuarios de chat: según el estado puede actuar ante el envío de un mensaje
Maquina de café: puede tener diferentes estados: lista, sirviendo, apagado, no disponible (sin café, sin leche o sin agua) y actuar distintamente ante diferentes eventos como ingreso de dinero o selección de una taza.

④ STRATEGY ⇒ encapsula algoritmos en clases, permitiendo que estos sean reutilizados e intercambiables. En base a un parámetro que puede ser cualquier objeto, permite a una aplicación decidir en tiempo de ejecución el algoritmo que debe ejecutar.

- ESTRUCTURA



- INTENCIÓN: familia de algoritmos encapsulados en objetos, haciendo intercambiables. Permite variar el algoritmo de forma independiente del cliente que lo usa.
- MOTIVACIÓN: se utiliza cuando tengo muchos clientes que varían solo en algún comportamiento.
Cuando necesito variantes en un algoritmo:
Para entar if anidados o switchs complejos.
Se resuelve con una jerarquía de algoritmos (o estrategias) donde el objeto instancia la estrategia concreta en el momento indicado
- EJEMPLO: Forma de pago, luego diferentes cosas de acuerdo a la forma de pago (efectivo, tarjeta, cuenta corriente).