PATRONES DE DISEÑO.

Solución probada a un problema que aparece con frecuencia. Elementos de un patrón: Nombre, resuelve un problema, solución y consecuencias.

Categorías:

- Creacionales.
- Organización del trabajo.
- Control de acceso.
- Variación de servicios.
- Extensión de servicios.
- Descomposición estructural.

SINGLETON.

Garantiza que una clase solo tenga una instancia y proporciona un punto de acceso global a ella.

La propia clase es responsable de crear la única instancia. Permite el acceso global a dicha instancia mediante un método de clase. Declara el constructor de clase como privado para que no sea instanciable directamente.

Consecuencias: Acceso controlado a la instancia. Se logra el objeto pero a cambio de ensuciar la clase. Se lo considera un patrón fácil pero intrusivo.

Posible ejemplo práctico: Tablero.

```
public class Singleton {
	private static Singleton INSTANCE = new Singleton();

	private Singleton(){}
	public static Singleton getInstance(){
		return INSTANCE;
	}
}
```

No se puede hacer new Singleton() en “otros lugares”, la clase se encarga de crear la instancia y lo hace solamente una vez. Le damos una responsabilidad más a la clase, se está violando el principio de única responsabilidad.

MULTITON.

Garantiza que una clase solo tenga varias instancias conocidas, y proporciona un punto de acceso global a ella.

Solución: Se implementa igual que el singleton pero con un mapa (indentificador, instancia) en vez de con un atributo. El método getInstance() recible el nombre de la instancia.

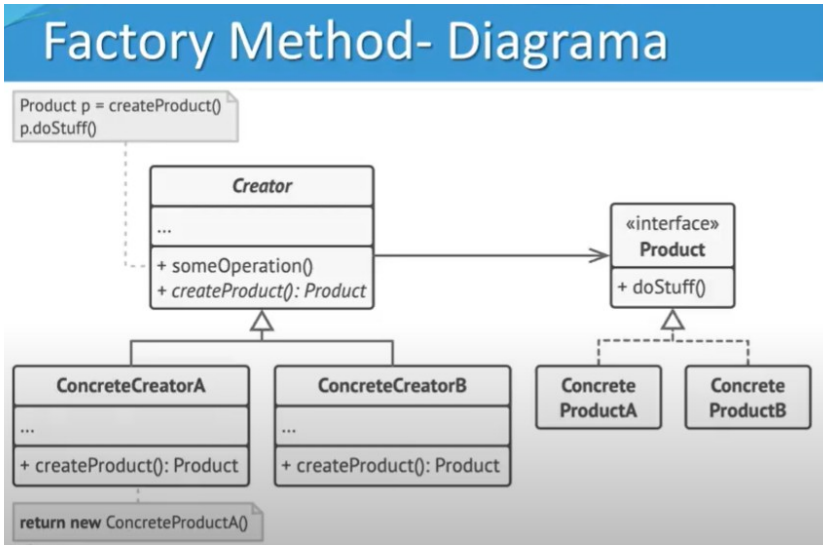
Aplicabilidad: Aplicación de loggers (log de producción, log de debug).

FACTORY METHOD

Proporciona una interfaz para crear un objeto, pero delega en sus hijas la decisión de qué objeto instanciar.

Solución: Crear una clase Factory/Creador que sea abstracta y que provea una interfaz común para la creación del objeto en cuestión. Crear clases Factories que hereden de la clase Factory abstracta y que implementen **EL** método definido creando la instancia concreta. En la aplicación utilizar solo la clase FactoryAbstracta / Creador una vez determinado el objeto a crear.

Consecuencias: Independencia de las clases concretas. Permite intercambiar el objeto creado de manera rápida y transparente.

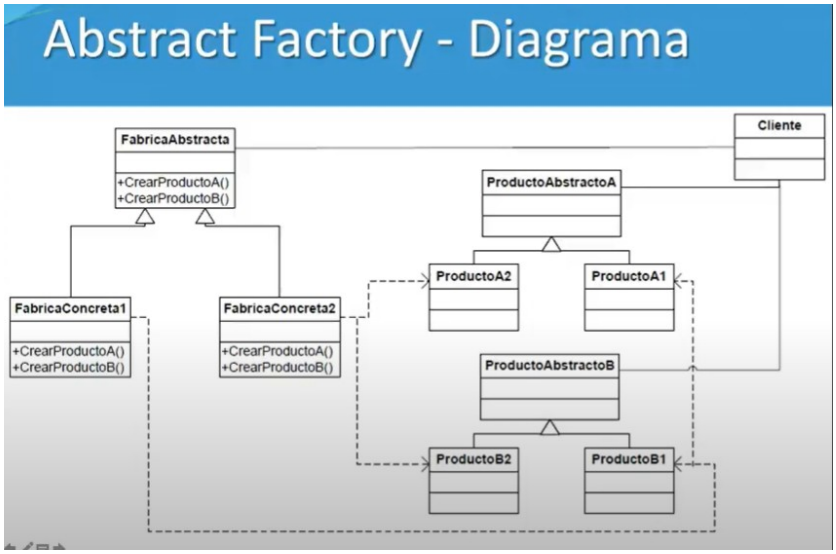


ABSTRACT FACTORY

Proporciona una interfaz para crear familias de objetos relaciones o que dependen entre sí, sin especificar sus clases concretas.

Solución: Crear una clase Factory que sea abstracta que provea una interfaz común de creación de familia de objetos. Crear clases Factories que hereden de la clase Factory abstracta y que implementen los métodos definidos creando las instancias concretas. En la aplicación utilizar solo la clase FactoryAbstracta una vez determinada la familia de objetos a crear.

Consecuencias: Independencia de las clases concretas. Permite intercambio de familias de objetos de manera rápida y transparente.



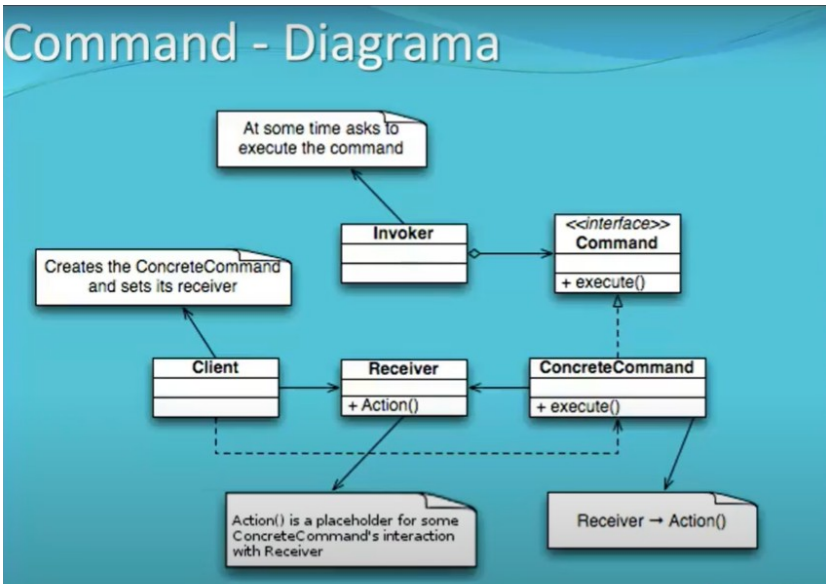
COMMAND

Encapsula una petición en un objeto permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de peticiones y poder deshacer las operaciones. Desacopla el código que solicita un servicio del que lo presta.

Solución: Crear una clase abstracta o una interfaz con un solo método execute(). Cada clase descendiente implementará el método. Para invocar el método se instanciará una de las clases y se invocará al metodo execute().

Motivación: Objetos como botones y menús que realizan una petición en respuesta a una entrada de usuario. Transacciones.

Consecuencias: Permite mantener referencias a métodos. Desacopla el objeto que invoca la operación de aquel que sabe como realizarla. Las órdenes son objetos de primera clase. Pueden ser manipulados y extendidos como cualquier otro objeto. Se pueden ensamblar órdenes en una orden compuesta. Es fácil añadir nuevas ordenes, ya que no hay que cambiar las clases existentes.



Ejemplo: ctr + Z, ctr + C, la acción la trata como si fuese un objeto entero. Define una clase que, por ejemplo se va llamar ctr C o abrir, o etc y todas entienden el mensaje execute();

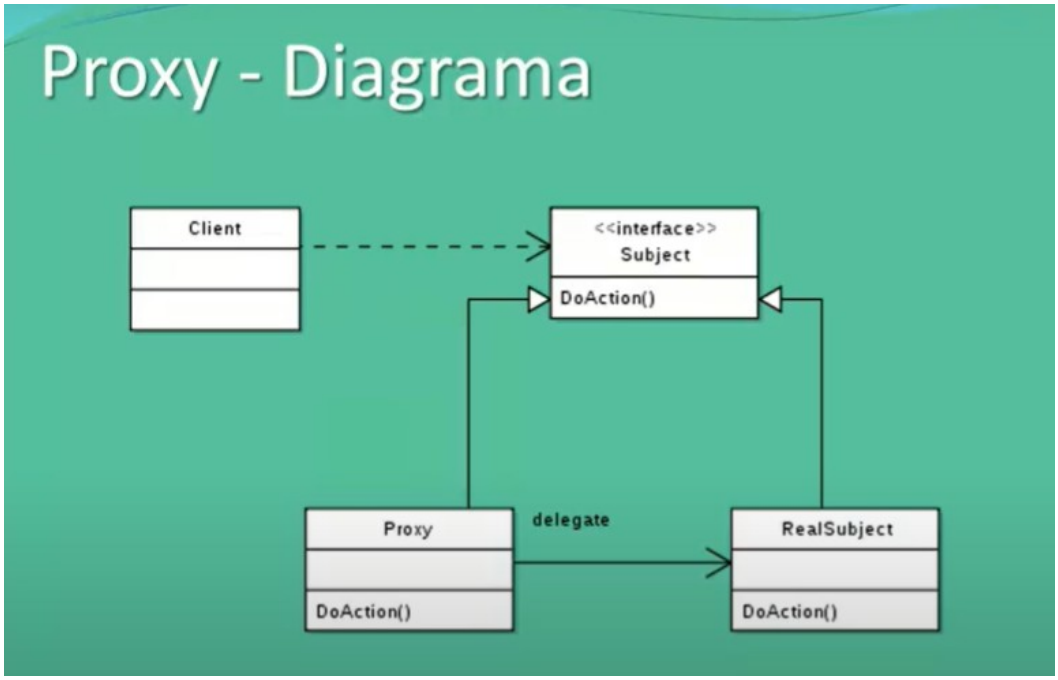
PROXY.

Proporciona un representante o sustituto de otro objeto para controlar el acceso a este.

Solución: Crear una jerarquía en la que intervengan el objeto original y el objeto proxy. En el objeto proxy habrá una referencia al objeto original. Se redefinen todas las llamadas en el proxy incorporando código antes de derivarlas al objeto original.

Motivación: Retrasar el costo de creación e inicialización hasta que realmente sea necesario (Ej: archivo de imágenes)

Consecuencias: Posibilidad de agregar funcionalidad de manera transparente a la aplicación. Permite realizar optimizaciones, ocultar complejidad, establecer mecanismos de seguridad en los accesos.



Ejemplo. El cliente puede ser un alumno o un profesor. Tengo un objeto sujeto y le pido que me cambie la nota a algo. Si tengo al “RealSubject”, este debe verificar antes de cambiar la nota que el cliente no sea un alumno y luego recién cambiar la nota. Para evitar este acomplamiento que no le interesa a RealSubject, el Proxy es quien se va a encargar de la validación y recién, si todo cumple, le va a delegar la tarea al RealSubject osea cambiar la nota.

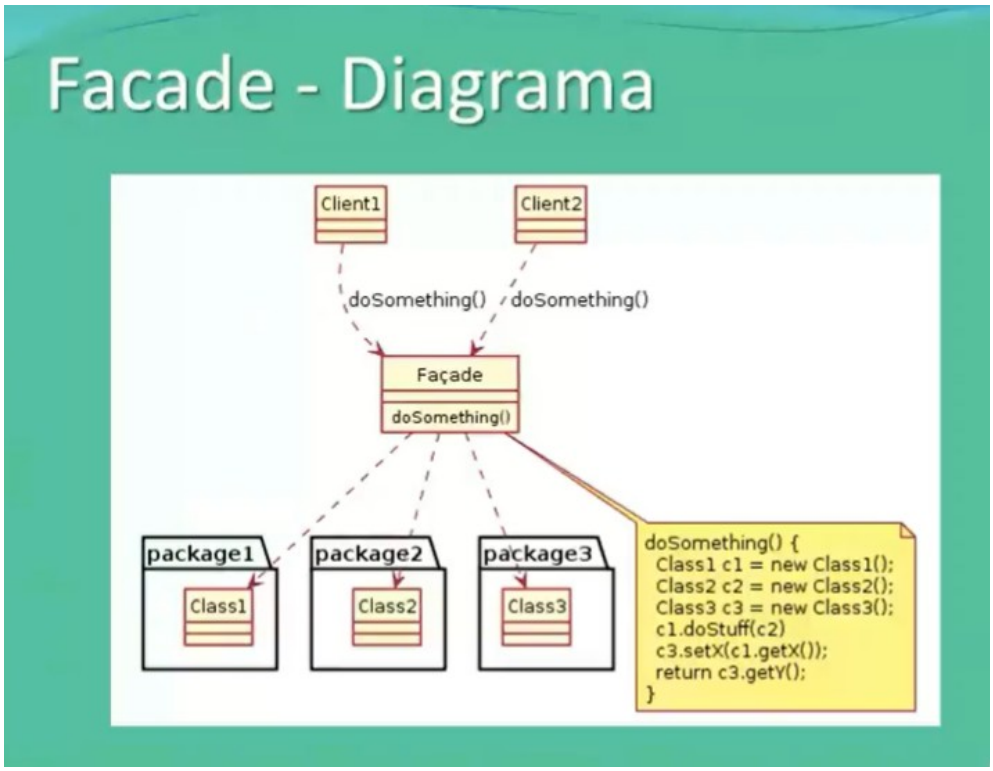
FACADE

Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más difícil de usar.

Motivación: Minimizar la comunicación y dependencias entre subsistemas.

Aplicabilidad: Queremos proporcionar una interfaz simple para un subsistema complejo. Cuando aya muchas dependencias entre los clientes y las clases que implementan una abstracción. Queremos dividir en capas nuestros subsistemas.

Beneficios: Oculta a los clientes los componentes del sistema reduciendo así el número de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar. Promueve un débil acoplamiento entre el subsistema y sus clientes. No impide que las aplicaciones usen las clases del subsistema en caso de que sea necesario. De este modo se puede elegir entre facilidad de uso y generalidad.

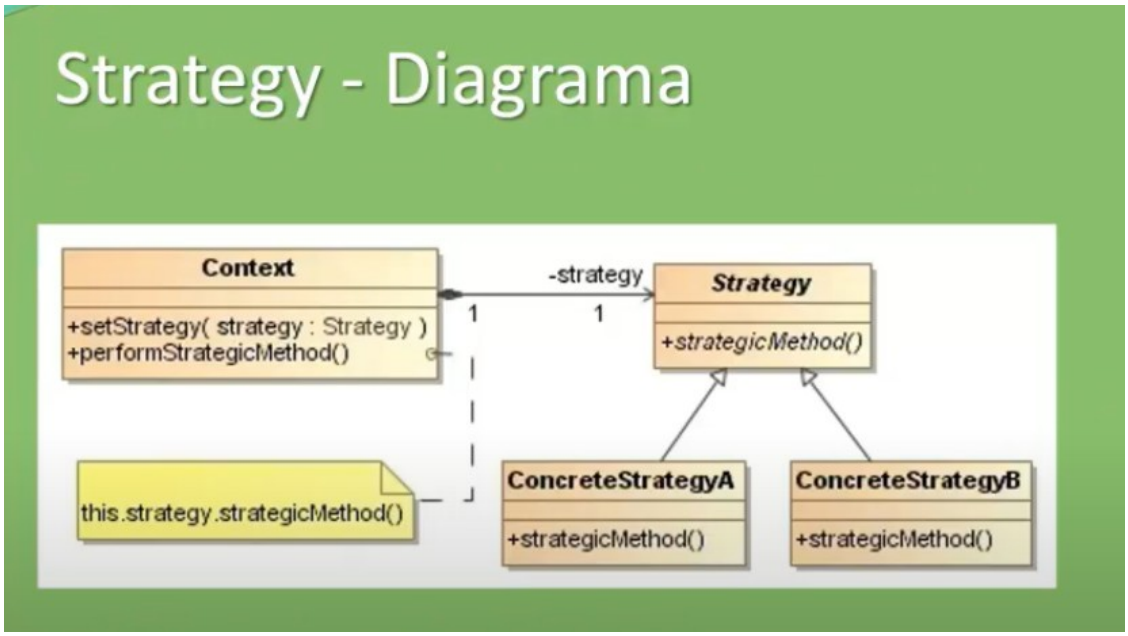


STRATEGY

Un mismo objeto debe poder tener un comportamiento que debe ser determinado en tiempo de ejecución.

Solución: Delegar el comportamiento en otro objeto. Armar una jerarquía con los diferentes comportamientos. Inyectar el comportamiento al objeto a través de un método o de su constructor.

Consecuencias: Se eliminan los condicionales. Se crea una jerarquía paralela a la jerarquía base. Los comportamientos quedan agrupados por las familias. A veces no es tan fácil aislar el comportamiento. A veces no alcanza.

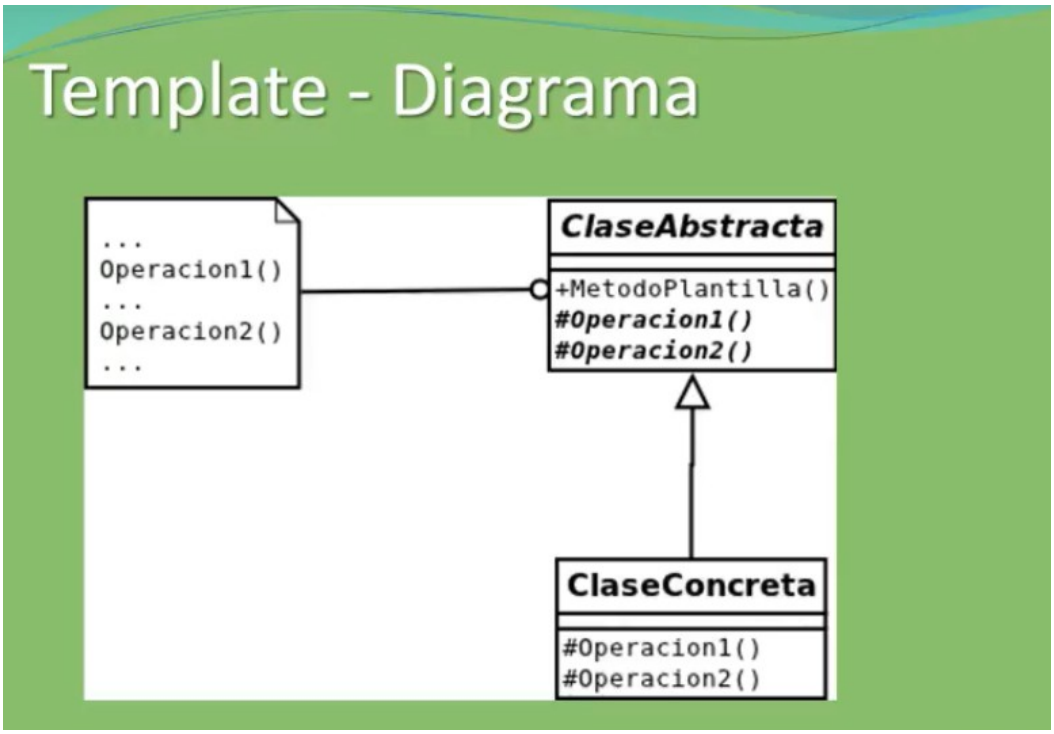


TEMPLATE

Define una operación de un algoritmo, delegando en las subclases alguno de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

Aplicabilidad. Para implementar las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento que puede variar. Cuando el comportamiento repetido de varias subclases debería factorizarse y ser localizado en una clase común para evitar código duplicado. Para controlar las extensiones de extensiones de las subclases.

Beneficios: Son una técnica fundamental de reutilización de código. Extraen el comportamiento común de las clases de la biblioteca. “Principio de Hollywood”, una clase padre llama a las operaciones de una subclase y no al revés. Operaciones de enganche, proporcionan el comportamiento predeterminado que puede ser modificado por las subclases.



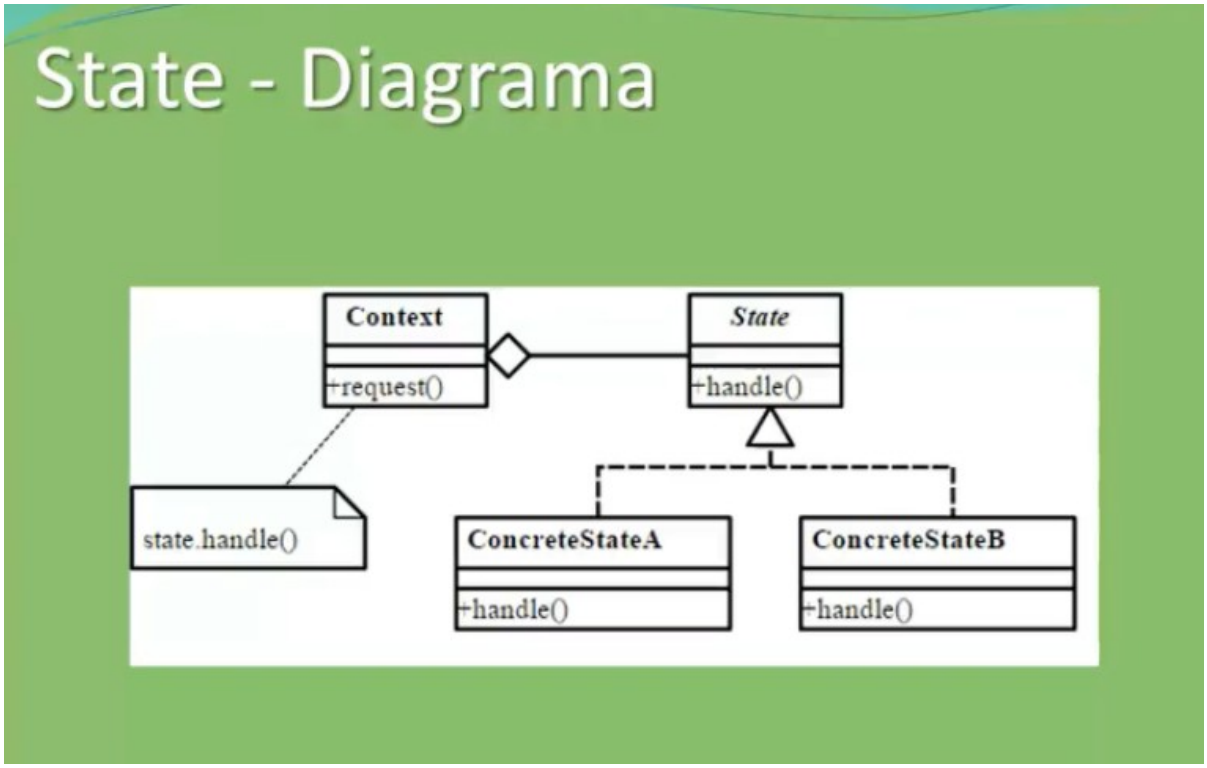
STATE

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

Motivación: Conexión TCP que presenta 2 estados, establecida, escuchando y cerrada.

Aplicabilidad: El comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución dependiendo de ese estado. Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Este estado se suele representar por una o más constantes numeradas.

Beneficios: Localiza el comportamiento dependiendo del estado y divide dicho comportamiento en diferentes estados. Hace explícitas las transiciones entre estados. Los objetos estado pueden compartirse.



¿En qué se diferencia el state del strategy? El Strategy desde afuera tiene un método publico que se llama setStrategy() en cambio con el state no tengo un método para que me cambie el estado. El objeto mismo se autosetea una nueva instancia.

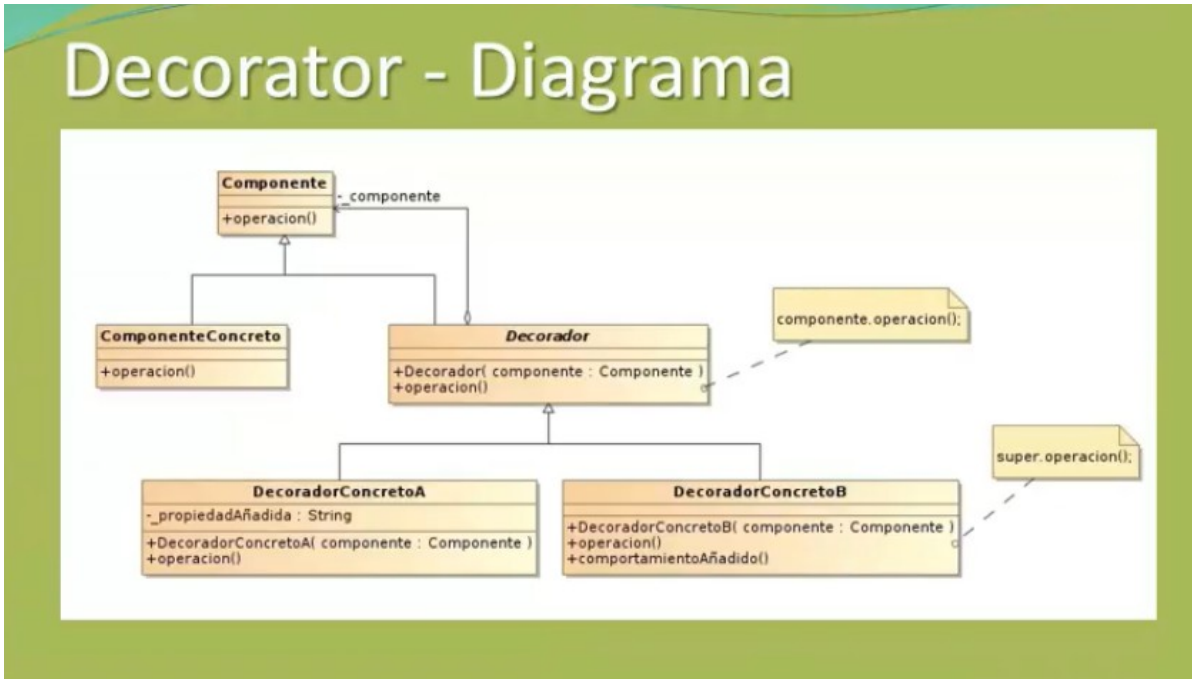
DECORATOR
Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Motivación: Interfaces de usuarios a las que se le agregan propiedades o comportamientos. Cambio de piel.

Aplicabilidad: Para añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos. Para responsabilidades que pueden ser retiradas. Cuando la extensión mediante herencia no es viable.

Beneficios: Más flexibilidad que la herencia estática. Evitar clases cargadas de funciones en la parte de arriba de la jerarquía.

Desventajas: Un decorador y su componente no son idénticos. Muchos objetos pequeños.



COMPOSITE

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

Motivación: Las aplicaciones gráficas como los editores de dibujo y los sistemas de diseño permiten agrupar componentes simples en más grandes. Manejos de excepciones.

Aplicabilidad: Quiera representar jerarquías de objetos parte-todo. Quiera que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

Beneficios: Define jerarquías de clases formadas por objetos primitivos y compuestos. Simplifica el cliente. Facilita añadir nuevos tipos de componentes.

Desventajas: Puede hacer que un diseño sea bastante general.

Composite - Diagrama