PATRONES DE DISEÑO

SINGLETON PATTERN

-CANVAS Y TOOLBAR (+PUBSUB)

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.

Utiliza el patrón Singleton cuando necesites un control más estricto de las variables globales.

UTILIDADES:

1. **Garantizar que una clase tenga una única instancia.**
2. **Proporcionar un punto de acceso global a dicha instancia.**

COMMAND PATTERN (PATRON DE COMPORTAMIENTO):

-PARA PONER EN COLA LOS COMANDOS QUE RECIBIMOS.

Command es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

Utiliza el patrón Command cuando quieras parametrizar objetos con operaciones.

Utiliza el patrón Command cuando quieras poner operaciones en cola, programar su ejecución, o ejecutarlas de forma remota.

Utiliza el patrón Command cuando quieras implementar operaciones reversibles.

FALLO O PROBLEMA:

Este método tiene dos desventajas. Primero, no es tan fácil guardar el estado de una aplicación, porque parte de ella puede ser privada. Este problema puede mitigarse con el patrón **Memento**.

Segundo, las copias de seguridad de estado pueden consumir mucha memoria RAM.

MEMENTO:

Memento es un patrón de diseño de comportamiento que te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

Utiliza el patrón Memento cuando quieras producir instantáneas del estado del objeto para poder restaurar un estado previo del objeto.

Utiliza el patrón cuando el acceso directo a los campos, consultores o modificadores del objeto viole su encapsulación.

RELACION CON OTROS PATRONES:

Puedes utilizar **Command** y **Memento** juntos cuando implementes "deshacer". En este caso, los comandos son responsables de realizar varias operaciones sobre un objeto destino, mientras que los mementos guardan el estado de ese objeto justo antes de que se ejecute el comando.

En ocasiones, **Prototype** puede ser una alternativa más simple al patrón **Memento**. Esto funciona si el objeto cuyo estado quieres almacenar en el historial es suficientemente sencillo y no tiene enlaces a recursos externos, o estos son fáciles de restablecer.

VER PATRON PROTOTYPE (+ SIMPLE)

PROTOTYPE (PATRON CREACIONAL): -PROBAR ANTES EL FACTORY METHOD

Prototype es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

El patrón Prototype delega el proceso de clonación a los propios objetos que están siendo clonados. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método `clonar`.

Un objeto que soporta la clonación se denomina *prototipo*. Cuando tus objetos tienen decenas de campos y miles de configuraciones posibles, la clonación puede servir como alternativa a la creación de subclases.

Funciona así: se crea un grupo de objetos configurados de maneras diferentes. Cuando necesites un objeto como el que has configurado, clonas un prototipo en lugar de construir un nuevo objeto desde cero.

Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.

RESUMEN: -UTIL PARA CLASE COLOR , POINT O SHAPE

Utiliza el patrón cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases para poder crear objetos con una configuración específica.

Muchos diseños empiezan utilizando el **Factory Method** (menos complicado y más personalizable mediante las subclases) y evolucionan hacia **Abstract Factory**, **Prototype**, o **Builder** (más flexibles, pero más complicados).

PREFERIBLE USAR FACTORY METHOD

FACTORY METHOD:

-SHAPES

Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Utiliza el Método Fábrica cuando no conozcas de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código.

Utiliza el Factory Method cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez.

Por lo tanto, necesitas un método regular capaz de crear nuevos objetos, además de reutilizar los existentes. Eso suena bastante a lo que hace un patrón Factory Method.

INTENCION DEL FACTORY METHOD

El patrón Factory Method sugiere que, en lugar de llamar al operador `new` para construir objetos directamente, se invoque a un método *fábrica* especial. No te preocupes: los objetos se siguen creando a través del operador `new`, pero se invocan desde el método fábrica. Los objetos devueltos por el método fábrica a menudo se denominan *productos*.

A simple vista, puede parecer que este cambio no tiene sentido, ya que tan solo hemos cambiado el lugar desde donde invocamos al constructor. Sin embargo, piensa en esto: ahora puedes sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

OBSERVER PATTERN (): -CANVAS O TOOLBAR

Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora. ¡No temas! No es tan complicado como parece. En realidad, este mecanismo consiste en: 1) un campo matriz para almacenar una lista de referencias a objetos suscriptores y 2) varios métodos públicos que permiten añadir suscriptores y eliminarlos de esa lista.

El objeto que tiene un estado interesante suele denominarse *sujeto*, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos *notificador* (en ocasiones también llamado *publicador*). El resto de los objetos que quieren conocer los cambios en el estado del notificador, se denominan *suscriptores*.

Utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.

Utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.