Taller Ejercicios Patrones de Diseño

1. SINGLETON

Ejercicio: Singleton en un Registro de Usuarios

Descripción: Crear una implementación del patrón de diseño Singleton para un registro de usuarios en una aplicación. El registro de usuarios debe ser único en toda la aplicación y debe permitir el acceso a la lista de usuarios desde diferentes partes del código.

Instrucciones:

- Crea una clase llamada **`UserRegistry`** que implemente el patrón Singleton.
- La clase **`UserRegistry`** debe tener un método estático **`getInstance()`** que devuelva la única instancia de la clase.
- Dentro de la clase **`UserRegistry`**, implementa una lista privada para almacenar objetos de tipo **`User`**.
- Implementa métodos públicos para agregar un usuario a la lista (**`addUser(User user)`**), obtener la lista completa de usuarios (**`List<User> getUsers()`**), y buscar un usuario por su nombre de usuario (**`User getUserByUsername(String username)`**).
- Asegúrate de que la clase **`User`** tenga al menos propiedades como **`username`**, **`email`** y **`password`**.

2.

Ejercicio: Configuración Global del Juego

Descripción: En un juego de consola, se requiere una configuración global que almacene la configuración actual del juego, como la dificultad, el volumen del sonido y las opciones de gráficos. Esta configuración debe ser accesible desde diferentes partes del código del juego y debe mantenerse consistente en toda la ejecución.

Instrucciones:

- Crea una clase llamada **`GameConfiguration`** que implemente el patrón Singleton.
- La clase **`GameConfiguration`** debe tener un método estático **`getInstance()`** que retorne la única instancia de la clase.
- Dentro de la clase **`GameConfiguration`**, implementa propiedades privadas para almacenar la dificultad del juego, el volumen del sonido y las opciones gráficas.
- Implementa métodos públicos para establecer y obtener los valores de dificultad, volumen de sonido y opciones gráficas.
- Asegúrate de que la clase **`GameConfiguration`** maneje adecuadamente la modificación de sus propiedades y proporcione una forma de acceder a la configuración desde diferentes partes del juego.

3.

Ejercicio: Control de Acceso a un Sistema

Descripción: En un sistema de control de acceso a una instalación, se necesita una manera de administrar el acceso de usuarios y controlar el estado de las puertas. Cada usuario debe tener un único punto de acceso para validar su identidad y acceder a diferentes áreas según sus permisos.

Instrucciones:

- Crea una clase llamada **`AccessControlSystem`** que implemente el patrón Singleton.
- La clase **`AccessControlSystem`** debe tener un método estático **`getInstance()`** que retorne la única instancia de la clase.
- Dentro de la clase **`AccessControlSystem`**, implementa una lista privada de usuarios registrados y una lista de puertas con su estado (abiertas/cerradas).
- Implementa métodos públicos para agregar usuarios (**`void addUser(User user)`**), validar el acceso de un usuario a una puerta (**`boolean validateAccess(User user, String doorId)`**), y cambiar el estado de una puerta (**`void changeDoorState(String doorId, boolean isOpen)`**).

- Asegúrate de que la clase **`AccessControlSystem`** gestione adecuadamente la autenticación de usuarios y el control del estado de las puertas.

4. PATRON DE DISEÑO FACTORY

Ejercicio: Creación de Personajes en un Videojuego RPG

Descripción: En un videojuego de rol (RPG), se necesita una manera de crear diferentes tipos de personajes, como guerreros, magos y arqueros, cada uno con sus propias características y habilidades únicas. Se busca implementar un sistema que permita crear personajes de manera eficiente y flexible.

Instrucciones:

- Crea una clase llamada **`Personaje`** que servirá como clase base para todos los tipos de personajes.
- Define clases concretas que hereden de **`Personaje`** para representar diferentes tipos de personajes, como **`Guerrero`**, **`Mago`** y **`Arquero`**.
- Crea una interfaz llamada **`Habilidad`** con un método **`usar()`** para representar las habilidades únicas de cada tipo de personaje.
- Implementa clases que implementen la interfaz **`Habilidad`** para cada tipo de habilidad, como **`AtaqueEspada`**, **`LanzarHechizo`** y **`DispararFlecha`**.
- Crea una clase llamada **`FabricaPersonajes`** que implemente el patrón Factory.
- Dentro de la clase **`FabricaPersonajes`**, implementa un método **`crearPersonaje(String tipo)`** que acepte un tipo de personaje y devuelva una instancia correspondiente.
- En el método principal del programa, crea una instancia de **`FabricaPersonajes`** y utiliza este objeto para crear diferentes tipos de personajes y sus habilidades únicas.

5.

Ejercicio: Creación de Widgets en una Interfaz Gráfica

Descripción: En una aplicación de interfaz gráfica de usuario (GUI), se necesita una manera de crear diferentes tipos de widgets, como botones, campos de texto y paneles, de manera consistente y personalizable. Se busca implementar un sistema que permita crear widgets de manera eficiente y centralizada.

Instrucciones:

- Crea una interfaz llamada **`Widget`** con métodos que representen las acciones comunes de los widgets, como **`dibujar()`**, **`clic()`**, etc.
- Implementa clases concretas que implementen la interfaz **`Widget`** para representar diferentes tipos de widgets, como **`Boton`**, **`CampoTexto`** y **`Panel`**.
- Crea una clase llamada **`FabricaWidgets`** que implemente el patrón Factory.
- Dentro de la clase **`FabricaWidgets`**, implementa un método **`crearWidget(String tipo)`** que acepte un tipo de widget y devuelva una instancia correspondiente.
- En el método principal del programa, crea una instancia de **`FabricaWidgets`** y utiliza este objeto para crear diferentes tipos de widgets y realizar acciones específicas en función del tipo de widget.

6.

Ejercicio: Creación de Comidas en un Sistema de Pedidos

Descripción: En un sistema de pedidos de alimentos, se necesita una manera de crear diferentes tipos de comidas, como hamburguesas, pizzas y ensaladas, cada una con sus propios ingredientes y opciones. Se busca implementar un sistema que permita crear comidas de manera consistente y personalizada.

Instrucciones:

- Crea una interfaz llamada **`Comida`** con métodos que representen las acciones comunes de las comidas, como **`mostrarDescripcion()`**, **`calcularPrecio()`**, etc.

- Implementa clases concretas que implementen la interfaz **`Comida`** para representar diferentes tipos de comidas, como **`Hamburguesa`**, **`Pizza`** y **`Ensalada`**.
- Crea una clase llamada **`FabricaComidas`** que implemente el patrón Factory.
- Dentro de la clase **`FabricaComidas`**, implementa métodos para crear diferentes tipos de comidas, cada una con ingredientes y opciones específicas.
- En el método principal del programa, crea una instancia de **`FabricaComidas`** y utiliza este objeto para crear varias comidas, mostrar sus descripciones y calcular sus precios.

7. PATRON DE DISEÑO PROTOTYPE

Ejercicio: Clonación de Documentos

Descripción: Se necesita implementar un sistema que permita clonar documentos de manera eficiente. Cada documento puede tener contenido y formato propio. El patrón Prototype será utilizado para crear copias exactas de documentos existentes.

Instrucciones:

- Crea una clase abstracta llamada **`Documento`** con métodos que representen las acciones comunes de los documentos, como **`mostrarContenido()`** y **`obtenerFormato()`**.
- Implementa clases concretas que hereden de **`Documento`** para representar diferentes tipos de documentos, como **`DocumentoTexto`** y **`DocumentoImagen`**.
- Implementa el método **`clone()`** en cada clase concreta para permitir la clonación profunda de documentos.
- En el método principal del programa, crea instancias de documentos y clona diferentes tipos de documentos para mostrar sus contenidos y formatos.

8.

Ejercicio: Creación y Modificación de Prototipos de Juegos de Estrategia

Descripción: Se necesita implementar un sistema para crear prototipos de juegos de estrategia que permita clonar unidades de juego y realizar modificaciones en sus atributos. Cada unidad tiene estadísticas y puede ser personalizada. El patrón Prototype será utilizado para crear y modificar las unidades de juego.

Instrucciones:

- Crea una interfaz llamada **`Unidad`** con métodos que representen las acciones y estadísticas comunes de las unidades, como **`mostrarDescripcion()`**, **`atacar()`** y **`defender()`**.
- Implementa clases concretas que implementen la interfaz **`Unidad`** para representar diferentes tipos de unidades de juego, como **`Soldado`** y **`Arquero`**.
- Implementa el método **`clone()`** en cada clase concreta para permitir la clonación de unidades y personalización.
- Agrega métodos en las clases concretas para modificar atributos específicos de las unidades, como **`modificarAtaque(int nuevoAtaque)`** y **`modificarDefensa(int nuevaDefensa)`**.
- En el método principal del programa, crea instancias de unidades de juego, clona unidades y realiza modificaciones personalizadas en las unidades clonadas.

9.

Contexto: Gestión de Cuentas Bancarias

Descripción: Se requiere desarrollar un sistema para la gestión de cuentas bancarias en un banco. Cada cuenta bancaria tiene un número único, saldo, y puede estar asociada a un titular. El banco ofrece diferentes tipos de cuentas, como cuentas de ahorro y cuentas corrientes, cada una con reglas y características específicas.

Instrucciones:

- Crea una clase abstracta llamada **`CuentaBancaria`** con propiedades y métodos comunes a todas las cuentas, como **`obtenerNumero()`**, **`obtenerSaldo()`** y **`realizarTransaccion(double monto)`**.
- Implementa clases concretas que hereden de **`CuentaBancaria`** para representar diferentes tipos de cuentas, como **`CuentaAhorro`** y **`CuentaCorriente`**.
- Define métodos específicos en las clases concretas para manejar las reglas y características de cada tipo de cuenta, como **`calcularIntereses()`** en la **`CuentaAhorro`** y **`cobrarComision()`** en la **`CuentaCorriente`**.
- Implementa el patrón Prototype en las clases concretas para permitir la clonación de cuentas y personalización.
- En el método principal del programa, crea instancias de cuentas, realiza transacciones, calcula intereses o cobra comisiones según el tipo de cuenta y muestra la información de las cuentas.