UNIVERSIDAD AUTÓNOMA DE QUERÉTARO

Nombre: Diego Mendoza Chávez

Expediente: 327967

Docente: Enrique Aguilar Orozco

Materia / Taller : Patrones de Diseño

Facultad de Informática

Universidad Autónoma de Querétaro

Trabajo a entregar: Entregable (práctica) de Patrones de Diseño y Pilares del POO

Primera Sección: Ejercicios de los Pilares del POO

-Dichos ejercicios tipo "Typescript" están en la carpeta "Ejercicios-Pilares-del-POO" de este entregable.

Segunda Sección: Identificar Patrones tipo Singleton, Strategy y Proxy

-Se encuentran en la carpeta "IdentificarPatrones" de este entregable.

-Tipo "Singleton" (creacional)

```typescript
class Singleton {
  getConfiguracion(arg0: string): any {
    throw new Error("Method not implemented.");
  }
  setConfiguracion(arg0: string, arg1: string) {
    throw new Error("Method not implemented.");
  }
  private static instance: Singleton; // Instancia única de la clase

  public constructor() { } // Constructor privado

  public static getInstance(): Singleton {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
}


class Configuracion extends Singleton {
  private configuracion: any;

  public constructor() {
    super(); // Llama al constructor de la clase base Singleton
    this.configuracion = {}; // Inicializa la configuración vacía
  }

  public setConfiguracion(clave: string, valor: any): void {
    this.configuracion[clave] = valor;
  }

  public getConfiguracion(clave: string): any {
    return this.configuracion[clave];
  }
}
```

```typescript

// Ejemplo de uso

const configuracion = Configuracion.getInstance();
configuracion.setConfiguracion("idioma", "español");
console.log(configuracion.getConfiguracion("idioma")); // Imprime "español"

```

En el presente código esta la Clase Configuración: usa el patrón base Singleton.

Posteriormente el constructor inicializa la configuración como un objeto vacío, y se usa el Método "setConfiguracion" y "getConfiguracion" que permite establecer un valor para una clave específica en la configuración; y obtener el valor de una clave específica en la configuración.

Ejemplo de uso: Se crea una instancia de la configuración, se establece un valor para la clave "idioma" y se obtiene el valor configurado.

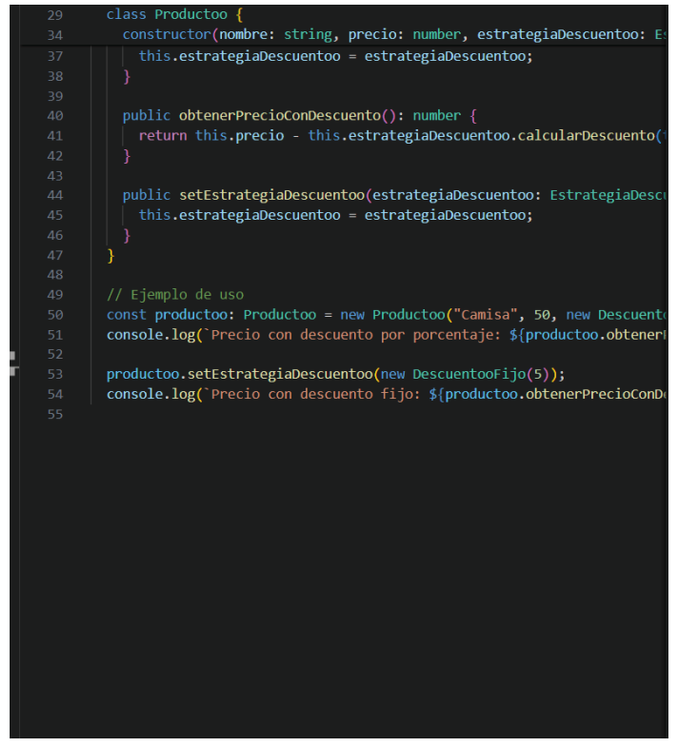
La función de la clase "Configuración" se enfoca en almacenar y acceder a configuraciones, además de almacenar un objeto arbitrario para la configuración

Casos de uso:

Almacenar preferencias de usuario: Guardar y cargar preferencias de idioma, tema, etc.
Configurar parámetros de aplicación: Definir valores predeterminados para parámetros de conexión, rutas de archivos, etc.
Gestionar entornos: Adaptar el comportamiento de la aplicación según el entorno (desarrollo, producción, etc.).

-Tipo "Strategy" (comportamiento)

```
interface EstrategiaDescuentoo {
    calcularDescuento(precio: number): number;
}

class DescuentooPorcentaje implements EstrategiaDescuentoo {
    private porcentaje: number;

    constructor(porcentaje: number) {
        this.porcentaje = porcentaje;
    }

    calcularDescuento(precio: number): number {
        return precio * (this.porcentaje / 100);
    }
}

class DescuentooFijo implements EstrategiaDescuentoo {
    private descuentooFijo: number;

    constructor(descuentooFijo: number) {
        this.descuentooFijo = descuentooFijo;
    }

    calcularDescuento(precio: number): number {
        return precio - this.descuentooFijo;
    }
}

class Productoo {
    private nombre: string;
    private precio: number;
    private estrategiaDescuentoo: EstrategiaDescuentoo;

    constructor(nombre: string, precio: number, estrategiaDescuentoo: EstrategiaDes
        this.nombre = nombre;
        this.precio = precio;
        this.estrategiaDescuentoo = estrategiaDescuentoo;
```



```
class Productoo {
    constructor(nombre: string, precio: number, estrategiaDescuentoo: E
        this.estrategiaDescuentoo = estrategiaDescuentoo;
    }

    public obtenerPrecioConDescuento(): number {
        return this.precio - this.estrategiaDescuentoo.calcularDescuento(
    }

    public setEstrategiaDescuentoo(estrategiaDescuentoo: EstrategiaDesc
        this.estrategiaDescuentoo = estrategiaDescuentoo;
    }
}

// Ejemplo de uso
const productoo: Productoo = new Productoo("Camisa", 50, new Descuent
console.log(`Precio con descuento por porcentaje: ${productoo.obtener

productoo.setEstrategiaDescuentoo(new DescuentooFijo(5));
console.log(`Precio con descuento fijo: ${productoo.obtenerPrecioConD
```

En este código esta el uso de la interfaz "EstrategiaDescuentoo" que determina y define la firma del método calcularDescuento que todas las estrategias de descuento deben implementar.

Posteriormente las Clases "DescuentooPorcentaje y DescuentoFijoo" implementan "EstrategiaDescuentoo" y calculan el descuento según su algoritmo específico.

Mientras que la Clase Producto almacena el nombre, precio y la estrategia de descuento actual del producto; usa también la estrategias para calcular el precio con descuento y cambiar la estrategia de descuento en el tiempo de ejecución, ubicada en el método "obtenerPrecioConDescuento" y "setEstrategiaDescuentoo".

Ejemplo de uso:
Se crea un producto con un precio inicial y una estrategia de descuento por porcentaje.
Se calcula y muestra el precio con descuento.
Se cambia la estrategia a descuento fijo y se calcula y muestra el nuevo precio con descuento.

Su función se enfoca en la flexibilidad y permitir la variación del comportamiento del cliente sin modificar su estructura.

Casos de uso:

Cálculos de descuentos: Aplicar diferentes tipos de descuentos (porcentaje, fijo, cupón) en un sistema de ventas.
Validación de datos: Implementar diferentes reglas de validación para distintos tipos de datos (formato de correo electrónico, números de teléfono).

-Tipo "Proxy" (estructural)

```typescript
interface Sujetoo {
  operacion(): string;
}

class SujetooReal implements Sujetoo {
  public operacion(): string {
    return "Operación realizada en el sujeto real";
  }
}

class Proxy1 implements Sujetoo {
  private sujetooReal: Sujetoo;

  constructor(sujetooReal: Sujetoo) {
    this.sujetooReal = sujetooReal;
  }

  public operacion(): string {
    console.log("Antes de la operación real"); // Registro previo
    const resultado = this.sujetooReal.operacion();
    console.log("Después de la operación real"); // Registro posterior
    return resultado;
  }
}

// Ejemplo de uso
const sujetooReal = new SujetooReal();
const proxy1 = new Proxy1(sujetooReal);

console.log(proxy1.operacion()); // Imprime:
                                 // Antes de la operación real
                                 // Operación realizada en el sujeto real
                                 // Después de la operación real
```

Este mismo código hace uso de la Interfaz "Sujetoo" que define la firma del método "operación" que todos los sujetos (real y proxy) deben implementar.

Ademas hace uso presente de la Clase "SujetooReal" que implementa la interfaz "Sujetoo" y realiza la operación real.
A su vez la interfaz "Sujetoo" realiza un almacenamiento haciendo referencia al objeto "sujetooReal".

Finalmente el método operacion realiza las siguientes acciones:

-Registra un mensaje antes de la operación real.
-Llama al método operacion del objeto sujetooReal.
-Registra un mensaje después de la operación real.
-Devuelve el resultado de la operación real.

Ejemplo de uso:
Se crea una instancia de SujetooReal y un proxy asociado. Se llama al método operacion del proxy, lo que desencadena el registro previo, la operación real y el registro posterior.

Casos de uso:

Acceso a recursos remotos: Controlar el acceso y el uso de recursos remotos como servicios web o bases de datos.
Operaciones sensibles: Registrar y auditar operaciones sensibles en un sistema, como transferencias bancarias o cambios de contraseña.

El Proxy permite controlar quién puede acceder al sujeto real y qué operaciones puede realizar y lo protege de accesos o modificaciones no deseados.