UNIVERSIDAD TECNOLÓGICA DE QUERÉTARO

Nombre del trabajo:

Arquitectura de Software

Nombre de la materia:
Patrones de Diseño

Ing. Desarrollo y Gestión de Software

Presenta:
URIEL ISAI ORTIZ PEREZ

Matricula:
2024378002

Santiago de Querétaro, Qro. 27 de diciembre 2024

Contenido

Singletón

En este ejemplo voy a usar singleton con un ejemplo de servicios, lo primero que haré es mostrar cómo sería un caso sin singleton. Esto nos dirá que servicio 1 y Servicio2 son Iguales y esto es lo que queremos evitar con Singleton.

```
export class MyService {
    constructor(public name: string) {}

    getName() {
        return this.name;
    }
}

// Crear instancias de MyService
const servicio1 = new MyService("Servicio1");
const servicio2 = new MyService("Servicio2");

// Imprimir los nombres
console.log(servicio1.getName()); // Imprime "Servicio1"
console.log(servicio2.getName()); // Imprime "Servicio2"

// Comparar si son diferentes
if (servicio1 !== servicio2) {
    console.log("Servicio1 y Servicio2 son diferentes.");
} else {
    console.log("Servicio1 y Servicio2 son iguales.");
}
```

Ahora lo que haremos es colocar en forma privada el constructor para solo poder acceder dentro del método, para eso vamos a crear el método estático llamado créate, solo vamos a usar una variable que se le pasará al constructor.
Ahora necesitamos algo que indique si previamente hemos creado una instancia de este servicio, por que pues solo se puede crear una instancia de este servicio, esta la debemos guardar en memoria por lo que vamos a crear un estático llamado instance que es de tipo Myservice el cual se va a iniciar en null = null.

```
    private constructor(private name: string) {}
```

Ahora lo que necesitamos es comprobar en el método créate y si es igual a nulo vamos a crear la instancia, el único lugar dentro de la clase donde se puede llamar al constructor. Y lo retornamos.

```
static create(name: string) {
```

```
    if (MyService.instance === null) {
      console.log('debería entrar una vez');
      MyService.instance = new MyService(name);
    }
    return MyService.instance;
  }
```

Ahora vamos a llamar a MyService y su método crear las veces que queramos para comprobar los imprimimos y corroboraremos que solo entra una vez y que por más que agregamos otros nombres solo se crea una vez.

```
  }debería entrar una vez
  service 1
  service 1
  service 1
```

Y como podemos ver solo permite que entre una vez y los 3 Service se siguen llamado igual

```
export class MyService {
  // Variable estática para almacenar una única instancia de la clase
  static instance: MyService | null = null;

  // Constructor privado para evitar la creación de múltiples instancias
  private constructor(private name: string) {}

  // Método para obtener el nombre de la instancia
  getName() {
    return this.name;
  }

  // Método estático para crear o obtener la instancia
  static create(name: string) {
    if (MyService.instance === null) {
      console.log('debería entrar una vez');
      MyService.instance = new MyService(name);
    }
    return MyService.instance;
  }
}

const myService1 = MyService.create('Service 1');
const myService2 = MyService.create('Service 2');
const myService3 = MyService.create('Service 3');
```

```
// Imprimir los nombres para verificar el contenido de la instancia única
console.log( myService1.getName());
console.log( myService2.getName());
console.log( myService3.getName());

console.log('myService1 === myService2:', myService1 === myService2); // true
console.log('myService2 === myService3:', myService2 === myService3); // true
console.log('myService1 === myService3:', myService1 === myService3); // true
```

Factory:

Patrón de diseño que nos provee de una interfaz para crear objetos basados en una superclase, posibilitando que las subclases creadoras alteren el tipo de objetos a retornar en su proceso de fabricación.

1. El patrón sugiere que en lugar de usar el operador new() se invoque a un método fábrica que se encargue de la creación de los objetos. Estos objetos son llamados productos.
2. Internamente, este método seguirá usando el operador new().

En este ejemplo:

Esta clase actúa como **interfaz base** para los distintos tipos de coches que vamos a crear. Define un método mostrarPrecio().Este método lo vamos a implementor por las clases derivadas de CocheBase, asegurando que todos los tipos de coches tengan un método para mostrar el precio.

```
class CocheBase {
   mostrarPrecio() {
     throw new Error('¡Método no implementado!');
   }
 }
```

CocheTitan y **CocheLeón** son **subclases** de CocheBase. Cada una implementa el método de mostrarPrecio(), que imprime el precio específico del coche cuando se llama. Estos son los productos concretos que se van a crear a través de las fábricas.

FabricaCoches es la **interfaz base de fábrica**. Aquí declaramos un método crearCoche() que debe ser implementado por las fábricas concretas. Esta clase no

sabe qué tipo de coche va a crear; garantiza que todas las fábricas derivadas sigan un contrato común.

```
class FabricaCocheLeón extends FabricaCoches {
  crearCoche() {
    return new CocheLeón();
  }
}
```

Estas son las **fábricas concretas** que implementan el método crearCoche().

- FabricaCocheTitan crea una instancia de CocheTitan.
- FabricaCocheLeón crea una instancia de CocheLeón.

Estas fábricas se encargan de crear los productos específicos (CocheTitan y CocheLeón), y es aquí donde se "decide" qué tipo de coche se va a crear.

```
class FabricaCocheTitan extends FabricaCoches {
  crearCoche() {
    return new CocheTitan();
  }
}

class FabricaCocheLeón extends FabricaCoches {
  crearCoche() {
    return new CocheLeón();
  }
}
```

La función aplicarFabrica recibe una fábrica (como FabricaCocheTitan o FabricaCocheLeón), llama al método crearCoche() de esa fábrica para crear un coche y luego llama a mostrarPrecio() para mostrar el precio del coche creado.

Esta función es la que usa la fábrica para crear el coche y realizar alguna acción con él (en este caso, mostrar el precio).

```
function aplicarFabrica(fabrica) {
    const coche = fabrica.crearCoche();
    coche.mostrarPrecio();
  }
```

La función CrearFabrica recibe un tipo de coche (por ejemplo, 'titan' o 'león'), selecciona la fábrica correspondiente (FabricaCocheTitan o FabricaCocheLeón) y retorna una nueva instancia de la fábrica seleccionada.

Este código nos va permitir

- Reducir el acoplamiento entre los objetos que crean los coches y aquellos que los utilizan.
- La creación de coches ocurre en un único lugar.
- Agregar nuevos coches no requiere modificar el código ya existente.

Prototype:

Patrón de diseño que nos permite hacer clones de objetos existentes sin que dependamos de clases concretas.
Para esto vamos a necesitar

- Declarar una clase base o interfaz prototipo que contenga los métodos de clonación.
- Crea productos concretos que heredan o implementan de la interfaz creada en el punto 1, con esto se asegura que tenga el método clone (Se prioriza la configuración)

El método Clone permite que un objeto de la clase CompleteUser cree una copia exacta de sí mismo. La lógica detrás de este método es la siguiente:

```
public clone() {
  const props: ICompleteUserProps = {
    name: this.person.getName,
    lastName: this.person.getLastName,
    username: this.user.getUsername,
    directory: this.userConfiguration.getDirectory,
    homeFolder: this.userConfiguration.getHomeFolder,
  };

  return new CompleteUser(props);
}
```

el método obtiene las propiedades del objeto original utilizando los getters definidos para las instancias de Person, User y UserConfiguration. Estas propiedades se almacenan en un objeto props que es una representación estructurada de todas las propiedades necesarias para crear un nuevo objeto.

El método clone() usa el objeto props para crear una nueva instancia de la clase CompleteUser. De este modo, se crea una **nueva instancia de CompleteUser** que tiene los mismos valores, pero es una **referencia diferente en memoria**. Este es el principio del patrón Prototype: la clonación de un objeto ya existente.

```
const originalUser = new CompleteUser({
  name,
  lastName,
  username,
  directory,
  homeFolder,
});
```

Aquí, se crea el primer objeto (originalUser) de tipo CompleteUser. Este objeto tiene las propiedades proporcionadas (nombre, apellido, nombre de usuario, directorio y carpeta principal) que son pasadas al constructor de la clase CompleteUser.

```
  const userClone = originalUser.clone();
```

El método clone() crea una nueva instancia de CompleteUser con las mismas propiedades que el objeto original, pero es un objeto distinto estos comparten los mismos valores, son diferentes en memoria. Esto lo podemos comprobar con

```
  const userClone = originalUser.clone();
  console.log(
    originalUser === userClone
      ? 'Son los mismosxdsxdxd'
      : 'DiferentesObjetos, prototype trabajando'
  );
};
```

Observador

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.

La clase TextManager representa el sujeto que se observa. Tiene la capacidad de:

- Registrar observadores con el método addListener.
- Eliminar observadores con removeListener.
- Notificar a los observadores usando notifyAll.

```
class Subject {
    constructor(){
        this.observers = [];
    }

    subscribe(o){
        this.observers.push(o);
    }

    unsubscribe(o){
        this.observers = this.observers.filter(e=> e!=0);
    }

    notify(model){
        this.observers.forEach(observer =>{
            observer.notify(model);
```

```
        })
    }
}
```

La clase TextManager hereda de Observable y añade:

currentText para almacenar el texto actual.

updateText que actualiza el texto y notifica a todos los observadores.

```
class TextManager extends Observable {
    constructor() {
        super();
        this.currentText = "";
    }

    updateText(newText) {
        this.currentText = newText;
        super.notifyAll(this);
    }
}
```

Ahora cada clase observadora reacciona de forma específica al estado del sujeto. Todos implementan un método update(subject), que es llamado cuando el sujeto notifica cambios.

```
class TextDisplayObserver {
    update(subject) {
        document.getElementById("textDisplay").innerHTML = subject.currentText;
    }
}

class TextLengthObserver {
    update(subject) {
                                document.getElementById("textLength").innerHTML        =
subject.currentText.length;
    }
}

class KeywordObserver {
    update(subject) {
        if (subject.currentText.toLowerCase().includes("cerveza")) {
            document.getElementById("keywordCheck").innerHTML = "¡Salud!";
```

```
        } else {
            document.getElementById("keywordCheck").innerHTML = ":(";
        }
    }
}
```

Esta conexión entre ellos se realiza con

Se crea una instancia del sujeto (TextManager). Después Se crean instancias de los observadores y Cada observador se registra con el sujeto usando addListener.

```
const textManager = new TextManager();
const displayObserver = new TextDisplayObserver();
const lengthObserver = new TextLengthObserver();
const keywordObserver = new KeywordObserver();

// Registrar observadores
textManager.addListener(displayObserver);
textManager.addListener(lengthObserver);
textManager.addListener(keywordObserver);
```

Y como podemos ver los divs se actualizan de acuerdo al texto que se introduce en el primer div, el segundo cuenta el total de letras que si intedroduce, y si en el primer observador= cerveza el tercero se actualiza a salud.

Builder

Builder es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

En el ejemplo que tenemos:

El constructor inicializa las propiedades del producto con valores predeterminados vacíos o neutros.

```
  constructor() {
      this.nombre = '';
      this.precio = 0;
      this.descripcion = '';
    }
```

Después cade método Estable el valor de una propiedad por la que va a pasar.

```
    establecerNombre(nombre) {
        this.nombre = nombre;
        return this;
    }

    establecerPrecio(precio) {
        this.precio = precio;
        return this;
    }

    establecerDescripcion(descripcion) {
        this.descripcion = descripcion;
        return this;
    }
```

Finalmente en este patrón el método construir se usa para generar el objeto final

```
construir() {
        return {
            nombre: this.nombre,
            precio: this.precio,
            descripcion: this.descripcion
        };
    }
```

Ahora vamos a crear el objeto utilizando la clase ConstructorProducto configurando las propiedades en pasos claros y consecutivos:

```
const producto = new ConstructorProducto()
    .establecerNombre('Laptop')
    .establecerPrecio(1000)
    .establecerDescripcion('Con esta laptop puedes nunca parar de aprender')
    .construir();
```

Adapter

El patrón Adapter permite que dos objetos con interfaces incompatibles trabajen juntos. Este patrón se aplica cuando queremos utilizar una clase existente en un sistema, pero su interfaz no cumple con las necesidades de nuestro código.
podemos implementar el patrón Adapter creando un objeto intermedio que actúe como intermediario entre la clase existente y nuestro código. Este objeto adapter tendrá una interfaz que cumpla con las necesidades de nuestro código y que se comunique con la clase existente mediante la adaptación de sus métodos y propiedades.

Para este ejemplo tenemos una clase “Auto” con un método “encender” que recibe un parámetro “llave”. Sin embargo, necesitamos que este método se ejecute sin recibir un parámetro. Para ello, podemos crear un adapter que se encargue de llamar al método “encender” con el parámetro adecuado.

```
class Auto {
    encender(llave) {
        console.log('El auto ha sido encendido con la llave:', llave);
    }
}

class AutoAdapter {
    constructor(auto) {
        this.auto = auto;
    }

    encender() {
        this.auto.encender('predeterminada');
    }
}

const miAuto = new Auto();
const adapter = new AutoAdapter(miAuto);

adapter.encender(); // El auto ha sido encendido con la llave: predeterminada
```

Como podemos creamos una clase “AutoAdapter” que recibe un objeto “auto” y que tiene un método “encender” que llama al método “encender” del objeto “auto” pasándole un parámetro predeterminado. De esta manera, podemos utilizar la clase “AutoAdapter” en nuestro código sin tener que preocuparnos por la recibir un parámetro en el método “encender”.

Decorator

El Patrón Decorador en JavaScript permite agregar responsabilidades adicionales a un objeto de forma dinámica durante la ejecución, sin necesidad de modificar su código original. Este patrón facilita la ampliación de la funcionalidad de un objeto a través de la creación de objetos decoradores que envuelven al objeto base, añadiendo nuevas funcionalidades o modificando las existentes.

El Patrón Decorador se implementa generalmente mediante clases decoradoras que siguen una estructura similar a la clase que están decorando. Estas clases invocan al objeto original y proporcionan una nueva funcionalidad sobre él.

En este ejemplo, definimos una clase base llamada “Bebida” que tiene un método precio que retorna 0. Luego, creamos dos clases decoradoras, AgregadoLeche y AgregadoChocolate, que extienden la clase base y modifican el comportamiento del método precio agregando un valor adicional. Finalmente, instanciamos estas clases decoradoras sobre el objeto base, y vemos cómo se van acumulando los cambios de funcionalidad al invocar el método precio.

Este patrón es particularmente útil cuando se necesita agregar o modificar la funcionalidad de un objeto sin alterar su estructura original.

```
// Clase base
class Bebida {
    precio() {
        return 0;
    }
}

// Clase Decoradora
class AgregadoLeche extends Bebida {
    constructor(bebida) {
        super();
        this.bebida = bebida;
    }

    precio() {
        return this.bebida.precio() + 0.5;
    }
}

// Clase Decoradora
class AgregadoChocolate extends Bebida {
    constructor(bebida) {
        super();
```

```
        this.bebida = bebida;
    }

    precio() {
        return this.bebida.precio() + 1;
    }
}

// Uso
const bebida = new Bebida();
console.log(bebida.precio()); // 0

const bebidaConLeche = new AgregadoLeche(bebida);
console.log(bebidaConLeche.precio()); // 0.5

const bebidaConLecheYChocolate = new AgregadoChocolate(bebidaConLeche);
console.log(bebidaConLecheYChocolate.precio()); // 1.5
```

STATE

El patrón State es un patrón de diseño de comportamiento que permite a un objeto cambiar su comportamiento cuando su estado interno cambia. Este patrón es especialmente útil cuando un objeto necesita cambiar su comportamiento en función de su estado.
En mi ejemplo el objeto (Pedido) puede cambiar su comportamiento dependiendo de su estado interno. El cambio de comportamiento se logra mediante la delegación a objetos de estado específicos (PedidoEnEsperaEstado, EnPreparacionEstado).

El estado inicial del pedido es PedidoEnEsperaEstado.

Este estado maneja el método prepararPedido() cambiando el estado del pedido a EnPreparacionEstado y mostrando el mensaje:
"El pedido está siendo preparado...".
pedido.servirPedido();

Ahora el estado es EnPreparacionEstado.
En este estado, el método servirPedido() cambia el estado a PedidoListoEstado y muestra:
"El pedido está listo para ser servido."
pedido.servirPedido();

El estado actual es PedidoListoEstado.
Este estado maneja servirPedido() cambiando el estado a PedidoFinalizadoEstado y mostrando:
"El pedido ha sido servido."

pedido.cancelarPedido();

El estado ahora es PedidoFinalizadoEstado.
Este estado maneja el método cancelarPedido() mostrando:
"El pedido ya ha sido finalizado, no se puede cancelar."
No ocurre ningún cambio de estado porque no se permite cancelar un pedido finalizado.

```
class EstadoPedido {
  constructor(pedido) {
    this.pedido = pedido;
  }

  prepararPedido() {
    throw new Error('Este método debe ser implementado');
  }

  servirPedido() {
    throw new Error('Este método debe ser implementado');
  }

  cancelarPedido() {
    throw new Error('Este método debe ser implementado');
  }
}

class PedidoEnEsperaEstado extends EstadoPedido {
  prepararPedido() {
    console.log('El pedido está siendo preparado...');
    this.pedido.cambiarEstado(this.pedido.enPreparacionEstado);
  }

  servirPedido() {
    console.log('El pedido aún no está listo para ser servido.');
  }

  cancelarPedido() {
    console.log('El pedido ha sido cancelado.');
    this.pedido.cambiarEstado(this.pedido.pedidoCanceladoEstado);
  }
}

class EnPreparacionEstado extends EstadoPedido {
  prepararPedido() {
```

```
    console.log('El pedido ya está siendo preparado.');
  }

  servirPedido() {
    console.log('El pedido está listo para ser servido.');
    this.pedido.cambiarEstado(this.pedido.pedidoListoEstado);
  }

  cancelarPedido() {
    console.log('El pedido no puede ser cancelado, ya está en preparación.');
  }
}

class PedidoListoEstado extends EstadoPedido {
  prepararPedido() {
    console.log('El pedido ya ha sido preparado y está listo.');
  }

  servirPedido() {
    console.log('El pedido ha sido servido.');
    this.pedido.cambiarEstado(this.pedido.pedidoFinalizadoEstado);
  }

  cancelarPedido() {
    console.log('El pedido ya ha sido servido, no puede ser cancelado.');
  }
}

class PedidoCanceladoEstado extends EstadoPedido {
  prepararPedido() {
    console.log('El pedido fue cancelado, no se puede preparar.');
  }

  servirPedido() {
    console.log('El pedido fue cancelado, no se puede servir.');
  }

  cancelarPedido() {
    console.log('El pedido ya ha sido cancelado.');
  }
}

class PedidoFinalizadoEstado extends EstadoPedido {
  prepararPedido() {
    console.log('El pedido ha sido finalizado, no se puede preparar.');
```

```
  }

  servirPedido() {
    console.log('El pedido ya ha sido servido, no se puede volver a servir.');
  }

  cancelarPedido() {
    console.log('El pedido ya ha sido finalizado, no se puede cancelar.');
  }
}

class Pedido {
  constructor() {
    this.pedidoEnEsperaEstado = new PedidoEnEsperaEstado(this);
    this.enPreparacionEstado = new EnPreparacionEstado(this);
    this.pedidoListoEstado = new PedidoListoEstado(this);
    this.pedidoCanceladoEstado = new PedidoCanceladoEstado(this);
    this.pedidoFinalizadoEstado = new PedidoFinalizadoEstado(this);

    // El estado inicial es "PedidoEnEspera"
    this.estadoActual = this.pedidoEnEsperaEstado;
  }

  cambiarEstado(estado) {
    this.estadoActual = estado;
  }

  prepararPedido() {
    this.estadoActual.prepararPedido();
  }

  servirPedido() {
    this.estadoActual.servirPedido();
  }

  cancelarPedido() {
    this.estadoActual.cancelarPedido();
  }
}

const pedido = new Pedido();

// Acciones sobre el pedido
pedido.prepararPedido();  // El pedido está siendo preparado...
pedido.servirPedido();    // El pedido está listo para ser servido.
```

```
pedido.servirPedido();    // El pedido ha sido servido.
pedido.cancelarPedido();  // El pedido ya ha sido finalizado, no se puede cancelar.
```

Strategy

es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

En este ejemplo Se crea una clase base o interfaz, en este caso **EstrategiaDescuento**. Esta clase define el método común que todas las estrategias deben implementar, en este caso aplicarDescuento(monto).

```
class EstrategiaDescuento {
  aplicarDescuento(monto) {
    throw new Error('¡Este método debe ser sobrescrito!');
  }
}
```

La implementación de las llamadas estrategias se hace de la siguiente manera:
Las clases concretas SinDescuento, DescuentoPorcentaje y DescuentoFijo extienden de EstrategiaDescuento y proporcionan una implementación específica de cómo se debe aplicar el descuento.
Es importante tomar encuenta que La clase **ContextoDescuento** es la que utiliza una de las estrategias para aplicar el descuento. Esta clase tiene una propiedad **estrategia**, que se puede asignar a cualquier estrategia concreta. Luego, en el método **aplicarDescuento**, simplemente delega la acción a la estrategia seleccionada.

```
class ContextoDescuento {
  constructor(estrategia) {
    this.estrategia = estrategia;
  }

  aplicarDescuento(monto) {
    this.estrategia.aplicarDescuento(monto);
  }
}
```

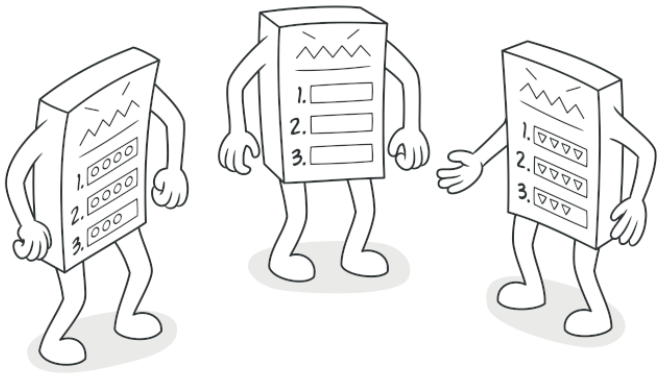
Y asi llegamos al punto clave de este patrón es que podemos cambiar de estrategia en cualquier tiempo de ejecución

Eso lo podemos ver en los logs de lo siguiente

```
const sinDescuento = new SinDescuento();
  const descuentoPorcentaje = new DescuentoPorcentaje(10);  // Descuento del 10%
  const descuentoFijo = new DescuentoFijo(50);  // Descuento de $50

  const contextoDescuento = new ContextoDescuento(sinDescuento);
  contextoDescuento.aplicarDescuento(200); // No se aplica descuento. El monto total es 200

  contextoDescuento.estrategia = descuentoPorcentaje;
  contextoDescuento.aplicarDescuento(200); // Aplicando un descuento del 10%. El monto final es 180

  contextoDescuento.estrategia = descuentoFijo;
  contextoDescuento.aplicarDescuento(200); // Aplicando un descuento de $50. El monto final es 150
```

Template:

es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.
El patrón Template Method sugiere que dividas un algoritmo en una serie de pasos, conviertas estos pasos en métodos y coloques una serie de llamadas a esos métodos dentro de un único *método plantilla*.



Clase Base (Beverage):

- Define el método prepareRecipe(), que describe los pasos generales para preparar una bebida.
- Implementa pasos comunes como hervir agua y verter en una taza.
- métodos abstractos brew() y addCondiments() para ser implementados por las subclases.
- la clase Beverage actúa como la clase base y define el método prepareRecipe() como una plantilla.

Subclases Concretas (Coffee y Tea):

    - Heredan de la clase Beverage.
    - Implementan los métodos abstractos brew() y addCondiments() para definir acciones específicas para cada bebida.

Con esto logramos que Los pasos comunes en el método prepareRecipe() se definen una vez en la clase base, reduciendo la duplicación de código.

Con este patrón podemos agregar Nuevas bebidas fácilmente creando subclases que implementen los métodos abstractos.