UT5 - TA4
Grupo 3
ADA1

Ejercicio 0

Principios SOLID que se aplican:

1. **Single Responsibility Principle (SRP):**
    - Cada decorador agrega una única responsabilidad al componente base, manteniendo cohesión en el diseño.
2. **Open/Closed Principle (OCP):**
    - Permite extender el comportamiento de un objeto sin modificar su código existente, a través de la composición de decoradores.
3. **Liskov Substitution Principle (LSP):**
    - Los decoradores son sustituibles por sus componentes base, ya que implementan la misma interfaz.

Principio violado:

- **Dependency Inversion Principle (DIP):**
    - Puede violarse si los decoradores dependen directamente de las clases concretas en lugar de depender de abstracciones. Esto puede acoplarse demasiado al componente base en lugar de a una interfaz genérica, reduciendo la flexibilidad y extensibilidad del diseño.

Ejercicio 1

Adapter para manejar de manera más eficiente y flexible la exportación de datos a diferentes formatos (JSON, XML, TXT).

```
// Interfaz común para todas las estrategias de exportación
public interface IDataExporter
{
    string ExportData(object data);
}

// Adaptador para exportar datos a JSON
public class JsonExporter : IDataExporter
{
    public string ExportData(object data)
    {
        return JsonConvert.SerializeObject(data);
    }
}

// Adaptador para exportar datos a XML
public class XmlExporter : IDataExporter
{
    public string ExportData(object data)
    {
        XmlSerializer xmlSerializer = new XmlSerializer(data.GetType());
        using (StringWriter textWriter = new StringWriter())
        {
            xmlSerializer.Serialize(textWriter, data);
            return textWriter.ToString();
        }
    }
}

// Adaptador para exportar datos a TXT
public class TxtExporter : IDataExporter
{
    public string ExportData(object data)
    {
        return data.ToString();
    }
}
```

```
// Servicio de datos que utiliza el adaptador
public class DataService
{
    private IDataExporter dataExporter;

    public void SetExporter(IDataExporter exporter)
    {
        dataExporter = exporter;
    }

    public string ExportData(object data)
    {
        if (dataExporter == null)
        {
            throw new InvalidOperationException("No se ha establecido un exportador.");
        }
        return dataExporter.ExportData(data);
    }
}
```

Ejercicio 2

Para hacer que la tienda en línea sea compatible con "SafePay" sin cambiar el código existente que utiliza "QuickPay", podemos utilizar el patrón **Adapter**. Este patrón nos permitirá adaptar la interfaz de "SafePay" a la interfaz esperada por la tienda en línea, manteniendo así la flexibilidad y la compatibilidad con ambos proveedores de servicios de pago.

```
public interface IPaymentProvider
{
    bool MakePayment(double amount, string currency);
}

public class SafePayAdapter : IPaymentProvider
{
    private SafePayService _safePayService;

    public SafePayAdapter(SafePayService safePayService)
    {
        _safePayService = safePayService;
    }

    public bool MakePayment(double amount, string currency)
    {
        // Adaptamos la llamada de MakePayment de SafePay a Transact
        // Aquí podrías implementar la lógica de adaptación necesaria
        // Por simplicidad, simularemos que siempre se completa la transacción
        _safePayService.Transact("MiCuenta", "CuentaDestino", currency, amount);
        Console.WriteLine($"Pagado {amount} {currency} usando SafePay.");
        return true; // Simular éxito
    }
}

public class OnlineStore
{
    private IPaymentProvider _paymentProvider;

    public OnlineStore(IPaymentProvider paymentProvider)
    {
        _paymentProvider = paymentProvider;
    }

    public void Checkout(double amount, string currency)
    {
        if (_paymentProvider.MakePayment(amount, currency))
```

```
        {
            Console.WriteLine("Pago exitoso!");
        }
        else
        {
            Console.WriteLine("El pago ha fallado.");
        }
    }
}
```

Ejercicio 3

Para introducir nuevas formas de envío de notificaciones (SMS, Mensajes Directos en Twitter y Mensajes en Facebook) utilizando un patrón estructural, el **Bridge** es una opción adecuada. Este patrón permite desacoplar una abstracción de su implementación, de modo que ambos puedan variar independientemente. En este caso, la abstracción sería el tipo de notificación (Email, SMS, Twitter, Facebook), y la implementación sería el método concreto de envío.

```
// Abstracción: Interface que define las operaciones de notificación
public interface INotification
{
    void Send(string message);
}

// Implementación: Clase abstracta que define el método de envío
public abstract class NotificationSender
{
    protected INotification notification;

    public NotificationSender(INotification notification)
    {
        this.notification = notification;
    }

    public abstract void SendNotification(string message);
}

// Implementación 1: Envío de notificación por correo electrónico
public class EmailNotification : INotification
{
    public void Send(string message)
    {
        Console.WriteLine($"Enviando correo electrónico: {message}");
    }
}

// Implementación 2: Envío de notificación por SMS
public class SmsNotification : INotification
{
    public void Send(string message)
    {
        Console.WriteLine($"Enviando SMS: {message}");
    }
}
```

```
// Implementación 3: Envío de notificación por Mensaje Directo en Twitter
public class TwitterNotification : INotification
{
    public void Send(string message)
    {
        Console.WriteLine($"Enviando Mensaje Directo en Twitter: {message}");
    }
}

// Implementación 4: Envío de notificación por Mensaje en Facebook
public class FacebookNotification : INotification
{
    public void Send(string message)
    {
        Console.WriteLine($"Enviando Mensaje en Facebook: {message}");
    }
}

// Abstracción refinada: Servicio que permite configurar la implementación concreta
public class NotificationService : NotificationSender
{
    public NotificationService(INotification notification) : base(notification)
    {
    }

    public override void SendNotification(string message)
    {
        notification.Send(message);
    }
}

// Abstracción refinada: Servicio que permite configurar la implementación concreta
public class NotificationService : NotificationSender
{
    public NotificationService(INotification notification) : base(notification)
    {
    }

    public override void SendNotification(string message)
    {
        notification.Send(message);
    }
}
```

Ejercicio 4

Para manejar múltiples subsistemas de un sistema de gestión de hotel de manera más eficiente y siguiendo el principio de diseño estructural, podemos utilizar el patrón **Facade**. Este patrón proporciona una interfaz unificada y simplificada para un conjunto de interfaces más complejas de subsistemas, permitiendo así que el cliente interactúe con el sistema de manera más sencilla y cohesiva.

```
// Facade que proporciona una interfaz unificada para interactuar con los subsistemas del hotel
public class HotelFacade
{
    private ReservationSystem reservationSystem;
    private RestaurantManagementSystem restaurantSystem;
    private CleaningServiceSystem cleaningSystem;

    public HotelFacade()
    {
        reservationSystem = new ReservationSystem();
        restaurantSystem = new RestaurantManagementSystem();
        cleaningSystem = new CleaningServiceSystem();
    }

    public void BookHotelServices(string roomType, string tableType, string roomNumber)
    {
        reservationSystem.ReserveRoom(roomType);
        restaurantSystem.BookTable(tableType);
        cleaningSystem.ScheduleRoomCleaning(roomNumber);
    }
}

class Program
{
    static void Main()
    {
        HotelFacade hotelFacade = new HotelFacade();

        // Utilizar la fachada para reservar servicios del hotel
        hotelFacade.BookHotelServices("Deluxe", "VIP", "101");
    }
}
```

Ejercicio 5

Para implementar un mecanismo de control de acceso a los documentos almacenados en un sistema de gestión de documentos, podemos utilizar el patrón **Proxy**. Este patrón nos permite controlar el acceso al objeto real (`Document` en este caso) mediante un objeto representante (`DocumentProxy`) que actúa como intermediario. De esta manera, podemos agregar lógica adicional antes o después de permitir el acceso al documento real, como verificar los permisos de acceso.

```
// Interfaz común para el documento y su proxy
public interface IDocument
{
    void Display();
}

// Clase concreta que representa el documento real
public class Document : IDocument
{
    private string _content;

    public Document(string content)
    {
        _content = content;
    }

    public void Display()
    {
        Console.WriteLine($"Contenido del documento: {_content}");
    }
}

// Proxy que controla el acceso al documento real
public class DocumentProxy : IDocument
{
    private Document _document;
    private readonly List<string> _authorizedUsers;

    public DocumentProxy(string content, List<string> authorizedUsers)
    {
        _document = new Document(content);
        _authorizedUsers = authorizedUsers;
    }

    public void Display()
```

```
    {
        // Lógica de verificación de permisos antes de mostrar el documento
        if (UserIsAuthorized())
        {
            _document.Display();
        }
        else
        {
            Console.WriteLine("Acceso denegado. No tienes permiso para ver este documento.");
        }
    }

    private bool UserIsAuthorized()
    {
        // Aquí se simula la verificación de permisos basada en una lista de usuarios autorizados
        string currentUser = "Alice"; // Supongamos que el usuario actual es Alice
        return _authorizedUsers.Contains(currentUser);
    }
}
```

Ejercicio 6

Para manejar la interacción entre los diferentes subsistemas de manera más eficiente y desacoplada, podemos aplicar el patrón **Facade**. Este patrón nos permitirá proporcionar una interfaz simplificada para realizar operaciones complejas que involucran varios subsistemas, como agregar productos al carrito, reducir el stock del inventario y generar facturas.

```
// Facade que proporciona una interfaz unificada para interactuar con los subsistemas de compra en línea
public class OnlineShoppingFacade
{
    private CartSystem cartSystem;
    private InventorySystem inventorySystem;
    private BillingSystem billingSystem;

    public OnlineShoppingFacade()
    {
        cartSystem = new CartSystem();
        inventorySystem = new InventorySystem();
        billingSystem = new BillingSystem();
    }

    public void ProcessOrder(string product, int quantity)
    {
        // Llamar a los subsistemas según sea necesario
        cartSystem.AddToCart(product, quantity);
        inventorySystem.ReduceStock(product, quantity);
        billingSystem.GenerateInvoice(product, quantity);
    }
}

class Program
{
    static void Main()
    {
        OnlineShoppingFacade shoppingFacade = new OnlineShoppingFacade();

        // Definir los parámetros del pedido
        string product = "Libro";
        int quantity = 2;

        // Procesar el pedido utilizando la fachada
        shoppingFacade.ProcessOrder(product, quantity);
    }
```

```
}
```

Ejericio 7

En el ejercicio proporcionado, tenemos una serie de clases que interactúan para autenticarse en la API de Twitter, hacer una solicitud y analizar la respuesta para extraer información específica. Para mejorar la estructura y flexibilidad del código, podemos aplicar el patrón **Facade** para encapsular la complejidad de estas interacciones detrás de una interfaz más simple y unificada.

```
// Facade que proporciona una interfaz simplificada para interactuar con Twitter
public class TwitterFacade
{
    private TwitterAuthenticator _authenticator;
    private TwitterApi _api;
    private TwitterDataParser _parser;

    public TwitterFacade()
    {
        _authenticator = new TwitterAuthenticator();
        _api = new TwitterApi();
        _parser = new TwitterDataParser();
    }

    public int GetPostCount(string apiKey, string apiSecret, string username)
    {
        // Autenticarse en la API de Twitter
        string accessToken = _authenticator.Authenticate(apiKey, apiSecret);

        // Hacer una solicitud a la API de Twitter para obtener información del usuario
        string jsonResponse = _api.MakeApiRequest($"https://api.twitter.com/users/{username}",
accessToken);

        // Parsear la respuesta JSON para extraer la cantidad de posts
        int postCount = _parser.ParsePostCount(jsonResponse);

        return postCount;
    }
}

class Program
{
    static void Main()
    {
        TwitterFacade twitterFacade = new TwitterFacade();
```

```
        // Definir las credenciales y el nombre de usuario
        string apiKey = "api_key";
        string apiSecret = "api_secret";
        string username = "john_doe";

        // Obtener la cantidad de posts utilizando la fachada
        int postCount = twitterFacade.GetPostCount(apiKey, apiSecret, username);

        // Mostrar la cantidad de posts
        Console.WriteLine($"Cantidad de posts del usuario {username}: {postCount}");
    }
}
```

Ejercicio 8

En el ejercicio proporcionado, tenemos una clase `ElementoTexto` que representa un elemento de texto al cual se le pueden aplicar estilos de fuente, color y decoración. Sin embargo, el diseño actual podría beneficiarse de la implementación del patrón **Builder** para facilitar la creación de objetos complejos y mejorar la legibilidad del código.

```
// Builder para construir un elemento de texto con estilos opcionales
public class ElementoTextoBuilder
{
    private ElementoTexto _elementoTexto;

    public ElementoTextoBuilder(string texto)
    {
        _elementoTexto = new ElementoTexto(texto);
    }

    public ElementoTextoBuilder AgregarEstiloFuente(string estiloFuente)
    {
        _elementoTexto.SetEstiloFuente(estiloFuente);
        return this;
    }

    public ElementoTextoBuilder AgregarColor(string color)
    {
        _elementoTexto.SetColor(color);
        return this;
    }

    public ElementoTextoBuilder AgregarDecoracion(string decoracion)
    {
        _elementoTexto.SetDecoracion(decoracion);
        return this;
    }

    public ElementoTexto Construir()
    {
        return _elementoTexto;
    }
}

class Program
{
    static void Main()
```

```
    {
        // Utilizar el builder para construir el elemento de texto
        ElementoTexto elementoTexto = new ElementoTextoBuilder("Hola, mundo!")
            .AgregarEstiloFuente("Arial")
            .AgregarColor("red")
            .AgregarDecoracion("underline")
            .Construir();

        // Obtener el texto con la apariencia personalizada
        string textoPersonalizado = elementoTexto.ObtenerTexto();

        Console.WriteLine(textoPersonalizado); // <span style="font-family: Arial; color: red;
text-decoration: underline">Hola, mundo!</span>
    }
}
```