UT5_TA3 Ignacio Villarreal, Bruno Albín, Santiago Aurrecochea, Joaquin Gasco, José Varela y Gonzalo Paz.

Patrones de diseño

Ejercicio 0

En el patrón Decorator, ¿qué principios SOLID se aplican? Justifique su respuesta.
¿Qué principios se violan, si los hubiera? Justifiquen su respuesta.

En el patrón Decorator, se aplican principalmente dos principios SOLID: el Principio de Responsabilidad Única y el Principio de Abierto/Cerrado.

SRP: El SRP establece que una clase debe tener solo una razón para cambiar. En el contexto del patrón Decorator, cada clase concreta (tanto la clase base como los decoradores) tiene una única responsabilidad específica.

OCP: El OCP establece que las clases deben estar abiertas para la extensión pero cerradas para la modificación. En el patrón Decorator, esto se logra porque se puede introducir nuevas funcionalidades añadiendo nuevos decoradores sin modificar el componente base ni los decoradores existentes. La estructura existente (componente y decoradores) no se altera cuando se añaden nuevas funcionalidades mediante decoradores.

Los principios que podrian violar son:

LSP: El patrón Decorator en sí mismo no viola el LSP, pero es importante diseñar las interfaces de manera que los decoradores puedan sustituir al componente base sin cambiar el comportamiento esperado.

DIP: En principio, el patrón Decorator no viola el DIP, ya que los decoradores dependen de la abstracción proporcionada por el componente base (o interface). Sin embargo, si la implementación del componente base cambia significativamente, puede requerir modificaciones en los decoradores existentes, lo cual podría considerarse una violación potencial en términos de dependencias más rígidas.

Para cada uno de los siguientes ejercicios, en equipo:

- Determine que patrón puede resolver el problema de una forma más eficiente.
- Agregue las clases, interfaces, métodos que considere necesarios para remediar la situación.

Ejercicio 1

```
public class DataService
{
```

```
    public string ExportAsJson(object data)
    {
        return JsonConvert.SerializeObject(data);
    }

    public string ExportAsXml(object data)
    {
        XmlSerializer xmlSerializer = new XmlSerializer(data.GetType());
        using (StringWriter textWriter = new StringWriter())
        {
            xmlSerializer.Serialize(textWriter, data);
            return textWriter.ToString();
        }
    }

    public string ExportAsTxt(object data)
    {
        return data.ToString();
    }
}

class Program
{
    static void Main()
    {
        // Datos a exportar
        var data = new { Name = "Juancito", Age = 30 };

        // Crear servicio de datos
        DataService dataService = new DataService();

        // Exportar y mostrar datos en formato JSON
        Console.WriteLine("Datos en formato JSON:");
        Console.WriteLine(dataService.ExportAsJson(data));

        // Exportar y mostrar datos en formato XML
        Console.WriteLine("\nDatos en formato XML:");
        Console.WriteLine(dataService.ExportAsXml(data));

        // Exportar y mostrar datos en formato TXT
        Console.WriteLine("\nDatos en formato TXT:");
        Console.WriteLine(dataService.ExportAsTxt(data));
    }
}
```

El patron Adapter nos permitirá definir una interfaz común para la exportación de datos y luego crear diferentes implementaciones de esa interfaz para cada formato (JSON, XML, TXT). De esta forma, podemos agregar nuevos formatos de exportación sin modificar la clase `DataService`.

```
public interface IDataExporter
{
    string Export(object data);
}

public class JsonDataExporter : IDataExporter
{
    public string Export(object data)
    {
        return JsonConvert.SerializeObject(data);
    }
}

public class XmlDataExporter : IDataExporter
{
    public string Export(object data)
    {
        XmlSerializer xmlSerializer = new XmlSerializer(data.GetType());
        using (StringWriter textWriter = new StringWriter())
        {
            xmlSerializer.Serialize(textWriter, data);
            return textWriter.ToString();
        }
    }
}

public class TxtDataExporter : IDataExporter
{
    public string Export(object data)
    {
        return data.ToString();
    }
}

public class DataService
{
    private readonly IDataExporter _dataExporter;

    public DataService(IDataExporter dataExporter)
    {
        _dataExporter = dataExporter;
```

```
    }

    public string Export(object data)
    {
        return _dataExporter.Export(data);
    }
}

class Program
{
    static void Main()
    {
        // Datos a exportar
        var data = new { Name = "Juancito", Age = 30 };

        // Crear y usar servicio de datos con exportador JSON
        DataService jsonDataService = new DataService(new
JsonDataExporter());
        Console.WriteLine("Datos en formato JSON:");
        Console.WriteLine(jsonDataService.Export(data));

        // Crear y usar servicio de datos con exportador XML
        DataService xmlDataService = new DataService(new
XmlDataExporter());
        Console.WriteLine("\nDatos en formato XML:");
        Console.WriteLine(xmlDataService.Export(data));

        // Crear y usar servicio de datos con exportador TXT
        DataService txtDataService = new DataService(new
TxtDataExporter());
        Console.WriteLine("\nDatos en formato TXT:");
        Console.WriteLine(txtDataService.Export(data));
    }
}
```

La Interfaz `IDataExporter` define un método `Export` que toma un objeto y devuelve una cadena. Esta interfaz es implementada por cada adaptador específico de formato.

Los adaptadores (`JsonDataExporter`, `XmlDataExporter`, `TxtDataExporter`) implementan la interfaz `IDataExporter` y define el método `Export` para exportar los datos en el formato específico.

La clase `DataService` toma un objeto `IDataExporter` en su constructor. El método `Export` llama al método `Export` del `IDataExporter` proporcionado.

En el `Main` se crea instancias de `DataService` con los diferentes adaptadores y exporta los datos en los distintos formatos.

Ejercicio 2

Imagina que eres un desarrollador en una tienda en línea. Tu tienda actualmente solo soporta pagos a través de un proveedor de servicios de pago llamado "QuickPay". Ahora, la tienda quiere soportar un nuevo proveedor de servicios de pago llamado "SafePay". El problema es que "SafePay" tiene una interfaz de programación completamente diferente a la de "QuickPay".

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

public class QuickPayService : IQuickPay
{
    public bool MakePayment(double amount, string currency)
    {
        Console.WriteLine($"Pagado {amount} {currency} usando
QuickPay.");
        return true; // Simular éxito
    }
}

public class OnlineStore
{
    private IQuickPay _paymentService;

    public OnlineStore(IQuickPay paymentService)
    {
        _paymentService = paymentService;
    }

    public void Checkout(double amount, string currency)
    {
        if (_paymentService.MakePayment(amount, currency))
        {
            Console.WriteLine("Pago exitoso!");
        }
        else
        {
            Console.WriteLine("El pago ha fallado.");
        }
```

```
    }
}

public class SafePayService
{
    public void Transact(string fromAccount, string toAccount, string
currencyType, double amount)
    {
        Console.WriteLine($"Transfiriendo {amount} {currencyType} de
{fromAccount} a {toAccount} usando SafePay.");
    }
}
```

El objetivo es hacer que la tienda en línea sea compatible con "SafePay" sin cambiar el código existente de las clases que procesan pagos con "QuickPay".

Para hacer que la tienda en línea sea compatible con "SafePay" sin cambiar el código existente de las clases que procesan pagos con "QuickPay", podemos utilizar el patrón de diseño Adapter. Este patrón nos permitirá crear un adaptador que convierta la interfaz de "SafePay" a la interfaz esperada por "QuickPay".

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

public class QuickPayAdapter : IPaymentService
{
    private readonly IQuickPay _quickPay;

    public QuickPayAdapter(IQuickPay quickPay)
    {
        _quickPay = quickPay;
    }

    public bool MakePayment(double amount, string currency)
    {
        return _quickPay.MakePayment(amount, currency);
    }
}

public class SafePayAdapter : IPaymentService
{
    private readonly SafePayService _safePayService;
```

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

    public bool MakePayment(double amount, string currency)
    {
        // Adaptar la llamada al método Transact de SafePay
        _safePayService.Transact("tienda", "cliente", currency, amount);
        return true; // Asumir éxito
    }
}

public class OnlineStore
{
    private readonly IPaymentService _paymentService;

    public OnlineStore(IPaymentService paymentService)
    {
        _paymentService = paymentService;
    }

    public void Checkout(double amount, string currency)
    {
        if (_paymentService.MakePayment(amount, currency))
        {
            Console.WriteLine("Pago exitoso!");
        }
        else
        {
            Console.WriteLine("El pago ha fallado.");
        }
    }
}

class Program
{
    static void Main()
    {
        // Usar QuickPay
        IQuickPay quickPay = new QuickPayService();
        IPaymentService quickPayAdapter = new QuickPayAdapter(quickPay);
        OnlineStore storeWithQuickPay = new
OnlineStore(quickPayAdapter);
        storeWithQuickPay.Checkout(100.0, "USD");
```

```
        // Usar SafePay
        SafePayService safePay = new SafePayService();
        IPaymentService safePayAdapter = new SafePayAdapter(safePay);
        OnlineStore storeWithSafePay = new OnlineStore(safePayAdapter);
        storeWithSafePay.Checkout(200.0, "EUR");
    }
}
```

La interfaz `IPaymentService` define un método `MakePayment` que toma un monto y una moneda y devuelve un booleano. Esta interfaz es implementada por los adaptadores de `QuickPay` y `SafePay`.

El adaptador `QuickPayAdapter` implementa `IPaymentService` y convierte la llamada a `MakePayment` a una llamada a `QuickPay`.

El adaptador `SafePayAdapter` implementa `IPaymentService` y convierte la llamada a `MakePayment` a una llamada a `Transact` de `SafePay`, adaptando los parámetros según sea necesario.

La Clase `OnlineStore` toma un objeto `IPaymentService` en su constructor. El método `Checkout` llama al método `MakePayment` del `IPaymentService` proporcionado.

El método `Main` crea instancias de `OnlineStore` con los adaptadores de `QuickPay` y `SafePay` y realiza el proceso de pago con ambos.

Ejercicio 3

Una aplicación que permite a los usuarios recibir notificaciones. Inicialmente, la aplicación solo soportaba notificaciones por correo electrónico. Ahora, se planea introducir nuevas formas de envío de notificaciones como SMS, Mensajes Directos en Twitter y Mensajes en Facebook.

Actualmente tienes la siguiente estructura en tu aplicación:

```
public abstract class Notification
{
    public abstract void Send(string message);
}

public class EmailNotification : Notification
{
    public override void Send(string message)
    {
        Console.WriteLine($"Enviando correo electrónico: {message}");
```

```
    }
}
```

Refactoriza el código existente e introduce nuevas clases o interfaces si es necesario para soportar las nuevas formas de notificación (SMS, Mensajes Directos en Twitter, Mensajes en Facebook).

Para soportar nuevas formas de notificación como SMS, Mensajes Directos en Twitter y Mensajes en Facebook, podemos utilizar el patrón de diseño Bridge. Este patrón nos permitirá separar la abstracción (la notificación) de su implementación (el método de envío), lo que facilitará la adición de nuevas formas de notificación sin cambiar el código existente.

```
public interface IMessageSender
{
    void SendMessage(string message);
}

public class EmailSender : IMessageSender
{
    public void SendMessage(string message)
    {
        Console.WriteLine($"Enviando correo electrónico: {message}");
    }
}

public class SmsSender : IMessageSender
{
    public void SendMessage(string message)
    {
        Console.WriteLine($"Enviando SMS: {message}");
    }
}

public class TwitterSender : IMessageSender
{
    public void SendMessage(string message)
    {
        Console.WriteLine($"Enviando mensaje directo en Twitter:
{message}");
    }
}

public class FacebookSender : IMessageSender
{
    public void SendMessage(string message)
```

```
    {
        Console.WriteLine($"Enviando mensaje en Facebook: {message}");
    }
}

public abstract class Notification
{
    protected IMessageSender _messageSender;

    protected Notification(IMessageSender messageSender)
    {
        _messageSender = messageSender;
    }

    public abstract void Send(string message);
}

public class UserNotification : Notification
{
    public UserNotification(IMessageSender messageSender) :
base(messageSender) { }

    public override void Send(string message)
    {
        _messageSender.SendMessage(message);
    }
}

class Program
{
    static void Main()
    {
        IMessageSender emailSender = new EmailSender();
        Notification emailNotification = new
UserNotification(emailSender);
        emailNotification.Send("Hola, esto es un correo electrónico.");

        IMessageSender smsSender = new SmsSender();
        Notification smsNotification = new UserNotification(smsSender);
        smsNotification.Send("Hola, esto es un SMS.");

        IMessageSender twitterSender = new TwitterSender();
        Notification twitterNotification = new
UserNotification(twitterSender);
        twitterNotification.Send("Hola, esto es un mensaje directo en
Twitter.");
```

```
        IMessageSender facebookSender = new FacebookSender();
        Notification facebookNotification = new
UserNotification(facebookSender);
        facebookNotification.Send("Hola, esto es un mensaje en
Facebook.");
    }
}
```

La interfaz `IMessageSender` define un método `SendMessage` que toma un mensaje y lo envía. Esta interfaz es implementada por las diferentes clases que representan los métodos de envío (correo electrónico, SMS, Twitter, Facebook).

Las clases de envío de mensajes (`EmailSender`, `SmsSender`, `TwitterSender`, `FacebookSender`) implementan `IMessageSender` y define el método `SendMessage` para enviar el mensaje de la manera correspondiente.

La clase `Notification` es una clase abstracta que contiene una referencia a un `IMessageSender`. El método `Send` es abstracto y será implementado por clases concretas.

La clase `UserNotification` hereda de `Notification` y usa el `IMessageSender` proporcionado para enviar el mensaje.

En el `Main` se crean instancias de `UserNotification` con los diferentes `IMessageSender` y envía mensajes utilizando cada método de envío.

Ejercicio 4

Un sistema de gestión de hotel que tiene múltiples subsistemas como el sistema de reservas, sistema de gestión de restaurantes, sistema de gestión de limpieza, etc. Cada subsistema tiene su propia interfaz compleja y son independientes entre sí.

Actualmente tienes la siguiente estructura en tu aplicación:

```
public class ReservationSystem
{
    public void ReserveRoom(string roomType)
    {
        Console.WriteLine($"Reservando una habitación de tipo:
{roomType}");
    }
}
```

```
public class RestaurantManagementSystem
{
    public void BookTable(string tableType)
    {
        Console.WriteLine($"Reservando una mesa de tipo: {tableType}");
    }
}

public class CleaningServiceSystem
{
    public void ScheduleRoomCleaning(string roomNumber)
    {
        Console.WriteLine($"Programando la limpieza para la habitación
número: {roomNumber}");
    }
}

class Program
{
    static void Main()
    {
        ReservationSystem reservationSystem = new ReservationSystem();
        reservationSystem.ReserveRoom("Deluxe");

        RestaurantManagementSystem restaurantSystem = new
RestaurantManagementSystem();
        restaurantSystem.BookTable("VIP");

        CleaningServiceSystem cleaningSystem = new
CleaningServiceSystem();
        cleaningSystem.ScheduleRoomCleaning("101");

        //... Do stuff... reservationSystem + restaurantSystem +
cleaningSystem
    }
}
```

Para gestionar la complejidad de múltiples subsistemas en un sistema de gestión de hotel, podemos utilizar el patrón de diseño Facade. Este patrón proporciona una interfaz simplificada para interactuar con un conjunto de interfaces en un subsistema, haciendo que el sistema sea más fácil de usar.

```
public class HotelFacade
{
    private readonly ReservationSystem _reservationSystem;
```

```
    private readonly RestaurantManagementSystem _restaurantSystem;
    private readonly CleaningServiceSystem _cleaningSystem;

    public HotelFacade()
    {
        _reservationSystem = new ReservationSystem();
        _restaurantSystem = new RestaurantManagementSystem();
        _cleaningSystem = new CleaningServiceSystem();
    }

    public void ReserveRoom(string roomType)
    {
        _reservationSystem.ReserveRoom(roomType);
    }

    public void BookTable(string tableType)
    {
        _restaurantSystem.BookTable(tableType);
    }

    public void ScheduleRoomCleaning(string roomNumber)
    {
        _cleaningSystem.ScheduleRoomCleaning(roomNumber);
    }
}

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

        hotelFacade.ReserveRoom("Deluxe");
        hotelFacade.BookTable("VIP");
        hotelFacade.ScheduleRoomCleaning("101");

        //... Realizar operaciones adicionales usando hotelFacade
    }
}
```

La clase `HotelFacade` esta clase encapsula las instancias de `ReservationSystem`, `RestaurantManagementSystem` y `CleaningServiceSystem`. Proporciona métodos simplificados (`ReserveRoom`, `BookTable`, `ScheduleRoomCleaning`) que internamente llaman a los métodos de los subsistemas correspondientes.

El `Main` en lugar de interactuar directamente con cada subsistema, se crea una instancia de `HotelFacade` y se utilizan sus métodos para realizar las operaciones necesarias.

Ejercicio 5

Un sistema de gestión de documentos y se te ha pedido que implementes un mecanismo de control de acceso a los documentos almacenados. Los usuarios solo deben poder acceder a un documento si tienen el permiso adecuado.

Tienes la siguiente clase que representa un documento:

```
public class Document
{
    private string _content;

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

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

class Program
{
    static void Main()
    {
        Document document = new Document("Este es un documento
importante.");
        document.Display();
    }
}
```

Para implementar un mecanismo de control de acceso a los documentos, podemos utilizar el patrón de diseño Proxy. Este patrón nos permitirá controlar el acceso a los documentos a través de una clase intermediaria que verificará los permisos antes de permitir el acceso al documento real.

```
public interface IDocument
{
    void Display();
}
```

```
public class Document : IDocument
{
    private string _content;

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

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

public class DocumentProxy : IDocument
{
    private Document _document;
    private string _userRole;

    public DocumentProxy(string content, string userRole)
    {
        _document = new Document(content);
        _userRole = userRole;
    }

    public void Display()
    {
        if (HasAccess())
        {
            _document.Display();
        }
        else
        {
            Console.WriteLine("Acceso denegado. No tiene los permisos
necesarios para ver este documento.");
        }
    }

    private bool HasAccess()
    {
        // Verificar si el usuario tiene el permiso adecuado
        // Aquí se puede agregar la lógica de verificación de permisos
según sea necesario
        return _userRole == "Admin" || _userRole == "Editor";
```

```
    }
}

class Program
{
    static void Main()
    {
        IDocument document = new DocumentProxy("Este es un documento
importante.", "Admin");
        document.Display();

        IDocument document2 = new DocumentProxy("Este es un documento
importante.", "Viewer");
        document2.Display();
    }
}
```

La interfaz `IDocument` define el método `Display` que será implementado por `Document` y `DocumentProxy`.

La clase `Document` implementa `IDocument` y define el método `Display` para mostrar el contenido del documento.

La clase `DocumentProxy` implementa `IDocument` y controla el acceso al documento. En el constructor, toma el contenido del documento y el rol del usuario. El método `Display` verifica si el usuario tiene el permiso adecuado antes de delegar la llamada al método `Display` de `Document`. La lógica de permisos se implementa en el método `HasAccess`.

En el `Main` se crean instancias de `DocumentProxy` con diferentes roles de usuario y llama al método `Display` para mostrar el contenido del documento si el usuario tiene los permisos adecuados.

Ejercicio 6

```
public class CartSystem
{
    public void AddToCart(string product, int quantity)
    {
        // Simular llamada a la API del sistema de carrito de compras
        Console.WriteLine($"API llamada: Agregando {quantity} de
{product} al carrito.");
    }
}
```

```
public class InventorySystem
{
    public void ReduceStock(string product, int quantity)
    {
        // Simular llamada a la API del sistema de inventario
        Console.WriteLine($"API llamada: Reduciendo el stock de
{product} en {quantity}.");
    }
}

public class BillingSystem
{
    public void GenerateInvoice(string product, int quantity)
    {
        // Simular llamada a la API del sistema de facturación
        Console.WriteLine($"API llamada: Generando factura para
{quantity} de {product}.");
    }
}

class Program
{
    static void Main()
    {
        // Crear instancias de cada subsistema
        CartSystem cartSystem = new CartSystem();
        InventorySystem inventorySystem = new InventorySystem();
        BillingSystem billingSystem = new BillingSystem();

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

        // Llamar a la API del sistema de carrito de compras para
agregar el producto al carrito
        cartSystem.AddToCart(product, quantity);

        // Llamar a la API del sistema de inventario para reducir el
stock
        inventorySystem.ReduceStock(product, quantity);

        // Llamar a la API del sistema de facturación para generar la
factura
        billingSystem.GenerateInvoice(product, quantity);
    }
}
```

Para gestionar la interacción entre los diferentes subsistemas (carrito de compras, inventario y facturación) en un sistema de gestión de compras, podemos utilizar el patrón de diseño Facade. Este patrón proporciona una interfaz simplificada para interactuar con estos subsistemas, haciendo que el sistema sea más fácil de usar y mantener.

```
public class OrderFacade
{
    private readonly CartSystem _cartSystem;
    private readonly InventorySystem _inventorySystem;
    private readonly BillingSystem _billingSystem;

    public OrderFacade()
    {
        _cartSystem = new CartSystem();
        _inventorySystem = new InventorySystem();
        _billingSystem = new BillingSystem();
    }

    public void PlaceOrder(string product, int quantity)
    {
        _cartSystem.AddToCart(product, quantity);
        _inventorySystem.ReduceStock(product, quantity);
        _billingSystem.GenerateInvoice(product, quantity);
    }
}

class Program
{
    static void Main()
    {
        // Crear una instancia del OrderFacade
        OrderFacade orderFacade = new OrderFacade();

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

        // Realizar el pedido usando la fachada
        orderFacade.PlaceOrder(product, quantity);
    }
}
```

La clase `OrderFacade` encapsula las instancias de `CartSystem`, `InventorySystem`, y `BillingSystem`. Proporciona un método simplificado `PlaceOrder` que realiza todas las operaciones necesarias para procesar un pedido (agregar al carrito, reducir el stock, y generar la factura) en una única llamada.

En el `Main` en lugar de interactuar directamente con cada subsistema, se crea una instancia de `OrderFacade` y se utiliza su método `PlaceOrder` para procesar el pedido.

Ejercicio 7

```
public class TwitterAuthenticator
{
    public string Authenticate(string apiKey, string apiSecret)
    {
        // Lógica para autenticarse en la API de Twitter y obtener un
token de acceso.
        // Para simplificar el ejemplo, vamos a suponer que siempre
recibimos un token "ABC123".
        Console.WriteLine("Autenticando en la API de Twitter...");
        return "ABC123";
    }
}

public class TwitterApi
{
    public string MakeApiRequest(string endpoint, string accessToken)
    {
        // Lógica para hacer una solicitud a la API de Twitter.
        // Para simplificar el ejemplo, vamos a suponer que siempre
recibimos un JSON de respuesta.
        Console.WriteLine($"Haciendo una solicitud a {endpoint}...");
        return "{\"user\": \"john_doe\", \"post_count\": 42}";
    }
}

public class TwitterDataParser
{
    public int ParsePostCount(string jsonResponse)
    {
        // Lógica para parsear el JSON y extraer la cantidad de posts.
        // Para simplificar, vamos a suponer que siempre recibimos 42.
        Console.WriteLine("Parseando la respuesta de la API...");
        return 42;
    }
}

class Program
{
    static void Main()
    {
```

```
        // Crear instancias de las clases necesarias
        TwitterAuthenticator authenticator = new TwitterAuthenticator();
        TwitterApi twitterApi = new TwitterApi();
        TwitterDataParser dataParser = new TwitterDataParser();

        // Autenticarse en la API de Twitter
        string accessToken = authenticator.Authenticate("api_key",
"api_secret");

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

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

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

Para simplificar la interacción con la API de Twitter y mejorar la gestión del código, podemos utilizar el patrón de diseño Facade. Este patrón nos permitirá encapsular la lógica de autenticación, solicitud a la API y el análisis de datos en una única interfaz simplificada.

```
public class TwitterFacade
{
    private readonly TwitterAuthenticator _authenticator;
    private readonly TwitterApi _twitterApi;
    private readonly TwitterDataParser _dataParser;

    public TwitterFacade()
    {
        _authenticator = new TwitterAuthenticator();
        _twitterApi = new TwitterApi();
        _dataParser = new TwitterDataParser();
    }

    public int GetPostCount(string apiKey, string apiSecret, string
endpoint)
    {
        // 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 = _twitterApi.MakeApiRequest(endpoint,
accessToken);

        // Parsear la respuesta JSON para extraer la cantidad de posts
        return _dataParser.ParsePostCount(jsonResponse);
    }
}

class Program
{
    static void Main()
    {
        // Crear una instancia de TwitterFacade
        TwitterFacade twitterFacade = new TwitterFacade();

        // Definir las credenciales y el endpoint
        string apiKey = "api_key";
        string apiSecret = "api_secret";
        string endpoint = "https://api.twitter.com/users/john_doe";

        // Obtener la cantidad de posts del usuario
        int postCount = twitterFacade.GetPostCount(apiKey, apiSecret,
endpoint);

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

La clase `TwitterFacade` encapsula las instancias de `TwitterAuthenticator`, `TwitterApi`, y `TwitterDataParser`. Proporciona un método simplificado `GetPostCount` que realiza todas las operaciones necesarias para obtener la cantidad de posts de un usuario de Twitter (autenticación, solicitud a la API y análisis de datos) en una única llamada.

El método `Main` en lugar de interactuar directamente con cada subsistema, se crea una instancia de `TwitterFacade` y se utiliza su método `GetPostCount` para obtener la cantidad de posts del usuario.

Ejercicio 8

```
public class ElementoTexto
{
    private string _texto;
    private string _estiloFuente;
    private string _color;
    private string _decoracion;

    public ElementoTexto(string texto)
    {
        _texto = texto;
    }

    public void SetEstiloFuente(string estiloFuente)
    {
        _estiloFuente = estiloFuente;
    }

    public void SetColor(string color)
    {
        _color = color;
    }

    public void SetDecoracion(string decoracion)
    {
        _decoracion = decoracion;
    }

    public string ObtenerTexto()
    {
        string textoDecorado = _texto;

        if (!string.IsNullOrEmpty(_estiloFuente))
        {
            textoDecorado = $"<span
style=\"font-family:{_estiloFuente}\">{textoDecorado}</span>";
        }

        if (!string.IsNullOrEmpty(_color))
        {
            textoDecorado = $"<span
style=\"color:{_color}\">{textoDecorado}</span>";
        }

        if (!string.IsNullOrEmpty(_decoracion))
```

```
        {
            textoDecorado = $"<span
style=\"text-decoration:{_decoracion}\">{textoDecorado}</span>";
        }

        return textoDecorado;
    }
}

class Program
{
    static void Main()
    {
        // Crear una instancia de un elemento de texto
        ElementoTexto elementoTexto = new ElementoTexto("Hola, mundo!");

        // Personalizar la apariencia del elemento de texto
        elementoTexto.SetEstiloFuente("Arial");
        elementoTexto.SetColor("red");
        elementoTexto.SetDecoracion("underline");

        // 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>
    }
}
```

Para mejorar la flexibilidad y la capacidad de expansión del código, podemos utilizar el patrón de diseño Decorator. Este patrón nos permite agregar responsabilidades a un objeto de manera dinámica. En este caso, podemos usar el patrón Decorator para agregar estilos y decoraciones al texto sin necesidad de modificar la clase base `ElementoTexto`.

```
public interface IElementoTexto
{
    string ObtenerTexto();
}

public class ElementoTexto : IElementoTexto
{
    private string _texto;

    public ElementoTexto(string texto)
    {
```

```
        _texto = texto;
    }

    public string ObtenerTexto()
    {
        return _texto;
    }
}

public abstract class TextoDecorator : IElementoTexto
{
    protected IElementoTexto _elementoTexto;

    public TextoDecorator(IElementoTexto elementoTexto)
    {
        _elementoTexto = elementoTexto;
    }

    public abstract string ObtenerTexto();
}

public class EstiloFuenteDecorator : TextoDecorator
{
    private string _estiloFuente;

    public EstiloFuenteDecorator(IElementoTexto elementoTexto, string
estiloFuente) : base(elementoTexto)
    {
        _estiloFuente = estiloFuente;
    }

    public override string ObtenerTexto()
    {
        return $"<span
style=\"font-family:{_estiloFuente}\">{_elementoTexto.ObtenerTexto()}</s
pan>";
    }
}

public class ColorDecorator : TextoDecorator
{
    private string _color;

    public ColorDecorator(IElementoTexto elementoTexto, string color) :
base(elementoTexto)
    {
```

```
        _color = color;
    }

    public override string ObtenerTexto()
    {
        return $"<span
style=\"color:{_color}\">{_elementoTexto.ObtenerTexto()}</span>";
    }
}

public class DecoracionDecorator : TextoDecorator
{
    private string _decoracion;

    public DecoracionDecorator(IElementoTexto elementoTexto, string
decoracion) : base(elementoTexto)
    {
        _decoracion = decoracion;
    }

    public override string ObtenerTexto()
    {
        return $"<span
style=\"text-decoration:{_decoracion}\">{_elementoTexto.ObtenerTexto()}<
/span>";
    }
}

class Program
{
    static void Main()
    {
        // Crear una instancia de un elemento de texto
        IElementoTexto elementoTexto = new ElementoTexto("Hola,
mundo!");

        // Aplicar los decoradores para personalizar la apariencia del
texto
        elementoTexto = new EstiloFuenteDecorator(elementoTexto,
"Arial");
        elementoTexto = new ColorDecorator(elementoTexto, "red");
        elementoTexto = new DecoracionDecorator(elementoTexto,
"underline");

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

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

La interfaz `IElementoTexto` define el método `ObtenerTexto` que será implementado por `ElementoTexto` y los decoradores.

La clase `ElementoTexto` implementa `IElementoTexto` y define el método `ObtenerTexto` para retornar el texto simple.

La clase abstracta `TextoDecorator` implementa `IElementoTexto` y sirve como base para todos los decoradores. Mantiene una referencia a un objeto `IElementoTexto` que se decorará.

Cada decorador concreto (`EstiloFuenteDecorator`, `ColorDecorator`, `DecoracionDecorator`) implementa el método `ObtenerTexto` para agregar su propio estilo al texto retornado por el objeto `IElementoTexto` que decora.

En el `Main` se crean instancias de `ElementoTexto` y aplica los decoradores para personalizar la apariencia del texto. Finalmente, obtiene y muestra el texto decorado.