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

Patrones de diseño

Ejercicio 0

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

En el patrón Chain of Responsibility, los principios SOLID que se aplican son los siguientes:

SRP: Se promueve el principio de responsabilidad única al separar las responsabilidades de manejar una solicitud en diferentes clases o componentes. Cada handler en la cadena tiene la responsabilidad de decidir si puede manejar la solicitud y, si no puede, pasarla al siguiente handler. Esto permite que cada handler tenga un único motivo para cambiar: cómo procesa o pasa la solicitud.

OCP: Se facilita la extensión y la modificación sin modificar el código existente, cumpliendo así con el principio de open/closed. Se pueden agregar nuevos handlers a la cadena sin necesidad de alterar los existentes. Cada handler puede ser extendido o reemplazado por uno nuevo sin afectar el comportamiento de los otros.

Posibles Violaciones y Cómo Evitarlas:

LSP: Si los handlers en la cadena no están correctamente diseñados para manejar las solicitudes de manera consistente, podría haber violaciones del LSP. Por ejemplo, si un handler en la cadena asume una responsabilidad diferente o no maneja adecuadamente la solicitud, podría romper la expectativa de comportamiento del patrón.

ISP: En algunos casos, si los handlers en la cadena implementan una interfaz que contiene métodos innecesarios para algunos handlers, podría considerarse una violación del ISP. Esto podría llevar a handlers que implementan métodos que no utilizan, lo cual es una señal de una mala separación de responsabilidades.

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

Este código es bastante básico y no es escalable. Por ejemplo, si queremos notificar a más estudiantes o si queremos que los estudiantes se suscriban a las notificaciones de diferentes exámenes, este diseño no sería adecuado.

```
class Program
{
    static void Main()
    {
        var exam = new Exam("Matemáticas");
        var student1 = new Student("Alice");
        var student2 = new Student("Bob");

        exam.NotifyStudents(student1, student2);
    }
}

class Exam
{
    public string Subject { get; }

    public Exam(string subject)
    {
        Subject = subject;
    }

    public void NotifyStudents(params Student[] students)
    {
        foreach (var student in students)
        {
            Console.WriteLine($"{student.Name}, hay un nuevo examen de
{Subject}!");
        }
    }
}

class Student
{
    public string Name { get; }

    public Student(string name)
    {
        Name = name;
    }
}
```

Para resolver el problema de notificar a los estudiantes de una manera más escalable y flexible, podemos usar el patrón Observer. Este patrón permitirá que los estudiantes se suscriban a notificaciones de diferentes exámenes y maneja de manera eficiente la adición de más estudiantes.

```
// Interfaz IObserver
public interface IObserver
{
    void Update(string subject);
}

// Clase concreta Student implementando IObserver
public class Student : IObserver
{
    public string Name { get; }

    public Student(string name)
    {
        Name = name;
    }

    public void Update(string subject)
    {
        Console.WriteLine($"{Name}, hay un nuevo examen de {subject}!");
    }
}

// Interfaz IObservable
public interface IObservable
{
    void Subscribe(IObserver observer);
    void Unsubscribe(IObserver observer);
    void NotifyObservers();
}

// Clase concreta Exam implementando IObservable
public class Exam : IObservable
{
    public string Subject { get; }
    private readonly List<IObserver> _observers;

    public Exam(string subject)
    {
        Subject = subject;
        _observers = new List<IObserver>();
    }

    public void Subscribe(IObserver observer)
    {
        _observers.Add(observer);
    }
```

```
    public void Unsubscribe(IObserver observer)
    {
        _observers.Remove(observer);
    }

    public void NotifyObservers()
    {
        foreach (var observer in _observers)
        {
            observer.Update(Subject);
        }
    }
}

class Program
{
    static void Main()
    {
        var exam = new Exam("Matemáticas");
        var student1 = new Student("Alice");
        var student2 = new Student("Bob");

        exam.Subscribe(student1);
        exam.Subscribe(student2);

        exam.NotifyObservers();
    }
}
```

La interfaz `IObserver` define el método `Update` que será implementado por los observadores para recibir notificaciones.

La clase `Student` implementa la interfaz `IObserver` y el método `Update` que recibe las notificaciones del examen.

La interfaz `IObservable` define los métodos `Subscribe`, `Unsubscribe` y `NotifyObservers` que serán implementados por las clases que puedan ser observadas.

La clase `Exam` implementa la interfaz `IObservable` y mantiene una lista de observadores. Implementa los métodos `Subscribe` para agregar observadores, `Unsubscribe` para eliminarlos y `NotifyObservers` para notificar a todos los observadores registrados.

La clase `Program` crea una instancia de `Exam` y varias instancias de `Student`. Los estudiantes se suscriben al examen. Se llama al método `NotifyObservers` del examen para notificar a todos los estudiantes suscritos.

Ejercicio 2

```
class Program
{
    static void Main()
    {
        var gameCharacter = new GameCharacter
        {
            Name = "John",
            Health = 100,
            Mana = 50
        };

        Console.WriteLine("Estado inicial:");
        gameCharacter.DisplayStatus();

        Console.WriteLine("\nGuardando estado...");
        var savedState = gameCharacter;

        Console.WriteLine("\nCambiando estados...");
        gameCharacter.Health -= 30;
        gameCharacter.Mana += 20;
        gameCharacter.DisplayStatus();

        Console.WriteLine("\nRestaurando estado...");
        gameCharacter = savedState;
        gameCharacter.DisplayStatus();
    }
}

class GameCharacter
{
    public string Name { get; set; }
    public int Health { get; set; }
    public int Mana { get; set; }

    public void DisplayStatus()
    {
        Console.WriteLine($"{Name} tiene {Health} de salud y {Mana} de
mana.");
    }
```

```
}
```

Para este ejercicio, el patrón Memento es adecuado para guardar y restaurar el estado de un objeto. El patrón Memento permite capturar y externalizar el estado interno de un objeto sin violar la encapsulación, para que el objeto pueda volver a este estado más tarde.

```
class Program
{
    static void Main()
    {
        var gameCharacter = new GameCharacter
        {
            Name = "John",
            Health = 100,
            Mana = 50
        };

        Console.WriteLine("Estado inicial:");
        gameCharacter.DisplayStatus();

        Console.WriteLine("\nGuardando estado...");
        var savedState = gameCharacter.SaveState();

        Console.WriteLine("\nCambiando estados...");
        gameCharacter.Health -= 30;
        gameCharacter.Mana += 20;
        gameCharacter.DisplayStatus();

        Console.WriteLine("\nRestaurando estado...");
        gameCharacter.RestoreState(savedState);
        gameCharacter.DisplayStatus();
    }
}

// Clase Memento
public class GameCharacterMemento
{
    public string Name { get; }
    public int Health { get; }
    public int Mana { get; }

    public GameCharacterMemento(string name, int health, int mana)
    {
        Name = name;
        Health = health;
```

```
        Mana = mana;
    }
}

// Clase GameCharacter
public class GameCharacter
{
    public string Name { get; set; }
    public int Health { get; set; }
    public int Mana { get; set; }

    public void DisplayStatus()
    {
        Console.WriteLine($"{Name} tiene {Health} de salud y {Mana} de
mana.");
    }

    // Método para crear un Memento
    public GameCharacterMemento SaveState()
    {
        return new GameCharacterMemento(Name, Health, Mana);
    }

    // Método para restaurar el estado desde un Memento
    public void RestoreState(GameCharacterMemento memento)
    {
        Name = memento.Name;
        Health = memento.Health;
        Mana = memento.Mana;
    }
}
```

La clase `GameCharacterMemento` es la clase Memento que almacena el estado del objeto `GameCharacter`. Contiene propiedades de solo lectura para `Name`, `Health` y `Mana`.

La clase `GameCharacter` representa el personaje del juego con propiedades `Name`, `Health` y `Mana`. Proporciona el método `DisplayStatus` para mostrar el estado actual del personaje. Proporciona el método `SaveState` para crear y devolver un objeto `GameCharacterMemento` con el estado actual. Proporciona el método `RestoreState` para restaurar el estado del personaje desde un objeto `GameCharacterMemento`.

La clase `Program` crea una instancia de `GameCharacter` y muestra su estado inicial. Guarda el estado del personaje creando un memento. Cambia el estado del personaje y muestra el nuevo estado. Restaura el estado del personaje desde el memento guardado y muestra el estado restaurado.

Ejercicio 3

```
class Program
{
    static void Main()
    {
        var alice = new User("Alice");
        var bob = new User("Bob");

        alice.SendMessage("Hola Bob!", bob);
        bob.SendMessage("Hola Alice!", alice);
    }
}

class User
{
    public string Name { get; }

    public User(string name)
    {
        Name = name;
    }

    public void SendMessage(string message, User recipient)
    {
        Console.WriteLine($"{Name} to {recipient.Name}: {message}");
    }
}
```

Para resolver el problema de enviar mensajes entre usuarios de una manera más escalable y flexible, podemos usar el patrón Mediator. Este patrón define un objeto que encapsula cómo interactúan un conjunto de objetos. Al usar un mediador, se promueve el desacoplamiento al evitar que los objetos se refieran entre sí explícitamente y permite variar sus interacciones independientemente.

```
// Interfaz IMediator
public interface IMediator
{
    void SendMessage(string message, User sender, User recipient);
}

// Clase concreta ChatMediator implementando IMediator
public class ChatMediator : IMediator
{
    private readonly List<User> _users;
```

```
    public ChatMediator()
    {
        _users = new List<User>();
    }

    public void RegisterUser(User user)
    {
        if (!_users.Contains(user))
        {
            _users.Add(user);
        }
    }

    public void SendMessage(string message, User sender, User recipient)
    {
        if (_users.Contains(sender) && _users.Contains(recipient))
        {
            Console.WriteLine($"{sender.Name} to {recipient.Name}:
{message}");
        }
    }
}

// Clase User
public class User
{
    public string Name { get; }
    private readonly IMediator _mediator;

    public User(string name, IMediator mediator)
    {
        Name = name;
        _mediator = mediator;
        _mediator.RegisterUser(this);
    }

    public void SendMessage(string message, User recipient)
    {
        _mediator.SendMessage(message, this, recipient);
    }
}

class Program
{
    static void Main()
```

```
    {
        IMediator chatMediator = new ChatMediator();

        var alice = new User("Alice", chatMediator);
        var bob = new User("Bob", chatMediator);

        alice.SendMessage("Hola Bob!", bob);
        bob.SendMessage("Hola Alice!", alice);
    }
}
```

La interfaz `IMediator` define el método `SendMessage` que será implementado por el mediador concreto.

La clase concreta `ChatMediator` implementa la interfaz `IMediator` y mantiene una lista de usuarios registrados. Proporciona el método `RegisterUser` para registrar usuarios en el mediador. Implementa el método `SendMessage` para enviar mensajes entre usuarios registrados.

La clase `User` representa a un usuario en el sistema. Al crear un usuario, se registra en el mediador. Proporciona el método `SendMessage` que utiliza el mediador para enviar mensajes a otros usuarios.

La clase `Program` crea una instancia del mediador `ChatMediator`. Crea instancias de `User` y las registra en el mediador. Los usuarios envían mensajes entre sí a través del mediador.

Ejercicio 4

```
class Program
{
    static void Main()
    {
        var television = new Television();
        string input = "";

        while (input != "exit")
        {
            Console.WriteLine("Escribe 'on' para encender, 'off' para
apagar, 'volumeup' para subir volumen, 'volumedown' para bajar volumen,
'exit' para salir.");
            input = Console.ReadLine();

            switch (input)
```

```
            {
                case "on":
                    television.TurnOn();
                    break;
                case "off":
                    television.TurnOff();
                    break;
                case "volumeup":
                    television.VolumeUp();
                    break;
                case "volumedown":
                    television.VolumeDown();
                    break;
            }
        }
    }
}

class Television
{
    private bool isOn = false;
    private int volume = 10;

    public void TurnOn()
    {
        isOn = true;
        Console.WriteLine("Televisión encendida.");
    }

    public void TurnOff()
    {
        isOn = false;
        Console.WriteLine("Televisión apagada.");
    }

    public void VolumeUp()
    {
        if (isOn)
        {
            volume++;
            Console.WriteLine($"Volumen: {volume}");
        }
    }

    public void VolumeDown()
    {
```

```
        if (isOn)
        {
            volume--;
            Console.WriteLine($"Volumen: {volume}");
        }
    }
}
```

Para este ejercicio, el patrón State es adecuado, ya que permite que un objeto altere su comportamiento cuando su estado interno cambia. Al usar el patrón State, podemos manejar las transiciones entre los estados de la televisión (encendido, apagado) de manera más organizada y extensible.

```
// Interfaz IState
public interface IState
{
    void TurnOn(Television tv);
    void TurnOff(Television tv);
    void VolumeUp(Television tv);
    void VolumeDown(Television tv);
}

// Estado concreto: Encendido
public class OnState : IState
{
    public void TurnOn(Television tv)
    {
        Console.WriteLine("La televisión ya está encendida.");
    }

    public void TurnOff(Television tv)
    {
        tv.SetState(new OffState());
        Console.WriteLine("Televisión apagada.");
    }

    public void VolumeUp(Television tv)
    {
        tv.Volume++;
        Console.WriteLine($"Volumen: {tv.Volume}");
    }

    public void VolumeDown(Television tv)
    {
        tv.Volume--;
```

```
        Console.WriteLine($"Volumen: {tv.Volume}");
    }
}

// Estado concreto: Apagado
public class OffState : IState
{
    public void TurnOn(Television tv)
    {
        tv.SetState(new OnState());
        Console.WriteLine("Televisión encendida.");
    }

    public void TurnOff(Television tv)
    {
        Console.WriteLine("La televisión ya está apagada.");
    }

    public void VolumeUp(Television tv)
    {
        Console.WriteLine("No se puede cambiar el volumen cuando la
televisión está apagada.");
    }

    public void VolumeDown(Television tv)
    {
        Console.WriteLine("No se puede cambiar el volumen cuando la
televisión está apagada.");
    }
}

// Clase Television
public class Television
{
    private IState _state;
    public int Volume { get; set; }

    public Television()
    {
        _state = new OffState();
        Volume = 10;
    }

    public void SetState(IState state)
    {
        _state = state;
```

```
    }

    public void TurnOn()
    {
        _state.TurnOn(this);
    }

    public void TurnOff()
    {
        _state.TurnOff(this);
    }

    public void VolumeUp()
    {
        _state.VolumeUp(this);
    }

    public void VolumeDown()
    {
        _state.VolumeDown(this);
    }
}

class Program
{
    static void Main()
    {
        var television = new Television();
        string input = "";

        while (input != "exit")
        {
            Console.WriteLine("Escribe 'on' para encender, 'off' para
apagar, 'volumeup' para subir volumen, 'volumedown' para bajar volumen,
'exit' para salir.");
            input = Console.ReadLine();

            switch (input)
            {
                case "on":
                    television.TurnOn();
                    break;
                case "off":
                    television.TurnOff();
                    break;
                case "volumeup":
```

```
                    television.VolumeUp();
                    break;
                case "volumedown":
                    television.VolumeDown();
                    break;
            }
        }
    }
}
```

La interfaz `IState` define los métodos `TurnOn`, `TurnOff`, `VolumeUp` y `VolumeDown` que serán implementados por los estados concretos.

La clase concreta `OnState` implementa el estado de la televisión encendida. Define el comportamiento de la televisión cuando está encendida.

La clase concreta `OffState` implementa el estado de la televisión apagada. Define el comportamiento de la televisión cuando está apagada.

La clase `Television` mantiene una referencia al estado actual (`_state`) y el volumen (`Volume`). Proporciona métodos para cambiar el estado y delega las acciones al estado actual.

La clase `Program` crea una instancia de `Television` y lee la entrada del usuario para realizar acciones en la televisión.

Ejercicio 5

```
class Program
{
    static void Main()
    {
        Animal[] animals = { new Lion(), new Monkey(), new Elephant() };

        foreach (var animal in animals)
        {
            animal.Feed();
            // Nota: Con el tiempo, aquí tendrás que agregar más
operaciones,
            // lo que hará que el código sea menos mantenible.
        }
    }
}
```

```
abstract class Animal
{
    public abstract void Feed();
}

class Lion : Animal
{
    public override void Feed()
    {
        Console.WriteLine("El león está siendo alimentado con carne.");
    }
}

class Monkey : Animal
{
    public override void Feed()
    {
        Console.WriteLine("El mono está siendo alimentado con
bananas.");
    }
}

class Elephant : Animal
{
    public override void Feed()
    {
        Console.WriteLine("El elefante está siendo alimentado con
pastito.");
    }
}
```

Para resolver el problema de agregar nuevas operaciones a diferentes tipos de animales de manera mantenible y escalable, podemos usar el patrón Visitor. Este patrón permite definir nuevas operaciones sobre una estructura de objetos sin cambiar las clases de los objetos sobre los cuales opera.

```
// Interfaz IVisitor
public interface IVisitor
{
    void Visit(Lion lion);
    void Visit(Monkey monkey);
    void Visit(Elephant elephant);
}

// Clase concreta Visitor: FeedVisitor
```

```
public class FeedVisitor : IVisitor
{
    public void Visit(Lion lion)
    {
        Console.WriteLine("El león está siendo alimentado con carne.");
    }

    public void Visit(Monkey monkey)
    {
        Console.WriteLine("El mono está siendo alimentado con
bananas.");
    }

    public void Visit(Elephant elephant)
    {
        Console.WriteLine("El elefante está siendo alimentado con
pastito.");
    }
}

// Clase abstracta Animal
public abstract class Animal
{
    public abstract void Accept(IVisitor visitor);
}

// Clases concretas de animales
public class Lion : Animal
{
    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

public class Monkey : Animal
{
    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

public class Elephant : Animal
{
    public override void Accept(IVisitor visitor)
```

```
    {
        visitor.Visit(this);
    }
}

// Clase Program
class Program
{
    static void Main()
    {
        Animal[] animals = { new Lion(), new Monkey(), new Elephant() };
        IVisitor feedVisitor = new FeedVisitor();

        foreach (var animal in animals)
        {
            animal.Accept(feedVisitor);
        }
    }
}
```

La interfaz `IVisitor` define los métodos `Visit` para cada tipo concreto de animal.

La clase concreta `FeedVisitor` implementa la interfaz `IVisitor` y define el comportamiento de alimentación para cada tipo de animal.

La clase abstracta `Animal` define el método abstracto `Accept`, que tomará un `IVisitor`.

Las clases concretas de animales (`Lion`, `Monkey`, `Elephant`) implementan el método `Accept` para aceptar un visitante y llamar al método `Visit` correspondiente en el visitante.

La clase `Program` crea instancias de diferentes animales y un visitante (`FeedVisitor`). Itera a través de los animales y les hace aceptar el visitante, lo que ejecuta la operación de alimentación correspondiente.

Ejercicio 6

Problema: agregar una luz amarillo intermitente entre el amarillo y el rojo.

```
class Program
{
    static void Main()
    {
        var trafficLight = new TrafficLight();
```

```
        for (int i = 0; i < 5; i++)
        {
            trafficLight.ChangeLight();
            Thread.Sleep(1000); // Wait 1 second
        }
    }
}

class TrafficLight
{
    enum Light { Red, Yellow, Green }
    private Light currentLight;

    public TrafficLight()
    {
        currentLight = Light.Red;
        Console.WriteLine("Luz inicial es Roja.");
    }

    public void ChangeLight()
    {
        switch (currentLight)
        {
            case Light.Red:
                currentLight = Light.Green;
                Console.WriteLine("Cambio a Verde.");
                break;
            case Light.Green:
                currentLight = Light.Yellow;
                Console.WriteLine("Cambio a Amarillo.");
                break;
            case Light.Yellow:
                currentLight = Light.Red;
                Console.WriteLine("Cambio a Rojo.");
                break;
        }
    }
}
```

Para agregar una luz amarilla intermitente (blink) entre el amarillo y el rojo, podemos usar el patrón State. Este patrón nos permite representar cada estado de la luz del semáforo como una clase separada, lo que hace que la adición de nuevos estados sea más fácil y el código sea más limpio y mantenible.

```
// Interfaz ILightState
```

```
public interface ILightState
{
    void ChangeLight(TrafficLight trafficLight);
}

// Clase concreta RedLightState
public class RedLightState : ILightState
{
    public void ChangeLight(TrafficLight trafficLight)
    {
        Console.WriteLine("Cambio a Verde.");
        trafficLight.SetState(new GreenLightState());
    }
}

// Clase concreta GreenLightState
public class GreenLightState : ILightState
{
    public void ChangeLight(TrafficLight trafficLight)
    {
        Console.WriteLine("Cambio a Amarillo.");
        trafficLight.SetState(new YellowLightState());
    }
}

// Clase concreta YellowLightState
public class YellowLightState : ILightState
{
    public void ChangeLight(TrafficLight trafficLight)
    {
        Console.WriteLine("Cambio a Amarillo Intermitente.");
        trafficLight.SetState(new YellowBlinkingLightState());
    }
}

// Clase concreta YellowBlinkingLightState
public class YellowBlinkingLightState : ILightState
{
    public void ChangeLight(TrafficLight trafficLight)
    {
        Console.WriteLine("Cambio a Rojo.");
        trafficLight.SetState(new RedLightState());
    }
}

// Clase TrafficLight
```

```
public class TrafficLight
{
    private ILightState currentState;

    public TrafficLight()
    {
        currentState = new RedLightState();
        Console.WriteLine("Luz inicial es Roja.");
    }

    public void SetState(ILightState state)
    {
        currentState = state;
    }

    public void ChangeLight()
    {
        currentState.ChangeLight(this);
    }
}

// Clase Program
class Program
{
    static void Main()
    {
        var trafficLight = new TrafficLight();

        for (int i = 0; i < 8; i++)
        {
            trafficLight.ChangeLight();
            Thread.Sleep(1000); // Wait 1 second
        }
    }
}
```

La interfaz `ILightState` define el método `ChangeLight` que será implementado por cada estado de la luz del semáforo.

Las clases concretas `RedLightState`, `GreenLightState`, `YellowLightState`, `YellowBlinkingLightState` representan un estado específico de la luz del semáforo. Implementan el método `ChangeLight` para cambiar al siguiente estado apropiado.

La clase `TrafficLight` mantiene una referencia al estado actual (`currentState`). Proporciona el método `SetState` para cambiar el estado actual y el método `ChangeLight` para delegar la acción al estado actual.

La clase `Program` crea una instancia de `TrafficLight` y llama al método `ChangeLight` en un bucle, esperando un segundo entre cada cambio.

Ejercicio 7

Un sistema de cálculo de envío para un servicio de e-commerce. Inicialmente, el sistema podría estar usando condicionales para determinar qué algoritmo de cálculo de envío usar. Este enfoque puede volverse difícil de mantener y no es muy flexible si se desean agregar más algoritmos de envío.

```
class Program
{
    static void Main()
    {
        var shippingCalculator = new ShippingCalculator();

        Console.WriteLine("Costo de envío con UPS: " +
shippingCalculator.CalculateShippingCost("UPS", 5));
        Console.WriteLine("Costo de envío con FedEx: " +
shippingCalculator.CalculateShippingCost("FedEx", 5));
        Console.WriteLine("Costo de envío con DAC: " +
shippingCalculator.CalculateShippingCost("DAC", 5));
    }
}

class ShippingCalculator
{
    public double CalculateShippingCost(string courier, double weight)
    {
        switch (courier)
        {
            case "UPS":
                return weight * 0.75;
            case "FedEx":
                return weight * 0.85;
            case "DAC":
                return weight * 0.65;
            default:
                throw new Exception("Courier no soportado.");
        }
    }
```

```
}
```

Para resolver el problema de determinar el algoritmo de cálculo de envío utilizando condicionales, podemos utilizar el patrón Strategy. Este patrón nos permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables.

```
// Interfaz IShippingStrategy
public interface IShippingStrategy
{
    double CalculateShippingCost(double weight);
}

// Clase concreta UPS
public class UPS : IShippingStrategy
{
    public double CalculateShippingCost(double weight)
    {
        return weight * 0.75;
    }
}

// Clase concreta FedEx
public class FedEx : IShippingStrategy
{
    public double CalculateShippingCost(double weight)
    {
        return weight * 0.85;
    }
}

// Clase concreta DAC
public class DAC : IShippingStrategy
{
    public double CalculateShippingCost(double weight)
    {
        return weight * 0.65;
    }
}

// Clase ShippingCalculator
public class ShippingCalculator
{
    private IShippingStrategy _shippingStrategy;

    public void SetShippingStrategy(IShippingStrategy shippingStrategy)
```

```
    {
        _shippingStrategy = shippingStrategy;
    }

    public double CalculateShippingCost(double weight)
    {
        if (_shippingStrategy == null)
        {
            throw new Exception("Shipping strategy not set.");
        }
        return _shippingStrategy.CalculateShippingCost(weight);
    }
}

// Clase Program
class Program
{
    static void Main()
    {
        var shippingCalculator = new ShippingCalculator();

        shippingCalculator.SetShippingStrategy(new UPS());
        Console.WriteLine("Costo de envío con UPS: " +
shippingCalculator.CalculateShippingCost(5));

        shippingCalculator.SetShippingStrategy(new FedEx());
        Console.WriteLine("Costo de envío con FedEx: " +
shippingCalculator.CalculateShippingCost(5));

        shippingCalculator.SetShippingStrategy(new DAC());
        Console.WriteLine("Costo de envío con DAC: " +
shippingCalculator.CalculateShippingCost(5));
    }
}
```

La interfaz `IShippingStrategy` define el método `CalculateShippingCost` que será implementado por cada estrategia concreta de cálculo de envío.
Las clases concretas `UPS`, `FedEx`, `DAC` representan una estrategia específica de cálculo de envío. Implementan el método `CalculateShippingCost` para calcular el costo según la estrategia particular.

La clase `ShippingCalculator` mantiene una referencia a la estrategia actual de envío (`_shippingStrategy`). Proporciona el método `SetShippingStrategy` para cambiar la estrategia de envío y el método `CalculateShippingCost` para calcular el costo de envío utilizando la estrategia actual.

La clase `Program` crea una instancia de `ShippingCalculator` y establece diferentes estrategias de envío para calcular el costo de envío con diferentes mensajerías.

Ejercicio 8

```
class Program
{
    static void Main()
    {
        var emailService = new EmailService();
        emailService.SendEmail("john.doe@example.com", "Nueva
promoción", "¡Revisa nuestra nueva promoción!");
        emailService.SendNewsletter("john.doe@example.com", "Newsletter
de Junio", "Aquí está nuestro newsletter de Junio.");
    }
}

class EmailService
{
    public void SendEmail(string recipient, string subject, string
message)
    {
        Console.WriteLine($"Enviando correo a {recipient} con el asunto
'{subject}': {message}");
        // Agregar código para enviar correo
    }

    public void SendNewsletter(string recipient, string subject, string
message)
    {
        Console.WriteLine($"Enviando newsletter a {recipient} con el
asunto '{subject}': {message}");
        // Agregar código para enviar newsletter
    }
}
```

Para mejorar la flexibilidad y escalabilidad del sistema de envío de correos y newsletters, podemos utilizar el patrón Strategy. Este patrón nos permitirá definir diferentes estrategias de envío, como enviar un correo o enviar un newsletter, y aplicar estas estrategias en tiempo de ejecución.

```
// Interfaz IEmailStrategy
public interface IEmailStrategy
{
```

```
    void Send(string recipient, string subject, string message);
}

// Clase concreta EmailStrategy
public class EmailStrategy : IEmailStrategy
{
    public void Send(string recipient, string subject, string message)
    {
        Console.WriteLine($"Enviando correo a {recipient} con el asunto
'{subject}': {message}");
        // Agregar código para enviar correo
    }
}

// Clase concreta NewsletterStrategy
public class NewsletterStrategy : IEmailStrategy
{
    public void Send(string recipient, string subject, string message)
    {
        Console.WriteLine($"Enviando newsletter a {recipient} con el
asunto '{subject}': {message}");
        // Agregar código para enviar newsletter
    }
}

// Clase EmailService
public class EmailService
{
    private IEmailStrategy _emailStrategy;

    public void SetEmailStrategy(IEmailStrategy emailStrategy)
    {
        _emailStrategy = emailStrategy;
    }

    public void Send(string recipient, string subject, string message)
    {
        if (_emailStrategy == null)
        {
            throw new Exception("Email strategy not set.");
        }
        _emailStrategy.Send(recipient, subject, message);
    }
}

// Clase Program
```

```
class Program
{
    static void Main()
    {
        var emailService = new EmailService();

        emailService.SetEmailStrategy(new EmailStrategy());
        emailService.Send("john.doe@example.com", "Nueva promoción",
"¡Revisa nuestra nueva promoción!");

        emailService.SetEmailStrategy(new NewsletterStrategy());
        emailService.Send("john.doe@example.com", "Newsletter de Junio",
"Aquí está nuestro newsletter de Junio.");
    }
}
```

La interfaz `IEmailStrategy` define el método `Send` que será implementado por cada estrategia concreta de envío.

Las clases concretas `EmailStrategy` y `NewsletterStrategy` representan una estrategia específica de envío. Implementan el método `Send` para enviar un correo o un newsletter, respectivamente.

La clase `EmailService` mantiene una referencia a la estrategia actual de envío (`_emailStrategy`). Proporciona el método `SetEmailStrategy` para cambiar la estrategia de envío y el método `Send` para enviar un mensaje utilizando la estrategia actual.

La clase `Program` crea una instancia de `EmailService` y establece diferentes estrategias de envío para enviar un correo o un newsletter.

Ejercicio 9

En un sistema de soporte técnico donde las consultas de los clientes se pueden manejar en diferentes niveles, como soporte de nivel 1, nivel 2, y nivel 3. Inicialmente, el sistema podría estar usando condicionales para determinar qué nivel de soporte debe manejar una consulta. Este enfoque puede volverse difícil de mantener y poco flexible si se desean agregar más niveles de soporte o cambiar las condiciones para el manejo de consultas.

```
class Program
{
    static void Main()
    {
        SupportSystem supportSystem = new SupportSystem();
        supportSystem.HandleSupportRequest(1, "No puedo iniciar
```

```
sesión.");
        supportSystem.HandleSupportRequest(2, "Mi cuenta ha sido
bloqueada.");
        supportSystem.HandleSupportRequest(3, "Necesito recuperar datos
borrados.");
    }
}

class SupportSystem
{
    public void HandleSupportRequest(int level, string message)
    {
        if (level == 1)
        {
            Console.WriteLine("Soporte de Nivel 1: Manejando consulta -
" + message);
        }
        else if (level == 2)
        {
            Console.WriteLine("Soporte de Nivel 2: Manejando consulta -
" + message);
        }
        else if (level == 3)
        {
            Console.WriteLine("Soporte de Nivel 3: Manejando consulta -
" + message);
        }
        else
        {
            Console.WriteLine("Consulta no soportada.");
        }
    }
}
```

Para mejorar la flexibilidad y la escalabilidad del sistema de soporte técnico, podemos utilizar el patrón Chain of Responsibility. Este patrón nos permite crear una cadena de manejadores de solicitudes, donde cada manejador decide si procesa la solicitud o la pasa al siguiente manejador de la cadena.

```
// Interfaz IHandler
public interface IHandler
{
    IHandler SetNext(IHandler handler);
    void Handle(int level, string message);
}
```

```
// Clase abstracta Handler
public abstract class Handler : IHandler
{
    private IHandler _nextHandler;

    public IHandler SetNext(IHandler handler)
    {
        _nextHandler = handler;
        return handler;
    }

    public virtual void Handle(int level, string message)
    {
        if (_nextHandler != null)
        {
            _nextHandler.Handle(level, message);
        }
        else
        {
            Console.WriteLine("Consulta no soportada.");
        }
    }
}

// Clase concreta Level1SupportHandler
public class Level1SupportHandler : Handler
{
    public override void Handle(int level, string message)
    {
        if (level == 1)
        {
            Console.WriteLine("Soporte de Nivel 1: Manejando consulta -
" + message);
        }
        else
        {
            base.Handle(level, message);
        }
    }
}

// Clase concreta Level2SupportHandler
public class Level2SupportHandler : Handler
{
    public override void Handle(int level, string message)
```

```
    {
        if (level == 2)
        {
            Console.WriteLine("Soporte de Nivel 2: Manejando consulta -
" + message);
        }
        else
        {
            base.Handle(level, message);
        }
    }
}

// Clase concreta Level3SupportHandler
public class Level3SupportHandler : Handler
{
    public override void Handle(int level, string message)
    {
        if (level == 3)
        {
            Console.WriteLine("Soporte de Nivel 3: Manejando consulta -
" + message);
        }
        else
        {
            base.Handle(level, message);
        }
    }
}

// Clase Program
class Program
{
    static void Main()
    {
        // Crear los manejadores de soporte
        var level1Support = new Level1SupportHandler();
        var level2Support = new Level2SupportHandler();
        var level3Support = new Level3SupportHandler();

        // Configurar la cadena de responsabilidad
        level1Support.SetNext(level2Support).SetNext(level3Support);

        // Procesar las solicitudes de soporte
        level1Support.Handle(1, "No puedo iniciar sesión.");
        level1Support.Handle(2, "Mi cuenta ha sido bloqueada.");
```

```
        level1Support.Handle(3, "Necesito recuperar datos borrados.");
    }
}
```

La interfaz `IHandler` define los métodos `SetNext` para establecer el siguiente manejador en la cadena y `Handle` para manejar las solicitudes.

La clase abstracta `Handler` implementa la interfaz `IHandler` y maneja la cadena de responsabilidades. La implementación por defecto del método `Handle` pasa la solicitud al siguiente manejador si existe.

Las clases concretas `Level1SupportHandler`, `Level2SupportHandler`, y `Level3SupportHandler` manejan las solicitudes de soporte de su nivel específico. Si no pueden manejar la solicitud, la pasan al siguiente manejador en la cadena.

La clase `Program` configura la cadena de responsabilidad estableciendo los manejadores en el orden deseado. Procesa las solicitudes de soporte pasando las solicitudes al primer manejador en la cadena.

Ejercicio 10

```
class Program
{
    static void Main()
    {
        GreetingSystem greetingSystem = new GreetingSystem();

        greetingSystem.Greet("USA", "John");
        greetingSystem.Greet("Spain", "Juan");
        greetingSystem.Greet("Japan", "Yuki");
    }
}

class GreetingSystem
{
    public void Greet(string nationality, string name)
    {
        if (nationality == "USA")
        {
            Console.WriteLine($"Hello, {name}!");
        }
        else if (nationality == "Spain")
        {
            Console.WriteLine($"¡Hola, {name}!");
```

```
        }
        else if (nationality == "Japan")
        {
            Console.WriteLine($"こんにちは, {name}!");
        }
        else
        {
            Console.WriteLine("Nationality not supported.");
        }
    }
}
```

Para hacer el sistema de saludos más flexible y escalable, podemos utilizar el patrón Strategy. Este patrón permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. De esta manera, el sistema puede seleccionar el algoritmo de saludo apropiado en tiempo de ejecución sin utilizar condicionales.

```
// Interfaz IGreetingStrategy
public interface IGreetingStrategy
{
    void Greet(string name);
}

// Estrategia para saludar en inglés
public class EnglishGreetingStrategy : IGreetingStrategy
{
    public void Greet(string name)
    {
        Console.WriteLine($"Hello, {name}!");
    }
}

// Estrategia para saludar en español
public class SpanishGreetingStrategy : IGreetingStrategy
{
    public void Greet(string name)
    {
        Console.WriteLine($"¡Hola, {name}!");
    }
}

// Estrategia para saludar en japonés
public class JapaneseGreetingStrategy : IGreetingStrategy
{
    public void Greet(string name)
```

```
    {
        Console.WriteLine($"こんにちは, {name}!");
    }
}

// Clase GreetingSystem
public class GreetingSystem
{
    private Dictionary<string, IGreetingStrategy> _strategies;

    public GreetingSystem()
    {
        _strategies = new Dictionary<string, IGreetingStrategy>
        {
            { "USA", new EnglishGreetingStrategy() },
            { "Spain", new SpanishGreetingStrategy() },
            { "Japan", new JapaneseGreetingStrategy() }
        };
    }

    public void Greet(string nationality, string name)
    {
        if (_strategies.ContainsKey(nationality))
        {
            _strategies[nationality].Greet(name);
        }
        else
        {
            Console.WriteLine("Nationality not supported.");
        }
    }
}

// Clase Program
class Program
{
    static void Main()
    {
        GreetingSystem greetingSystem = new GreetingSystem();

        greetingSystem.Greet("USA", "John");
        greetingSystem.Greet("Spain", "Juan");
        greetingSystem.Greet("Japan", "Yuki");
    }
}
```

La interfaz `IGreetingStrategy` define el método `Greet` que todas las estrategias de saludo deben implementar.

Las clases concretas `EnglishGreetingStrategy`, `SpanishGreetingStrategy`, y `JapaneseGreetingStrategy` implementan la interfaz `IGreetingStrategy` y proporciona su propia implementación del método `Greet`.

La clase `GreetingSystem` contiene un diccionario de estrategias de saludo, donde la clave es la nacionalidad y el valor es la instancia de la estrategia correspondiente. El método `Greet` selecciona y utiliza la estrategia adecuada en función de la nacionalidad proporcionada.

La clase `Program` configura el sistema de saludos y demuestra cómo utilizarlo para saludar a diferentes usuarios según su nacionalidad.