UT5 - TA5
Grupo 3
ADA1

Ejercicio 0
Para el patrón Chain of responsibility los diferentes principios que se cumplen son :

1. **Single Responsibility Principle (SRP):**
    - **Aplicación:** Cada manejador tiene una única responsabilidad: procesar la solicitud o pasarla al siguiente manejador.
    - **Justificación:** Cada clase maneja una tarea específica, cumpliendo con el SRP.
2. **Open/Closed Principle (OCP):**
    - **Aplicación:** Se pueden añadir nuevos manejadores a la cadena sin modificar el código existente.
    - **Justificación:** La cadena se puede extender sin alterar las clases actuales.
3. **Liskov Substitution Principle (LSP):**
    - **Aplicación:** Los manejadores deben ser intercambiables, permitiendo reemplazar cualquier manejador sin afectar el sistema.
    - **Justificación:** Todos los manejadores implementan una interfaz común.
4. **Interface Segregation Principle (ISP):**
    - **Aplicación:** Los manejadores implementan interfaces específicas y simples para el procesamiento de solicitudes.
    - **Justificación:** Mantiene las interfaces pequeñas y enfocadas.
5. **Dependency Inversion Principle (DIP):**
    - **Aplicación:** Los manejadores dependen de abstracciones (interfaces) en lugar de implementaciones concretas.
    - **Justificación:** Facilita la extensión y modificación del sistema.

Como este patron aplica todos los principios SOLID por ende no viola ninguno.

Ejercicio 1

El patrón **Observer** permite que los objetos (en este caso, los estudiantes) se suscriban y reciban notificaciones sobre eventos (nuevos exámenes) sin que el sujeto (examen) conozca los detalles de los observadores (estudiantes).

Lo resolveremos con 2 interfaces IObserver y ISubject

```
interface IObserver {
void Update(Exam exam);
}

interface ISubject {
void Subscribe(IObserver observer);
```

```
void Unsubscribe(IObserver observer);
void NotifyStudents();
}
```
Y luego 2 clases Exam que implementa ISubject y Student que implementa IObserver

```
class Exam : ISubject {
public string Subject { get; }
private List<IObserver> _students = new List<IObserver>();
public Exam(string subject) { Subject = subject; }

public void Subscribe(IObserver observer) { _students.Add(observer); }
public void Unsubscribe(IObserver observer) { _students.Remove(observer); }
public void NotifyStudents() { foreach (var student in _students) { student.Update(this); } }
}

class Student : IObserver {
public string Name { get; }
public Student(string name) { Name = name; }
public void Update(Exam exam) { Console.WriteLine($"{Name}, hay un nuevo examen de
{exam.Subject}!"); }
}
```

Ejercicio 2:

Para mejorar este problema se puede utilizar el patrón de comportamiento "Memento". Este patrón permite capturar y restaurar el estado interno de un objeto sin violar la encapsulación.

```
using System;

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

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

        Console.WriteLine("\nGuardando estado...");
        Caretaker caretaker = new Caretaker();
        caretaker.Memento = gameCharacter.SaveState();

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

        Console.WriteLine("\nRestaurando estado...");
        gameCharacter.RestoreState(caretaker.Memento);
        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.");
    }

    public Memento SaveState()
    {
        return new Memento(Name, Health, Mana);
    }

    public void RestoreState(Memento memento)
    {
        Name = memento.Name;
        Health = memento.Health;
        Mana = memento.Mana;
    }
}

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

    public Memento(string name, int health, int mana)
    {
        Name = name;
        Health = health;
        Mana = mana;
    }
}

class Caretaker
{
    public Memento Memento { get; set; }
}
```

Ejercicio 3

Para mejorar este diseño y hacerlo más escalable, podemos utilizar el patrón **Mediator**. Este patrón facilita la comunicación entre objetos (en este caso, los usuarios) al evitar que se refieran directamente entre sí. En lugar de eso, los usuarios se comunican a través de un mediador central.

Lo implementaremos así:

```
// Interfaz IMediator que define el método SendMessage
interface IMediator {
void SendMessage(string message, User sender, User recipient);
}

// Clase ChatRoom que implementa la interfaz IMediator
class ChatRoom : IMediator {
public void SendMessage(string message, User sender, User recipient) {
Console.WriteLine($"{sender.Name} to {recipient.Name}: {message}"); }
}

// Clase User que representa a los usuarios class User {
public string Name { get; } private IMediator _mediator;
public User(string name, IMediator mediator) { Name = name; _mediator = mediator; }
public void SendMessage(string message, User recipient) { _mediator.SendMessage(message,
this, recipient); }
}
```

Ejercicio 4

Para mejorar este diseño y hacerlo más flexible y escalable, podemos utilizar el patrón **Command**. Este patrón encapsula las solicitudes como objetos, lo que permite parametrizar a otros objetos con diferentes solicitudes, encolar o registrar solicitudes, y soportar operaciones que se pueden deshacer.

```
// Interfaz ICommand que define el método Execute
interface ICommand
{
    void Execute();
}

// Clase TurnOnCommand que implementa ICommand
class TurnOnCommand : ICommand
{
    private Television _television;

    public TurnOnCommand(Television television)
    {
        _television = television;
    }

    public void Execute()
    {
        _television.TurnOn();
    }
}

// Clase TurnOffCommand que implementa ICommand
class TurnOffCommand : ICommand
{
    private Television _television;

    public TurnOffCommand(Television television)
    {
        _television = television;
    }

    public void Execute()
    {
```

```
        _television.TurnOff();
    }
}

// Clase VolumeUpCommand que implementa ICommand
class VolumeUpCommand : ICommand
{
    private Television _television;

    public VolumeUpCommand(Television television)
    {
        _television = television;
    }

    public void Execute()
    {
        _television.VolumeUp();
    }
}

// Clase VolumeDownCommand que implementa ICommand
class VolumeDownCommand : ICommand
{
    private Television _television;

    public VolumeDownCommand(Television television)
    {
        _television = television;
    }

    public void Execute()
    {
        _television.VolumeDown();
    }
}

// Clase Television que contiene la lógica de la televisión
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}");
        }
    }
}

// Clase RemoteControl que gestiona los comandos
class RemoteControl
{
    private Dictionary<string, ICommand> _commands = new Dictionary<string, ICommand>();

    public void SetCommand(string action, ICommand command)
    {
        _commands[action] = command;
    }

    public bool HasCommand(string action)
    {
        return _commands.ContainsKey(action);
    }
```

```
    public void ExecuteCommand(string action)
    {
        if (_commands.ContainsKey(action))
        {
            _commands[action].Execute();
        }
    }
}
```

Ejercicio 5

Para mejorar la mantenibilidad del código y facilitar la adición de nuevas operaciones sin modificar las clases `Animal` y sus subclases, podemos utilizar el patrón **Visitor**. Este patrón permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

```
// Interfaz IVisitor que define el método Visit para cada tipo de Animal
interface IVisitor
{
    void Visit(Lion lion);
    void Visit(Monkey monkey);
    void Visit(Elephant elephant);
}

// Clase Feeder que implementa IVisitor y define la lógica de alimentar a cada animal
class Feeder : 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 HealthChecker que implementa IVisitor y define la lógica de chequear la salud de cada animal
class HealthChecker : IVisitor
{
    public void Visit(Lion lion)
    {
        Console.WriteLine("El león está siendo chequeado de salud.");
    }

    public void Visit(Monkey monkey)
```

```
    {
        Console.WriteLine("El mono está siendo chequeado de salud.");
    }

    public void Visit(Elephant elephant)
    {
        Console.WriteLine("El elefante está siendo chequeado de salud.");
    }
}

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

// Clase Lion que hereda de Animal e implementa el método Accept
class Lion : Animal
{
    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

// Clase Monkey que hereda de Animal e implementa el método Accept
class Monkey : Animal
{
    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

// Clase Elephant que hereda de Animal e implementa el método Accept
class Elephant : Animal
{
    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

Ejercicio 6

Para mejorar el diseño del semáforo y agregar una luz amarilla intermitente entre el amarillo y el rojo, podemos utilizar el patrón **State**. Este patrón permite que un objeto altere su comportamiento cuando su estado interno cambia, encapsulando los estados en clases separadas.

```
// Interfaz ILightState que define el método Handle
interface ILightState
{
    void Handle(TrafficLight context);
}

// Clase TrafficLight que gestiona los estados del semáforo
class TrafficLight
{
    private ILightState currentState;

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

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

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

// Clase RedLight que implementa ILightState
class RedLight : ILightState
{
    public void Handle(TrafficLight context)
    {
        Console.WriteLine("Cambio a Verde.");
        context.SetState(new GreenLight());
    }
}
```

```
// Clase GreenLight que implementa ILightState
class GreenLight : ILightState
{
    public void Handle(TrafficLight context)
    {
        Console.WriteLine("Cambio a Amarillo.");
        context.SetState(new YellowLight());
    }
}

// Clase YellowLight que implementa ILightState
class YellowLight : ILightState
{
    public void Handle(TrafficLight context)
    {
        Console.WriteLine("Cambio a Amarillo Intermitente.");
        context.SetState(new FlashingYellowLight());
    }
}

// Clase FlashingYellowLight que implementa ILightState
class FlashingYellowLight : ILightState
{
    public void Handle(TrafficLight context)
    {
        Console.WriteLine("Cambio a Rojo.");
        context.SetState(new RedLight());
    }
}
```

Ejercicio 7

Para mejorar el diseño del sistema de cálculo de envío y hacerlo más flexible, podemos utilizar el patrón **Strategy**. Este patrón permite definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. El patrón Strategy permite que el algoritmo varíe independientemente de los clientes que lo usan.

```
// Interfaz IShippingStrategy que define el método Calculate
interface IShippingStrategy
{
    double Calculate(double weight);
}

// Clase UPS que implementa IShippingStrategy
class UPS : IShippingStrategy
{
    public double Calculate(double weight)
    {
        return weight * 0.75;
    }
}

// Clase FedEx que implementa IShippingStrategy
class FedEx : IShippingStrategy
{
    public double Calculate(double weight)
    {
        return weight * 0.85;
    }
}

// Clase DAC que implementa IShippingStrategy
class DAC : IShippingStrategy
{
    public double Calculate(double weight)
    {
        return weight * 0.65;
    }
}

// Clase ShippingCalculator que gestiona las estrategias de envío
class ShippingCalculator
{
    private IShippingStrategy _strategy;
```

```
    public void SetStrategy(IShippingStrategy strategy)
    {
        _strategy = strategy;
    }

    public double CalculateShippingCost(double weight)
    {
        if (_strategy == null)
        {
            throw new Exception("Estrategia de envío no establecida.");
        }
        return _strategy.Calculate(weight);
    }
}
```

Ejercicio 8

Para mejorar el diseño del servicio de envío de correos y newsletters, podemos utilizar el patrón **Template Method**. Este patrón permite definir el esqueleto de un algoritmo en una clase base y delegar la implementación de ciertos pasos a clases derivadas.

```
// Clase abstracta EmailService que define el método Template Method Send
abstract class EmailService
{
    public void Send(string recipient, string subject, string message)
    {
        Console.WriteLine($"Preparando para enviar a {recipient}...");
        SendMessage(recipient, subject, message);
        Console.WriteLine("Correo enviado.\n");
    }

    protected abstract void SendMessage(string recipient, string subject, string message);
}

// Clase PromotionalEmailService que implementa EmailService
class PromotionalEmailService : EmailService
{
    protected override void SendMessage(string recipient, string subject, string message)
    {
        Console.WriteLine($"Enviando correo promocional a {recipient} con el asunto '{subject}':
{message}");
        // Agregar código para enviar correo promocional
    }
}

// Clase NewsletterEmailService que implementa EmailService
class NewsletterEmailService : EmailService
{
    protected override void SendMessage(string recipient, string subject, string message)
    {
        Console.WriteLine($"Enviando newsletter a {recipient} con el asunto '{subject}':
{message}");
        // Agregar código para enviar newsletter
    }
}
```

Ejercicio 9

Para mejorar la flexibilidad y mantenibilidad del sistema de soporte técnico, podemos utilizar el patrón **Chain of Responsibility**. Este patrón permite que un objeto pase la solicitud a lo largo de una cadena de manejadores hasta que uno de ellos se haga cargo de la solicitud.

```
// Interfaz ISupportHandler que define el método HandleSupportRequest y SetNext
interface ISupportHandler
{
    ISupportHandler SetNext(ISupportHandler nextHandler);
    void HandleSupportRequest(int level, string message);
}

// Clase abstracta SupportHandler que implementa ISupportHandler
abstract class SupportHandler : ISupportHandler
{
    private ISupportHandler _nextHandler;

    public ISupportHandler SetNext(ISupportHandler nextHandler)
    {
        _nextHandler = nextHandler;
        return nextHandler;
    }

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

    protected abstract bool CanHandle(int level);
    protected abstract void Handle(string message);
}
```

```
// Clases concretas de soporte (Level1Support, Level2Support, Level3Support)
class Level1Support : SupportHandler
{
    protected override bool CanHandle(int level)
    {
        return level == 1;
    }

    protected override void Handle(string message)
    {
        Console.WriteLine("Soporte de Nivel 1: Manejando consulta - " + message);
    }
}

class Level2Support : SupportHandler
{
    protected override bool CanHandle(int level)
    {
        return level == 2;
    }

    protected override void Handle(string message)
    {
        Console.WriteLine("Soporte de Nivel 2: Manejando consulta - " + message);
    }
}

class Level3Support : SupportHandler
{
    protected override bool CanHandle(int level)
    {
        return level == 3;
    }

    protected override void Handle(string message)
    {
        Console.WriteLine("Soporte de Nivel 3: Manejando consulta - " + message);
    }
}
```

Ejercicio 10

El patrón **Strategy** nos permitirá definir una familia de algoritmos (en este caso, los diferentes saludos) encapsulados en clases separadas y hacerlos intercambiables.

```
// Interfaz IGreetStrategy que define el método Greet
interface IGreetStrategy
{
    void Greet(string name);
}

// Clase GreetingSystem que maneja las estrategias de saludo
class GreetingSystem
{
    private Dictionary<string, IGreetStrategy> _strategies = new Dictionary<string,
IGreetStrategy>();

    public void SetStrategy(string nationality, IGreetStrategy strategy)
    {
        _strategies[nationality] = strategy;
    }

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

// Clase concreta GreetInEnglish que implementa IGreetStrategy
class GreetInEnglish : IGreetStrategy
{
    public void Greet(string name)
    {
        Console.WriteLine($"Hello, {name}!");
    }
}
```

```
// Clase concreta GreetInSpanish que implementa IGreetStrategy
class GreetInSpanish : IGreetStrategy
{
    public void Greet(string name)
    {
        Console.WriteLine($"¡Hola, {name}!");
    }
}

// Clase concreta GreetInJapanese que implementa IGreetStrategy
class GreetInJapanese : IGreetStrategy
{
    public void Greet(string name)
    {
        Console.WriteLine($"こんにちは, {name}!");
    }
}
```