Ejercicio 0:

Los principios SOLID que se aplican en este patrón son:

SRP ya que cada manejador tiene una única responsabilidad, OCP porque la cadena se puede extender con nuevos manejadores sin la necesidad de modificar código que ya existe, LSP dado que los manejadores pueden ser sustituidos sin afectar la corrección de la cadena, y por último DIP debido a que los módulos dependen de abstracciones.

El principio que se viola es ISP, ya que los manejadores pueden implementar interfaces con métodos que no sean usados.

Ejercicio 1:

En este caso, para resolver el problema, el patrón a utilizar es Observer.

interface *INotifiable*

{

void Update(string message);

}

class Student : INotifiable

{

public string Name { get; }

public Student(string name)

{

Name = name;

}

public void Update(string message)

{

Console.WriteLine($"{Name}, {message}");

}

}

class Exam

{

private List<*INotifiable*> students = new List<*INotifiable*>();

public string Subject { get; }

public Exam(string subject)

{

Subject = subject;

}

public void Subscribe(*INotifiable* student)

{

students.Add(student);

}

public void Unsubscribe(*INotifiable* student)

{

students.Remove(student);

}

public void NotifyStudents()

{

foreach (var student in students)

{

student.Update($"hay un nuevo examen de {Subject}!");

}

}

}

class Program

{

static void Main(string[] args)

{

var exam = new Exam("Matemáticas");

var student1 = new Student("Alice");

var student2 = new Student("Bob");

exam.Subscribe(student1);

exam.Subscribe(student2);

exam.NotifyStudents();

}

}

Ejercicio 2:  
En este ejercicio el patrón a utilizar es Memento.

class GameCharacter

{

public string Name { get; set; }

public int Health { get; set; }

public int Mana { get; set; }

public GameCharacterMemento SaveState()

{

return new GameCharacterMemento(Name, Health, Mana);

}

public void RestoreState(GameCharacterMemento memento)

{

Name = memento.Name;

Health = memento.Health;

Mana = memento.Mana;

}

public void DisplayStatus()

{

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

}

}

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;

}

}

class Program

{

static void Main(string[] args)

{

var gameCharacter = new GameCharacter { Name = "John", Health = 100, Mana = 50 };

gameCharacter.DisplayStatus();

var savedState = gameCharacter.SaveState();

gameCharacter.Health -= 30;

gameCharacter.Mana += 20;

gameCharacter.DisplayStatus();

gameCharacter.RestoreState(savedState);

gameCharacter.DisplayStatus();

}

}

Ejercicio 3:

Para este caso utilizaremos Mediator.

class ChatRoom

{

public void ShowMessage(User user, string message)

{

Console.WriteLine($"{user.Name}: {message}");

}

}

class User

{

public string Name { get; }

private ChatRoom chatRoom;

public User(string name, ChatRoom chatRoom)

{

Name = name;

this.chatRoom = chatRoom;

}

public void SendMessage(string message)

{

chatRoom.ShowMessage(this, message);

}

}

class Program

{

static void Main(string[] args)

{

var chatRoom = new ChatRoom();

var alice = new User("Alice", chatRoom);

var bob = new User("Bob", chatRoom);

alice.SendMessage("Hola Bob!");

bob.SendMessage("Hola Alice!");

}

}

Ejercicio 4:

En este caso usamos el patrón State.

interface ITelevisionState

{

void TurnOn(Television context);

void TurnOff(Television context);

void VolumeUp(Television context);

void VolumeDown(Television context);

}

class TelevisionOffState : ITelevisionState

{

public void TurnOn(Television context)

{

Console.WriteLine("Televisión encendida.");

context.SetState(new TelevisionOnState());

}

public void TurnOff(Television context)

{

Console.WriteLine("La televisión ya está apagada.");

}

public void VolumeUp(Television context)

{

Console.WriteLine("No se puede subir el volumen porque la televisión está apagada.");

}

public void VolumeDown(Television context)

{

Console.WriteLine("No se puede bajar el volumen porque la televisión está apagada.");

}

}

class TelevisionOnState : ITelevisionState

{

public void TurnOn(Television context)

{

Console.WriteLine("La televisión ya está encendida.");

}

public void TurnOff(Television context)

{

Console.WriteLine("Televisión apagada.");

context.SetState(new TelevisionOffState());

}

public void VolumeUp(Television context)

{

context.Volume++;

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

}

public void VolumeDown(Television context)

{

context.Volume--;

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

}

}

class Television

{

private ITelevisionState state;

public int Volume { get; set; } = 10;

public Television()

{

state = new TelevisionOffState();

}

public void SetState(ITelevisionState state)

{

this.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;

}

}

}

}

Ejercicio 5:

En este ejercicio el patrón Visitor fue el que seleccionamos.

interface *IVisitor*

{

void Visit(Lion lion);

void Visit(Monkey monkey);

void Visit(Elephant elephant);

}

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 pasto.");

}

}

abstract class Animal

{

public abstract void Accept(*IVisitor* visitor);

}

class Lion : Animal

{

public override void Accept(IVisitor visitor)

{

visitor.Visit(this);

}

}

class Monkey : Animal

{

public override void Accept(IVisitor visitor)

{

visitor.Visit(this);

}

}

class Elephant : Animal

{

public override void Accept(IVisitor visitor)

{

visitor.Visit(this);

}

}

class Program

{

static void Main(string[] args)

{

var animals = new Animal[] { new Lion(), new Monkey(), new Elephant() };

var feeder = new FeedVisitor();

foreach (var animal in animals)

{

animal.Accept(feeder);

}

}

}

Ejercicio 6:

El patrón utilizado en este caso fue State.

interface *ILightState*

{

void ChangeLight(TrafficLight light);

}

class RedState : ILightState

{

public void ChangeLight(TrafficLight light)

{

Console.WriteLine("Cambio a Verde.");

light.SetState(new GreenState());

}

}

class YellowState : ILightState

{

public void ChangeLight(TrafficLight light)

{

Console.WriteLine("Cambio a Amarillo Intermitente.");

light.SetState(new BlinkingYellowState());

}

}

class BlinkingYellowState : ILightState

{

public void ChangeLight(TrafficLight light)

{

Console.WriteLine("Cambio a Rojo.");

light.SetState(new RedState());

}

}

class GreenState : ILightState

{

public void ChangeLight(TrafficLight light)

{

Console.WriteLine("Cambio a Amarillo.");

light.SetState(new YellowState());

}

}

class TrafficLight

{

private *ILightState* currentState;

public TrafficLight()

{

currentState = new RedState();

Console.WriteLine("Luz inicial es Roja.");

}

public void SetState(*ILightState* state)

{

currentState = state;

}

public void ChangeLight()

{

currentState.ChangeLight(this);

}

}

class Program

{

static void Main(string[] args)

{

var trafficLight = new TrafficLight();

for (int i = 0; i < 5; i++)

{

trafficLight.ChangeLight();

Thread.Sleep(1000); *// Wait 1 second*

}

}

}

Ejercicio 7:

Utilizamos el patrón Strategy.

interface *IShippingStrategy*

{

double CalculateShippingCost(double weight);

}

class UPS : IShippingStrategy

{

public double CalculateShippingCost(double weight)

{

return weight \* 0.75;

}

}

class FedEx : IShippingStrategy

{

public double CalculateShippingCost(double weight)

{

return weight \* 0.85;

}

}

class DAC : IShippingStrategy

{

public double CalculateShippingCost(double weight)

{

return weight \* 0.65;

}

}

class ShippingCalculator

{

private *IShippingStrategy* strategy;

public void SetStrategy(*IShippingStrategy* strategy)

{

this.strategy = strategy;

}

public double CalculateShippingCost(double weight)

{

return strategy.CalculateShippingCost(weight);

}

}

class Program

{

static void Main(string[] args)

{

var calculator = new ShippingCalculator();

calculator.SetStrategy(new UPS());

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

calculator.SetStrategy(new FedEx());

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

calculator.SetStrategy(new DAC());

Console.WriteLine("Costo de envío con DAC: " + calculator.CalculateShippingCost(5));

}

}

Ejercicio 8:

Para este caso consideramos que el patrón Strategy era el indicado.

interface IEmailStrategy

{

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

}

class EmailStrategy : IEmailStrategy

{

public void Send(string recipient, string subject, string message)

{

Console.WriteLine($"Enviando correo a {recipient} con el asunto '{subject}': {message}");

// Código para enviar correo

}

}

class NewsletterStrategy : IEmailStrategy

{

public void Send(string recipient, string subject, string message)

{

Console.WriteLine($"Enviando newsletter a {recipient} con el asunto '{subject}': {message}");

// Código para enviar newsletter

}

}

class EmailService

{

private IEmailStrategy \_emailStrategy;

public void SetEmailStrategy(IEmailStrategy emailStrategy)

{

\_emailStrategy = emailStrategy;

}

public void SendEmail(string recipient, string subject, string message)

{

if (\_emailStrategy == null)

{

throw new InvalidOperationException("No email strategy set");

}

\_emailStrategy.Send(recipient, subject, message);

}

}

class Program

{

static *void* Main()

{

var emailService = new EmailService();

// Enviar correo

emailService.SetEmailStrategy(new EmailStrategy());

emailService.SendEmail("john.doe@example.com", "Nueva promoción", "¡Revisa nuestra nueva promoción!");

// Enviar newsletter

emailService.SetEmailStrategy(new NewsletterStrategy());

emailService.SendEmail("john.doe@example.com", "Newsletter de Junio", "Aquí está nuestro newsletter de Junio.");

}

}

Ejercicio 9:

En este ejercicio utilizamos el patrón Chain of Responsibility.

abstract class Approver

{

protected Approver nextApprover;

public void SetNextApprover(Approver nextApprover)

{

this.nextApprover = nextApprover;

}

public abstract void ApproveRequest(int amount);

}

class Manager : Approver

{

public override void ApproveRequest(int amount)

{

if (amount < 1000)

{

Console.WriteLine($"Manager aprobó el gasto de {amount}.");

}

else if (nextApprover != null)

{

nextApprover.ApproveRequest(amount);

}

}

}

class Director : Approver

{

public override void ApproveRequest(int amount)

{

if (amount < 5000)

{

Console.WriteLine($"Director aprobó el gasto de {amount}.");

}

else if (nextApprover != null)

{

nextApprover.ApproveRequest(amount);

}

}

}

class CEO : Approver

{

public override void ApproveRequest(int amount)

{

if (amount >= 5000)

{

Console.WriteLine($"CEO aprobó el gasto de {amount}.");

}

}

}

class Program

{

static void Main(string[] args)

{

var manager = new Manager();

var director = new Director();

var ceo = new CEO();

manager.SetNextApprover(director);

director.SetNextApprover(ceo);

int[] expenseRequests = { 500, 2000, 10000 };

foreach (var amount in expenseRequests)

{

manager.ApproveRequest(amount);

}

}

}

Ejercicio 10:

En este caso usamos el patrón Strategy.

interface *ITaxStrategy*

{

double CalculateTax(double amount);

}

class RegularTax : ITaxStrategy

{

public double CalculateTax(double amount)

{

return amount \* 0.21;

}

}

class ReducedTax : ITaxStrategy

{

public double CalculateTax(double amount)

{

return amount \* 0.10;

}

}

interface *IDiscountStrategy*

{

double ApplyDiscount(double amount);

}

class NoDiscount : IDiscountStrategy

{

public double ApplyDiscount(double amount)

{

return amount;

}

}

class SeasonalDiscount : IDiscountStrategy

{

public double ApplyDiscount(double amount)

{

return amount \* 0.90;

}

}

class Invoice

{

private *ITaxStrategy* taxStrategy;

private *IDiscountStrategy* discountStrategy;

public Invoice(*ITaxStrategy* taxStrategy, *IDiscountStrategy* discountStrategy)

{

this.taxStrategy = taxStrategy;

this.discountStrategy = discountStrategy;

}

public double CalculateTotal(double amount)

{

var discountedAmount = discountStrategy.ApplyDiscount(amount);

var tax = taxStrategy.CalculateTax(discountedAmount);

return discountedAmount + tax;

}

}

class Program

{

static void Main(string[] args)

{

var invoice1 = new Invoice(new RegularTax(), new NoDiscount());

Console.WriteLine("Total factura (impuesto regular, sin descuento): " + invoice1.CalculateTotal(100));

var invoice2 = new Invoice(new ReducedTax(), new SeasonalDiscount());

Console.WriteLine("Total factura (impuesto reducido, descuento estacional): " + invoice2.CalculateTotal(100));

}

}