ASSIGNMENT 1
NAME:VANDANA S KAMBLI

Exercise 1: Problem Statement on Design patterns

Come up creatively with six different use cases to demonstrate your understanding of the following software design patterns by coding the

same.

1. Two use cases to demonstrate two behavioural design pattern.

2. Two use cases to demonstrate two creational design pattern.

3. Two use cases to demonstrate two structural design pattern.

Exercise 2: Problem Statements for Mini-projects

1. MUST Comply: No fancy looking application is required to be built as part of this exercise. It shall be a simple console/terminal based

application. Focus shall ONLY be on logic and code quality as described in the points below.

2. MUST Comply: Coding should be done adopting best practices - Behavioural/structural/creational design patterns, SOLID design

principles, OOPs programming, language of candidates choice.

3. Candidate shall pick ONE among the EIGHT problem statements provided below, and solve.

4. Note: Please feel free to assume unknowns, and be creative in enhancing the problem statements to demonstrate your excellence!

1. Astronaut Daily Schedule Organizer Programming Exercise

Problem Statement

Design and implement a console-based application that helps astronauts organize their daily schedules. The application should allow users

to add, remove, and view daily tasks. Each task will have a description, start time, end time, and priority level. The intent behind this problem

statement is to evaluate your ability to implement a basic CRUD (Create, Read, Update, Delete) application, manage data a efficiently, and

apply best coding practices.

Functional Requirements

Mandatory Requirements

1. Add a new task with description, start time, end time, and priority level.

2. Remove an existing task.

3. View all tasks sorted by start time.

4. Validate that new tasks do not overlap with existing tasks.

5. Provide appropriate error messages for invalid operations.

ASSIGNMENT 1
NAME:VANDANA S KAMBLI

Optional Requirements

1. Edit an existing task.

2. Mark tasks as completed.

3. View tasks for a specific priority level.

Non-functional Requirements

1. The application should handle exceptions gracefully.

2. Ensure the application is optimized for performance.

3. Implement a logging mechanism for tracking application usage and errors.

Key Focus

Design Patterns to be used

1. Singleton Pattern: Ensure there is only one instance of the schedule manager.

2. Factory Pattern: Use a factory to create task objects.

3. Observer Pattern: Notify users of task conflicts or updates.

Detailed Instructions

1. Use the Singleton Pattern to create a ScheduleManager class that manages all tasks.

2. Implement a TaskFactory class to create Task objects.

3. Use the Observer Pattern to alert users if a new task conflicts with an existing one.

Possible Inputs and Corresponding g Outputs

Positive Cases

1. Input: Add Task("Morning Exercise", "07:00", "08:00", "High") Output: Task added successfully. No conflicts.

2. Input: Add Task("Team Meeting", "09:00", "10:00", "Medium") Output: Task added successfully. No conflicts.

3. Input: View Tasks Output:

a. 07:00 - 08:00: Morning Exercise [High]

b. 09:00 - 10:00: Team Meeting [Medium]

4. Input: Remove Task("Morning Exercise") Output: Task removed successfully.

5. Input: Add Task("Lunch Break", "12:00", "13:00", "Low") Output: Task added successfully. No conflicts.

Negative Cases

1. Input: Add Task("Training Session", "09:30", "10:30", "High") Output: Error: Task conflicts with existing task "Team Meeting".

ASSIGNMENT 1
NAME:VANDANA S KAMBLI

2. Input: Remove Task("Non-existent Task") Output: Error: Task not found.

3. Input: Add Task("Invalid Time Task", "25:00", "26:00", "Low") Output: Error: Invalid time format.

4. Input: Add Task("Overlap Task", "08:30", "09:30", "Medium") Output: Error: Task conflicts with existing task "Team Meeting".

5. Input: View Tasks (when no tasks exist) Output: No tasks scheduled for the day.

Evaluation

1. Code Quality: Adherence to best practices, use of design patterns, SOLID principles, and OOP.

2. Functionality: All mandatory requirements implemented correctly.

3. Error Handling: Graceful handling of all errors and edge cases.

4. Performance: Code is optimized for performance.

5. Explanation: Candidate's ability to walk through the code and explain design decisions and logic.

6. Documentation: Code is well-documented, and usage instructions are clear.

The goal of this exercise is to assess the candidate's coding skills, understanding of design patterns, and ability to produce high-quality,

maintainable code.

Solution:

Here’s a plan to approach the assignment, split into two exercises:

Exercise 1: Design Patterns

You need to create six use cases to demonstrate your understanding of design patterns. These will be split across behavioral, creational, and structural patterns.

1. Behavioral Design Patterns

- **Observer Pattern**: Create a scenario where multiple objects need to be notified of changes in another object’s state, like a stock price monitoring system.
- **Command Pattern**: Implement a task manager that records and replays user commands (add, edit, delete).

2. Creational Design Patterns

- **Singleton Pattern**: Demonstrate a logger class where only one instance of the logger exists to ensure a single point for writing logs.
- **Factory Pattern**: Create a shape factory that generates different shapes (circle, square, rectangle) based on user input.

3. Structural Design Patterns

- **Adapter Pattern**: Build an application that integrates two incompatible interfaces, such as a payment processor that works with different payment APIs.

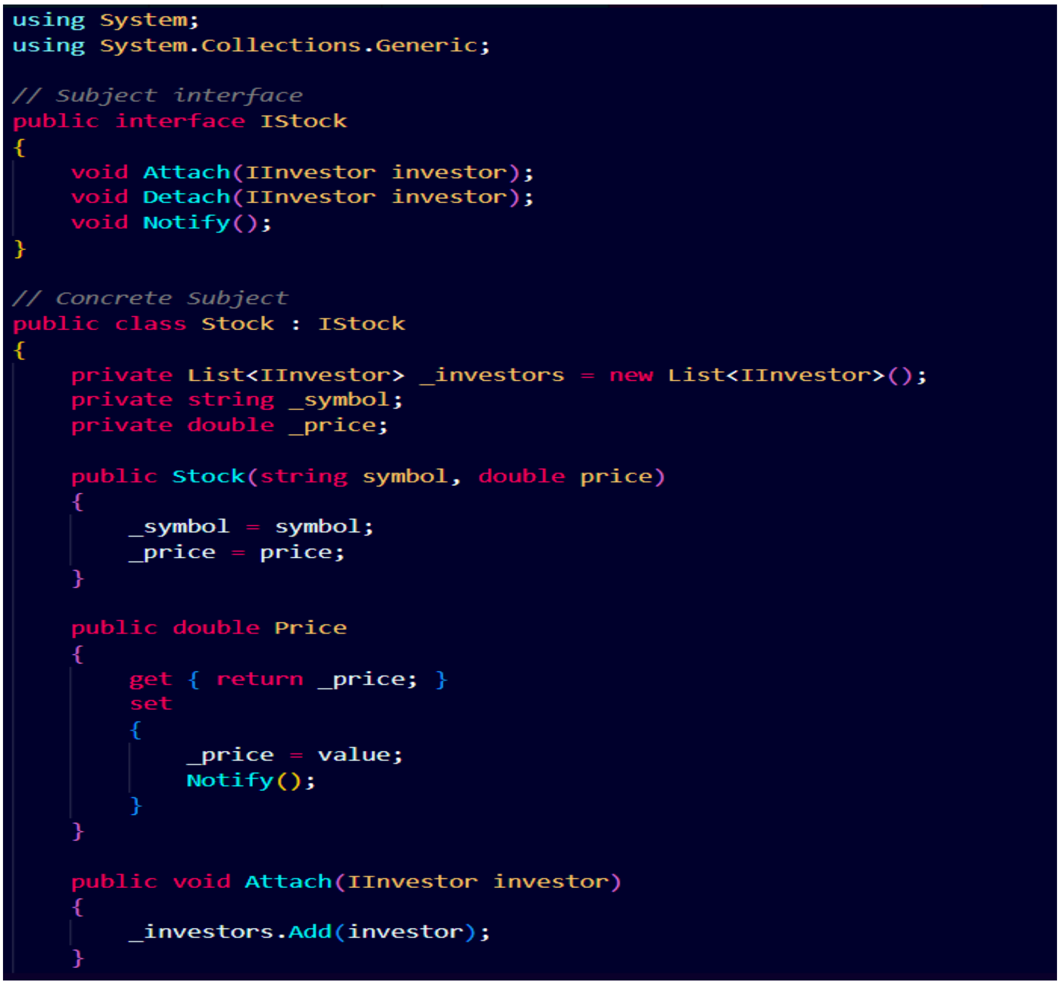
ASSIGNMENT 1
NAME:VANDANA S KAMBLI

- **Decorator Pattern**: Demonstrate an application where objects can be dynamically enhanced with new features (e.g., adding formatting to a text editor).

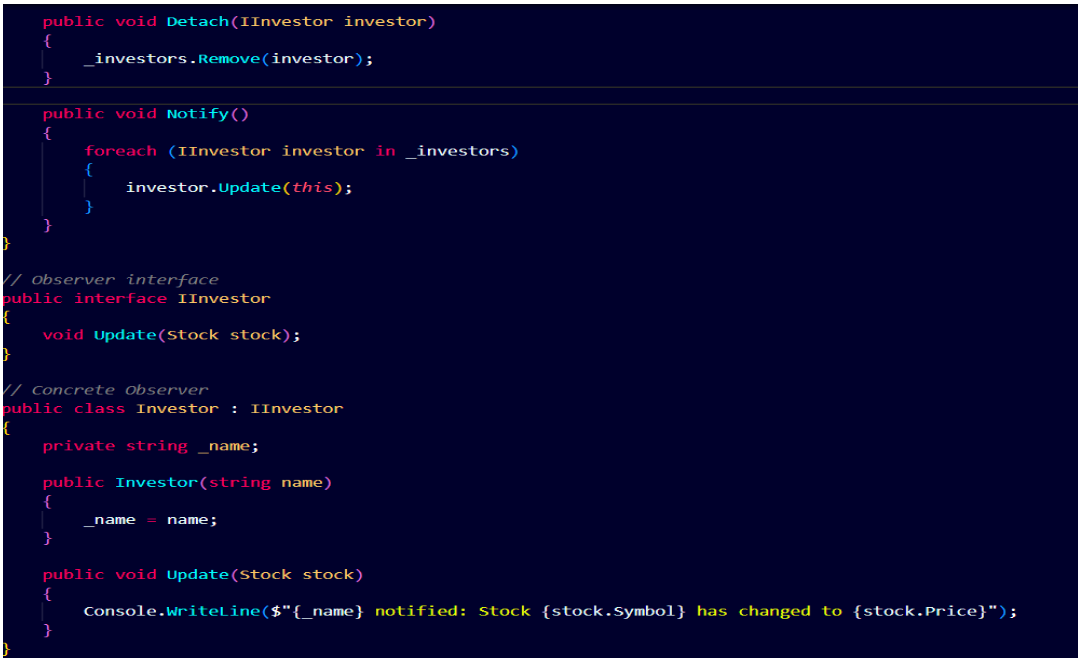
Exercise 1: Design Patterns with Six Use Cases in C#

1. Behavioral Patterns

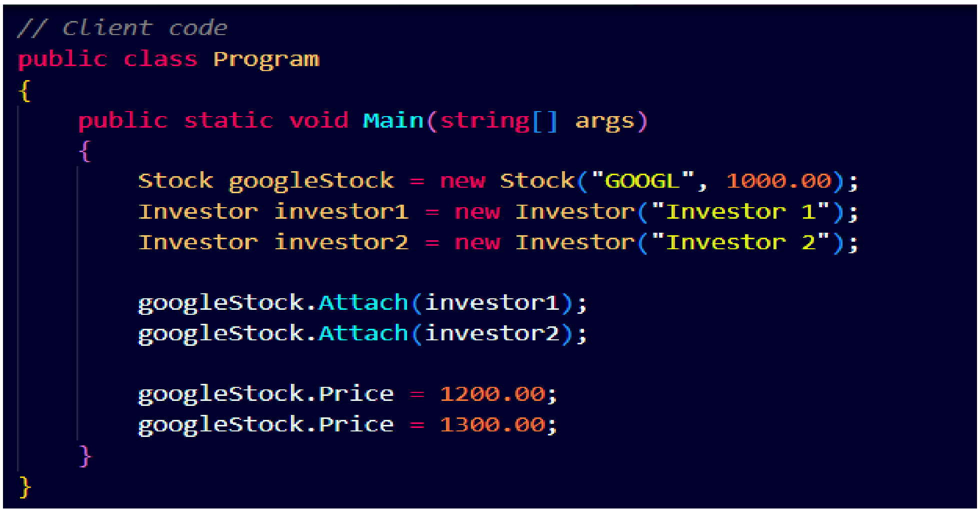
(a) Observer Pattern: Stock Price Monitoring System



```
using System;
using System.Collections.Generic;

// Subject interface
public interface IStock
{
    void Attach(IInvestor investor);
    void Detach(IInvestor investor);
    void Notify();
}

// Concrete Subject
public class Stock : IStock
{
    private List<IInvestor> _investors = new List<IInvestor>();
    private string _symbol;
    private double _price;

    public Stock(string symbol, double price)
    {
        _symbol = symbol;
        _price = price;
    }

    public double Price
    {
        get { return _price; }
        set
        {
            _price = value;
            Notify();
        }
    }

    public void Attach(IInvestor investor)
    {
        _investors.Add(investor);
    }
```



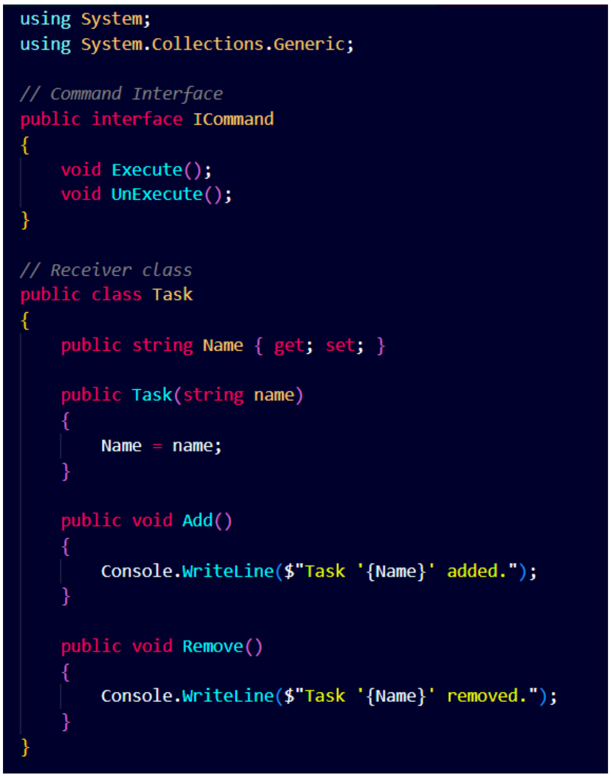
```
    public void Detach(IInvestor investor)
    {
        _investors.Remove(investor);
    }

    public void Notify()
    {
        foreach (IInvestor investor in _investors)
        {
            investor.Update(this);
        }
    }
}

// Observer interface
public interface IInvestor
{
    void Update(Stock stock);
}

// Concrete Observer
public class Investor : IInvestor
{
    private string _name;

    public Investor(string name)
    {
        _name = name;
    }

    public void Update(Stock stock)
    {
        Console.WriteLine($"{_name} notified: Stock {stock.Symbol} has changed to {stock.Price}");
    }
}
```



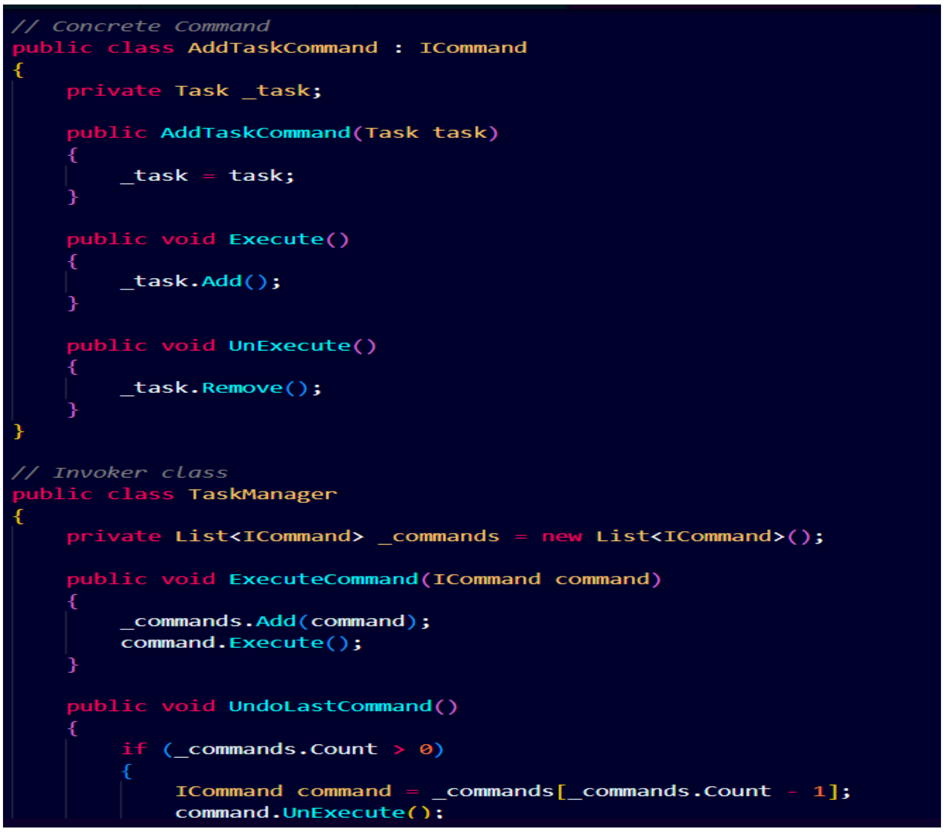
```
// Client code
public class Program
{
    public static void Main(string[] args)
    {
        Stock googleStock = new Stock("GOOGL", 1000.00);
        Investor investor1 = new Investor("Investor 1");
        Investor investor2 = new Investor("Investor 2");

        googleStock.Attach(investor1);
        googleStock.Attach(investor2);

        googleStock.Price = 1200.00;
        googleStock.Price = 1300.00;
    }
}
```

(b) **Command Pattern**: Task Manager (Command Logging)

ASSIGNMENT 1
NAME:VANDANA S KAMBLI

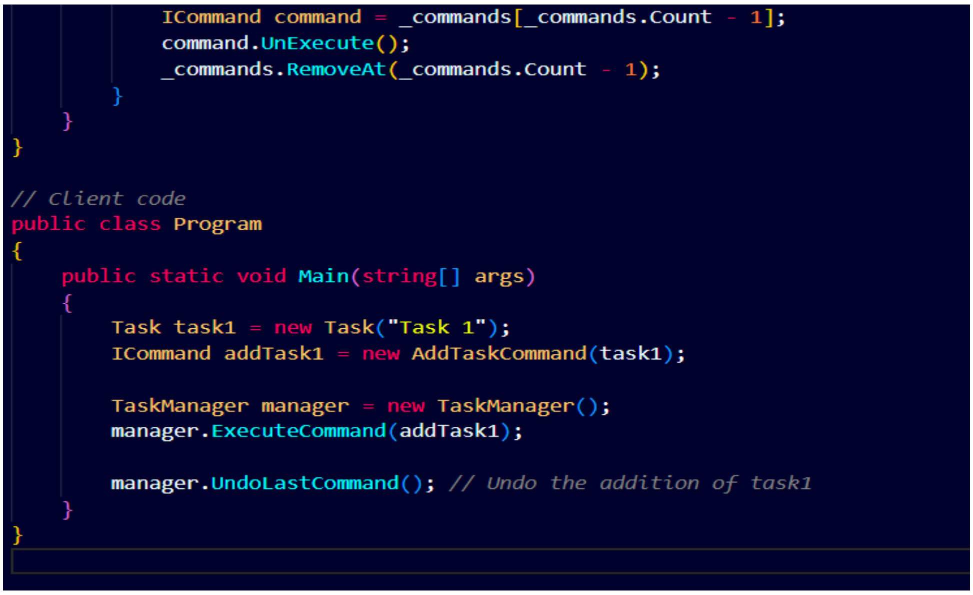


```
using System;
using System.Collections.Generic;

// Command Interface
public interface ICommand
{
    void Execute();
    void UnExecute();
}

// Receiver class
public class Task
{
    public string Name { get; set; }

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

    public void Add()
    {
        Console.WriteLine($"Task '{Name}' added.");
    }

    public void Remove()
    {
        Console.WriteLine($"Task '{Name}' removed.");
    }
}
```



```
// Concrete Command
public class AddTaskCommand : ICommand
{
    private Task _task;

    public AddTaskCommand(Task task)
    {
        _task = task;
    }

    public void Execute()
    {
        _task.Add();
    }

    public void UnExecute()
    {
        _task.Remove();
    }
}

// Invoker class
public class TaskManager
{
    private List<ICommand> _commands = new List<ICommand>();

    public void ExecuteCommand(ICommand command)
    {
        _commands.Add(command);
        command.Execute();
    }

    public void UndoLastCommand()
    {
        if (_commands.Count > 0)
        {
            ICommand command = _commands[_commands.Count - 1];
            command.UnExecute();
```

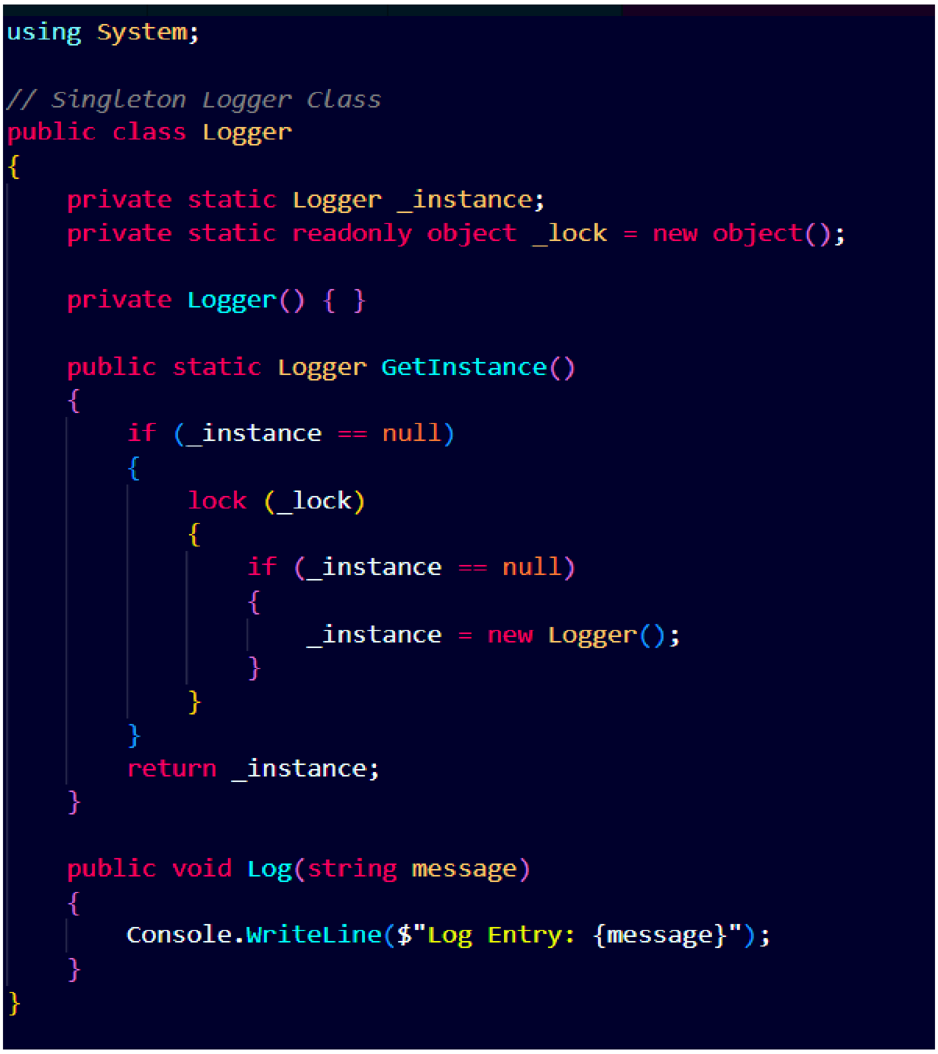
ASSIGNMENT 1
NAME:VANDANA S KAMBLI



```
            ICommand command = _commands[_commands.Count - 1];
            command.UnExecute();
            _commands.RemoveAt(_commands.Count - 1);
        }
    }
}

// Client code
public class Program
{
    public static void Main(string[] args)
    {
        Task task1 = new Task("Task 1");
        ICommand addTask1 = new AddTaskCommand(task1);

        TaskManager manager = new TaskManager();
        manager.ExecuteCommand(addTask1);

        manager.UndoLastCommand(); // Undo the addition of task1
    }
}
```

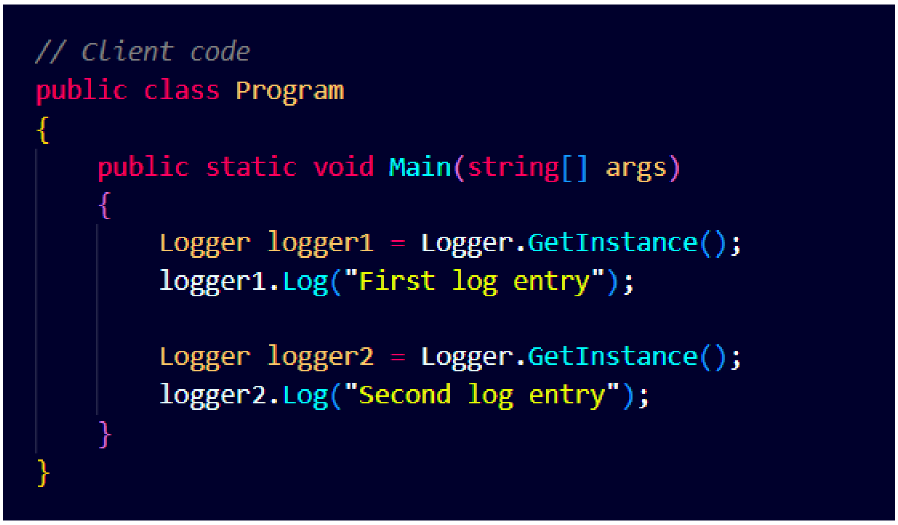
2. Creational Patterns

(a) Singleton Pattern: Logger



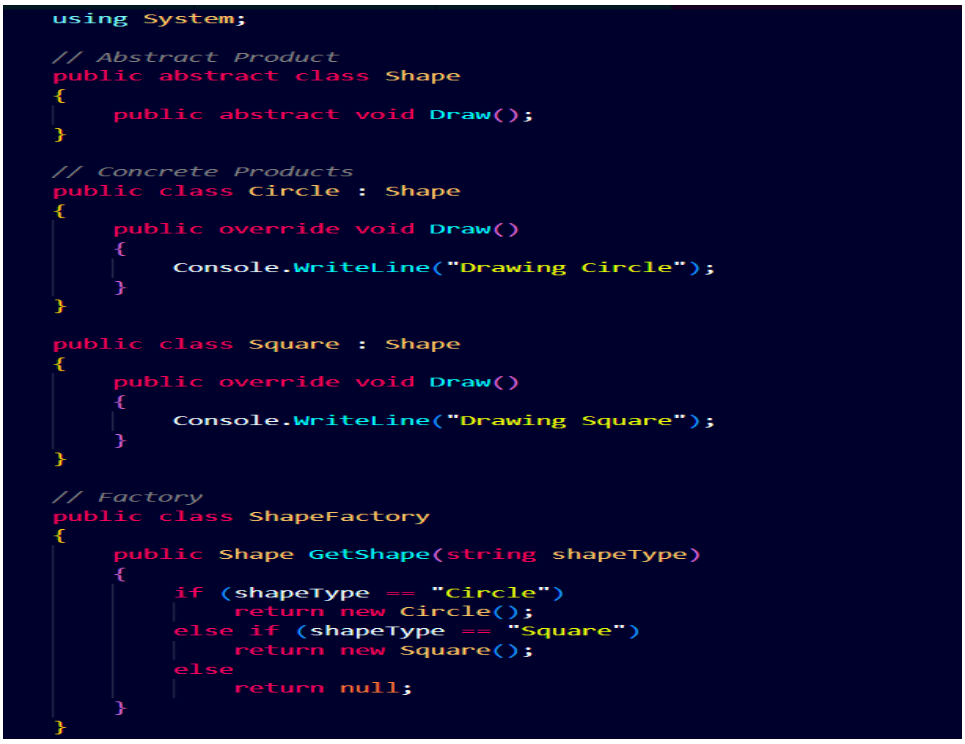
```
using System;

// Singleton Logger Class
public class Logger
{
    private static Logger _instance;
    private static readonly object _lock = new object();

    private Logger() { }

    public static Logger GetInstance()
    {
        if (_instance == null)
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    _instance = new Logger();
                }
            }
        }
        return _instance;
    }

    public void Log(string message)
    {
        Console.WriteLine($"Log Entry: {message}");
    }
}
```

ASSIGNMENT 1
NAME:VANDANA S KAMBLI



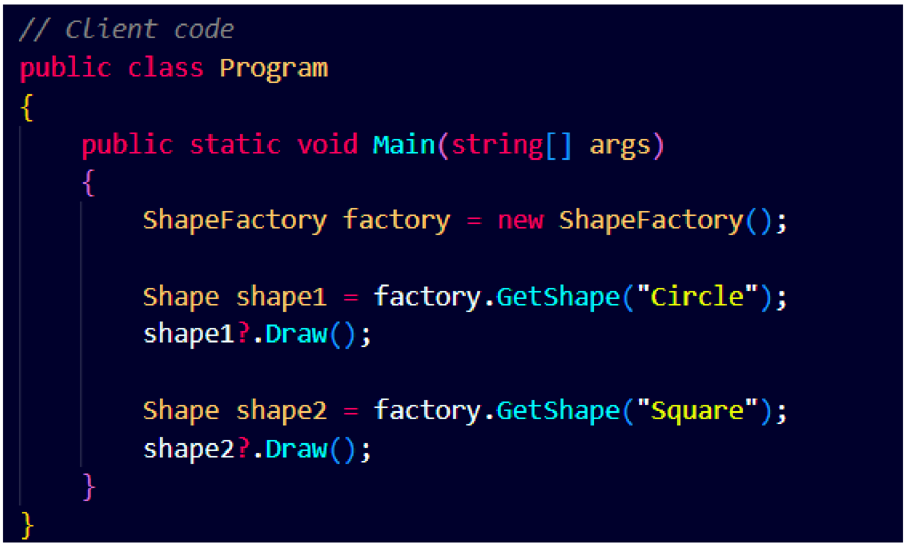
```
// Client code
public class Program
{
    public static void Main(string[] args)
    {
        Logger logger1 = Logger.GetInstance();
        logger1.Log("First log entry");

        Logger logger2 = Logger.GetInstance();
        logger2.Log("Second log entry");
    }
}
```

(b) **Factory Pattern**: Shape Factory



```
using System;

// Abstract Product
public abstract class Shape
{
    public abstract void Draw();
}

// Concrete Products
public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing Circle");
    }
}

public class Square : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing Square");
    }
}

// Factory
public class ShapeFactory
{
    public Shape GetShape(string shapeType)
    {
        if (shapeType == "Circle")
            return new Circle();
        else if (shapeType == "Square")
            return new Square();
        else
            return null;
    }
}
```

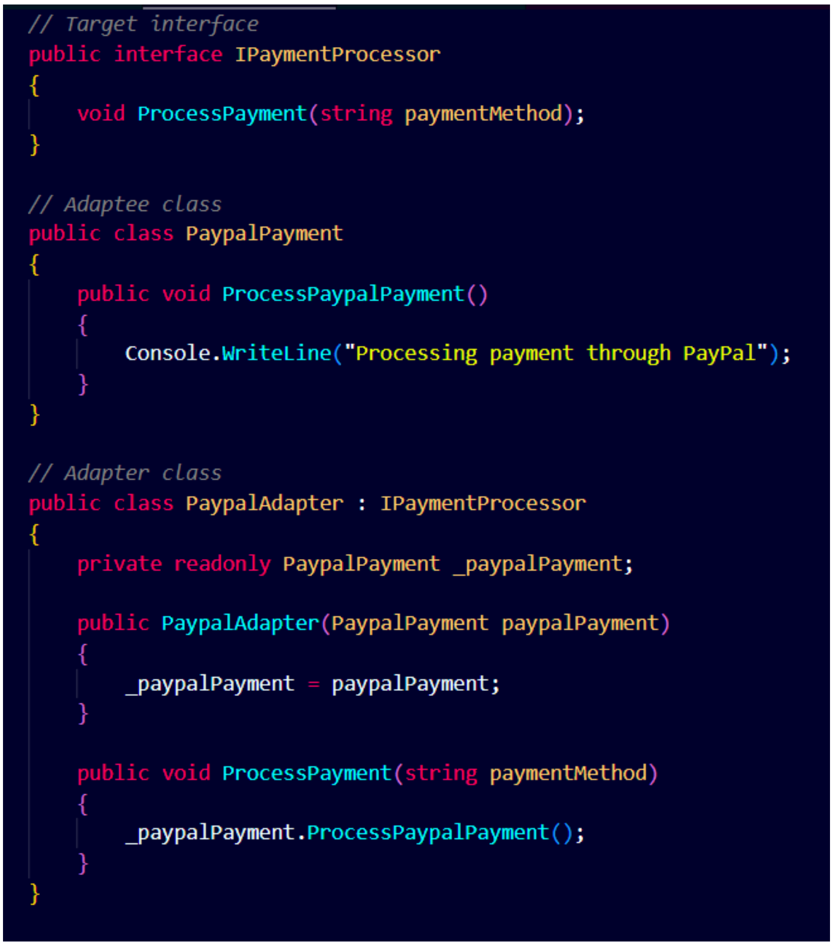
ASSIGNMENT 1
NAME:VANDANA S KAMBLI



```
// Client code
public class Program
{
    public static void Main(string[] args)
    {
        ShapeFactory factory = new ShapeFactory();

        Shape shape1 = factory.GetShape("Circle");
        shape1?.Draw();

        Shape shape2 = factory.GetShape("Square");
        shape2?.Draw();
    }
}
```

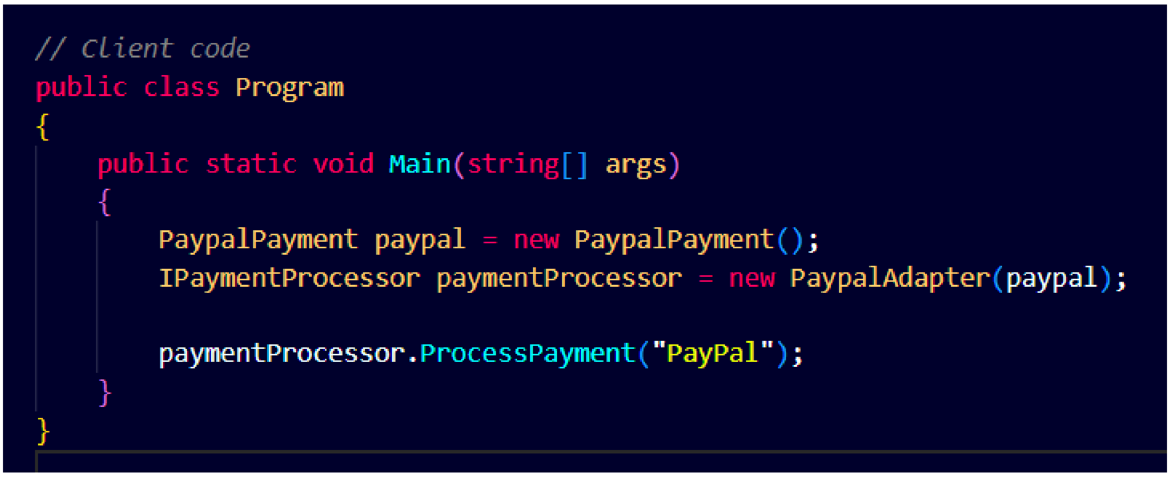
3. Structural Patterns

(a) Adapter Pattern: Payment Processor Integration



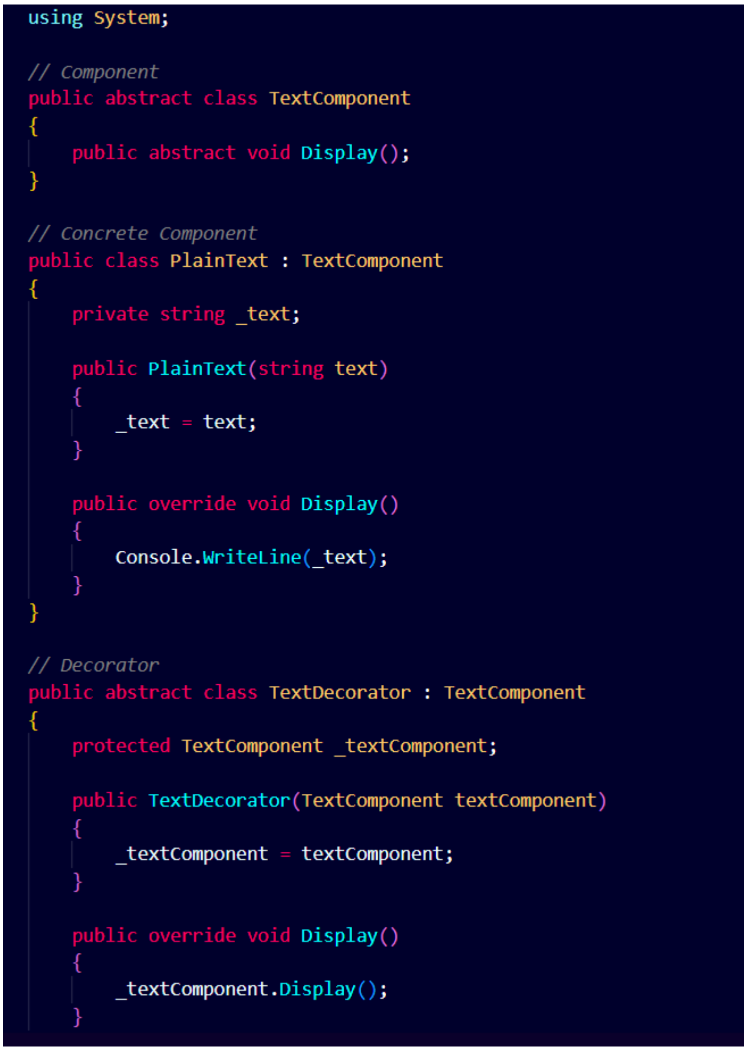
```
// Target interface
public interface IPaymentProcessor
{
    void ProcessPayment(string paymentMethod);
}

// Adaptee class
public class PaypalPayment
{
    public void ProcessPaypalPayment()
    {
        Console.WriteLine("Processing payment through PayPal");
    }
}

// Adapter class
public class PaypalAdapter : IPaymentProcessor
{
    private readonly PaypalPayment _paypalPayment;

    public PaypalAdapter(PaypalPayment paypalPayment)
    {
        _paypalPayment = paypalPayment;
    }

    public void ProcessPayment(string paymentMethod)
    {
        _paypalPayment.ProcessPaypalPayment();
    }
}
```

ASSIGNMENT 1
NAME:VANDANA S KAMBLI



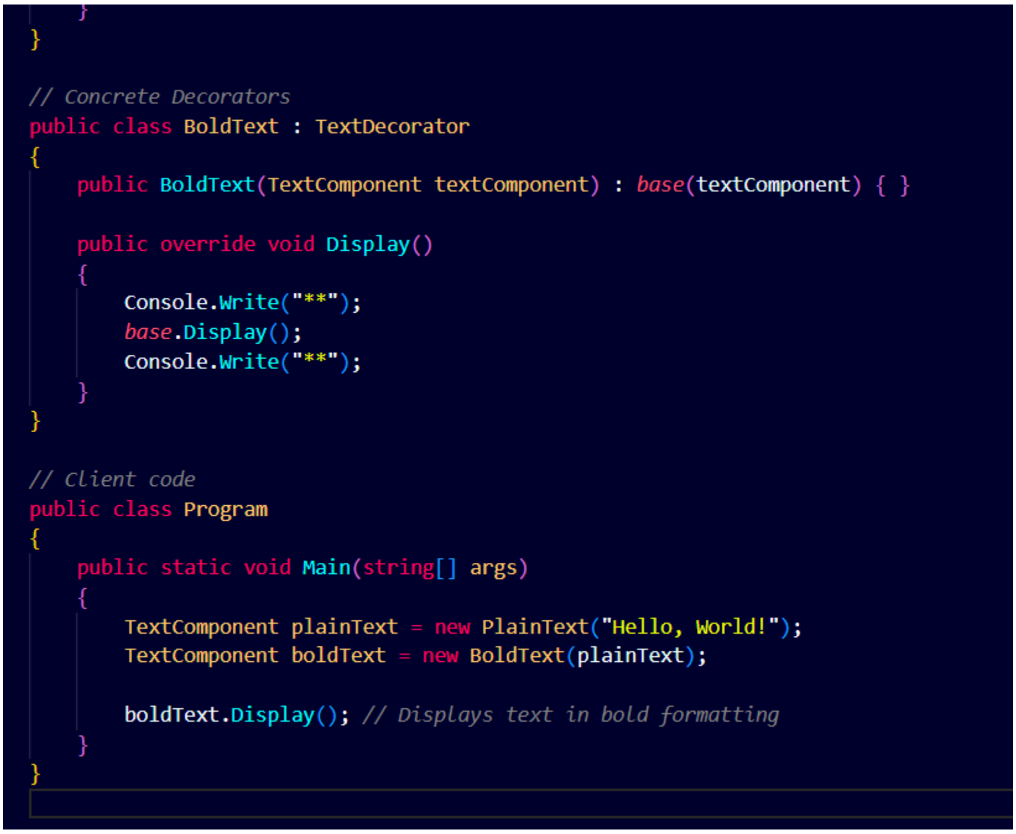
```
// Client code
public class Program
{
    public static void Main(string[] args)
    {
        PaypalPayment paypal = new PaypalPayment();
        IPaymentProcessor paymentProcessor = new PaypalAdapter(paypal);

        paymentProcessor.ProcessPayment("PayPal");
    }
}
```

(b) Decorator Pattern: Text Editor Formatting



```
using System;

// Component
public abstract class TextComponent
{
    public abstract void Display();
}

// Concrete Component
public class PlainText : TextComponent
{
    private string _text;

    public PlainText(string text)
    {
        _text = text;
    }

    public override void Display()
    {
        Console.WriteLine(_text);
    }
}

// Decorator
public abstract class TextDecorator : TextComponent
{
    protected TextComponent _textComponent;

    public TextDecorator(TextComponent textComponent)
    {
        _textComponent = textComponent;
    }

    public override void Display()
    {
        _textComponent.Display();
    }
```

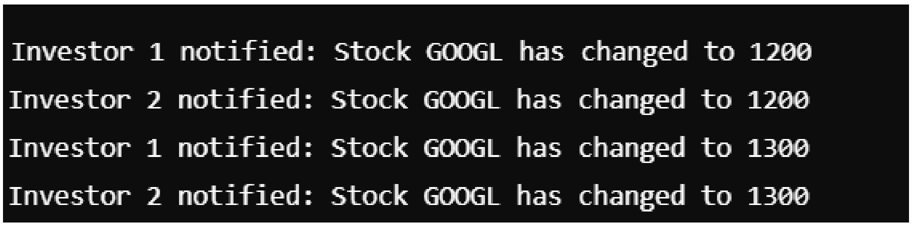
ASSIGNMENT 1
NAME:VANDANA S KAMBLI



```
    }
}

// Concrete Decorators
public class BoldText : TextDecorator
{
    public BoldText(TextComponent textComponent) : base(textComponent) { }

    public override void Display()
    {
        Console.Write("**");
        base.Display();
        Console.Write("**");
    }
}

// Client code
public class Program
{
    public static void Main(string[] args)
    {
        TextComponent plainText = new PlainText("Hello, World!");
        TextComponent boldText = new BoldText(plainText);

        boldText.Display(); // Displays text in bold formatting
    }
}
```

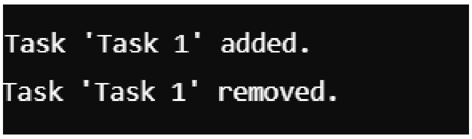
Exercise 1: Design Patterns - Outputs

1. Observer Pattern Output:



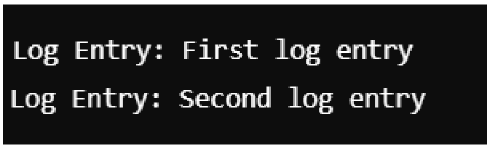
```
Investor 1 notified: Stock GOOGL has changed to 1200
Investor 2 notified: Stock GOOGL has changed to 1200
Investor 1 notified: Stock GOOGL has changed to 1300
Investor 2 notified: Stock GOOGL has changed to 1300
```

2. Command Pattern Output:



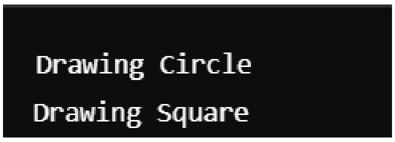
```
Task 'Task 1' added.
Task 'Task 1' removed.
```

3. Singleton Pattern Output:



```
Log Entry: First log entry
Log Entry: Second log entry
```

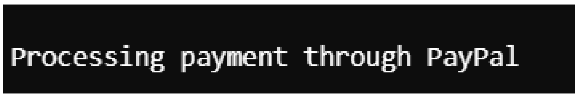
4. Factory Pattern Output:



```
Drawing Circle
Drawing Square
```

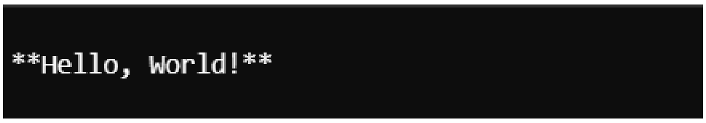
ASSIGNMENT 1
NAME:VANDANA S KAMBLI

5. Adapter Pattern Output:



```
Processing payment through PayPal
```

6. Decorator Pattern Output:



```
**Hello, World!**
```