Solid Exercises

Exercise: Refactoring to Follow the ISP

Problem Statement

Consider the following interface and classes for a media player system:

```
public interface IMediaPlayer
{
    void PlayAudio();
    void PlayVideo();
    void DisplaySubtitles();
    void LoadMedia(string filePath);
}

public class AudioPlayer : IMediaPlayer
{
    public void PlayAudio()
    {
        // Code to play audio
    }

    public void PlayVideo()
    {
        throw new NotImplementedException("Audio players cannot play
videos.");
    }

    public void DisplaySubtitles()
    {
        throw new NotImplementedException("Audio players cannot display
subtitles.");
    }

    public void LoadMedia(string filePath)
    {
        // Code to load audio file
    }
}

public class VideoPlayer : IMediaPlayer
{
    public void PlayAudio()
    {
        throw new NotImplementedException("Video players cannot play audio
```

```
without video.");
    }

    public void PlayVideo()
    {
        // Code to play video
    }

    public void DisplaySubtitles()
    {
        // Code to display subtitles
    }

    public void LoadMedia(string filePath)
    {
        // Code to load video file
    }
}
```

In this example, the `IMediaPlayer` interface defines methods for playing audio, playing video, displaying subtitles, and loading media. However, not all media players need to implement all of these methods, violating the ISP.

Tasks

1. Identify the ISP violation in the provided code.
2. Refactor the code to adhere to the ISP by segregating the interface into smaller, focused interfaces.
3. Update the `AudioPlayer` and `VideoPlayer` classes to implement only the relevant interfaces.
4. Write comments to explain your refactoring approach and how it addresses the ISP violation.

Guidelines

- Use appropriate naming conventions for interfaces and methods.
- Ensure that each interface is focused on a specific responsibility or behavior.
- Consider separating the loading functionality into a separate interface if needed.
- Write comments to explain the purpose and responsibilities of each interface.

Exercise: Refactoring to Follow the DIP

Problem Statement

Consider the following classes for a file processing system:

```
public class FileProcessor
{
    private FileReader _fileReader;
    private FileWriter _fileWriter;

    public FileProcessor()
    {
        _fileReader = new FileReader();
        _fileWriter = new FileWriter();
    }

    public void ProcessFile(string inputFilePath, string outputFilePath)
    {
        string fileContent = _fileReader.ReadFile(inputFilePath);
        // Process the file content
        _fileWriter.WriteFile(outputFilePath, fileContent);
    }
}

public class FileReader
{
    public string ReadFile(string filePath)
    {
        // Code to read file content
        return "File content";
    }
}

public class FileWriter
{
    public void WriteFile(string filePath, string content)
    {
        // Code to write file content
    }
}
```

In this example, the `FileProcessor` class is tightly coupled to the `FileReader` and `FileWriter` classes, violating the DIP. If we need to change the file reading or writing implementation, we would have to modify the `FileProcessor` class, making the system more rigid and harder to maintain.

Tasks

1. Identify the violation of the DIP in the provided code.
2. Refactor the code to adhere to the DIP by introducing abstractions and inverting the dependencies.
3. Update the `FileProcessor` class to use the new abstractions.

4. Write comments to explain your refactoring approach and how it addresses the DIP violation.

Full SOLID Exercise

```
public class ECommerceSystem
{
    private List<Product> products = new List<Product>();
    private List<Order> orders = new List<Order>();

    public void AddProduct(string name, decimal price, int quantity)
    {
        products.Add(new Product { Name = name, Price = price, Quantity =
quantity });
    }

    public void PlaceOrder(string customerName, List<int> productIds, string
paymentMethod)
    {
        decimal totalCost = 0;
        List<Product> orderedProducts = new List<Product>();

        foreach (int productId in productIds)
        {
            Product product = products.Find(p => p.Id == productId);
            if (product != null && product.Quantity > 0)
            {
                orderedProducts.Add(product);
                totalCost += product.Price;
                product.Quantity--;
            }
        }

        if (orderedProducts.Count > 0)
        {
            if (paymentMethod == "CreditCard")
            {
                ProcessCreditCardPayment(totalCost);
            }
            else if (paymentMethod == "PayPal")
            {
                ProcessPayPalPayment(totalCost);
            }

            Order order = new Order
            {
                CustomerName = customerName,
                Products = orderedProducts,
```

```
                TotalCost = totalCost
            };

            orders.Add(order);
            SendOrderConfirmationEmail(order);
        }
    }

    private void ProcessCreditCardPayment(decimal amount)
    {
        // Process credit card payment
        Console.WriteLine($"Processing credit card payment of ${amount}");
    }

    private void ProcessPayPalPayment(decimal amount)
    {
        // Process PayPal payment
        Console.WriteLine($"Processing PayPal payment of ${amount}");
    }

    private void SendOrderConfirmationEmail(Order order)
    {
        string message = $"Order confirmation for {order.CustomerName}:\n";
        message += $"Total Cost: ${order.TotalCost}\n";
        message += "Products:\n";
        foreach (Product product in order.Products)
        {
            message += $"- {product.Name} (${product.Price})\n";
        }

        // Send email
        Console.WriteLine(message);
    }
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
}

public class Order
{
    public string CustomerName { get; set; }
    public List<Product> Products { get; set; }
    public decimal TotalCost { get; set; }
}
```

Exercise:

1. Identify the SOLID principles that are being violated in the provided code.
2. Refactor the code to follow the SOLID principles by separating concerns, introducing interfaces, and applying appropriate design patterns.
3. Ensure that the refactored code maintains the same functionality as the original code while improving maintainability, extensibility, and testability.

You can start by separating the responsibilities of product management, order processing, payment handling, and notification sending into different classes or modules. Then, introduce interfaces and abstract classes to promote abstraction and loose coupling. Apply dependency inversion by letting higher-level modules depend on abstractions rather than concrete implementations.