SOLID Exercise Solution

Sure, let's refactor the provided code to follow the SOLID principles. I'll explain the thought process and the changes made at each step.

Step 1: Separate Responsibilities (Single Responsibility Principle)

We'll start by separating the responsibilities of product management, order management, payment processing, and notification sending into different classes or modules.

```
// Product Management
public class ProductRepository
{
    private List<Product> products = new List<Product>();

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

    public Product GetProductById(int id)
    {
        return products.Find(p => p.Id == id);
    }

    // Additional methods for managing products
}

// Order Management
public class OrderService
{
    private readonly ProductRepository _productRepository;
    private readonly IPaymentProcessor _paymentProcessor;
    private readonly INotificationService _notificationService;

    public OrderService(ProductRepository productRepository,
IPaymentProcessor paymentProcessor, INotificationService
notificationService)
    {
        _productRepository = productRepository;
        _paymentProcessor = paymentProcessor;
        _notificationService = notificationService;
    }

    public void PlaceOrder(string customerName, List<int> productIds, string
paymentMethod)
    {
        // Implementation ...
    }
}

// Payment Processing (Interface and concrete implementations)
public interface IPaymentProcessor
{
    void ProcessPayment(decimal amount);
}

public class CreditCardPaymentProcessor : IPaymentProcessor
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing credit card payment of ${amount}");
    }
}

public class PayPalPaymentProcessor : IPaymentProcessor
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing PayPal payment of ${amount}");
    }
}

// Notification Service (Interface and concrete implementation)
public interface INotificationService
{
    void SendNotification(string message);
}

public class EmailNotificationService : INotificationService
{
    public void SendNotification(string message)
    {
        Console.WriteLine(message);
    }
}
```

Explanation:

- We created separate classes for `ProductRepository`, `OrderService`, `IPaymentProcessor` (and its implementations), and `INotificationService` (and its implementation).

- Each class now has a single, well-defined responsibility, adhering to the Single Responsibility Principle.
- The `OrderService` depends on abstractions (`IPaymentProcessor` and `INotificationService`) instead of concrete implementations, following the Dependency Inversion Principle.

Step 2: Introduce Abstractions (Open/Closed Principle and Interface Segregation Principle)

We'll introduce an abstract base class for payment processors and an interface for notification services to promote abstraction and open/closed design.

```
// Payment Processing (Abstract base class and derived classes)
public abstract class PaymentProcessor : IPaymentProcessor
{
    public abstract void ProcessPayment(decimal amount);
}

public class CreditCardPaymentProcessor : PaymentProcessor
{
    public override void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing credit card payment of ${amount}");
    }
}

public class PayPalPaymentProcessor : PaymentProcessor
{
    public override void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing PayPal payment of ${amount}");
    }
}

// Notification Service (Interface)
public interface INotificationService
{
    void SendNotification(string message);
}
```

Explanation:

- We introduced an abstract base class `PaymentProcessor` that implements the `IPaymentProcessor` interface.
- Concrete payment processor classes (`CreditCardPaymentProcessor` and `PayPalPaymentProcessor`) now inherit from `PaymentProcessor` .

- This design allows for easy extension of new payment processors without modifying existing code, following the Open/Closed Principle.
- The `INotificationService` interface segregates the responsibility of notification sending, adhering to the Interface Segregation Principle.

Step 3: Ensure Liskov Substitution Principle

To ensure the Liskov Substitution Principle, we need to make sure that derived classes can be substituted for their base classes without breaking the application.

```
// Payment Processing (Abstract base class and derived classes)
public abstract class PaymentProcessor : IPaymentProcessor
{
    public abstract void ProcessPayment(decimal amount);

    // Optional: Additional shared behavior or properties
}

public class CreditCardPaymentProcessor : PaymentProcessor
{
    public override void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing credit card payment of ${amount}");
    }
}

public class PayPalPaymentProcessor : PaymentProcessor
{
    public override void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing PayPal payment of ${amount}");
    }
}
```

Explanation:

- The `PaymentProcessor` abstract base class and its derived classes (`CreditCardPaymentProcessor` and `PayPalPaymentProcessor`) follow the Liskov Substitution Principle.
- Any instance of a derived class can be substituted for an instance of the base class without breaking the application.
- If we need to add shared behavior or properties across all payment processors, we can add them to the `PaymentProcessor` base class.

Step 4: Dependency Injection and Usage

Finally, we'll use dependency injection to wire up the dependencies and see the refactored code in action.

```
// Usage
public static void Main(string[] args)
{
    // Instantiate dependencies
    var productRepository = new ProductRepository();
    var paymentProcessor = new CreditCardPaymentProcessor(); // or new
PayPalPaymentProcessor()
    var notificationService = new EmailNotificationService();

    // Add some products
    productRepository.AddProduct("Product A", 10.99m, 5);
    productRepository.AddProduct("Product B", 19.99m, 3);

    // Create an order service and place an order
    var orderService = new OrderService(productRepository, paymentProcessor,
notificationService);
    orderService.PlaceOrder("John Doe", new List<int> { 1, 2 },
"CreditCard");
}
```

Explanation:

- In the `Main` method, we instantiate the required dependencies (`ProductRepository` , `PaymentProcessor` , and `NotificationService`).
- We create an instance of `OrderService` and pass the dependencies through constructor injection.
- We can easily swap out the concrete implementations of `PaymentProcessor` and `NotificationService` without modifying the `OrderService` class, thanks to the dependency inversion principle.

By following these steps, we've refactored the original monolithic code to adhere to the SOLID principles. The resulting code is more modular, maintainable, and extensible. Each class has a single responsibility, and dependencies are managed through abstractions, promoting loose coupling and flexibility.