**Курс:** Шаблоны проектирования приложений

**Тема:** Модуль 04 Принципы проектирования

*Баллы:*

**Произведите корректную (правильную) по вашему мнению реализацию с применением принципа Single-Responsibility Principle (SRP):**

Рассмотрим класс **Invoice**, который нарушает принцип SRP, поскольку он занимается как расчетом стоимости, так и сохранением счета-фактуры в базу данных:

public class Invoice

{

public int Id { get; set; }

public List<Item> Items { get; set; }

public double TaxRate { get; set; }

public double CalculateTotal()

{

double subTotal = 0;

foreach (var item in Items)

{

subTotal += item.Price;

}

return subTotal + (subTotal \* TaxRate);

}

public void SaveToDatabase()

{

// Логика для сохранения счета-фактуры в базу данных

}

}

В этом примере класс **Invoice** имеет две ответственности: расчёт стоимости и сохранение в базу данных. Необходимо разделить обязанности на два класса: один для расчета суммы, другой — для сохранения в базу данных.

Код должен содержать три класса:

* Invoice — отвечает за представление данных счета-фактуры.
* InvoiceCalculator — отвечает за расчет суммы счета-фактуры.
* InvoiceRepository — отвечает за сохранение счета-фактуры в базу данных.

**Произведите корректную (правильную) по вашему мнению реализацию с применением принципа Open-Closed Principle, OCP:**

Рассмотрим пример, в котором класс **DiscountCalculator** нарушает принцип OCP, поскольку каждый раз при добавлении нового типа скидки нужно изменять существующий код:

public enum CustomerType

{

Regular,

Silver,

Gold

}

public class DiscountCalculator

{

public double CalculateDiscount(CustomerType customerType, double amount)

{

if (customerType == CustomerType.Regular)

{

return amount;

}

else if (customerType == CustomerType.Silver)

{

return amount \* 0.9; // 10% скидка

}

else if (customerType == CustomerType.Gold)

{

return amount \* 0.8; // 20% скидка

}

else

{

throw new ArgumentException("Неизвестный тип клиента");

}

}

}

Если потребуется добавить новый тип клиента, например, Platinum, то потребуется модифицировать метод **CalculateDiscount**, что нарушает принцип OCP.

Необходимо используя полиморфизм, сделать класс **DiscountCalculator** открытым для расширения, но закрытым для модификации.

**Произведите корректную (правильную) по вашему мнению реализацию с применением принципа** **Interface Segregation Principle, ISP:**

Рассмотрим пример интерфейса **IWorker**, который объединяет слишком много методов:

public interface IWorker

{

void Work();

void Eat();

void Sleep();

}

public class HumanWorker : IWorker

{

public void Work()

{

// Логика работы

}

public void Eat()

{

// Логика питания

}

public void Sleep()

{

// Логика сна

}

}

public class RobotWorker : IWorker

{

public void Work()

{

// Логика работы

}

public void Eat()

{

// Робот не ест, но вынужден реализовать метод

throw new NotImplementedException();

}

public void Sleep()

{

// Робот не спит, но вынужден реализовать метод

throw new NotImplementedException();

}

}

В этом примере класс **RobotWorker** вынужден реализовывать методы, которые ему не нужны (Eat и Sleep). Это нарушение принципа ISP.

Чтобы соблюсти принцип ISP, вам необходимо разделить интерфейс **IWorker** на несколько специализированных интерфейсов.

**Произведите корректную (правильную) по вашему мнению реализацию с применением принципа** **Dependency-Inversion Principle, DIP:**

Рассмотрим пример, где класс Notification зависит от конкретной реализации класса EmailService:

public class EmailService

{

public void SendEmail(string message)

{

Console.WriteLine($"Отправка Email: {message}");

}

}

public class Notification

{

private EmailService \_emailService;

public Notification()

{

\_emailService = new EmailService();

}

public void Send(string message)

{

\_emailService.SendEmail(message);

}

}

В этом примере класс Notification жестко связан с конкретной реализацией EmailService. Если в будущем нужно будет изменить способ отправки уведомлений (например, добавить SMS или push-уведомления), придется изменять класс Notification, что нарушает DIP.

Чтобы соблюдать DIP, вам необходимо использовать абстракцию в виде интерфейса для отделения высокоуровневого модуля от низкоуровневого.