**Курс:** Шаблоны проектирования приложений

**Тема:** Модуль 04 Принципы проектирования

*Цель задания:*

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

В этом примере класс **Order** отвечает за несколько вещей: хранение данных о заказе, расчет стоимости заказа с учетом скидок, обработку платежа и отправку уведомления пользователю.

public class Order

{

public string ProductName { get; set; }

public int Quantity { get; set; }

public double Price { get; set; }

public double CalculateTotalPrice()

{

// Рассчет стоимости с учетом скидок

return Quantity \* Price \* 0.9;

}

public void ProcessPayment(string paymentDetails)

{

// Логика обработки платежа

Console.WriteLine("Payment processed using: " + paymentDetails);

}

public void SendConfirmationEmail(string email)

{

// Логика отправки уведомления

Console.WriteLine("Confirmation email sent to: " + email);

}

}

**Проблемы:**

* Класс Order нарушает принцип SRP, так как он отвечает за несколько вещей: расчет цены, обработку платежа и отправку уведомлений. Это усложняет код и делает его менее гибким.

В этом примере Вам необходимо разделить ответственность между несколькими классами. Класс Order должен отвечает только за хранение данных о заказе. Другие задачи, такие как расчет цены, обработка платежа и отправка уведомлений, должны быть делегированы другим классам.

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

Расчет зарплаты сотрудников

В этом примере класс EmployeeSalaryCalculator нарушает принцип OCP, так как для добавления новой логики расчета зарплаты приходится изменять код метода CalculateSalary.

public class Employee

{

public string Name { get; set; }

public double BaseSalary { get; set; }

public string EmployeeType { get; set; } // "Permanent", "Contract", "Intern"

}

public class EmployeeSalaryCalculator

{

public double CalculateSalary(Employee employee)

{

if (employee.EmployeeType == "Permanent")

{

return employee.BaseSalary \* 1.2; // Permanent employee gets 20% bonus

}

else if (employee.EmployeeType == "Contract")

{

return employee.BaseSalary \* 1.1; // Contract employee gets 10% bonus

}

else if (employee.EmployeeType == "Intern")

{

return employee.BaseSalary \* 0.8; // Intern gets 80% of the base salary

}

else

{

throw new NotSupportedException("Employee type not supported");

}

}

}

**Проблемы:**

* Если нужно добавить новый тип сотрудника, например, "Freelancer", придется изменить метод CalculateSalary. Это нарушает принцип OCP, так как мы изменяем уже существующий код, что может привести к ошибкам.

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

Работа с принтерами

В этом примере интерфейс IPrinter содержит методы для различных типов принтеров: обычного принтера, сканера и факса. Но что если какой-то принтер поддерживает только печать и сканирование, но не поддерживает отправку факсов? Такой принтер будет вынужден реализовывать метод Fax, который ему не нужен.

public interface IPrinter

{

void Print(string content);

void Scan(string content);

void Fax(string content);

}

public class AllInOnePrinter : IPrinter

{

public void Print(string content)

{

Console.WriteLine("Printing: " + content);

}

public void Scan(string content)

{

Console.WriteLine("Scanning: " + content);

}

public void Fax(string content)

{

Console.WriteLine("Faxing: " + content);

}

}

public class BasicPrinter : IPrinter

{

public void Print(string content)

{

Console.WriteLine("Printing: " + content);

}

public void Scan(string content)

{

throw new NotImplementedException();

}

public void Fax(string content)

{

throw new NotImplementedException();

}

}

**Проблемы:**

* Класс BasicPrinter вынужден реализовывать методы Scan и Fax, даже если он их не поддерживает. Это нарушает принцип ISP, так как он зависит от методов, которые ему не нужны.

Вам необходимо разделить IPrinter на несколько более мелких интерфейсов, каждый из которых описывает отдельную функциональность.

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

Система уведомлений

В этом примере класс NotificationService напрямую зависит от конкретных классов EmailSender и SmsSender. Если нужно добавить новый тип уведомления, например, через мессенджер, придётся изменить класс NotificationService.

public class EmailSender

{

public void SendEmail(string message)

{

Console.WriteLine("Email sent: " + message);

}

}

public class SmsSender

{

public void SendSms(string message)

{

Console.WriteLine("SMS sent: " + message);

}

}

public class NotificationService

{

private EmailSender emailSender = new EmailSender();

private SmsSender smsSender = new SmsSender();

public void SendNotification(string message)

{

emailSender.SendEmail(message);

smsSender.SendSms(message);

}

}

**Проблемы:**

* Класс NotificationService жестко связан с конкретными реализациями EmailSender и SmsSender. Если потребуется изменить способ отправки уведомлений или добавить новый способ, придется изменять код NotificationService.