Министерство образования Республики Беларусь

Учреждение образования  
 БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
 ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

Рефакторинг и оптимизация программного кода

Отчет

по результатам выполнения задания к практическому занятию №1

на тему:

**ОСНОВНЫЕ ПРОБЛЕМЫ И ОШИБКИ  
 ПРИ НАПИСАНИИ ПРОГРАММНОГО КОДА**

|  |  |  |
| --- | --- | --- |
| Проверил |  | А.В. Шелест |
|  | (подпись) |  |
| зачтено |  |  |
|  | (дата защиты) |  |
|  |  |  |
| Выполнил |  | Ю-Ф.А Тюленев  гр. 114301 |
|  | (подпись) |  |

Минск 2025

**Цель**

Изучить литературные и Интернет источники на предмет наиболее распространенных проблем и ошибок, возникающих при написании кода на выбранном языке программирования.

**Задание**

Исследовать литературу и материалы в сети Интернет по теме практического занятия. Представить примеры некачественного программного кода (как собственные, так и из источников) характерные для выбранного языка программирования и/или фреймворка (примеры некачественного кода: длинные методы, дублирование кода, плохие имена функций, избыточные временные переменные, утечка памяти, длительное выполнение операций и прочие). Привести способы и методы устранения описанных ранее проблем и ошибок.

Вводные данные

Язык программирования – *C#*.

**Проблема несоблюдения принципов *SOLID.***

1. Принцип единственной обязанности (Single Responsibility Principle) можно сформулировать так: Каждый компонент должен иметь одну и только одну причину для изменения.

В C# в качестве компонента может выступать класс, структура, метод. А под обязанностью здесь понимается набор действий, которые выполняют единую задачу. То есть суть принципа заключается в том, что класс/структура/метод должны выполнять одну единственную задачу. Весь функционал компонента должен быть целостным, обладать высокой связностью (high cohesion).

Конкретное применение принципа зависит от контекста. В данном случае важно понимать, как изменяется компонент. Если он выполняет несколько различных действий, и они изменяются по отдельности, то это как раз тот случай, когда можно применить принцип единственной обязанности. То есть иными словами, у компонента несколько причин для изменения.

Допустим, нам надо определить класс отчета, по которому мы можем перемещаться по страницам и который можно выводить на печать. На первый взгляд мы могли бы определить следующий класс:

class Report

{

    public string Text { get; set; } = "";

    public void GoToFirstPage() =>

        Console.WriteLine("Переход к первой странице");

    public void GoToLastPage() =>

        Console.WriteLine("Переход к последней странице");

    public void GoToPage(int pageNumber) =>

        Console.WriteLine($"Переход к странице {pageNumber}");

    public void Print()

    {

        Console.WriteLine("Печать отчета");

        Console.WriteLine(Text);

    }

}

Ключевым понятием применительно к данному принципу является cohesion или связность/согласованность. Это понятие описывает, насколько близко связаны компоненты. Чем больше связность между компонентами, тем больше программа соответствует принципу единой ответственности

Например, первые три метода класса относятся к навигации по отчету и представляют одно единое функциональное целое, обладают высокой связностью. От них отличается метод Print, который производит печать. Что если нам понадобится печатать отчет на консоль или передать его на принтер для физической печати на бумаге? Или вывести в файл? Сохранить в формате html, txt, rtf и т.д.? Очевидно, что мы можем для этого поменять нужным образом метод Print(). Однако это вряд ли затронет остальные методы, которые относятся к навигации страницы.

Также верно и обратное - изменение методов постраничной навигации вряд ли повлияет на возможность вывода текста отчета на принтер или на консоль. Таким образом, у нас здесь прослеживаются две причины для изменения, значит, класс Report обладает двумя обязанностями, и от одной из них этот класс надо освободить. Решением было бы вынести каждую обязанность в отдельный компонент (в данном случае в отдельный класс):

class Report

{

    public string Text { get; set; } = "";

    public void GoToFirstPage() =>

        Console.WriteLine("Переход к первой странице");

    public void GoToLastPage() =>

        Console.WriteLine("Переход к последней странице");

    public void GoToPage(int pageNumber) =>

        Console.WriteLine($"Переход к странице {pageNumber}");

}

//  обязанность - печать отчета

class Printer

{

    public void PrintReport(Report report)

    {

        Console.WriteLine("Печать отчета");

        Console.WriteLine(report.Text);

    }

}

Теперь печать вынесена в отдельный класс Printer, который через метод Print получает объект отчета и выводит его текст на консоль.

2. Принцип открытости/закрытости

Принцип открытости/закрытости (Open/Closed Principle) можно сформулировать так: Сущности программы должны быть открыты для расширения, но закрыты для изменения.

Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего. Рассмотрим простейший пример - класс повара:

class Cook

{

    public string Name { get; set; }

    public Cook(string name)

    {

        this.Name = name;

    }

    public void MakeDinner()

    {

        Console.WriteLine("Чистим картошку");

        Console.WriteLine("Ставим почищенную картошку на огонь");

        Console.WriteLine("Сливаем остатки воды, разминаем варенный картофель в пюре");

        Console.WriteLine("Посыпаем пюре специями и зеленью");

        Console.WriteLine("Картофельное пюре готово");

    }

}

И с помощью метода MakeDinner любой объект данного класса сможет сделать картофельного пюре:

Cook bob = new Cook("Bob");

bob.MakeDinner();

Однако одного умения готовить картофельное пюре для повара вряд ли достаточно. Хотелось бы, чтобы повар мог приготовить еще что-то. И в этом случае мы подходим к необходимости изменения функционала класса, а именно метода MakeDinner. Но в соответствии с рассматриваемым нами принципом классы должны быть открыты для расширения, но закрыты для изменения. То есть, нам надо сделать класс Cook отрытым для расширения, но при этом не изменять.

Итак, изменим класс Cook следующим образом:

class Cook

{

    public string Name { get; set; }

    public Cook(string name)

    {

        this.Name = name;

    }

    public void MakeDinner(IMeal meal)

    {

        meal.Make();

    }

}

interface IMeal

{

    void Make();

}

class PotatoMeal : IMeal

{

    public void Make()

    {

        Console.WriteLine("Чистим картошку");

        Console.WriteLine("Ставим почищенную картошку на огонь");

        Console.WriteLine("Сливаем остатки воды, разминаем варенный картофель в пюре");

        Console.WriteLine("Посыпаем пюре специями и зеленью");

        Console.WriteLine("Картофельное пюре готово");

    }

}

class SaladMeal : IMeal

{

    public void Make()

    {

        Console.WriteLine("Нарезаем помидоры и огурцы");

        Console.WriteLine("Посыпаем зеленью, солью и специями");

        Console.WriteLine("Поливаем подсолнечным маслом");

        Console.WriteLine("Салат готов");

    }

}

Теперь приготовление еды абстрагировано в интерфейсе IMeal, а конкретные способы приготовления определены в реализациях этого интерфейса. А класс Cook делегирует приготовление еды методу Make объекта IMeal.

3. Принцип подстановки Лисков

Принцип подстановки Лисков (Liskov Substitution Principle) представляет собой некоторое руководство по созданию иерархий наследования. Кратко значит что должна быть возможность вместо базового типа подставить любой его подтип. Фактически принцип подстановки Лисков помогает четче сформулировать иерархию классов, определить функционал для базовых и производных классов и избежать возможных проблем при применении полиморфизма.

Рассмотрим пример, где нарушается принцип подстановки Лисков:

public class Rectangle

{

public virtual int Width { get; set; }

public virtual int Height { get; set; }

public int GetArea() => Width \* Height;

}

public class Square : Rectangle

{

public override int Width

{

set { base.Width = base.Height = value; }

}

public override int Height

{

set { base.Width = base.Height = value; }

}

}

Проблема в том что Square изменяет поведение Rectangle. Если вы замените Rectangle на Square, то код, использующий Rectangle, может начать работать неправильно, так как Square изменяет логику установки ширины и высоты. Чтобы следовать принципу LSP, можно изменить иерархию классов:

public abstract class Shape

{

public abstract int GetArea();

}

public class Rectangle : Shape

{

public int Width { get; set; }

public int Height { get; set; }

public override int GetArea() => Width \* Height;

}

public class Square : Shape

{

public int SideLength { get; set; }

public override int GetArea() => SideLength \* SideLength;

}

* Rectangle и Square являются независимыми реализациями Shape.
* Square и Rectangle теперь корректно следуют принципу подстановки, так как не изменяют поведение друг друга.
* Оба класса реализуют свой собственный метод GetArea, соответствующий их геометрической форме.

Следование принципу LSP помогает улучшить модульность и расширяемость кода, избегая неожиданных изменений поведения при использовании подтипов.

4. Принцип разделения интерфейсов (Interface Segregation Principle) относится к тем случаям, когда классы имеют "жирный интерфейс", то есть слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы. Принцип разделения интерфейсов можно сформулировать так: Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.

Рассмотрим на примере. Допустим у нас есть интерфейс отправки сообщения:

interface IMessage

{

    void Send();

    string Text { get; set;}

    string Subject { get; set;}

    string ToAddress { get; set; }

    string FromAddress { get; set; }

}

Интерфейс определяет все основное, что нужно для отправки сообщения: само сообщение, его тему, адрес отправителя и получателя и, конечно, сам метод отправки. И пусть есть класс EmailMessage, который реализует этот интерфейс:

class EmailMessage : IMessage

{

    public string Subject { get; set; } = "";

    public string Text { get; set; } = "";

    public string FromAddress { get; set; } = "";

    public string ToAddress { get; set; } = "";

    public void Send() => Console.WriteLine($"Отправляем Email-сообщение: {Text}");

}

Теперь определим класс, который бы отправлял данные по смс:

class SmsMessage : IMessage

{

    public string Text { get; set; } = "";

    public string FromAddress { get; set; } = "";

    public string ToAddress { get; set; } = "";

    public string Subject

    {

        get

        {

            throw new NotImplementedException();

        }

        set

        {

            throw new NotImplementedException();

        }

    }

    public void Send() => Console.WriteLine($"Отправляем Sms-сообщение: {Text}");

}

Здесь мы уже сталкиваемся с небольшой проблемой: свойство Subject, которое определяет тему сообщения, при отправке смс не указывается, поэтому оно в данном классе не нужно. Таким образом, в классе SmsMessage появляется избыточная функциональность, от которой класс SmsMessage начинает зависеть. Это не очень хорошо, но посмотрим дальше. Допустим, нам надо создать класс для отправки голосовых сообщений.

Класс голосовой почты также имеет отправителя и получателя, только само сообщение передается в виде звука, что на уровне C# можно выразить в виде массива байтов. И в этом случае было бы неплохо, если бы интерфейс IMessage включал бы в себя дополнительные свойства и методы для этого, например:

interface IMessage

{

    void Send();

    string Text { get; set;}

    string ToAddress { get; set; }

    string Subject { get; set; }

    string FromAddress { get; set; }

    byte[] Voice { get; set; }

}

Тогда класс голосовой почты VoiceMessage мог бы выглядеть следующим образом:

class VoiceMessage : IMessage

{

    public string ToAddress { get; set; } = "";

    public string FromAddress { get; set; } = "";

    public byte[] Voice { get; set; } = new byte[] {};

    public string Text

    {

        get

        {

            throw new NotImplementedException();

        }

        set

        {

            throw new NotImplementedException();

        }

    }

    public string Subject

    {

        get

        {

            throw new NotImplementedException();

        }

        set

        {

            throw new NotImplementedException();

        }

    }

    public void Send() => Console.WriteLine("Передача голосовой почты");

}

И здесь опять же мы сталкиваемся с ненужными свойствами. Плюс нам надо добавить новое свойство в предыдущие классы SmsMessage и EmailMessage, причем этим классам свойство Voice в принципе то не нужно. В итоге здесь мы сталкиваемся с явным нарушением принципа разделения интерфейсов.

Для решения возникшей проблемы нам надо выделить из классов группы связанных методов и свойств и определить для каждой группы свой интерфейс:

interface IMessage

{

    void Send();

    string ToAddress { get; set; }

    string FromAddress { get; set; }

}

interface IVoiceMessage : IMessage

{

    byte[] Voice { get; set; }

}

interface ITextMessage : IMessage

{

    string Text { get; set; }

}

interface IEmailMessage : ITextMessage

{

    string Subject { get; set; }

}

class VoiceMessage : IVoiceMessage

{

    public string ToAddress { get; set; } = "";

    public string FromAddress { get; set; } = "";

    public byte[] Voice { get; set; } = Array.Empty<byte>();

    public void Send() => Console.WriteLine("Передача голосовой почты");

}

class EmailMessage : IEmailMessage

{

    public string Text { get; set; } = "";

    public string Subject { get; set; } = "";

    public string FromAddress { get; set; } = "";

    public string ToAddress { get; set; } = "";

    public void Send() => Console.WriteLine("Отправляем по Email сообщение: {Text}");

}

class SmsMessage : ITextMessage

{

    public string Text { get; set; } = "";

    public string FromAddress { get; set; } = "";

    public string ToAddress { get; set; } = "";

    public void Send() => Console.WriteLine("Отправляем по Sms сообщение: {Text}");

}

Теперь классы больше не содержат неиспользуемые методы. Чтобы избежать дублирование кода, применяется наследование интерфейсов. В итоге структра классов получается проще, чище и яснее.

4. Принцип инверсии зависимостей

Принцип инверсии зависимостей (Dependency Inversion Principle) служит для создания слабосвязанных сущностей, которые легко тестировать, модифицировать и обновлять. Этот принцип можно сформулировать следующим образом: Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.

Рассмотрим следующий пример:

class Book

{

    public string Text { get; set; }

    public ConsolePrinter Printer { get; set; }

    public void Print()

    {

        Printer.Print(Text);

    }

}

class ConsolePrinter

{

    public void Print(string text)

    {

        Console.WriteLine(text);

    }

}

Класс Book, представляющий книгу, использует для печати класс ConsolePrinter. При подобном определении класс Book зависит от класса ConsolePrinter. Более того мы жестко определили, что печать книгу можно только на консоли с помощью класса ConsolePrinter. Другие же варианты, например, вывод на принтер, вывод в файл или с использованием каких-то элементов графического интерфейса - все это в данном случае исключено. Абстракция печати книги не отделена от деталей класса ConsolePrinter. Все это является нарушением принципа инверсии зависимостей.

Теперь попробуем привести наши классы в соответствие с принципом инверсии зависимостей, отделив абстракции от низкоуровневой реализации:

interface IPrinter

{

    void Print(string text);

}

class Book

{

    public string Text { get; set; }

    public IPrinter Printer { get; set; }

    public Book(IPrinter printer)

    {

        this.Printer = printer;

    }

    public void Print()

    {

        Printer.Print(Text);

    }

}

class ConsolePrinter : IPrinter

{

    public void Print(string text)

    {

        Console.WriteLine("Печать на консоли");

    }

}

class HtmlPrinter : IPrinter

{

    public void Print(string text)

    {

        Console.WriteLine("Печать в html");

    }

}

Теперь абстракция печати книги отделена от конкретных реализаций. В итоге и класс Book и класс ConsolePrinter зависят от абстракции IPrinter. Кроме того, теперь мы также можем создать дополнительные низкоуровневые реализации абстракции IPrinter и динамически применять их в программе.

**Ошибки в стиле кода C#**

1. Именование классов и интерфейсов

Неправильно:

public class customerdata

{

}

public interface idatabase

{

}

Правильно:

public class CustomerData

{

}

public interface IDatabase

{

}

2. Именование локальных переменных и переменных классов

Неправильно:

public class OrderProcessor

{

private int Ordercount;

public void Process()

{

int TEMP = 0;

}

}

Правильно:

public class OrderProcessor

{

private int orderCount;

public void Process()

{

int temp = 0;

}

}

3. Константы

Неправильно:

public class Circle

{

private const double pi = 3.14;

}

Правильно:

public class Circle

{

private const double PI = 3.14;

}

4. Форматирование скобок

Неправильно:

if (condition) {

DoSomething();

}

Правильно:

if (condition)

{

DoSomething();

}

6. Использование фигурных скобок

Неправильно:

if (isValid) DoSomething();

Правильно:

if (isValid)

{

DoSomething();

}

7. Пробелы вокруг операторов

Неправильно:

int sum=10+5;

Правильно:

int sum = 10 + 5;

8. Использование магических чисел

Неправильно:

double radius = 5;

double area = 3.14 \* radius \* radius;

Правильно:

const double Pi = 3.14;

double radius = 5;

double area = Pi \* radius \* radius;

9. Избыточные комментарии

// This variable stores the user's age

int age = 30;

int age = 30; // User's age

Соблюдение стандартов стиля кода делает ваш код более читаемым, поддерживаемым и понятным для других разработчиков.

**Правильно использование структур данных**

Пример 1: Использование массива вместо списка

Неправильно:

int[] numbers = new int[2];

numbers[0] = 1;

numbers[1] = 2;

// Добавление нового элемента требует создания нового массива

Правильно:

List<int> numbers = new List<int> { 1, 2 };

numbers.Add(3); // Легко добавить элемент

2. Линейный поиск в отсортированном массиве

Неправильно:

int[] sortedArray = { 1, 2, 3, 4, 5 };

int target = 3;

bool found = false;

foreach (int num in sortedArray)

{

if (num == target)

{

found = true;

break;

}

}

Правильно:

int[] sortedArray = { 1, 2, 3, 4, 5 };

int target = 3;

bool found = Array.BinarySearch(sortedArray, target) >= 0;

3. Использование List вместо HashSet для проверки уникальности

Неправильно:

List<string> names = new List<string> { "Alice", "Bob", "Alice" };

bool hasDuplicates = names.Count != names.Distinct().Count();

Правильно:

HashSet<string> names = new HashSet<string> { "Alice", "Bob", "Alice" };

bool hasDuplicates = names.Count != 3;

4. Ручной подсчет элементов вместо использования встроенных функций

Неправильно:

int[] numbers = { 1, 2, 3, 4, 5 };

int sum = 0;

foreach (int num in numbers)

{

sum += num;

}

Правильно:

int[] numbers = { 1, 2, 3, 4, 5 };

int sum = numbers.Sum();

5. Использование Dictionary для подсчета частоты элементов

Неправильно:

string[] words = { "apple", "banana", "apple" };

Dictionary<string, int> frequency = new Dictionary<string, int>();

foreach (string word in words)

{

if (frequency.ContainsKey(word))

{

frequency[word]++;

}

else

{

frequency[word] = 1;

}

}

Правильно:

string[] words = { "apple", "banana", "apple" };

var frequency = words.GroupBy(word => word)

.ToDictionary(g => g.Key, g => g.Count());

Правильный выбор структур данных и алгоритмов улучшает производительность и читаемость кода. Используйте встроенные функции и структуры данных, когда это возможно.

**Вывод**: В ходе выполнение практического задания были разобраны частые примеры ошибок в использовании принципов SOLID, ошибки в стиле кода и ошибки в использовании структур данных.