Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Кафедра програмної інженерії

Практична робота №1

з дисципліни: «Аналіз та рефакторинг коду»

на тему «Правила оформлення програмного коду»

|  |  |
| --- | --- |
| Перевірив:  Сокорчук І.П | Виконала:  ст. гр. ПЗПІ-22-7  Лаврович Оксана Степанівна |

Харків 2024

«Правила оформлення програмного коду»

1. Мета заняття

Ознайомити студентів з основними рекомендаціями щодо написання чистого, ефективного та підтримуваного коду для різних мов програмування, а також навчити аналізувати та рефакторити код для покращення його якості.

1. Завдання

Студент повинен обрати мову програмування, дослідити основні рекомендації для написання якісного коду у цій мові, продемонструвати ці рекомендації на прикладах коду, обґрунтувати кожну рекомендацію і надати детальні пояснення. Для кожного прикладу мають бути наведені фрагменти коду.

Вибрана мова програмування: C#

1. Хід роботи

ВСТУП

C# - Об'єктно-орієнтована мова програмування загального призначення. Розроблено в 1998-2001 роках групою інженерів компанії Microsoft під керівництвом Андерса Хейлсберга і Скотта Вільтаумота як мову розробки додатків для платформи Microsoft .NET Framework та .NET Core. Згодом був стандартизований як ECMA-334 та ISO/IEC 23270. Мова має статичну типізацію, підтримує поліморфізм, навантаження операторів (у тому числі операторів явного та неявного приведення типу), делегати, атрибути, події, змінні, властивості, узагальнені типи та методи, ітератори, анонімні функції з підтримкою замикань, LINQ, винятки, коментарі у форматі XML.

Стандарт оформлення коду (стандарт кодування, стиль у програмуванні) (англ. coding standards, coding convention або programming style) - набір правил і угод, що використовуються при написанні вихідного коду деякою мовою програмування. Наявність загального стилю програмування полегшує розуміння та підтримку вихідного коду, написаного більш ніж одним програмістом, а також спрощує взаємодію кількох людей під час розробки програмного забезпечення.

Дотримання стандартів оформлення коду є важливим аспектом для ефективної командної роботи. Коли всі учасники проекту дотримуються однакових стандартів, код стає більш передбачуваним і структурованим. Це дозволяє будь-якому розробнику легко розібратися в роботі програмного забезпечення, навіть якщо він не створював конкретну його частину. В результаті зменшується час на адаптацію нових членів команди й пришвидшується вирішення проблем. Стандартизація коду робить його більш зрозумілим і зручним для сприйняття, що дозволяє швидше орієнтуватися в проекті та скорочує час, необхідний для його підтримки та розвитку.

Переваги стандартів кодування:

* Уніфікація. Стандартизація забезпечує єдиний підхід до написання коду, що сприяє злагодженій роботі команди та запобігає плутанині.
* Зрозумілість. Код, написаний за стандартами, легше сприймається іншими розробниками, що прискорює процеси огляду та внесення змін.
* Ефективність. Стандарти дозволяють зосередитися на вирішенні задач, а не на інтерпретації різних стилів написання коду, що покращує продуктивність.
* Підтримуваність. Уніфікований код легше тестувати, оновлювати та масштабувати, що зменшує технічний борг.

ЗАГАЛЬНІ ПРАВИЛА ОФОРМЛЕННЯ КОДУ

* 1. **Іменування (іменування змінних, функцій, класів)**
* В Pascal casing описуються імена:

- всіх визначень типів, у тому числі класів, подій і структур;

- значення перерахувань;

- readonly полів і констант;

- інтерфейсів;

- методів;

- просторів імен (namespace);

- властивостей;

- публічних полів;

Цей стиль використовується для елементів, які, зазвичай, є важливими в загальній архітектурі програми, що дозволяє їм виглядати більш виділено та формально.

Приклад:

namespace ExampleApp

{

class ClassNamingConvention

{

public const string ConstantFieldNamingConvention = "C#";

public string PropertiesNamingConvention { get; set; }

public void MethodNamingConvention()

{

//type something here

}

}

}

* В Camel casing описуються імена:

- локальних змінних;

- аргументів методів;

- захищених (protected) полів.

Цей стиль застосовується для елементів, які мають більш локальний характер і використовуються лише у певних методах або класах. Camel Casing зменшує акцент на таких іменах і робить їх менш помітними, що корисно для структурованого коду.

Приклад:

private string fieldsNamingConvention;

public void MethodNamingConvention(string methodArgsNamingConvention)

{

string localVariables = "string here ...";

}

* Не використовуйте «магічні» числа

Не використовуйте літеральні значення, числа або строки у вашому коді ні для чого іншого, крім константи оголошення. Для прикладу:

// wrong

public double CalculateCircleArea(double radius) {

return 3.14 \* radius \* radius;

}

// right

public class Circle {

private const double Pi = 3.14159;

public double CalculateArea(double radius) {

return Pi \* radius \* radius;

}

}

Літеральні значення допускаються використовувати тільки тоді, коли їх сенс ясен із контексту, і їх не планується змінити.

* Вибирайте для ідентифікаторів імена, що легко читаються.

Наприклад, властивість з ім'ям HorizontalAlignment легше читається англійською, ніж AlignmentHorizontal.

* НЕ використовуйте символи підкреслення, дефіси та інші символи, які не є буквено-цифровими.

Наприклад використання FirstName і LastName, замість first\_name або first-name, робить код більш читабельним і уніфікованим. Це зменшує ймовірність помилок і покращує зрозумілість.

* НЕ використовуйте угорську нотацію.

Венгерська нотація, де тип змінної включається в її ім'я (наприклад, **strName** для рядка),  робить код менш читабельним і ускладнює його підтримку.

// Correct

int counter;

string name;

// Avoid

int iCounter;

string strName;

* НЕ використовуйте скорочення в іменах ідентифікаторів.

Наприклад, використовуйте GetWindow, а не GetWin.

* 1. **Суфікси та префікси**

Префікси та суфікси допомагають одразу зрозуміти призначення елемента. Це знижує необхідність додаткових пояснень або документації, оскільки назва вже дає уявлення про роль і використання класу або інтерфейсу.

Застосовуються такі суфікси та префікси:

* імена власних класів винятків завжди закінчуються суфіксом “Exception”;

public class SampleException: System.Exception  
{  
  public SampleException()  
  {  
  }  
}

* імена інтерфейсів завжди починаються з префікса I;

interface ISample  
{  
  void SampleMethod();  
}

* імена атрибутів користувача завжди закінчуються суфіксом «Attribute»;

[System.AttributeUsage(System.AttributeTargets.All, Inherited = false, AllowMultiple = true)]  
sealed class SampleAttribute: System.Attribute  
{  
  public SampleAttribute()  
  {  
  }  
}

* імена делегатів обробників подій завжди закінчуються суфіксом EventHandler, імена класів-спадкоємців від EventArgs завжди закінчуються суфіксом EventArgs.

public delegate void AnswerCreatedEventHandler(object sender, AnswerCreatedEventArgs e);

public class AnswerCreatedEventArgs: EventArgs  
  {  
    public int СreatedId;  
    public int ParentId;  
    public string CreatorName;  
  }

public class FileNotFoundException : Exception

{

public FileNotFoundException(string message) : base(message)

{

}

}

public interface IOrderProcessor

{

void ProcessOrder();

}

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]

public class LoggingAttribute : Attribute

{

public string LogMessage { get; }

public LoggingAttribute(string logMessage)

{

LogMessage = logMessage;

}

}

* 1. **Коментарі**
* Розмістіть коментар в окремому рядку, а не в кінці рядка коду.
* Починайте текст коментаря з великої літери.
* Завершіть текст коментаря крапкою.
* Вставте один пробіл між розділювачем коментаря ﴾//﴿ і текстом коментаря, як показано нижче
* Не створюйте форматовані блоки зірочок навколо коментарів.
* Не пишіть коментарі, якщо код легко зрозумілий без коментарів.

Менша кількість рядків коментарів зробить код кращим і більш елегантним. Пишіть чистий, читабельний код таким чином, щоб він не потребував жодних коментарів для розуміння. Виконайте перевірку орфографії в коментарях, а також переконайтеся, що використовується правильна граматика та пунктуація.

Поганий приклад:

//Підрахунок результату

int result = 10 + 5; //додаємо числа

int sum = result \* 2; // множимо результат на два

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*Кінець коду\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Хороший приклад:

// Підрахунок результату.

int result = 10 + 5;

// Множимо результат на два.

int sum = result \* 2;

ОФОРМЛЕННЯ ФАЙЛІВ ТА СТРУКТУРИ ПРОЄКТУ

* 1. **Іменування файлів**
* Назви файлів і каталогів мають регістр Pascal, напр. MyFile.cs.
* Якщо можливо, ім’я файлу має збігатися з ім’ям основного класу у файлі, наприклад. MyClass.cs.
* Загалом віддавайте перевагу одному основному класу на файл.
  1. **Структура коду**

Організація проєкту та файлів

* Модифікатори з’являються в такому порядку: public protected internal private new abstract virtual override sealed static readonly extern unsafe volatile async.
* Оголошення простору імен using йдуть угорі, перед усіма просторами імен. використання порядку імпорту є алфавітним, за винятком системного імпорту, який завжди йде першим.
* Namespace using оголошення йдуть угорі, перед будь-якими просторами імен.
* Використання порядку імпорту є алфавітним, за винятком System імпорту, який завжди йде першим.

using System;

using System.IO;

using Microsoft.Extensions.Configuration;

using Newtonsoft.Json;

namespace LibraryManagement

{

}

Порядок членів класу:

* Групуйте членів класу в такому порядку:
* Вкладені класи, переліки, делегати та події.
* Static, const та readonly поля.
* Поля та властивості.
* Конструктори та фіналізатори.
* Методи.
* У кожній групі елементи повинні бути в такому порядку:
* Public.
* Internal.
* Protected internal.
* Protected.
* Private.
* Де це можливо, групуйте реалізації інтерфейсу разом.
  1. **Архітектура**
* Завжди використовуйте багаторівневу (N-Tier) архітектуру
* Ніколи не звертайтеся до бази даних зі сторінок інтерфейсу користувача.
* Завжди майте клас рівня даних, який виконує всі пов’язані з базою даних завдання.
* Використовуйте try-catch на рівні даних, щоб перехопити все винятки бази даних.
* Розділіть свою програму на кілька збірок.
* Згрупуйте всі незалежні корисні класи в окрему бібліотеку класів.
* Усі пов’язані з базою даних файли можуть бути в іншій бібліотеці класів

**N-рівнева архітектура** це перевірена в галузі модель архітектури програмного забезпечення. Він підходить для підтримки клієнт-серверних програм корпоративного рівня, надаючи рішення для масштабованості, безпеки, відмовостійкості, багаторазового використання та зручності обслуговування. Це допомагає розробникам створювати гнучкі та багаторазово використовувані програми.

ФОРМАТУВАННЯ КОДУ

* 1. **Layout Conventions**

Хороший макет використовує форматування, щоб підкреслити структуру вашого коду та зробити код легшим для читання. Приклади та зразки Microsoft відповідають таким умовам:.

* Використовуйте параметри редактора коду за замовчуванням ﴾розумні відступи, чотирисимвольні відступи)
* Пишіть лише одне твердження в рядку.
* Пишіть лише одне оголошення на рядок.
* Якщо рядки продовження не мають відступу автоматично, зробіть для них відступ на одну позицію табуляції ﴾чотири пробіли﴿.
* Додайте принаймні один порожній рядок між визначеннями методів і властивостей.
* Використовуйте дужки, щоб зробити речення у виразі очевидними, як показано в наступному коді.

if ((val1 > val2) && (val1 > val3))

{

// Take appropriate action.

}

* 1. **Правила використання круглих і фігурних дужок**
* Використовуйте стиль Allman для фігурних скобок. Фігурні скобки відповідають поточному рівню відступу.

Відкриваючі фігурні дужки для класів, методів і умовних операторів розміщуються на нових рядках, що підвищує читабельність та візуальну структурованість коду.

* Фігурні дужки ( {} ) мають бути на одному рівні з кодом поза фігурними дужками

class ClassName

{

void MethodName()

{

// Код

}

}

* 1. **Об'єм коду**
* уникайте файлів із більш ніж 500 рядками коду;
* уникайте методів із більш ніж 200 рядками коду;
* уникайте методів з більш ніж 5 аргументами, використовуйте структури передачі великої кількості параметрів;
* Обмежте довжину рядка до 65 символів.

Це полегшує читання коду на різних екранах.

Файли з великою кількістю рядків важко підтримувати, тому краще розділити функціональність на менші, спеціалізовані класи або файли.Обмеження довжини рядка до 65 символів полегшує читання коду на різних екранах і в різних середовищах розробки.

Приклад:

class Example

{

public void PrintMessage()

{

Console.WriteLine("Цей рядок є занадто довгим і перевищує рекомендовану довжину в 65 символів, що ускладнює його читання на малих екранах.");

Console.WriteLine("Цей рядок є коротким і зрозумілим, "

+ "дотримуючись рекомендацій щодо довжини.");

}

}

ОФОРМЛЕННЯ МЕТОДІВ

* Уникайте написання дуже довгих методів.
* Використовуйте конструкцію дієслово-об'єкт для іменування методів

ShowUserInfo()

В окремому випадку, для методів, які повертають значення, використовуйте в парі дієслово-об'єкт для дієслова «Get», а для об'єкта – опис значення, що повертається.

GetUserId()

* Обмежте методи однією функціональністю

Найкраще обмежити свої методи однією функціональністю. Не намагайтеся об'єднати кілька функцій класу в один метод. Це забезпечує читабельність коду.

class AppNotification

{

public void SendNotification()

{

//. . .

}

public void ReceiveNotification()

{

//. . .

}

public void MuteNotification()

{

//. . .

}

}

* Уникайте публічних методів і властивостей, якщо до них справді не потрібен доступ ззовні
* Використовуйте «internal», якщо доступ до них доступний лише в межах однієї збірки
* Уникайте передачі методу занадто великої кількості параметрів.

Краще визначте клас або структуру.

* Якщо у вас є метод, який повертає колекцію, повертайте порожню колекцію замість null

// Клас замість численних параметрів методу

public class Order

{

public string CustomerName { get; set; }

public string OrderDetails { get; set; }

public string PaymentMethod { get; set; }

public string ShippingAddress { get; set; }

public string PromoCode { get; set; }

}

public class OrderProcessor

{

// Використовуємо internal для доступу тільки всередині збірки

internal void ProcessOrder(Order order)

{

// Логіка обробки замовлення

}

// Приватна властивість замість публічної

private int orderCount;

// Метод, який повертає порожню колекцію замість null

public List<string> GetAvailableItems()

{

return new List<string>(); // Повертаємо порожню колекцію

}

}

РЕКОМЕНДАЦИИ ПО СТВОРЕННЮ ДОКУМЕНТАЦІЇ

* Для опису методів, класів, полей і всіх загальнодоступних елементів використовуються xml-коментарі.

Вони починаються з трьох косих рисок (**///**) і дозволяють створювати структуровану документацію. Компілятор C# створює файл XML, що містить структуровані дані, що представляють коментарі та сигнатури API. Інші засоби можуть обробляти ці вихідні дані у форматі XML і створювати зручну для читання документацію, наприклад, у вигляді веб-сторінок або PDF-файлів. (за допомогою таких інструментів, як DocFX або [Sandcastle](https://github.com/EWSoftware/SHFB))

/// <summary>

/// This class performs an important function.

/// </summary>

public class MyClass {}

* Пишіть коментарі і документацію англійською.

Англійська є міжнародною мовою програмування, тому коментарі, написані англійською, зрозумілі більшості розробників.

* Документуйте всі public, protected та internal типи та члени

Документування вашого коду дозволить Visual Studio виводити підказки, коли ваш клас буде використовуватися. Крім цього, коли ви добре документуєте ваші класи, ви можете генерувати документацію до коду, який виглядає професійно.

* Використовуйте стиль MSDN написання документації

Дотримуйтесь стилю онлайн довідки MSDN, щоб допомогти іншому розробнику швидше розібратися у документації. GhostDoc за допомогою поєднання гарячих клавіш може генерувати коментарі xml для створення документації.

РЕКОМЕНДАЦІЇ ЩОДО ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ

* Використовуйте async тільки для довготривалих та низькоінтенсивних завдань

Використання async не запустить автоматично щось у робочому потоці, як це робить Task.Run. Async просто додає необхідну логіку, яка служить для того, щоб дозволити вивільняти поточний потік і повернути результат на той же потік після завершення асинхронної операції. Іншими словами, використовуйте async тільки для операцій, пов'язаних з I/O.

* Використовуйте Task.Run для високоінтенсивних завдань

Якщо вам потрібно виконати операцію, пов'язану із виділенням додаткових ресурсів процесора, використовуйте Task.Run, щоб вивантажити роботу на потік із пулу потоків. Просто не забувайте про те, що вам доведеться вручну повертати результат у ваш основний потік.

* Уникайте використання await/async з Task.Wait

await не заблокує поточний потік, а просто проінформує компілятор про необхідність побудови станів машини. Однак Task.Wait заблокує потік і може призвести до взаємних блокувань.

public async Task ProcessFileAsync()

{

// Операція з I/O, тому async підходить

var fileContent = await ReadFileAsync("file.txt");

Console.WriteLine(fileContent);

}

// Використання Task.Run для ресурсоємної операції

public Task ProcessDataAsync()

{

return Task.Run(() =>

{

// Ресурсоємна операція, виконується в окремому потоці

var result = DoIntensiveTask();

Console.WriteLine(result);

});

}

* 1. **Тестування**
* Використовуйте Assert.Equal

Використовуйте Assert.Equal(expected, actual) замість Assert.True(actual == expected).

ВИСНОВКИ

У результаті виконання даної роботи було досягнуто глибокого розуміння важливості дотримання стандартів написання чистого, ефективного та підтримуваного коду на мові програмування C#. Ознайомилася з основними принципами організації проектів, форматування коду, іменування змінних та функцій, а також з правильним використанням коментарів і документації. Правильне форматування, зокрема відступи та стиль іменування, покращує читабельність, що дозволяє розробникам швидше орієнтуватися в коді та знижує ймовірність помилок. Коментарі та документація надають чітке пояснення логіки коду, що полегшує його подальшу підтримку та адаптацію. У підсумку, дотримання цих правил не тільки підвищує якість коду, але й сприяє ефективній командній роботі, що дозволяє швидше досягати результатів у розробці програм.

Без стандарту кодування розробники використовуватимуть власні методи кодування, і це має певні негативні наслідки.

1. Проблеми безпеки. Основними причинами вразливостей програмного забезпечення, які часто використовують, є невідповідності, помилки та помилки в логіці. Більшість із цих проблем виникає через помилки програмування, які є результатом поганої практики кодування.

2. Проблеми з продуктивністю сайту: погане кодування впливає на загальну продуктивність сайту. Різні проблеми продуктивності включають:

* Взаємодія з користувачем
* Відповідь сервера
* Багаторазове використання коду
* Проблеми з потоком

Впровадження стандартів кодування подолає ці проблеми, надаючи безпечний сайт із мінімальними проблемами продуктивності.

1. **Використані джерела**

1. Microsoft. Конвенції стилю коду для C#. Доступно за посиланням: <https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/coding-style/coding-conventions>

2.Google. C# Coding Style Guide . – Доступно за посиланням: [https://google.github.io/styleguide/csharp-style.html](https://google.github.io/styleguide/csharp-style.html" \t "_new)

3. Microsoft. XML-документація для C#. Доступно за посиланням: <https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/xmldoc/>

4. Microsoft. Керівництво з кодування для .NET. Доступно за посиланням: <https://github.com/dotnet/runtime/blob/main/docs/coding-guidelines/coding-style.md>

5. ELMA BPM. Рекомендації щодо стилю кодування для C#. Доступно за посиланням: [https://kb.elma-bpm.com/article-6658.html](https://kb.elma-bpm.com/article-6658.html" \t "_new)

ДОДАТОК А

Слайди презентації







































