C# Codestyle Qoollo

Alpha version 24.09.2020

Документ, описывающий корпоративные соглашения и код-стайл, для написания программ на языке C# в компании Qoollo.

1. Соглашения по наименованию

1.1. Общие правила

1. В языке C# принят стиль UpperCamelCase, однако для некоторых конструкций для лучшей читаемости могут существовать исключения.
2. Нельзя давать имена с приставкой *My* любым конструкциям.

   Пример:

```
// Неправильно
public class MyTextSplitter
{
    ...
}
```

```
// Правильно
public class TextSplitter
{
    ...
}
```

3. Нельзя использовать название конструкций языка в любых идентификаторах. Например, недопустимо использование слов *Class*, *Struct*, *Method*, *Event*, *Enum*, *Property* и пр.

   Пример:

```
// Неправильно
public class TextSplitterClass
{
      public string FilePathProperty { get; set; }

      public event EventHandler<FileOpenEventArgs> FileOpenedEvent;

      public void ReadTextMethod()
```

```
        {
            ...
        }
}

// Правильно
public class TextSplitter
{
        public string FilePath { get; set; }

        public event EventHandler<FileOpenEventArgs> FileOpened;

        public void ReadText()
        {
            ...
        }
}
```

4. Нельзя добавлять в имена любых лексем слова, не относящиеся к выполняющемуся действию или имеющие эмоциональную окраску. Добавлять технические характеристики в название можно только если существует аналогичный идентификатор без характеристик.

```
// Базовая версия
public class TextSplitter
{
    ...
}

// Неправильно
public class SuperTextSplitter
{
    ...
}

// Правильно
public class OptimizedByMemoryTextSplitter
{
    ...
}
```

5. Нельзя использовать в идентификаторах порядковые номера. Вместо этого необходимо конкретизировать отличительные характеристики.

   Пример:

```
// Неправильно
int count1;
int count2;

// Правильно
int stringLength;
int spaceCount;
```

6. Нельзя использовать сокращения кроме общепринятых.

```
// Неправильно
int usrInd;
int wepView;
int effStyle;

// Правильно
int userIndex;
int weaponView;
int effectStyle;
```

7. Необходимо использовать латинские буквы и существующие слова на английском языке и не писать транслитом.

```
// Неправильно
string kotik;
int kolichestvo;

// Правильно
string cat;
int count;
```

8. Нужно использовать легко читаемые, простые и грамматически правильные имена. Необходимо стремиться к тому, чтобы названия мог понимать даже человек с минимальным уровнем английского:

```
// Неправильно
public void DeductSustainableEssences()
{
    ...
}

// Правильно
public void CalculateSingleObjects()
{
```

```
    ...
}
```

9. Для всех наименований необходимо использовать единственное число. Исключениями, которые всегда должны быть названы во множественном числе, являются:
    - Свойства, члены-данные классов, локальные переменные и параметры методов, имеющие тип массивов и коллекций
    - Перечисления, имеющие атрибут *[flags]*.
10. Коллекции не должны иметь слово Collection, Array, List и т.п. в названии, а должны вместо иметь последнее слово в названии во множественном числе:

```
// Неправильно
var filteredClientNameCollection = new List<string>();

// Правильно
var filteredClientNames = new List<string>();
```

1.2. Решения

В случае монолитных приложений название *.sln* файла должно совпадать с названием проекта из реальной жизни, например Aria, ArmadaSd, Jeton и т.п. В противном случае, когда у проекта нет формального названия или он состоит из нескольких решений, то название должно отражать функциональное назначение группы проектов. Например, AriaBackend, ArmadaSdServer.

С морфологической точки зрения, название решения должно быть существительным.

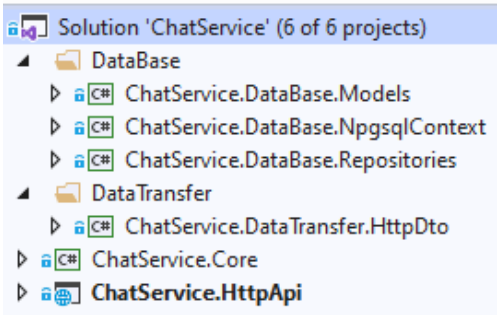
1.3. Проекты

Проектные файлы *.csproj* должны носить такие имена, чтобы автоматически генерируемые средой Visual Studio неймспейсы были корректными и не требовали ручного исправления. Для этого они должны удовлетворять шаблону:

<Название решения>.[Название архитектурной группы проектов.]<Назначение проекта>

Например, ChatService.DataBase.Models, где ChatService - название солюшена, DataBase - название архитектурного слоя с группой проектов и Models - назначение проекта. Группа проектов должна находиться в своей папке, как на диске, так и в логическом дереве солюшена.

Пример:



С морфологической точки зрения, название каждой части названия проекта должно быть существительным.

Из-за особенностей работы пространств имен в .NET последнюю составную часть названия проекта с его назначением лучше не называть так же, как любой из содержащихся в нем классов, так как тогда при использовании этого класса перед ним каждый раз придется явно указывать неймспейс, что довольно непрактично.

1.4. Файлы исходных кодов

Необходимо соблюдать правило: в одном файле должна быть расположен только одна из следующих лексем в единственном экземпляре:

- класс
- интерфейс
- структура
- перечисление.

При этом файл должен называться точно так же, как и содержащаяся в нем вышеописанная лексема и иметь расширение *.cs*.

1.5. Пространства имен

Пространство имен должно отражать логическую структуру решения со всеми вложенными папками и проектами до текущего файла. Для этого оно должно удовлетворять шаблону:

<Название проекта>.[Вложенные в проект папки, ведущие к файлу]

Подобный шаблон стандартен для Visual Studio и создается автоматически при создании файла. Крайне важно не редактировать его руками, так как в случае введения нестандартной системы неймспейсов, часто происходит её нарушение новыми участниками проекта, да и вообще очень легко забыть про необходимость

вручную поправить неймспейс при создании файла. Поэтому, чтобы автоматически проставленный неймспейс всегда был корректен, крайне важно соблюдать правила наименования проектов.

1.6. Аббревиатуры

В аббревиатурах необходимо приводить к верхнему регистру только первый символ. Пример:

```
// Неправильно
public class HTTPUtil
{
    ...
}
```

```
// Неправильно
long ID;
long cardID
long CARDID;
```

```
// Правильно
public class HttpUtil
{
    ...
}
```

```
// Правильно
long id;
long cardId;
```

1.7. Классы

Имена классов должны быть существительным в единственном числе и, желательно, иметь окончание -er.

Также необходимо всегда явно указывать модификатор доступа: *public*, *protected*, *internal*, *private*.

```
public class DatasetViewer
{
```

```
    ...
}
```

1.8. Структуры

Аналогично классам.

1.9. Интерфейсы

Аналогично классам, только в начале должна быть большая буква i для явного обозначения, что это именно интерфейс. Пример:

```
public interface IAbbacyService
{
    ...
}
```

1.10. Методы

Названия методов должны начинаться с глагола и отражать суть выполняемого методом действия.

Пример:

```
public void CalculateFrequency()
{
    ...
}
```

В случае, когда метод асинхронный, к названию добавляется суффикс *Async*.

Пример:

```
public async Task CalculateFrequencyAsync()
{
    ...
}
```

1.11. Параметры методов

Параметры методов пишутся с маленькой буквы и являются существительными.

```
public void PrintExampleString(string clientName)
{
```

```
    ...
}
```

1.12. Локальные переменные

Имена локальным переменным дается аналогично параметрам методов. Имя переменной не должно содержать в себе тип переменной.

Пример:

```
// Неверно
var inputInt = Console.ReadLine();
```

```
// Верно
var clientName = Console.ReadLine();
```

Также можно использовать *var* только тогда, когда тип переменной очевиден.

Пример:

```
// Неверно, так как не понятен тип (int? uint? float?)
var i = 3;
```

```
// Верно, тип Client
var client = new Client();
```

Важно избегать коротких имен или имен, которые можно спутать с другими наименованиями.

Пример:

```
bool b001 = (lo == l0) ? (I1 == 11) : (lOl != 101);
```

1.13. Члены-данные классов

Имена приватных полей класса должны быть существительными в единственном числе и начинаться с нижнего подчеркивания. Имена публичных полей подчиняются правилам именования для локальных переменных, однако их крайне не рекомендуется использовать. Вместо публичных полей-данных в классах и структурах лучше использовать свойства. Переменные, не используемые вне области одного метода, должны быть объявлены как *локальные переменные* в самом теле метода.

Пример:

```
// Приватное поле.
private DateTime _date;
```

```
// Публичное поле (не рекомендуется использовать)
public string day;
```

1.14. Константы

Имена константам внутри локальных областей видимости даются аналогично именам для локальных переменных. А константам - членам класса -- аналогично для членов-данных класса.

Пример:

```
const int maxClients = 6;
```

1.15. Свойства

Имена свойств должны начинаться с заглавной буквы и быть существительными.

Имеют те же модификаторы доступа, что и классы.

Пример:

```
public int Bonus { get; set; }
```

1.16. Перечисления

Имена даются аналогично свойствам, за исключением случая, когда *enum* имеет флаги.

```
public enum DiscountType
{
   Percent = 0,
   Cost = 1,
}
```

Причем литеральным значениям энама всегда рекомендуется проставлять целое значение явно во избежание проблем при сериализации/десериализации, а также ставить после последнего значения запятую, чтобы при добавлении нового git фиксировал только новую строчку без предыдущей.

1.17. Делегаты

1.18. События

1.19. Параметры лямбда-выражений

Параметрам лямбда-выражений необходимо давать значимые имена, но допускается сокращение имени до одной или нескольких букв.

Пример:

```
// Верно
Clients.Any(client => client.Id == id)
```

```
// Допускается
Clients.Any(c => c.Id == id)
```

2. Соглашения по пробелам

2.1. Оператор ;

Перед оператором ; (точка с запятой) пробел не ставится:

```
int a = 5;
```

2.2. Оператор ,

Перед оператором , (запятая) никогда не ставится пробел, но всегда ставится после неё:

```
int a = 5, b = 6;
```

2.3. Бинарные операторы

До и после всех бинарных операторов всегда должен стоять пробел:

```
int a = 5 + 3 * b / (7 - c);
bool d = a < 10 && a > 3;
```

2.4. Условные операторы и циклы

После *if*, *else*, *for*, *do*, *while* и *foreach* всегда ставится пробел:

```
if (a > 5)
	a++;
else if (a < 3)
	a--;
else a = 0;

for (int i = 0; i < 10; i++)
	a++;

while (a > 0)
{
	a--;
}

do
{
	a--;
} while (a > 0);

foreach (var i in Enumerable.Range(0, 10))
{
	a += i;
}
```

2.5. Унарные операторы

Между операндом и унарным оператором пробелы не ставятся. Если оператор префиксный, то пробел ставится перед ним, а если постфиксный, то после него (кроме случаев соседства со скобками или запятой или точкой с запятой). Пример:

```
int c = 5 + ++a - b--;
bool d = !(c > 5);
```

2.6. Оператор .

До и после оператора точки пробел не ставится:

```
string s = d.ToString();
```

3. Соглашения по переносам строк и табуляции

3.1. Фигурные скобки

Фигурные скобки всегда располагаются на отдельных строках:

```
void Foo()
{
	for (int i = 0; i < 10; i++)
	{
		if (i % 2 == 0)
		{
			Console.WriteLine(i.ToString());
		}
	}
}
```

3.2. Длина строки

Максимальная длина строки кода без переноса составляет 120 символов. При превышении этого значения остаток инструкции стоит перенести на следующую строку.

3.3. Общие правила переносов и табуляции

Если инструкция вместе с табуляцией не умещается в 120, либо текущая вложенная закончена, то необходимо сделать перенос строки и он осуществляется по следующим правилам:

- Если это первый перенос для текущей простой или вложенной инструкции, то следующая строка сдвигается на один tab вправо
- Если это не первый перенос для текущей вложенной инструкции, то следующая строка идет на том же уровне табуляции, что и предыдущая
- Если текущая вложенная инструкция завершена, то следующая строка сдвигается на один tab влево относительно текущей
- Если новая строка содержит открывающую фигурную скобку *{*, то количество табов не изменяется по сравнению с предыдущей строкой
- Если новая строка содержит закрывающую фигурную скобку *}*, то она должна быть сдвинута на один таб влево по сравнению с предыдущей строкой

3.4. Параметры методов

Если объявление метода вместе с его табуляцией и списком параметров превышает 120 символов, либо количество параметров больше 4, то параметры необходимо разнести по разным строкам: один параметр -- одна строка. Причем соблюдается правило общей табуляции со сдвигом на один таб вправо от предыдущей строки. Пример:

```
// Неправильно (в одну строчку)
public TextSplitter(string fileName, string directoryPath, bool rewrite,
int maxLineLength, byte[] inputBuffer, DateTime timeStamp, bool binary,
Guid fileId, int maxBufferSize, int threadCount)
{
    ...
}

// Неправильно (в три строчки)
public TextSplitter(string fileName, string directoryPath, bool rewrite,
      int maxLineLength, byte[] inputBuffer, DateTime timeStamp,
      bool binary, Guid fileId, int maxBufferSize, int threadCount)
{
    ...
}

// Неправильно (в количество строк, равное количеству параметров, но не
// соблюдается правило табуляции)
public TextSplitter(string fileName,
                    string directoryPath,
                    bool rewrite,
                    int maxLineLength,
                    byte[] inputBuffer,
                    DateTime timeStamp,
                    bool binary,
                    Guid fileId,
                    int maxBufferSize,
                    int threadCount)
{
    ...
}

// Правильно (в количество строк, равное количеству параметров и один
// таб вправо)
public TextSplitter(string fileName,
      string directoryPath,
      bool rewrite,
```

```
      int maxLineLength,
      byte[] inputBuffer,
      DateTime timeStamp,
      bool binary,
      Guid fileId,
      int maxBufferSize,
      int threadCount)
{
    ...
}
```

3.5. Логические выражения

При переносе логических выражений на новую строку, логический оператор переносится на неё, а не остается на предыдущей. Правила табуляции должны соответствовать общим:

```
if (client.ClientMetrics != null
      && client.ClientMetrics.ChurnRisk != null
      && client.ClientMetrics.ChurnRisk <= 2
      && client.AverageReceiptCategory != ClientMetricCategory.NoData)
{
    ...
}
```

3.6. Linq запросы

LINQ-запросы, содержащие более одного linq-метода, необходимо логически разбивать по разным строкам для большей читабельности. Первая коллекция располагается на первой строке, а затем каждый применяющийся к ней метод располагается на следующей строке. Пустые строки посреди одного Linq-запроса не допускаются.

Пример:

```
Client client = await _dbContext.Clients
    .AsNoTracking()
    .FirstAsync(c => c.Id == id);
```

В случае вложенных запросов Linq, они так же сдвигаются на таб вправо и переносить строку надо перед символом =>:

```
category.SkuSubcategories = category.SkuSubcategories
      .OrderByDescending(sc
            => sc.SkuGroups.Max(g
                  => g.GroupPurchaseHistories.FirstOrDefault(gph
                        => gph.ClientId == clientId)?.Affinity ?? 0))
      .ToList();
```

3.7. Пустые конструкторы

В случае, если необходимо прописать пустой конструктор, фигурные скобки проставляются по всем правилам, но между ними не должно быть пустых строк:

```
// Неправильно
public TextSplitter() { }

// Неправильно
public TextSplitter()
{

}

// Правильно
public TextSplitter()
{
}
```

3.8. Использование this и base в конструкторах

Так как операторы this и base в конструкторах идут после бинарного оператора :, то к ним применяются правила расположения операндов вокруг бинарных операторов, то есть до и после : ставится пробел, однако конструкции : this(...) и : base(...) принято переносить на новую строчку со сдвигом на один таб вправо:

```
public TextSplitter(string filename)
      : base(filename)
{
}

public TextSplitter(string filename)
      : this(filename)
{
}
```

4. XML комментарии

4.1. Что стоит комментировать

Необходимо проставлять xml-комментарии на английском языке всем объявлениям:

- классов
- интерфейсов
- публичных методов
- событий
- свойств с нестандартными сеттерами или геттерами, которые хочется пояснить
- приватных методов, которые нуждаются в пояснении.

Для комментариев к классам, интерфейсам, свойствам и событиям должна быть обязательно заполнена только секция *summary*. Остальные опционально.

Для методов должны быть заполнены секции:

- *summary*
- *param* для каждого параметра
- *returns*
- *exception*, если доподлинно известно, что метод может кидать конкретные исключения при определенных обстоятельствах, которые необходимо описать

Секция *summary* должна содержать общие сведения о том что делает класс или метод.

Секция *param* описывает чем является каждый параметр в контексте метода.

Секция *returns* отвечает на вопрос что возвращает метод.

Секция *exception* описывает условия при которых метод может кинуть определенное исключение.

Ещё в случае упоминания в комментарии других синтаксических единиц, например другого класса, желательно использование тэга *seealso*.

4.2. Как нужно комментировать

Все комментарии необходимо начинать с большой буквы и завершать точками.

Не стоит засорять код такими капитанскими комментариями, как например, конструктор - *This method constructs instance of class*. Лучше тогда не писать ничего вовсе.

Если комментарий состоит из одной только секции *summary* и она однострочная, то в целях экономии места допускается запись всего комментария вместе с тэгами в одну строчку. Пример:

Классический стиль

```
/// <summary>
/// Example empty class.
/// </summary>
public class ExampleClass
{
}
```

Компактный стиль (рекомендуется для коротких комментариев)

```
/// <summary>Example empty class.</summary>
public class ExampleClass
{
}
```

Для всех секций кроме *summary* у классов описание должно быть построено в третьем лице и начинаться с глагола. Пример:

```
/// <summary> Invoked when a task is sent for execution.</summary>
event EventHandler<TaskStartedEventArgs> TaskStarted;

/// <summary>Updates tracking task asynchronously.</summary>
Task UpdateTrackingTaskAsync();

/// <summary>Gets and sets id of camera.</summary>
int CameraId { get; set; }
```

4.3. Наследование комментариев

Для отсутствия копипасты комментариев при наследовании классов или реализации интерфейсов необходимо пользоваться тэгом *Inheritdoc*. Например, при проставлении его классу-реализации какого-либо интерфейса, имеющего XML-комментарии, он автоматически добавляет ему все комментарии из интерфейса. Однако при проставлении членам класса собственных комментариев, комментарии из интерфейса или базового класса не применяются. Для комбинирования необходимо специально проставлять *Inheritdoc* данному члену класса с указанием откуда брать остальные тэги.

4.4. Пример

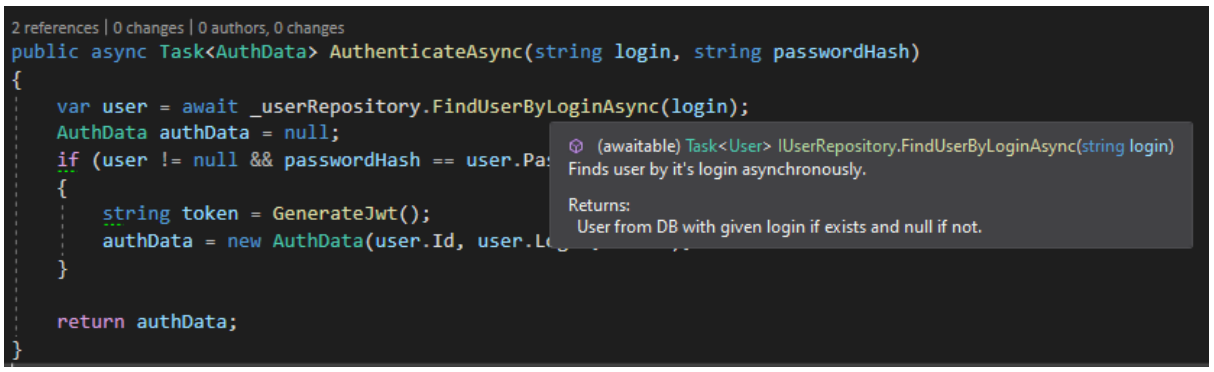
Интерфейс:

```
/// <summary>
/// Repository for working with users in DB.
/// </summary>
public interface IUserRepository
{
    /// <summary>
    /// Finds user by it's login asynchronously.
    /// </summary>
    /// <param name="login">Login of user.</param>
    /// <returns>
    /// User from DB with given login if exists and null if not.
    /// </returns>
    Task<User> FindUserByLoginAsync(string login);

    /// <summary>
    /// Adds given user to DB asynchronously.
    /// </summary>
    /// <param name="user">Filled <see cref="User"/> object.</param>
    Task AddUserAsync(User user);

    /// <summary>
    /// Removes user from DB by it's Id asynchronously.
    /// </summary>
    /// <param name="id">Id of user to remove from DB.</param>
    Task RemoveUserAsync(Guid id);
}
```

Реализация:

```
/// <inheritdoc cref="IUserRepository"/>
public class UserRepository : IUserRepository
{
    private readonly PostgresContext _postgresContext;

    public UserRepository(PostgresContext postgresContext)
    {
        _postgresContext = postgresContext;
    }

    public async Task AddUserAsync(User coreUser)
    {
        DbModels.User dbUser =
DbToCoreUserConverter.ConvertBack(coreUser);
        _postgresContext.Users.Add(dbUser);
        await _postgresContext.SaveChangesAsync();
    }

    public async Task<User> FindUserByLoginAsync(string login)
    {
        DbModels.User dbUser = await _postgresContext.Users
            .AsNoTracking()
            .FirstOrDefaultAsync(u => u.Login == login);
        return DbToCoreUserConverter.Convert(dbUser);
    }

    /// <inheritdoc cref="IUserRepository.RemoveUserAsync"/>
    /// <exception cref="GatewayRepositoryUserNotFoundException">
    /// Raised when there is no user with given Id in DB.
    /// </exception>
    public async Task RemoveUserAsync(Guid id)
    {
        var dbUser = await _postgresContext.Users.FindAsync(id);
        if (dbUser == null)
        {
            throw new GatewayRepositoryUserNotFoundException($"User with
id '{id}' can't be removed because it " +
                $"wasn't found in db.");
        }

        _postgresContext.Users.Remove(dbUser);
        await _postgresContext.SaveChangesAsync();
    }
```

После этого при наведении мышки на любой из методов реализации или интерфейса или параметра метода, будет отображаться подобная картина:



4.5. Полезные ссылки

Ссылка на msdn:
https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/xmldoc/

5. Модульные тесты

<Здесь про arrange, act, assert>

6. Источники

1. https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/inside-a-program/coding-conventions
2. https://habr.com/ru/post/272053/
3.