ANTHILL

Соглашение о кодировании на языке C#

Август 2018

Андрей Лежебоков
CTO, Anthill Company
Таганрог, Ростовская область, Россия

Назначение

Соглашения о написании кода предназначены для реализации следующих целей.

- Создание согласованного вида кода, позволяющего сосредоточиться на содержимом, а не на структуре.
- Предоставление возможности делать предположения, основанные на опыте, и поэтому быстрее понимать код.
- Упрощение процессов копирования, изменения и обслуживания кода.
- Предоставление разработчикам лучших методик программирования на C#.

Общие рекомендации по работе в команде

1. Обязательно соблюдайте описанные ниже рекомендации по кодированию!
2. Перед написанием кода обязательно уделите некоторое время обдумыванию задачи и проектированию (визуальное, схематичное).
    a. Следует наметить для себя основные пути реализации той или иной функциональности, определить перспективы по дальнейшему возможному улучшению и расширению данной функциональности.
    b. В случае если при реализации необходимо использовать несколько классов нарисуйте их диаграмму и определите отношения между ними. Возможно, в процессе этой работы вы найдете лучший путь реализации заданной функциональности.
3. Старайтесь, по возможности, находить как можно более гибкие и изящные пути реализации необходимой функциональности.
    a. Возможно, это позволит в дальнейшем упростить поддержку написанного вами кода, а также упростить его возможную модернизацию.
    b. Изящность решения ни в коем случае не означает запутанность и сложность, а также максимально большое количество классов.
    c. Старайтесь, по возможности, обходится минимальным числом небольших классов, возможно стандартных.
4. В случае если путь решения до конца не ясен не стесняйтесь посоветоваться с коллегами. Помните, что потерянное вами время может значительно сильнее отразится на вашем профессиональном авторитете, нежели четко и конкретно сформулированные вопросы коллегам.
5. С другой стороны старайтесь отнимать у коллег как можно меньше времени. Для этого формулируйте вопросы как можно более конкретно и лаконично.

   a. Лучше всего задавать такие вопросы по электронной почте/чату и как можно дольше сохранять эти письма/сообщения.
   b. Это позволит вам в дальнейшем вернуться к данному вопросу и уточнить необходимые детали не отвлекая человека, с которым вы вели диалог.
6. Используйте личные заметки, блокноты, тетради для документирования идей/мыслей/требований/пожеланий/пояснений по дальнейшему процессу работы над задачей/проектом

Общие правила разработки ASP.NET кода

1. Все элементы управления должны иметь осмысленные имена в соответствии с их назначением. Желательно (но необязательно) отражать в них также и тип элемента управления для улучшения читаемости кода.
2. HTML-код в файлах должен быть по возможности отформатирован в древовидную структуру для облегчения ее восприятия.
3. В случае если JavaScript-код используется в нескольких страницах обязательно выносить такой код в отдельные js-файлы. Таким же образом следует поступать если JavaScript-код имеет большой объем по сравнению с содержанием всей страницы.
4. HTML-код, который генерируется приложением должен соответствовать спецификации XHTML и по возможности должен быть отформатирован в древовидную структуру для облегчения отладки.
5. Старайтесь по возможности использовать стандартные элементы управления .NET вместо написания своих собственных.
6. В случае если сложная совокупность элементов управления с обособленным поведением может быть потенциально использована в других формах оформляйте ее в виде отдельного пользовательского элемента управления.
7. При проектировании формы старайтесь минимизировать число возвратов формы (postback).
8. Желательно тестировать функциональность написанных форм в разных браузерах.

Общие правила разработки классов

1. Допускается определять в классе публичные (public и internal) поля, только если они являются readonly.
2. Старайтесь реализовать в виде свойств только то, что отражает состояние класса или объекта. Например, если вы делаете свою коллекцию, то количество элементов (Count) должно быть свойством, а операцию преобразования ее в

массив (GetArray) лучше сделать методом. Причем вместо Get в данном случае лучше использовать другое слово, например, ToXxxx. То есть метод будет называться ToArray().

3. Используйте методы, если выполняемая операция является преобразованием, имеет побочный эффект или долго выполняется.
4. Используйте метод, если важен порядок выполнения операций.
5. Свойство не должно менять своего значения от вызова к вызову, если состояние объекта не изменяется. Если результат при новом вызове может быть другим при том же состоянии объекта, используйте метод.
6. Не используйте свойства «только для записи». Потребность в таком свойстве может быть признаком плохого проектирования.
7. Функциональность, выполняемая написанным вами методом должна на 100% соответствовать его имени. Т.е. из названия метода должно быть понятно, что и как метод делает. Возвращаемое значение также должно быть очевидно из названия метода. (например, методы, начинающиеся на Is должны обязательно возвращать Boolean, не выбрасывать исключения и не делать никаких побочных операций (за исключением, может быть, логирования) кроме возврата результата)
8. Запечатанные (sealed) классы не должны иметь ни защищенных, ни виртуальных методов. Такие методы используются в производных классах, а sealed-класс не может быть базовым.
9. Классы, определяющие только статические методы и свойства, должны быть обязательно помечены как static.
10. Члены класса должны быть разбиты на регионы в соответствии с уровнем доступа.
11. Регионы должны быть использованы в следующем порядке: constants, static members (public comes first), public constructors, public events, public properties, public methods, protected properties, protected methods, private properties, private methods, protected fields, private fields.
12. Не используйте литеральные константы (магические числа, зашитые в код размеры буферов, времена ожидания и тому подобное). Лучше определите константу (если вы никогда не будете ее менять) или переменную только для чтения (если она может измениться в будущих версиях вашего класса). Строковые константы сохраняется в соответствующей таблице в ресурсах приложения.
13. Старайтесь обрабатывать только известные вам исключения. Если вы все же обрабатываете все исключения, то или, проведя необходимую обработку, генерируйте исключение повторно, чтобы его могли обработать последующие фильтры, или выводите пользователю максимально полную информацию об ошибке. Если целью перехвата исключений является очистка ресурсов после сбоя, лучше воспользоваться секцией finally.
14. Если требуется повторный выброс исключения из блока catch используйте только throw без параметров и не в коем случае не throw new Exception, т.к. в этом случае будет потеряна вся информация из StackTrace.

15. Старайтесь не использовать блоки try/catch с перехватом всех исключений. Этот блок может приводить к появлению трудно-обнаруживаемых ошибок, внешне ничем себя не проявляющих. Если все же использование такого блока необходимо, это следует отразить в комментариях к этому блоку. Компромиссным вариантом в этом случае может быть перехват всех исключений с последующим их логированием, но этот подход также должен применяться только в крайнем случае.
16. Используйте метод TryParse вместо Parse для базовых типов.
17. При использовании автоматической генерации кода методы добавляемые в автоматически сгенерированные классы следует добавлять в отдельный файл.
18. **Код должен компилироваться без предупреждений!**

Соглашения об именах

Стили использования регистра букв

В проектах допускается использование следующих стилей:

Паскаль – указание этого стиля оформления идентификатора означает, что первая буква заглавная и все последующие первые буквы слов тоже заглавные. Например, BackColor,LastModified, DateTime.

Кэмел – указание этого стиля означает, что первая буква строчная, а остальные первые буквы слов заглавные. Например, borderColor, accessTime, templateName.

Предпочтение отдается Кэмел.

Общие правила именования объектов

1. **Важно помнить!** Код чаще читается, чем пишется, поэтому не экономьте на понятности и чистоте кода ради скорости набора.
2. Не используйте малопонятные префиксы или суффиксы (например, венгерскую нотацию), современные языки и средства разработки позволяют контролировать типы данных на этапе разработки и сборки.
3. Не используйте подчеркивание для отделения слов внутри идентификаторов, это удлиняет идентификаторы и затрудняет чтение. Вместо этого используйте стиль именования Кемел или Паскаль.
4. Старайтесь не использовать сокращения лишний раз, помните о тех, кто читает ваш код.
5. Старайтесь делать имена идентификаторов как можно короче (но не в ущерб читабельности). Помните, что современные языки позволяют формировать имя из пространств имен и типов. Главное, чтобы смысл идентификатора был понятен в

используемом контексте. Например, количество элементов коллекции лучше назвать Count, а не CountOfElementsInMyCollection.

6. Когда придумываете название для нового, общедоступного (public) класса, пространства имен или интерфейса, старайтесь не использовать имена, потенциально или явно конфликтующие со стандартными идентификаторами.
7. Предпочтительно использовать имена, которые ясно и четко описывают предназначение и/или смысл сущности.
8. Старайтесь не использовать для разных сущностей имена, отличающиеся только регистром букв. Разрабатываемые вами компоненты могут быть использованы из языков, не различающих регистр, и некоторые методы (или даже весь компонент) окажутся недоступными.
9. Старайтесь использовать имена с простым написанием. Их легче читать и набирать. Избегайте (в разумных пределах) использования слов с двойными буквами, сложным чередованием согласных. Прежде, чем остановиться в выборе имени, убедитесь, что оно легко пишется и однозначно воспринимается на слух. Если оно с трудом читается, и вы ошибаетесь при его наборе, возможно, стоит выбрать другое.

Использования директив using

В коротких примерах кода, не содержащих директив using, рекомендуется использовать полные указания для пространства имен. Если известно, что пространство имен импортировано в проект по умолчанию, не требуется указывать полные имена из этого пространства имен. Полные имена, если они слишком длинные для одной строки, можно разбить после точки (.), как показано в следующем примере.

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

Нет необходимости изменять имена объектов, созданных с помощью инструментов разработки Visual Studio, чтобы привести их в соответствие с другими соглашениям.

Соглашения о расположении

Чтобы выделить структуру кода и облегчить чтение кода, в хорошем макете используется форматирование:

- Использование параметров редактора кода по умолчанию (логичные отступы, отступы по четыре символа, использование пробелов для табуляции).
- Запись только одного оператора в строке.
- Запись только одного объявления в строке.
- Если отступ для дополнительных строк не ставится автоматически, необходимо сделать для них отступ на одну позицию табуляции (четыре пробела).
- Добавление по крайней мере одной пустой строки между определениями методов и свойств.
- Использование скобок для ясности предложений в выражениях, как показано в следующем коде.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Соглашения о комментариях

- Комментарий размещается на отдельной строке, а не в конце строки кода.
- Текст комментария начинается с заглавной буквы.
- Текст комментария завершается точкой.
- Между разделителем комментария (/ /) и текстом комментария вставляется один пробел, как показано в следующем примере.

```
// The following declaration creates a query. It does not run
// the query.
```

- Не используйте блоки звездочек (*****) вокруг комментариев.

Соглашения по языку C#

В следующих подразделах описаны методики, которыми руководствуется команда C# для подготовки примеров и образцов кода.

Тип данных String

- Используйте строчную интерполяцию **$** для объединения коротких строк, как показано в следующем коде

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- Чтобы добавить строки в цикле, особенно когда вы работаете с большим количеством текста, используйте объект **StringBuilder**.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

Неявно типизированные локальные переменные

- Используйте неявное типирование для локальных переменных, когда тип переменной очевиден с правой стороны задания или когда точный тип не важен.

```
// When the type of a variable is clear from the context, use var
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

- Не используйте var, если тип не отображается с правой стороны задания.

```
// When the type of a variable is not clear from the context, use an
// explicit type.
int var4 = ExampleClass.ResultSoFar();
```

- Не следует полагаться на имя переменной, чтобы указать тип переменной.

```
// Naming the following variable inputInt is misleading.
```

```
// It is a string.
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Используйте неявное типирование для определения типа переменной цикла для циклов for и foreach.

```
var syllable = "ha";
var laugh = "";
for (var i = 0; i < 10; i++)
{
    laugh += syllable;
    Console.WriteLine(laugh);
}
```

```
foreach (var ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

Беззнаковые типы данных

- В общем случае используйте типы int, а не unsigned. Использование int является общим в C#, и при использовании int легче взаимодействовать с другими библиотеками.

Массивы

- Используйте сжатый синтаксис при инициализации массивов в строке объявления.

```
// Preferred syntax. Note that you cannot use var here instead of string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

```
// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a
time.
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

Делегаты

- Используйте сжатый синтаксис для создания экземпляров типа делегата.

```
// First, in class Program, define the delegate type and a method that
// has a matching signature.

// Define the type.
public delegate void Del(string message);

// Define a method that has a matching signature.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

```
// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.
Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.
Del exampleDel1 = new Del(DelMethod);
```

try-catch и использование выражений в обработке исключений

- Используйте инструкцию try-catch для обработки большинства исключений.

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
```

```
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        throw;
    }
}
```

- Упростите свой код, используя оператор C# **using**. Если у вас есть оператор **try-finally**, в котором единственный код в блоке finally является вызовом метода Dispose, используйте вместо него инструкцию using.

```
// This try-finally statement only calls Dispose in the finally block.
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}

// You can do the same thing with a using statement.
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

Операторы **&&** и **||**

- Чтобы избежать исключений и повысить производительность, пропуская ненужные сравнения, используйте && вместо & и || вместо | когда вы выполняете сравнения.

```
Console.Write("Enter a dividend: ");
var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
```

```
var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following condition
// causes a run-time error. The && operator short circuits when the
// first expression is false. That is, it does not evaluate the
// second expression. The & operator evaluates both, and causes
// a run-time error when divisor is 0.
if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

Оператор New

- Используйте краткую форму экземпляра объекта с неявной типизацией, как показано в следующем объявлении.

```
var instance1 = new ExampleClass();

// this is the same
ExampleClass instance2 = new ExampleClass();
```

- Используйте инициализаторы объектов для упрощения создания объектов.

```
// Object initializer.
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

Обработка событий

- Если вы определяете обработчик событий, который вам не нужно удалять в дальнейшем, то используйте лямбда-выражение

```
// Using a lambda expression shortens the following traditional definition.
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}

// You can use a lambda expression to define an event handler.
public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

Статические члены

- Вызывайте статические члены, используя имя класса: ClassName.StaticMember. Эта практика делает код более читаемым, делая статический доступ понятным.
- Не квалифицируйте статический член, определенный в базовом классе с именем производного класса. Хотя этот код компилируется, читаемость кода вводит в заблуждение, и в будущем код может сломаться, если вы добавите статический член с тем же именем в производный класс.

LINQ запросы

- Используйте значащие имена для переменных запроса. Следующий пример использует имя seattleCustomers для клиентов, которые находятся в Сиэтле.

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.Name;
```

- Используйте псевдонимы с использованием нотации Pascal.

```
var localDistributors =
```

```
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Переименуйте свойства, если имена свойств в результате выборки будут неоднозначными. Например, если ваш запрос возвращает имя клиента и идентификатор дистрибьютора, вместо того, чтобы оставлять их в качестве имени и идентификатора, переименуйте их, чтобы уточнить, что имя является именем клиента, а идентификатор - идентификатором дистрибьютора.

```
var localDistributors2 =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorID = dist.ID };
```

- Используйте неявное типирование в объявлении переменных запроса и переменных диапазона.

```
var seattleCustomers = from cust in customers
                       where cust.City == "Seattle"
                       select cust.Name;
```

- Используйте выравнивание предложения запроса в предложении from, как показано в предыдущих примерах.
- Используйте предложения where перед другими предложениями запроса, чтобы гарантировать, что более поздние предложения запроса работают с уменьшенным, отфильтрованным набором данных.

```
var seattleCustomers2 = from cust in customers
                        where cust.City == "Seattle"
                        orderby cust.Name
                        select cust;
```

- Для доступа к внутренним коллекциям используйте несколько **from** предложений вместо предложения join. Например, коллекция объектов Student может содержать

коллекцию тестов. Когда выполняется следующий запрос, он возвращает каждый балл, который превышает 90, вместе с фамилией учащегося, получившего оценку.

```
// Use a compound from to access the inner sequence within each element.
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

Резюме

Данный документ не является статическим, он живет и развивается вместе с развитием технологий, программных инструментов и языковых средств C#. Любые неточности, устаревающие соглашения и требования являются объектом наблюдения данного документа и должны найти свое отражение в нем по мере необходимости. Указанные в документе соглашения не являются “жесткими”, но помогают избежать серьезных временных затрат в случае командной работы над длительным проектом.

P.S. Все замечания/пожелания можно оставлять комментариями в этом документе или прислать на почту legebokov@gmail.com