СТИЛЬ КОДИРОВАНИЯ

– Пишите код так, как будто сопровождать его будет склонный к насилию психопат, который знает, где вы живете.

Стив Макконнелл, "Совершенный код".

В разделе «Основы программирования на языке C#» мы уже немного говорили о том, насколько важно для улучшения читабельности и удобства модификации программы правильно определять имена идентификаторов и записывать операторы языка. Теперь мы переходим к разработке относительно больших программ, в которых нам придется создавать собственные классы и целые иерархии классов. В этих условиях проблема создания читабельного кода становится очень актуальной.

Цитата, вынесенная в заголовок данного раздела, является своего рода «криком души» любого разработчика, который хоть раз был вынужден разбираться в коде, написанным его собратом по профессии, или модифицировать свой собственной код, созданный когда-то давно. Представьте ситуацию, что в этом коде нет ни единого комментария, все команды записаны в одну большую строчку, а переменные называются, например, x, xx, xxx, xxxx ... x_217 … Вы очень быстро поймете, как важно правильно оформлять программу.

В данном разделе мы рассмотрим основные приемы правильного оформления программ, при этом за основу будет взят неофициальный стандарт, предложенный Майком Крюгером (Mike Krueger) и получивший довольно широкое распространение среди разработчиков.

Работа с файлами и каталогами

Один класс – один файл

Каждый класс должен находиться в отдельном файле, имя которого совпадает с именем класса и имеет расширение cs. Например, класс Program должен находиться в файле с именем Program.cs. Никаких других классов в этом же файле быть не может, так как найти нужный класс по имени файла в грамотно организованном дереве каталогов гораздо проще, чем лихорадочно вспоминать, где же он лежит, или запускать глобальный поиск по всему проекту, особенно если вы не помните точное имя класса.

Размер файлов

Рекомендуемый максимальный размер одного файла – 2000 строк кода (при этом подразумевается, что на каждой строчке записана ровно одна команда). Если размер вашего класса не укладывается в указанное число строк, имеет смысл подумать о том, а нужен ли вам один большой класс, который делает ВСЕ? Возможно, его лучше разбить на несколько более мелких классов.

Каталоги

Не сваливайте все файлы в одну большую кучу в корневом каталоге проекта. Лучше грамотно организовать дерево каталогов проекта. Например, можно поместить все вспомогательные служебные классы в папку Utilities, а классы, отвечающие за работу с текстом – в папку TextUtilities.

Пространства имен и каталоги

Структура пространств имен должна соответствовать структуре каталогов. Например, если файлы классов расположены в папке Utilities проекта Project, то все классы, расположенные в данных файлах должны располагаться в пространстве имен Project.Utilities, или EPAM.Project.Utilities, если есть необходимость явного задания префикса имени организации (в нашем случае, EPAM).

Форматирование текста

Длина строк

Длина каждой строки программного кода не должна превышать 80 символов (примерно столько символов умещается на экране без горизонтальной прокрутки). В случае необходимости следует осуществлять перенос кода на следующую строку.

Правила переноса

Перенос можно осуществлять после запятой, точки с запятой или после оператора/операции. При этом переносимая часть строки должна быть сдвинута вправо, как минимум, на один стандартный символ табуляции.

Примеры, правильных переносов:

```
Console.WriteLine("Результат вычисления выражения равен {0}",
          1 + Math.pow(2, x / (y – 5 * Math.sin(z))) - tmp);

res = 1 + Math.pow(2, x /
          (y – 5 * Math.sin(z))) – tmp);
```

Пример, неправильных переносов:

```
// выравнивание происходит по одной и той же вертикальной линии
// без сдвига
Console.WriteLine("Результат вычисления выражения равен {0}", 1 +
Math.pow(2, x / (y – 5 * Math.sin(z))) - tmp);

// перенос произошел до операции
res = 1 + Math.pow(2, x / (y
          – 5 * Math.sin(z))) – tmp);
```

Символы табуляции

Обеспечить требуемый сдвиг при переносах можно двумя основными способами. Во-первых, нужным числом пробелов; во-вторых, символом табуляции. Несмотря на то, что Visual Studio, по умолчанию, обеспечивает отступы именно пробелами, вместо них рекомендуется использовать символы табуляции. Основными преимуществами их применения является:

- уменьшение размеров файлов исходных кодов (один символ табуляции занимает меньше памяти, чем три-четыре пробела);
- возможность индивидуальной настройки величины отступов в зависимости от личных пожеланий.

Изменить настройки Visual Studio в отношении используемых отступов можно, выполнив следующую последовательность команд: Tools→Options→Text Editor→C#→Tabs. Указанное действие отображено на Рис. 11.

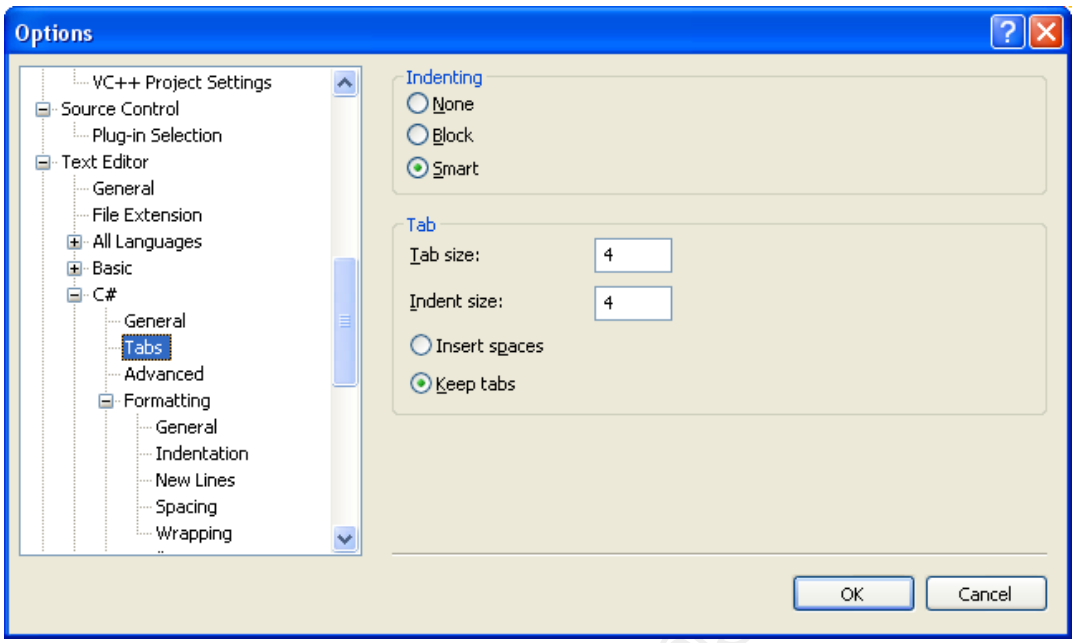


Рис. 11. Настройка табуляции.

Пустые строки

Рекомендуется использовать пустые строки, не содержащие в себе кода для отделения друг от друга отдельных методов, а также логических блоков внутри метода для повышения читабельности кода. Например:

```
...
// Вводим аргументы с клавиатуры
Console.WriteLine("Введите первое число");
arg1 = Console.ReadLine();
x = double.Parse(arg1);

// Проверяем, возможно ли деление (на ноль делить нельзя)
if (y == 0)
{
...
```

Пробелы

Одиночные пробелы используются для отделения друг от друга параметров при передаче их в методы, а также для выделения операндов в выражениях. Текст программы смотрится гораздо лучше, если писать:

```
z = Math.pow(x, y);   вместо   z=Math.pow(x,y);
```

и

```
res = x + y;   вместо   res=x+y;
```

Следует отметить, что указанное форматирование Visual Studio делает автоматически.

Табличное форматирование

При объявлении и инициализации нескольких переменных желательно использовать табличное форматирование. Следующий фрагмент программного кода

```
int x      = 5;
int radius = 12;
int len    = 2 * Math.PI * radius;
```

воспринимается гораздо лучше, чем

```
int x = 5;
int radius = 12;
int len = 2 * Math.PI * radius;
```

Правила объявления идентификаторов

При задании имен идентификаторов необходимо придерживаться правил именования.

Осмысленность имен

Имена идентификаторов должны быть осмысленными и отражать их содержание. Например, для обозначения суммы чисел может использоваться переменная с именем *summa*, но не *s*, *x*, или *var*. Оптимальным является название, состоящее из полных английских слов. Если вы не уверены в своем английском – не используйте сокращения. Ваши сокращения могут оказаться не вполне очевидными для других людей, более того, они могут оказаться самостоятельными словами, возможно, не вполне приличного смысла. Например, при использовании переменной, объявленной как:

```
int sesID;
```

возможно, как минимум, три варианта толкования сокращения ses:

1. Satellite Earth Station – земная станция спутниковой связи;
2. Surface-Effect Ship – судно на воздушной подушке, СВП;
3. SESion – сессия.

Какой именно из этих вариантов толкования имеется в виду?

Не менее жесткие правила касаются использования транслита. Далеко не каждый англоговорящий программист в состоянии понять, что MailFolder и PochtovayaPapka – это одно и то же.

Допустимые символы

В имена рекомендуется включать только заглавные и строчные латинские буквы, цифры и знак подчеркивания.

Хотя, формально, кодировка Unicode позволяет использовать для именования идентификаторов почти любые символы, в том числе и русские буквы, делать этого настоятельно не рекомендуется. Например, приведенная ниже программа будет корректно работать, но использовать написанные в ней классы сможет только программист, понимающий русский язык.

```
class Программа
{
   static void Main(string[] args)
   {
      int первоеСлагаемое;
      int второеСлагаемое;
      int сумма;
      первоеСлагаемое = int.Parse(Console.ReadLine());
```

```
            вторoеСлагаемое = int.Parse(Console.ReadLine());
            сумма = первоеСлагаемое + вторoеСлагаемое;
            Console.WriteLine(сумма);
            Console.Read();
        }
    }
```

Одна строка – одна переменная

На одной строке лучше объявлять только одну переменную. Это необходимо для того, чтобы можно было прокомментировать каждую из них. Например, вместо объявления вида:

```
int x, y;
```

лучше использовать следующий вариант:

```
int x;
int y;
```

Форматы именований

Существует несколько видов нотаций – соглашений о правилах создания имен.

В нотации Pascal каждое слово, входящее в идентификатор, начинается с заглавной буквы, например:

```
Age, LastName, TimeOfDeath
```

В нотации Camel с заглавной буквы начинается каждое слово идентификатора, кроме первого, например:

```
age, lastName, timeOfDeath
```

Венгерская нотация отличается от предыдущей наличием префикса, соответствующего типу величины, например:

```
fAge, sName, iTime
```

Существует еще и нотация Upper case, которая допускает в названии программных объектов использовать только заглавные буквы, например:

```
E, PI, SUM
```

Наиболее часто используются нотации Pascal и Camel. Рассмотрим для каких идентификаторов какую нотацию рекомендуется использовать.

Классы и структуры

При задании имен классов и структур используются следующие правила:

- название должно представлять собой существительное в единственном, или множественном числе;
- имя должно записываться в соответствии с нотацией Pascal;
- какие-либо префиксы в имени должны отсутствовать.

Примерами корректных имен классов и структур могут быть: Circle, FileInfo и Graphics. Необходимо менять стандартные имена, присваиваемые средой разработки по умолчанию: Class1, Form1 и т.д.

Интерфейсы

При задании имен интерфейсов используется следующий набор правил:

- название должно представлять собой существительное в единственном, или множественном числе;
- имя должно записываться в соответствии с венгерской нотацией, в качестве префикса перед именем ставится заглавная латинская буква I (сокращение от Interface).

Примерами корректных имен интерфейсов могут быть: IEnumerator, IComponent и IComparer.

Перечисления

К именованию перечислений и их элементов предъявляются следующие требования:

- имена как собственно перечислений, так и их элементов должны записываться в соответствии с нотацией Pascal;
- название перечисления должно записываться во множественном числе;
- названия элементов должны записываться в единственном числе.

Пример корректного именования перечисления:

```
enum DaysOfWeek
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}
```

Замечание

Отметим, что если в коде нужно использовать названия дней недели, то не нужно пользоваться перечислением. Для этого есть специальные методы.

Методы классов

Названия методов классов рекомендуется записывать в соответствии со следующими правилами:

- имена методов должны образовываться глагольными формами единственного, или множественного числа;
- имена методов должны писаться в соответствии с нотацией Pascal.

Например, метод, который вычисляет и возвращает площадь фигуры, может называться GetArea, метод вставки имеет смысл называть Insert, а метод удаления Delete.

Параметры методов и поля классов

При задании имен параметров методов и полей классов рекомендуется придерживаться нотации Camel. Например, корректными будут названия полей: length, arrayLength, area.

А при задании имен констант и read-only полей рекомендуется использовать нотацию Pascal. Для случая коротких названий допустимо использование нотации Upper case.

Локальные переменные методов

Для именования локальных переменных методов принято использовать нотацию Camel. В данном случае корректными будут считаться следующие имена переменных: localVariable, summa, temporaryArray.

Для счетчиков циклов допустимо использовать однобуквенные имена, например, i, j, k. В циклах более чем тройной степени вложенности дополнительно могут использоваться имена l, m, n. Соответственно, использовать имена i, j, k для каких-либо иных целей запрещается, имена l, m, n – не рекомендуется.

В цикле foreach в качестве имени элемента обычно используется item, или используют имя сущности в единственном числе, например:

```
foreach (User user in UserList)
{
   ...
}
```

Комментарии

Грамотное документирование является очень важной составляющей оформления программного кода. Принято считать, что программа продокументирована на приемлемом уровне, если объем комментариев в ней составляет не менее 30% от общего объема исходного кода программы. Причем речь идет именно об осмысленных комментариях, поясняющих логику работы программы, а не о формальных действиях вида:

```
int x; // объявляем переменную x целого типа
```

Данный комментарий не имеет смысла.

Язык C# поддерживает три вида комментариев: одно- и многострочные, а также документационные.

Однострочные комментарии

Однострочные комментарии начинаются с пары символов // – два слеша, идущие подряд. Любой идущий за ними текст до конца строки считается комментарием и не участвует в компиляции программы и ее исполнении. Однострочные комментарии записываются либо перед комментируемым фрагментом кода:

```
// Вводим аргументы с клавиатуры
Console.WriteLine("Введите первое число");
arg1 = Console.ReadLine();
x = double.Parse(arg1);
```

либо непосредственно в самой строке:

```
double radius; // радиус окружности
```

Многострочные комментарии

Многострочные комментарии начинаются с пары символов /* и заканчиваются парой символов */. Они могут использоваться как для комментирования нескольких строк кода идущих подряд:

```
/* это
   многострочный
   комментарий */
```

так и для комментирования небольших фрагментов кода внутри какой-либо одной строки, например, для изъятия из процесса компиляции какого-либо фрагмента исходного кода:

```
x = 2 * y /* - z  * 3 */ + 1;
```

Разумеется, такое изъятие кода может быть лишь временной мерой. Следует понимать, зачем делается комментарий. Если только для того, чтобы удалить часть кода – лучше удалить ее совсем. Комментировать код подобным образом можно только на время отладки.

Документационные комментарии

В основе концепции документационных комментариев лежит идея использования уже написанных комментариев для автоматической генерации документации и в качестве автоматически всплывающих подсказок так, как это происходит при работе со стандартными классами. Для того, чтобы система IntelliSense и компилятор документации могли понимать и обрабатывать комментарии, необходимо, чтобы они писались в соответствии с определенным форматом.

Первым признаком того, что комментарий является документационным, являются три слеша (///) в его начале. Visual Studio распознает по этому признаку документационный комментарий и автоматически генерирует для него заготовку в зависимости от того, в каком месте поставлены три слеша. После генерации заготовки разработчику достаточно лишь заполнить содержимое соответствующих тегов и документационный комментарий готов.

Полный список тегов, используемых при работе с документационными комментариями, приведен в следующей таблице:

Тег	Описание
<c>	Помечает текст как программный код.
<code>	Помечает множество строк как программный код.
<example>	Помечает текст как пример кода.
<exception>	Документирует класс исключения.
<include>	Включает комментарии из другого файла документации.
<list>	Описывает список.
<param>	Задает описание передаваемого в метод аргумента. Имеет атрибут name, задающий имя аргумента.
<paramref>	Указывает, что слово является параметром метода.
<permission>	Документирует доступ к члену класса.
<remarks>	Добавляет описание члена класса.
<returns>	Документирует возвращаемое методом значение.
<see>	Предоставляет перекрестную ссылку на другой параметр.
<seealso>	Задает раздел «Смотри также».
<summary>	Документирует краткое описание класса, или его членов.
<value>	Описывает свойство.

Отметим, что если речь не идет о создании полноценной библиотеки классов, которую предполагается распространять как самостоятельный программный продукт и которую необходимо снабдить максимально подробной помощью, сопоставимой по возможностям с MSDN, в использовании всех этих тегов нет необходимости. В повседневной жизни разработчику необходимо задавать лишь минимальное описание созданного им программного кода, для чего вполне достаточно ограничиться знанием тегов <summary> и <param>, отображаемых в качестве подсказок системой IntelliSense, а также тега <returns>, поскольку при задании описания для методов, возвращающих значение, он генерируется автоматически и его содержимое отображается в Object Browser.

В теле программы документировать с использованием тегов <summary>, <param> и <returns> необходимо, как минимум:

- все классы, структуры, перечисления и интерфейсы;
- все не закрытые (private) поля, свойства и методы классов;
- все элементы перечислений;
- закрытые поля, свойства и методы в том случае, если тело класса имеет достаточно большой размер, или допускается хотя бы мысль о возможности его дальнейшей модификации;

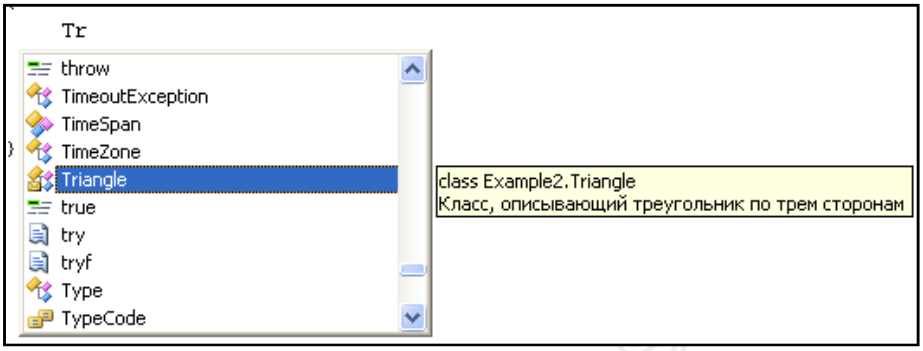
При разработке коммерческих библиотек необходимо проводить полноценное документирование с использованием всего спектра тегов.

Рассмотрим пример полностью документированного класса.

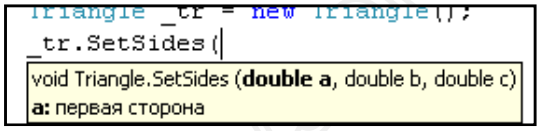
```
/// <summary>
/// Класс, описывающий треугольник по трем сторонам
/// </summary>
public class Triangle
{
   private double a; // первая сторона треугольника
   private double b; // вторая сторона треугольника
   private double c; // третья сторона треугольника
   /// <summary>
   /// Конструктор
   /// </summary>
   /// <param name="a">первая сторона</param>
   /// <param name="b">вторая сторона</param>
   /// <param name="c">третья сторона</param>
   public void Triangle(double a, double b, double c)
   {
      this.a = a;
      this.b = b;
      this.c = c;
   }
   /// <summary>
   /// Определение площади треугольника
   /// </summary>
   /// <returns>площадь треугольника</returns>
   public double GetArea()
   {
      double p = (a + b + c) / 2; // полупериметр треугольника
      // рассчитываем площадь по формуле Герона
```

```
            double s = Math.Sqrt(p * (p - a) * (p - b) * (p - c));
            return s;
        }
    }
```

После такого документирования, система IntelliSense начнет отображать документационные комментарии в качестве подсказок. Например, если начать набирать название класса, высветится подсказка, соответствующая описанию класса.



Аналогично, при работе с объектом класса, для его методов и их аргументов также будут отображаться подсказки:



Магические числа

Магическими в программировании принято называть числовые значения, появляющиеся в программе с целью, понятной только их создателю. Если вы видите в чужой программе числовое значение, и не можете мгновенно сказать, зачем оно здесь нужно, знайте – перед вами «магическое число».

Использование в программе магических чисел несет в себе три потенциальных угрозы. Во-первых, когда не ясно, зачем это число нужно, при модификации программы придется, как минимум, тратить некоторое время на проведение исследований с целью узнать, что будет, если его поменять. Во-вторых, если одно и то же магическое число встречается в программе несколько раз, то, при необходимости его изменить, придется активно лазить по коду и заниматься розысками, с тем, чтобы потом скомпилировать программу и узнать, что где-то его все-таки не исправили. В-третьих, если в коде встречаются два магических числа с одинаковыми значениями, и нужно изменить только одно из них, или, что еще хуже, два связанных магических числа с разными значениями...

Избежать использования этих чисел можно только заменой их константами, или readonly переменными. Сравните, например, два фрагмента кода. Этот:

```
gr.DrawRect(5, 5, width - 10, height - 10);
```

и этот:

```
const int Indent = 5; // величина отступа от края области рисования
gr.DrawRect(Indent, Indent, width - Indent * 2,
            height - Indent * 2);
```

В каком изменение величины отступа проще?