Validio Software, LLC
110 110th Ave. NE, Suite 310
Bellevue, WA 98004-5828

Tel: (425) 990-8803 Fax: (425) 990-8804
E-mail: info@validio.com Web:www.validio.com

Author: Pivovarova Irina
Last edited by: Pivovarova Irina
Modification Date: 1/4/2011

C# Coding Guidelines

Table of Contents

Introduction

These guidelines provide the requirements to the source code written in C# for the applications targeted at the Microsoft .NET platform. If there are no other stated requirements, the requirements outlined in this document are mandatory and must be strictly followed by programmers.

The main topics regulated by the guidelines are the source code formatting, naming and capitalization conventions. Given material also covers the code readability improvement, reuse policy, and specific issues for the technology areas like ASP .NET or .NET Windows Forms.

Project and File Structure

Project Structure and Recommended Settings

1. Each project must have its own sub-folder within a solution. Do not place the solution file and project files within the same folder;
2. Use folders within projects for organizing the source code; the namespaces must correspond to the folder structure;
3. Specify XML documentation files to force the compiler to produce documentation for your code and to warn you about the missing comments.

Source File Structure

The source code files must have the following structure (the order in which the source file elements are listed is also important):

1. "using" declarations;
2. Namespace declaration;
3. [Optional] Enumerations or helper structures relevant to the class stored in this source file. Please, remember this can affect the VS.NET designer behavior. These declarations can also be placed in separate files;
4. Class declaration. It is strongly recommended to declare only one class within a single source file, except for the unit test suites that must accompany the production code;

Class, Interface and Structure Layout

Classes and structures must be declared in the following pre-defined order:

1. Data members;
2. Constructor(s);
3. Properties (first public, then protected and then private);
4. Methods (first public, then protected and then private);
5. Delegates and events.

NOTE: Some items from this list may be inapplicable for a class, a structure or an interface. If this case, one must skip the item that cannot be applied under current circumstances and go on with the next item.

Source Code Formatting

All code must be formatted according to the following rules:

1. The "Tabs" option must be set to the "Keep tabs" mode;
2. The default indent size (4 characters) must be used, if there are no other conventions within your project team;

3. Conditional, looping and switch constructs must be written according to the following layout examples:

```
if (file.Exists(fileName))
{
  file.Open(fileName);
}
```

```
for (int i = 0; i < MAX_ELEMENTS; ++i)
{
  array[i] = i * MULTIPLY_FACTOR;
}
```

```
switch (workMode)
{
  case WorkMode.Add:
    DoAdd();
    break;
  case WorkMode.Update:
    DoUpdate();
    break;
}
```

4. Blank lines must be used to improve the code readability. Usually they are used to separate relatively independent sections of a code. One must always use blank lines:
    - After the last "using" declaration and between namespace declarations;
    - Between methods;
    - Between the declarations of local variables in a method and its first statement;
    - Before multiline or single-line comments except for the comment that is the first line after a curly brace opening a block of statements;
    - Before logically separated blocks of code within a method.
5. Blank spaces must also be used to improve the code readability. It is required to insert blank spaces:
    - When a keyword is followed by a parenthesis. A blank space must be inserted between the keyword and the parenthesis:

```
if (condition)
```

    - After commas in argument lists:

```
int result = Calculate(argumentOne, argumentTwo);
```

    - Between binary operators and their operands:

```
int result = argumentOne + argumentTwo;
```

    - Between parts of a "for" statement:

```
for (int i = 0; i < MAX_ELEMENTS; ++i)
```

    Arguments of unary operators must never be separated by blank spaces:

```
i++;
--counter;
```

6. All long lines must be wrapped. The recommended maximum length limit is 78 characters. When wrapping a long line, please try to stick to the following principles:
    - Break after a comma;
    - Break before an operator;
    - Prefer "logical", or "semantic" breaks to "physical" breaks;
    - Align the wrapped part to have one more indent than the beginning of the line being wrapped:

```
int i;
do
{

  // Here comes a long line.
  bool isSucceeded = a + b + c + d + e – h * MAX_FACTOR / DEFAULT_SCALE
       - correctionFactor;
}
```

7. If there are no other conventions within your project team, long argument lists must also be wrapped in method declarations as well as in the calling code:

```
// The method declaration.
public float DoSomethingFromManyArguments(
  int argumentOne,
  int argumentTwo,
  decimal argumentThree,
  string argumentFour)
{
  // Method body goes here.
}

// The calling code.
float result = DoSomethingFromManyArguments(
 1,
 1,
 0.5m,
 "Some text");
```

8. If there are no other conventions within your project team, put each attribute declaration on a separate line:

```
[Serializable]
[Description("Defines a background color")]
public int BackgroundColor
{
 // Getter and setter blocks.
}
```

9. Do not declare several variables on a single line. Use separate lines for each variable declaration:

```
int    customerID;
string customerName;
float  salary;
```

10. It is recommended to similarly align related lines of code, especially the variable declarations and statements containing binary operators. For example:

```
Position _position;
string   _customerName;
float    _salary;
_customerName = "John Doe";
_salary       = DEFAULT_SALARY;
_salary     += SALARY_BONUS;
_position   =  Position.Supervisor;
```

Use tab stops to perform the alignment.

Naming Conventions

1. One must not use the Hungarian notation or any other type prefixes unless explicitly permitted by these guidelines;
2. Local variable names and argument names must follow the "camel casing" convention:

```
string customerName;

private int DoSomething(int firstArgument, float secondArgument);
```

3. Private class fields must follow "camel casing" convention with the first letter preceded by an underscore:

```
private string _customerName;
```

4. Public class fields must follow the "Pascal casing" convention:

```
public int CustomerID;
```

5. Property and method names must follow the "Pascal casing" convention:

```
public bool ValidateAmount();
public bool FileExists
{
  get;
}
```

6. Class, structure and enumeration names must follow the "Pascal casing" convention:

```
public class CustomerAccount
{
};

private enum WorkMode
{
  Add,
  Update;
}
```

7. Method names must follow the "Verb" + "Noun" pattern – for example, "UpdateAccount";
8. Use singular form, not plural in naming enumerations. In other words, "WorkMode" is a correct name and "WorkModes" is not;
9. Use meaningful names even for private variables, methods, properties, classes etc. Avoid using short names like "a", "b", "n" except for the commonly used loop indexers "i" and "j", primarily for innermost loops. If you abbreviate a name, make sure the abbreviation will be easily understood by other developers.

General Development Issues

Code Reuse Policy

The "Cut and Paste" programming is strictly prohibited. Use any appropriate object-oriented means instead. The most popular technique to use is the inheritance; you may consider extracting a class that provides shared functionality or other refactoring techniques as well. The following basic principles give some guidance for it:

1. If you feel that some functionality you need could have been already written by someone else, try to look for it within the project before implementing this functionality yourself. If you find something, but it does not completely fit your needs, contact either the developer who wrote this code, or the project leader. In most cases, existing functionality can be extended to support your particular requirements;
2. If you develop something that could potentially be reused, design an up-front for reusability. Do not hesitate to contact a competent person whenever you have any uncertainty in your design or implementation.

Coding Culture

1. Procedural programming and functional decomposition must be avoided as much as possible. If you are not sure how to express your idea in object-oriented terms, please consult a more experienced person;
2. It is obligatory to replace almost all numerical and string literals with symbolic constants. Exceptions are made either only for self-evident constants like 0 or 1,

or literals that "speak for themselves" in a context. Nevertheless, try to use named constants as much as possible and always give the constants meaningful names;

3. It is also recommended to group related constants into enumerations for integral types or non-creatable classes with public static constant members for non-integral types;
4. Share commonly used constants and enumerations as well as auxiliary classes through the System Frameworks projects;
5. It is required to follow the best OOD/OOP practices whenever possible. Make the maximum use of the design patterns (GoF, Craig Larman's GRASP) but apply them in an appropriate context as it is explained in "Applicability" section of the pattern description template;
6. Try to avoid long and complex methods. Split your code into several smaller and manageable routines. Use the refactoring techniques to make changes to your code whenever possible;
7. Always pay attention to compiler warnings. Ideally, your code must compile with no warnings at all or at least with a few warnings that cannot be easily avoided.

Commenting Conventions

1. The code must be readable even without rich comments. Convey your thoughts and ideas to the reviewer or the maintainer by expressiveness of the programming language itself;
2. Comments must be separated from the leading slashes with a single white space. The first letter of each sentence must be capital. Comments must end with a period:

```
// This is a sample comment. This is yet another comment sentence.
```

3. Comments must follow the overall line wrapping rules;
4. Comment methods, properties, classes, interfaces and all other supported language constructs must be in the XML comment format. This format facilitated by the IDE and the code editor can generate comment templates for you. In addition, such comments can be easily transformed to an HTML documentation set;
5. Specify XML documentation files for your projects;
6. While the code is being compiled, presence of a "Missing XML comment…" warning is highly unwanted;
7. Do not use in-line comments and C-style comments: `/* … */`
8. Comments must not just paraphrase what has already been expressed by the code being commented. Try to provide a higher-level, abstract explanation. For example, commenting the statement "i++" as "Increment i by one" is a bad practice whereas the comment "Update the number of customer accounts processed" is much better.

Regions

C# language provides means to create an additional level of the code structure that is region. Although one must try to keep classes relatively small and simple (and definitely

avoid "The Blob"-like classes[1]), if code complexity grows up, one must split the code into several regions, for example, "Data Binding", "Rendering" and "Validation".

Regions must organize the code by its purpose from the class user's point of view. Therefore, logical groups like the preceding ones are much better candidates for the regions than "Public Properties", "Private Variables", and so on. Nevertheless, this kind of grouping can also be acceptable if the corresponding code areas are quite large.

Windows Forms Specifics

1. Follow the Microsoft User Interface Guidelines if project requirements do not state otherwise.
2. If two controls tend to share the same descriptive name (for example, a label and a text box both facilitate entering a customer's name), it is allowed to specify the type of the control as a postfix. Therefore, in the preceding example the controls names will be "customerNameLabel" and "customerNameText" respectively;
3. Try to move all business logic code out of forms and controls. Use only calls to business facade or business entity methods in the form and control event handlers. Again, avoid making "The Blobs" user interface classes.
4. Use the "camel casing" for naming controls within a form or a composite control – "optionsHeading";
5. Use the "Pascal casing" for naming forms and controls if they are the actual classes – "ProductList";
6. Reuse UI as well as the source code itself. Split your interface into reusable controls; inherit forms and controls from a common base if appropriate.

Materials Used

Design Guidelines for Class Library Developers

[1] "The Blob" is a class in which the most of the program functionality is concentrated, while the rest of the classes are just data holders or perform merely auxiliary operations (see "The Blob" AntiPattern for more information).