2011

C# Coding Conventions & Good Practices

Most of the information in this document is compiled from the coding standards and best practices published in various articles in dotnetspider.com. Also, we referred to the guidelines published by Microsoft and various other sources.

Tan Tran

DS Team - R&D Dept. - Logigear

8/11/2011

CONTENTS

INTRODUCTION

Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a piece program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, etc. Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier. Coding conventions are only applicable to the human maintainers and peer reviewers of a software project.

Reducing the cost of software maintenance is the most often cited reason for following coding conventions. In their introduction to code conventions for the Java Programming Language, Sun Microsystems provides the following rationale:

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

(from wikipedia)

PURPOSE OF CODING CONVENTIONS AND GOOD PRACTICES

To develop reliable and maintainable applications, **you must follow coding conventions and best practices**.

The naming conventions, coding conventions and best practices described in this document are compiled from our own experience and by referring to various Microsoft and non Microsoft guidelines.

There are several standards exists in the programming industry. None of them are wrong or bad and you may follow any of them. What is more important is, selecting one standard approach and ensuring that everyone is following it.

NOTES IN THIS DOCUMENT

The terms <u>Pascal Casing</u> and <u>Camel Casing</u> are used throughout this document.

Pascal Casing - First character of all words are Upper Case and other characters are lower case.

Example: <u>B</u>ack<u>C</u>olor

Camel Casing - First character of all words, <u>except the first word</u> are Upper Case and other characters are lower case.

Example: <u>b</u>ack<u>C</u>olor

NAMING CONVENTIONS AND STANDARDS

1. Use Pascal casing for Class names

```csharp
public class HelloWorld
{
        ...
}
```

2. Use Pascal casing for Method names

```csharp
void SayHello(string name)
{
        ...
}
```

3. Use Camel casing for variables and method parameters

```csharp
int totalCount = 0;
void SayHello(string name)
{
        string fullMessage = "Hello " + name;
        ...
}
```

4. Use the prefix "I" with Camel Casing for interfaces (Example: IEntity)

5. Do not use Hungarian notation to name variables.
In earlier days most of the programmers liked it - having the data type as a prefix for the variable name and using m_ as prefix for member variables. Eg:

```csharp
string m_sName;
int nAge;
```

However, in .NET coding standards, this is not recommended. Usage of data type and m_ to represent member variables should not be used. All variables should use camel casing.

6. Use Meaningful, descriptive words to name variables. Do not use abbreviations.
   Good:

```csharp
string address
int salary
```

Not Good:

```
string nam
string addr
int sal
```

7. Do not use single character variable names like `i, n, s` etc. Use names like `index, temp`

One exception in this case would be variables used for iterations in loops:

```
for ( int i = 0; i < count; i++ )
{
      ...
}
```

If the variable is used only as a counter for iteration and is not used anywhere else in the loop, many people still like to use a single char variable (i) instead of inventing a different suitable name.

8. Do not use underscores (_) for local variable names.

9. Do not use variable names that resemble keywords.

10. Prefix `boolean` variables, properties and methods with "`is`", "`can`", "`has`"

Ex: `private bool isFinished`

11. Namespace names should follow the standard pattern

```
<company name>.<product name>.<top level module>.<bottom level module>
```

12. File name should match with class name.

For example, for the class HelloWorld, the file name should be helloworld.cs (or, helloworld.vb)

13. Use Pascal Case for file names.

INDENTATION AND SPACING

1. Use TAB for indentation. Do not use SPACES.

2. Comments should be in the same level as the code (use the same level of indentation).

Good:

```
// Format a message and display
string fullMessage = "Hello " + name;
DateTime currentTime = DateTime.Now;
string message = fullMessage + ", the time is : " +
currentTime.ToShortTimeString();
MessageBox.Show ( message );
```

Not Good:

```
// Format a message and display
    string fullMessage = "Hello " + name;
    DateTime currentTime = DateTime.Now;
    string message = fullMessage + ", the time is : " +
    currentTime.ToShortTimeString();
    MessageBox.Show ( message );
```

3. Curly braces ({}) should be in the same level as the code outside the braces.

```
if ( ... )
{
        // Do something
        // ...

        return false;
}
```

4. Use one blank line to separate logical groups of code.
Good:

```
bool SayHello ( string name )
{
        string fullMessage = "Hello " + name;
        DateTime currentTime = DateTime.Now;

        string message = fullMessage + ", the time is : "
        + currentTime.ToShortTimeString();
```

```csharp
            MessageBox.Show ( message );

            if ( ... )
            {
                    // Do something
                    // ...

                    return false;
            }

            return true;
    }
```
Not Good:

```csharp
    bool SayHello (string name)
    {
            string fullMessage = "Hello " + name;
            DateTime currentTime = DateTime.Now;
            string message = fullMessage + ", the time is : "
            + currentTime.ToShortTimeString();
            MessageBox.Show ( message );
            if ( ... )
            {
                    // Do something
                    // ...
                    return false;
            }
            return true;
    }
```

5. There should be one and only one single blank line between each method inside the class.

6. The curly braces should be on a separate line and not in the same line as `if, for` etc.
    Good:

```csharp
        if ( ... )
        {
                // Do something
        }
```
    Not Good:

```csharp
        if ( ... ) {
                // Do something
        }
```


7. Use `#region` to group related pieces of code together. If you use proper grouping using `#region`, the page should like this when all definitions are collapsed.

```
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

8. Keep private member variables, properties and methods in the top of the file and public members in the bottom.

GOOD PROGRAMMING PRACTICES

1. Avoid writing very long methods. (A method should typically have 1~50 lines of code. If a method has more than 50 lines of code, you must consider re factoring into separate methods)

2. Method name should tell what it does.

Good:

```csharp
void SavePhoneNumber ( string phoneNumber )
{
    // Save the phone number.
}
```

Not Good:

```csharp
// This method will save the phone number.
void SaveDetails ( string phoneNumber )
{
    // Save the phone number.
}
```

3. A method should do only 'one job'. Do not combine more than one job in a single method, even if those jobs are very small.

Good:

```csharp
// Save the address.
SaveAddress (  address );

// Send an email to the supervisor to inform that the address is
updated.
SendEmail ( address, email );

void SaveAddress ( string address )
{
    // Save the address.
    // ...
}

void SendEmail ( string address, string email )
{
    // Send an email to inform the supervisor that the address is
changed.
    // ...
}
```

Not Good:

```csharp
// Save address and send an email to the supervisor to inform that
// the address is updated.
SaveAddress ( address, email );

void SaveAddress ( string address, string email )
{
    // Job 1.
    // Save the address.
    // ...

    // Job 2.
    // Send an email to inform the supervisor that the address is changed.
    // ...
}
```

4. Use the c# or VB.NET specific types (aliases), rather than the types defined in System namespace.

```csharp
int age;     (not Int16)
string name;  (not String)
object contactInfo; (not Object)
```

5. Always watch for unexpected values. For example, if you are using a parameter with 2 possible values, never assume that if one is not matching then the only possibility is the other value.

Good:

```csharp
If ( memberType == eMemberTypes.Registered )
{
    // Registered user… do something…
}
else if ( memberType == eMemberTypes.Guest )
{
    // Guest user... do something…
}
else
{
    // Un expected user type. Throw an exception
    throw new Exception ("Un expected value " + memberType.ToString()
+ "'.")

    // If we introduce a new user type in future, we can easily find
    // the problem here.
}
```

Not Good:

```
If ( memberType == eMemberTypes.Registered )
{
    // Registered user… do something…
}
else
{
    // Guest user... do something…

    // If we introduce another user type in future, this code will
    // fail and will not be noticed.
}
```

6. Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code.

   However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed.

7. Do not hardcode strings. Use resource files.

8. Convert strings to lowercase or upper case before comparing. This will ensure the string will match even if the string being compared has a different case.

```
if ( name.ToLower() == "john" )
{
        //…
}
```

9. Use String.Empty instead of ""

Good:

```
if ( name == String.Empty )
{
    // do something
}
```

Not Good:

```
if ( name == "" )
{
    // do something
}
```

10. Avoid using member variables. Declare local variables wherever necessary and pass it to other methods instead of sharing a member variable between methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.

11. Use `enum` wherever required. Do not use numbers or strings to indicate discrete values.

Good:

```
enum MailType
{
        Html,
        PlainText,
        Attachment
}

void SendMail (string message, MailType mailType)
{
        switch ( mailType )
        {
                case MailType.Html:
                        // Do something
                        break;
                case MailType.PlainText:
                        // Do something
                        break;
                case MailType.Attachment:
                        // Do something
                        break;
                default:
                        // Do something
                        break;
        }
}
```

Not Good:

```
void SendMail (string message, string mailType)
{
        switch ( mailType )
        {
                case "Html":
                        // Do something
                        break;
                case "PlainText":
                        // Do something
                        break;
                case "Attachment":
                        // Do something
                        break;
                default:
```

```
                    // Do something
                    break;
            }
        }
    }
```

12. Do not make the member variables public or protected. Keep them private and expose public/protected Properties.

13. The event handler should not contain the code to perform the required action. Rather call another method from the event handler.

14. Do not programmatically click a button to execute the same action you have written in the button click event. Rather, call the same method which is called by the button click event handler.

15. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.

16. Never assume that your code will run from drive "C:". You may never know, some users may run it from network or from a "Z:".

17. In the application start up, do some kind of "self check" and ensure all required files and dependencies are available in the expected locations. Check for database connection in start up, if required. Give a friendly message to the user in case of any problems.

18. If the required configuration file is not found, application should be able to create one with default values.

19. If a wrong value found in the configuration file, application should throw an error or give a message and also should tell the user what are the correct values.

20. Error messages should help the user to solve the problem. Never give error messages like "Error in Application", "There is an error" etc. Instead give specific messages like "Failed to update database. Please make sure the login id and password are correct."

21. When displaying error messages, in addition to telling what is wrong, the message should also tell what should the user do to solve the problem. Instead of message like "Failed to update database.", suggest what should the user do: "Failed to update database. Please make sure the login id and password are correct."

22. Show short and friendly message to the user. But log the actual error with all possible information. This will help a lot in diagnosing problems.

23. Do not have more than one class in a single file.

24. Have your own templates for each of the file types in Visual Studio. You can include your company name, copy right information etc in the template. You can view or edit the Visual Studio file templates in the folder `C:\Program Files\Microsoft Visual Studio 8\Common7\IDE\ItemTemplatesCache\CSharp\1033` (in Visual Studio.NET 2008). (This folder has the templates for C#, but you can easily find the corresponding folders or any other language)

25. Avoid having very large files. If a single file has more than 1000 lines of code, it is a good candidate for refactoring. Split them logically into two or more classes.

26. Avoid public methods and properties, unless they really need to be accessed from outside the class. Use "internal" if they are accessed only within the same assembly.

27. Avoid passing too many parameters to a method. If you have more than 7 parameters, it is a good candidate to define a class or structure.

28. If you have a method returning a collection, return an empty collection instead of null, if you have no data to return. For example, if you have a method returning an ArrayList, always return a valid ArrayList. If you have no items to return, then return a valid ArrayList with 0 items. This will make it easy for the calling application to just check for the "count" rather than doing an additional check for "null".

29. Use the AssemblyInfo file to fill information like version number, description, company name, copyright notice etc.

30. If you are opening database connections, sockets, file stream etc, always close them in the `finally` block. This will ensure that even if an exception occurs after opening the connection, it will be safely closed in the `finally` block.

31. Declare variables as close as possible to where it is first used. Use one variable declaration per line.

32. Use StringBuilder class instead of String when you have to manipulate string objects in a loop. The String object works in weird way in .NET. Each time you append a string, it is

actually discarding the old string object and recreating a new object, which is a relatively expensive operations.

Consider the following example:

```csharp
public string ComposeMessage (string[] lines)
{
    string message = String.Empty;

    for (int i = 0; i < lines.Length; i++)
    {
        message += lines [i];
    }

    return message;
}
```

In the above example, it may look like we are just appending to the string object 'message'. But what is happening in reality is, the string object is discarded in each iteration and recreated and appending the line to it.

If your loop has several iterations, then it is a good idea to use StringBuilder class instead of String object.

See the example where the String object is replaced with StringBuilder.

```csharp
public string ComposeMessage (string[] lines)
{
    StringBuilder message = new StringBuilder();

    for (int i = 0; i < lines.Length; i++)
    {
        message.Append( lines[i] );
    }

    return message.ToString();
}
```

COMMENTS

Good and meaningful comments make code more maintainable. However,

1. Do not write comments for every line of code and every variable declared.

2. Use `//` or `///` for comments. Avoid using `/* … */ or "flowerbox" comment blocks`

    ```
    // ***************************************
    // flowerbox comment block
    // ***************************************
    ```

3. Write comments wherever required. But good readable code will require very less comments. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.

4. Do not write comments if the code is easily understandable without comment. The drawback of having lot of comments is, if you change the code and forget to change the comment, it will lead to more confusion.

5. Fewer lines of comments will make the code more elegant. But if the code is not clean/readable and there are less comments, that is worse.

6. If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.

7. If you initialize a numeric variable to a special number other than 0, -1 etc, document the reason for choosing that value.

8. Perform spelling check on comments and also make sure proper grammar and punctuation is used.

EXCEPTION HANDLING

1. Never do a 'catch exception and do nothing'. If you hide an exception, you will never know if the exception happened or not. Lot of developers uses this handy method to ignore non significant errors. You should always try to avoid exceptions by checking all the error conditions programmatically. In any case, catching an exception and doing nothing is not allowed. In the worst case, you should log the exception and proceed.

2. In case of exceptions, give a friendly message to the user, but log the actual error with all possible details about the error, including the time it occurred, method and class name etc.

3. Always catch only the specific exception, not generic exception.

Good:

```csharp
void ReadFromFile ( string fileName )
{
    try
    {
        // read from file.
    }
    catch (FileIOException ex)
    {
        // log error.
        //  re-throw exception depending on your case.
        throw;
    }
}
```

Not Good:

```csharp
void ReadFromFile ( string fileName )
{
    try
    {
        // read from file.
    }
    catch (Exception ex)
    {
        // Catching general exception is bad... we will never know
whether
        // it was a file error or some other error.
        // Here you are hiding an exception.
        // In this case no one will ever know that an exception happened.

        return "";
    }
}
```

---

4. No need to catch the general exception in all your methods. Leave it open and let the application crash. This will help you find most of the errors during development cycle. You can have an application level (thread level) error handler where you can handle all general exceptions. In case of an 'unexpected general error', this error handler should catch the exception and should log the error in addition to giving a friendly message to the user before closing the application, or allowing the user to 'ignore and proceed'.

5. When you re throw an exception, use the `throw` statement without specifying the original exception. This way, the original call stack is preserved.
Good:

```csharp
catch
{
        // do whatever you want to handle the exception

        throw;
}
```

Not Good:

```csharp
catch (Exception ex)
{
        // do whatever you want to handle the exception

        throw ex;
}
```

6. Do not write try-catch in all your methods. Use it only if there is a possibility that a specific exception may occur and it cannot be prevented by any other means.

   For example, if you want to insert a record if it does not already exists in database, you should try to select record using the key. Some developers try to insert a record without checking if it already exists. If an exception occurs, they will assume that the record already exists. This is strictly not allowed. You should always explicitly check for errors rather than waiting for exceptions to occur. On the other hand, you should always use exception handlers while you communicate with external systems like network, hardware devices etc. Such systems are subject to failure anytime and error checking is not usually reliable. In those cases, you should use exception handlers and try to recover from error.

7. Do not write very large try-catch blocks. If required, write separate try-catch for each task you perform and enclose only the specific piece of code inside the try-catch. This will help you find which piece of code generated the exception and you can give specific error message to the user.

8. Write your own custom exception classes if required in your application. Do not derive your custom exceptions from the base class SystemException. Instead, inherit from ApplicationException.