

Ascertain.org

C# CODING STANDARD

Presented By: ming@Ascertain.org

Date: August 5, 2003

TABLE OF CONTENT

0.	概述.....	3
1.	針對的讀者.....	3
1.	程式碼命名規範.....	4
1.	名詞解釋.....	4
2.	快速入門.....	4
3.	Namespace 命名規範.....	5
4.	Class 命名規範.....	5
5.	Interface 命名規範.....	6
6.	Enumeration Type 命名規範.....	6
7.	Static Field 命名規範.....	6
8.	Parameter/ Attribute 命名規範.....	7
9.	Method 命名規範.....	7
10.	Property 命名規範.....	7
11.	Event 命名規範.....	8
12.	不要使用匈牙利方法來命名變數.....	8
13.	用有意義的，描述性的詞語來命名.....	8
2.	慣用編碼守則.....	10
1.	文件名要和類名匹配.....	10
2.	不要在每個運算符和括號的前後空格.....	11
3.	不要任意在括弧前後使用空格.....	11
4.	區段括弧不要分行.....	11
5.	減少無謂的分行及括弧.....	12
6.	良好的編碼習慣.....	12
7.	宣告 (Declaration).....	16
8.	Access Level.....	16
9.	註解.....	16
10.	異常處理 (Exception handling).....	16

0. 概述

誰都會寫程式碼！幾個月的編程經驗可以讓你寫出「可運行的應用程式」。讓它可運行容易，但以最有效率的方式編碼、能被小組成員承接的程式碼就需要下更多的功夫！

要知道，大多數程式設計師在寫「可運程式碼」，而不是「高效率程式碼」。寫高效程式碼是一項藝術，必須靠紀律及學習來實踐。建立一個高效率、有默契的開發團隊更需有一定的規範及標準流程。

想要讓開發出來的程式碼穩固且容易維護，除了有必要適當地採用設計模式（Design Patterns）之外，也有必要採用編碼指導方針（Coding Guidelines），做為寫程式時依循的準則。設計模式是在設計階段進行的，主要由系統設計者（System Designer）負責，所以和程式員的關係比較不是那麼密切。至於編碼方針，就和程式員的關係相當密切了。對C#來說，最重要的編程方針是美國微軟MSDN網站上的〈[Design Guidelines for Class Library Developers](#)〉。

雖然我們利用[reSharper](#)來自動化大部分的工作。閱讀本文件可避免您知其然不知所以然的窘境。

1. 針對的讀者

本文主要針對開發團隊的成員以及專案的管理團隊。本文假設讀者已有了一定的軟體開發專業基礎（例如，C#或SQL Server）。另外本文也適用對程式碼審查感興趣的測試人員。

1. 命名規範

程式碼命名及格式規範的目的為建立開發團隊的默契以及效率。檢視一篇與自己撰寫風格相同的程式碼不但能減少無謂的猜測，也能快速瞭解程式碼，立即進入程式撰寫的狀況。

以下章節介紹開發團隊使用的 c# 程式碼編寫標準。

1. 名詞解釋

Pascal case：大小寫形式－所有單詞第一個字母大寫，其他字母小寫。

Camel case：大小寫形式－除了第一個單詞，所有單詞第一個字母大寫，其他字母小寫。

2. 快速入門

以下為完全合乎格式的程式片段：

```
namespace CyberTutor.Kids.Foo {
    /// <summary>Foo is this and that.</summary>
    /// <remark>this is an example code</remark>
    /// <example><code>MyFoo = new MyFoo();</code></example>
    public class MyFoo : FooMaster {
        protected MyObject _myProtectedVariable;
        MyObject _myPrivateVariable; //No need to put private key

        /// <summary>
        /// Asserts that a condition is true. If it isn't it throws
        /// an <see cref="FooError"/>.
        /// </summary>
        /// <param name="message">The message to foo </param>
        /// <param name="condition">The Boolean condition to foo </param>
        public static void FooMethod(string message, bool condition) {
            if (!condition)
                Assertion.Fail(message);

            try
                (this) {
                    DoThis();
                    DoOneMoreThis();
                }
            catch(that) {
                throw new FooError(that);
            }
        }
    }
}
```

3. Namespace 命名規範

使用 {公司名}.{產品名} 的格式。

Namespace 中類的依賴關係應該體現在命名上，比如 System.Web.UI.Design 中的類以來於 System.Web.UI

1. 使用 Pascal case 命名
2. 當商標（產品名）的命名風格和 Pascal case 不相符的時候，以商標（產品名）為準
3. 在語意合適的情況下使用複數，比如 System.Collections。例外是縮寫和商標的情況。
4. Namespace 的名字不一定和 Assembly 一一對應。

好：

```
Namespace CyberTutor.Kids.MyNameSpace
```

不好：

```
Namespace KIDS.Mynamespace
```

4. Class 命名規範

1. 使用名詞或者名詞性詞組命名 class
2. 使用 Pascal case
3. 保守地使用縮寫
4. 不使用 type 前綴，例如 C 來標識 Class。比如，使用 FileStream 而不是 CFileStream。
5. 不使用下劃線
6. 偶爾的在 Class 名稱組成中需要使用 I 開頭的時候，比如 IdentityStore，just use it。
7. 在合適的時候，使用單詞復合來標識從某個基類繼承而來。比如 xxxException。

好：

```
public class HelloWorld {  
}
```

不好：

```
public class cHelloworld {  
}
```

5. Interface 命名規範

1. 使用名詞或者名詞性詞組命名 Interface
2. 使用 Pascal case
3. 保守地使用縮寫
4. 在 interface 名稱前加上字母 I 來表示 type 是 interface。
5. 實作 interface 時，用類似的名字來命名它們
6. 不要使用下底線

好：

```
public interface IMyObject {  
}
```

不好：

```
public interface Myobject {  
}
```

6. Enumeration Type 命名規範

1. 對 Enum 類型和值使用 Pascal case
2. 保守地使用縮寫
3. 不要在 Enum 類型名稱後面加上 Enum 後綴
4. 對於大多數 Enum 類型使用單數名稱，僅僅在這個 Enum 類型是 bit fields 地時候使用複數形式
5. 總是給 bit field Enum 類型添加 FlagsAttribute。

7. Static Field 命名規範

1. 使用名詞、名詞性詞組或名詞縮寫來命名 static fields
2. 使用 Pascal case
3. 不要在 static field 名稱中使用匈牙利命名法
4. 在任何可能的情況下使用 private static properties 而不是 public static fields。

8. Parameter/ Attribute 命名規範

1. 對於 parameter 名稱使用 camel case
2. 使用描述性的名稱。參數名字應該在大多數場合下它的名字加上類型足夠描述它的意義。
3. 使用描述參數的意義的名字而不是描述參數類型的名字。開發工具應該提供有關參數類型的有意義的信息。因而，參數的名字可以用於更好的描述意義。保守地使用基於類型的參數名字，僅僅在它們是合適的場合下使用。
4. 不要使用保留的參數
5. 不要使用匈牙利命名法

9. Method 命名規範

1. 使用動詞或者動詞性詞組命名
2. 使用 Pascal case

```
public class HelloWorld {
    void SayHello(string name) {
    }
}
```

10. Property 命名規範

1. 使用名詞或者名詞性詞組命名
2. 使用 Pascal case
3. 不要使用匈牙利命名法
4. 所有 private property 需使用下底線；如此我們可省略 private keyword

好：

```
public string name;
public int age;
int _age;
```

不好：

```
public string m_sName;
public int nAge;
public int _age;
private int _age;
```

1.1. Event 命名規範

1. 使用 Pascal case
2. 不要使用匈牙利命名法
3. 在 event handler 名字中使用 EventHandler 後綴

指定兩個名字分別為 sender 和 e 的參數。sender 參數代表了發出事件的對象。sender 參數總是類型 object，即使可能使用一個更加精確的類型。和事件相關的狀態封裝在名字為 e 的 event class 的實體之中。給 e 指定恰當而且明確的 event class。

1. 使用 EventArgs 後綴命名事件參數 class
2. 考慮使用動詞命名事件。使用進行時態來標識事件正在進行之中，使用完成時態標識事件已經完成，不要使用 BeforeXxx/AfterXxx 命名法。
3. 不要在事件聲明中使用前綴和後綴，比如，用 Close 而不是 OnClose。
4. 一般的，你應該同時提供一個名字為 OnXxx 的 protected method 供派生類來改寫。

1.2. 不要使用匈牙利方法來命名變數

VB 程式員喜歡把類型作為變量名的前綴如 m_。

不好：

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

然而，這種方式在 .NET 編碼規範中是不推薦的。所有變量都用 camel 大小寫形式，而不是用數據類型和 m_ 來作前綴。

好：

```
string name;  
int age;
```

1.3. 文件命名

對於 HelloWorld class, 相對應的文件名應為 HelloWorld.cs (或, helloworld.vb)。

1.4. 用有意義的，描述性的詞語來命名

1. 別用縮寫。用 name, address, salary 等代替 nam, addr, sal
2. 別使用單個字母的變量象 i, n, x 等. 使用 index, temp 等
3. 用於循環迭代的變量例外：

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

如果變量只用於迭代計數，沒有在循環的其他地方出現，許多人還是喜歡用單個字母的變量(i)，而不是另外取名。

1. 變量名中不使用下底線 (_)。
2. 命名空間需按照標準的模式命名

2. 慣用編碼守則

1. 編排基本原則

- ◆ 縮排用 TAB，不用 space
- ◆ 註釋需和程式碼對齊
- ◆ 括弧 ({}) 需和括號外的程式碼對齊
- ◆ 用空行來分開程式碼的邏輯分組
- ◆ 不要濫用空格

不好：

```
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;
}
```

這段程式碼看起來比上面的好：

```
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;
}
```

在一個類中，各個方法需用一空行，也只能是一行分開。括弧跟括號在同一行。

好：

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

不好：

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

2. 不要在每個運算符和括號的前後空格

好：

```
if (showResult==true) {
    for (int i= 0;i<10;i++) {
        //
    }
}
```

不好：

```
if ( showResult == true )
{
    for ( int i = 0; i < 10; i++ )
    {
        //
    }
}
```

3. 不要任意在括弧前後使用空格

Do not put a space after the opening parenthesis and the closing one

好：

```
method(a);
array[10];
```

不好：

```
method ( a ); array[ 10 ];
```

4. 區段括弧不要分行

Inside a code block, put the opening brace on the same line as the statement

好：

```
if (a) {
    code();
    code();
}
```

不好：

```
if (a)
```

```
{  
    code();  
    code();  
}
```

5. 減少無謂的分行及括弧

Avoid using unnecessary open/close braces, vertical space is usually limited

好：

```
if (a)  
    code();
```

```
if (a) code();
```

不好：

```
if (a) {  
    code ();  
}
```

6. 良好的編碼習慣

遵從以下良好的習慣以寫出可讀性高的程式碼

1. 避免使用大文件。如果一個文件裡的程式碼超過 300~400 行，必須考慮將程式碼分開到不同類中。
2. 避免寫太長的方法。一個典型的方法程式碼在 1~25 行之間。如果一個方法發程式碼超過 25 行，應該考慮將其分解為不同的方法。
3. 方法名需能看出它作什麼。別使用會引起誤解的名字。如果名字一目瞭然，就無需用文檔來解釋方法的功能了。

好：

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

不好：

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

6.1 一個方法(method)只完成一個任務。不要把多個任務組合到一個方法中，即使那些任務非常小。

好：

```
// 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.
}
```

不好：

```
// 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.
    // ...
}
```

6.2 使用C#的特有類型，而不是System命名空間中定義的別名類型

好：

```
int age;
string name;
object contactInfo;
```

不好：

```
Int16 age;
String name;
Object contactInfo;
```

6.3 必要時使用enum。別用數字或字符串來指示離散值

好：

```
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;
    }
}

```

不好：

```

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;
    }
}

```

6.4 Switch statements have the case at the same indentation as the switch

Argument names should use the camel casing for identifiers

Empty methods : They should have the body of code using two lines, in consistency with the rest

6.5 Line length: The line length for C# source code is 134 columns

If your function declaration arguments go beyond this point, please align your arguments to match the opening brace, like this :

```
void Function (int veryveryverylongarg, veryveryverylongObject argb,
              int argc) {
    DoThis();
    DoThat();
}
```

When invoking functions, the rule is different, the arguments are not aligned with the previous argument, and instead they begin at the tabbed position, like this :

```
void M() {
    MethodCall ("Very long string that will force",
               "Next argument on the 8-tab pos",
               "Just like this one")
}
```

Here are a couple of examples :

```
class X : Y {
    bool Method(int argument_1, int argument_2) {
        if (argument_1 == argument_2)
            throw new Exception (Locale.GetText ("They are equal!"));

        if (argument_1 < argument_2) {
            if (argument_1 * 3 > 4)
                return true;
            else
                return false;
        }

        //
        // This sample helps keep your sanity while using 4-spaces for tabs
        //
        VeryLongIdentifierWhichTakesManyArguments (
            Argument1, Argument2, Argument3,
            NestedCallHere (
                MoreNested));
    }

    bool MyProperty {
        get {return x; }
        set {x = value; }
    }

    void AnotherMethod() {
        if ((a + 5) != 4) {
        }
    }
}
```

```

while (blah) {
    if (a)
        continue;
    b++;
}
}

```

7. 宣告 (Declaration)

別在程式中使用固定數值，用常量代替。

別用字符串常數。用資源文件。

避免使用很多成員變量。聲明局部變量，並傳遞給方法。不要在方法間共享成員變量。如果在幾個方法間共享一個成員變量，那就很難知道是哪個方法在什麼時候修改了它的值。

8. Access Level

別把成員變量聲明為 `public` 或 `protected`。都聲明為 `private` 而使用 `public/protected` 的 Properties.

不在程式碼中使用具體的路徑和驅動器名。使用相對路徑，並使路徑可編程。

永遠別設想你的程式碼是在「C:」盤運行。你不會知道，一些使用者在網絡或「Z:」盤運程式。

應用程式啟動時作些「自檢」並確保所需文件和附件在指定的位置。必要時檢查數據庫連接。出現任何問題給使用者一個友好的提示。

如果需要的配置文件找不到，應用程式需能自己創建使用默認值的一份。

9. 註解

簡言之，可讀性高的程式碼無須太多註解就能理解。以下列出寫註解的基本原則：

1. 只在需要的地方註釋。可讀性強的程式碼不需要太多的註解。如果所有的變量和方法的命名都很有意義，會使程式碼可讀性很高，因此註解並無意義。
2. 如果應為某種原因使用了複雜艱澀的原理及程式碼，為程式碼配置良好的文件檔和註解。
3. 不需要每行程式碼，每個聲明的變量都做註釋。
4. 對一個數值變量採用不是 0, -1 等的數值初始化，給出選擇該值的理由。
5. 行數不多的註解會使程式碼看起來優雅。但如果程式碼不清晰，可讀性差，那就糟糕。
6. If you are modifying someone else's code, and your contribution is significant, please add yourself to the Authors list.

10. 異常處理 (Exception handling)

不要捕捉了異常卻什麼也不做。如果隱藏了一個異常，你將永遠不知道異常到底發生了沒有。

發生異常時，要精確記錄錯誤的所有可能細節，包括發生的時間，和相關方法，類名等。

在 `business logic layer` 或 `data layer` 發生錯誤時，必要拋出錯誤，給出正確的相關資訊給 `caller`。

在 `presentation layer` 發生錯誤時，必須要捕捉並精確記錄錯誤的所有可能細節，然後顯示有用的錯誤訊息給使用者。所謂「有用的錯誤訊息」是指訊息能提示使用者如何解決問題。永遠別用像「應用程式出錯」；「發現一個錯誤」等

無意義，沒有幫助的訊息。而應給出像「請確認登錄帳號和密碼正確」的具體資訊。
 顯示給使用者的訊息要簡短而友好。但要把所有可能的細節都記錄下來，以助診斷問題。
 顯示錯誤訊息時，除了說哪裡錯了，還應提示使用者如何解決問題。

只捕捉特定的異常，而不是一般的異常。

好：

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

不好：

```
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 "";
    }
}
```

不必在所有方法中捕捉一般異常。讓程式擲出錯誤。這將幫助在開發週期發現大多數的錯誤。

你可以用應用程式級（線程級）錯誤處理器處理所有一般的異常。遇到以外的一般性錯誤時，此錯誤處理器應該捕捉異常，給使用者提示消息，在應用程式關閉或使用者選擇忽略並繼續之前記錄錯誤信息。

不必每個方法都用 `try-catch`。當特定的異常可能發生時才使用。比如，當你寫文件時，處理異常的 `FileNotFoundException`。

別寫太大的 `try-catch` 模塊。如果需要，為每個執行的任務編寫單獨的 `try-catch` 模塊。這將幫你找出哪一段程式碼產生異常，並給使用者發出特定的錯誤消息。

如果應用程式需要，可以編寫自己的 `exception class`。自定異常不應從基類，如 `SystemException` 衍生，而要繼承於 `IApplicationException`。