Naming Conventions and Standards

Development using ASP.Net

Release: 2.0

Table Of Contents

1. Naming Conventions and Standards

> Note :
>
> The terms Pascal Casing and Camel Casing are used throughout this document.
>
> **Pascal Casing** - First character of all words are Upper Case and other characters are lower case.
>
> Example: BackColor
>
> **Camel Casing -** First character of all words, except the first word are Upper Case and other characters are lower case.
>
> Example: backColor

1. Use Pascal casing for Class names

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

2. Use Pascal casing for Method names

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

3. Use Camel casing for variables and method parameters

```
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:

```
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.

Some programmers still prefer to use the prefix **m_** to represent member variables, since there is no other easy way to identify a member variable.

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

    Good:

    ```
    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. All member variables must be prefixed with underscore (_) so that they can be identified from other local variables.

    ```
    Private _pageNumber As Integer

    Public Property PageNumber() As Integer

        Get
            Return _pageNumber
        End Get

        Set(ByVal value As Integer)
            _pageNumber = value
        End Set

    End Property
    ```

10. Do not use variable names that resemble keywords.

    ```
    Like in vb.net  "const" is a keyword
    Dim const as String  'This will throw an error
    ```

11. Prefix `boolean` variables, properties and methods with "`is`" or similar prefixes.

Ex: `private bool _isFinished`

12. Namespace names should follow the standard pattern

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

Example:
        Controller.Fashion.LinkBuilder
        Controller.Supplier.LinkBuilder
```

13. Use appropriate prefix for the UI elements so that you can identify them from the rest of the variables.

There are 2 different approaches recommended here.

a. Use a common prefix (ui_) for all UI elements. This will help you group all of the UI elements together and easy to access all of them from the intellisense.

b. Use appropriate prefix for each of the ui element. A brief list is given below. Since .NET has given several controls, you may have to arrive at a complete list of standard prefixes for each of the controls (including third party controls) you are using.

Control	Prefix
Label	lbl
TextBox	txt
DataGrid	dtg
Button	btn
ImageButton	imb
Hyperlink	hlk
DropDownList	ddl
ListBox	lst
DataList	dtl
Repeater	rep
Checkbox	chk
CheckBoxList	cbl

RadioButton	rdo
RadioButtonList	rbl
Image	img
Panel	pnl
PlaceHolder	phd
Table	tbl
Validators	val

14. File name should match with class name.

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

15. Use Pascal Case for file names.

2. Indentation and Spacing

1. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.

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();

        MessageBox.Show ( message );

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

                return false;
        }
```

```
        return true;
    }
```

Not Good:

```
    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:

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

Not Good:

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

7. Use a single space before and after each operator and brackets.

Good:

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

Not Good:

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

8. 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.

```csharp
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

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

3. Good Programming practices

1. Avoid writing very long methods. A method should typically have 1~25 lines of code. If a method has more than 25 lines of code, you must consider re factoring into separate methods.

2. Method name should tell what it does. Do not use mis-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

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

Not Good:
```
// 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:
```
// 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:
```
// 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.

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

> Some developers prefer to use types in Common Type System than language specific aliases.

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:

```
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. 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" )
{
          //…
}
```

8. Use String.Empty instead of ""

Good:

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

Not Good:

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

9. 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.

10. 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;
        }
}
```

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

```
Public int UserID {get; set;}
```

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

Not Good:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Load

        //Do not write anything here ...

End Sub
```

Good:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs)
Handles Me.Load

    BindSupplierDetails()

    SetupPanel()

    BindSupplierImages()

End Sub
```

13. 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.

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

Not Good:

```
   Dim imagePath as String ="D:\\Code\\Site\\Branches\\FashionAdminPhase-
2\Images"
```

Good:

```
   Dim imagePath as String ="/images/"
```

15. 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:".

16. In the application start up, do some kind of "self check" and ensure all required files and dependancies 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.

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

18. 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.

19. 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."

20. 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."

21. 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.

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

23. 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.

24. 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.

25. Avoid passing too many parameters to a method. If you have more than 4~5 parameters, it is a good candidate to define a class or structure.

Not Good:

```
   Public Function InsertRealWedding(ByVal brideName As String,ByVal grommName
As String,ByVal weddinDate As Date, etc.) As String

End Function
```

Good:

```
Public Function InsertRealWedding(ByVal realWedding As
Model.RealWedding.RealWeddings) As String

End Function
```

26. If you have a method returning a collection, return a null instead of empty collection, if you have no data to return. For example, if you have a method returning an ArrayList, always return a null if you have no items to return.

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

```
 System.Reflection.Assembly.GetExecutingAssembly.GetName.Name()

lblCompany.Text = "Company Name: " & Application.CompanyName.Trim
lblProduct.Text = "Product Name: " & Application.ProductName.Trim
lblVersion.Text = "Version " & Application.ProductVersion.Trim
```

16. Make sure you have a good logging class which can be configured to log errors, warning or traces. If you configure to log errors, it should only log errors. But if you configure to log traces, it should record all (errors, warnings and trace). Your log class should be written such a way that in future you can change it easily to log to Windows Event Log, SQL Server, or Email to administrator or to a File etc without any change in any other part of the application. Use the log class extensively throughout the code to record errors, warning and even trace messages that can help you trouble shoot a problem.

17. 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.

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

19. 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:

```
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.

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

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

    return message.ToString();
}
```

4. Architecture

1. Always use multi layer (N-Tier) architecture.

2. Never access database from the UI pages. Always have a data layer class which performs all the database related tasks. This will help you support or migrate to another database back end easily.

3. Use try-catch in your data layer to catch all database exceptions. This exception handler should record all exceptions from the database. The details recorded should include the name of the command being executed, stored proc name, parameters, connection string used etc. After recording the exception, it could be re thrown so that another layer in the application can catch it and take appropriate action.

4. Separate your application into multiple assemblies. Group all independent utility classes into a separate class library. All your database related files can be in another class library.

5. ASP.NET

1. Do not use session variables throughout the code. Use session variables only within the classes and expose methods to access the value stored in the session variables. A class can access the session using `System.Web.HttpCOntext.Current.Session`

2. Do not store large objects in session. Storing large objects in session may consume lot of server memory depending on the number of users.

3. Always use style sheet to control the look and feel of the pages. Never specify font name and font size in any of the pages. Use appropriate style class. This will help you to change the UI of your application easily in future. Also, if you like to support customizing the UI for each customer, it is just a matter of developing another style sheet for them

6. Comments

Good and meaningful comments make code more maintainable. However,

1. Use GhostDoc which is a Visual Studio Plug-in for adding comments.

2. Each file should have the following comments at the beginning of the file

```
//<File name>
//
//<Brief Description about the file>
//
//@author <Your Name>
//@copyright Copyright (c) <year>, <Project Name>
//@version <version number> Created On <created date> Modified On
<modified date>
//
```

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

4. Use **//** or **///** for comments. Avoid using **/*** … ***/**

5. 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.

6. 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.

7. 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.

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

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

10. The bottom line is, write clean, readable code such a way that it doesn't need any comments to understand.

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

7. 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:

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

Not Good:

```
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:

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

            throw;
    }
    ```

    Not Good:

    ```
    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.

8. Code Sample

A sample ASP.Net project is available with "SampleProject.rar".
Code snippet to perform a database operation in web application:

Web Service method:

```
/// <summary>
/// Gets the employee by ID.
/// </summary>
/// <param name="id">The id.</param>
/// <returns></returns>
public Model.Admin.Employee GetEmployeeByID(int id)
{
    Employee _employee = new Employee();

    return _employee.GetEmployeeByID(id);

}
```

Controller method:

```
/// <summary>
/// Gets the employee by ID.
/// </summary>
/// <param name="id">The id.</param>
/// <returns></returns>
public Model.Admin.Employee GetEmployeeByID(int id)
{
    List<SqlParameter> employeeParameters = new List<SqlParameter>();

    employeeParameters.Add(new SqlParameter("@id", id));

    DataTableReader reader =
DBOPS.SqlHelper.ExecuteTableReader(GlobalValues.ConnectionString,
CommandType.StoredProcedure, "GetEmployeeByID", employeeParameters.ToArray());

    if (reader.Read())
    {
    return BuildEmployeeItem(reader);
    }

    //if record does not exist then return a blank object of employee  or
    else you can return a null object.
    return new Model.Admin.Employee();

}


/// <summary>
/// Builds the employee item.
/// </summary>
/// <param name="reader">The reader.</param>
/// <returns></returns>
private Model.Admin.Employee BuildEmployeeItem(DataTableReader reader)
{
    Model.Admin.Employee _employeeItem = new Model.Admin.Employee();
```

```
    _employeeItem.ID = Convert.ToInt32(reader["id"]);
    _employeeItem.Name = Convert.ToString(reader["Name"]);
    _employeeItem.Department = Convert.ToString(reader["Department"]);
    _employeeItem.DateOfBirth = Convert.ToDateTime(reader["DateOfBirth"]);

    return _employeeItem;

  }
```

Calling Web service method from page:

```
/// <summary>
/// Gets the employee by ID.
/// </summary>
/// <param name="id">The id.</param>
/// <returns></returns>
private Employee GetEmployeeByID(int id)
{
    WSEmployee _wsEmployee = new WSEmployee();
    return _wsEmployee.GetEmployeeByID(id);
}


/// <summary>
/// Handles the Click event of the btnSearch control.
/// </summary>
/// <param name="sender">The source of the event.</param>
/// <param name="e">The <see cref="System.EventArgs"/> instance containing
the event data.</param>
protected void btnSearch_Click(object sender, EventArgs e)
{
    Employee _employee = GetEmployeeByID(1);
}
```