EGOVERNMENT AUTHORITY

Standards

[C# .NET Coding Standards]

[19/06/2009]

Table of Contents

1.0 Introduction

Anybody can write code. With a few months of programming experience, you can write 'working applications'. Making it work is easy, but doing it the right way requires more work, than just making it work.

Believe it, majority of the programmers write 'working code', but not ‘good code'. Writing 'good code' is an art and you must learn and practice it.

Following are the characteristics of good code.

- Reliable
- Maintainable
- Efficient

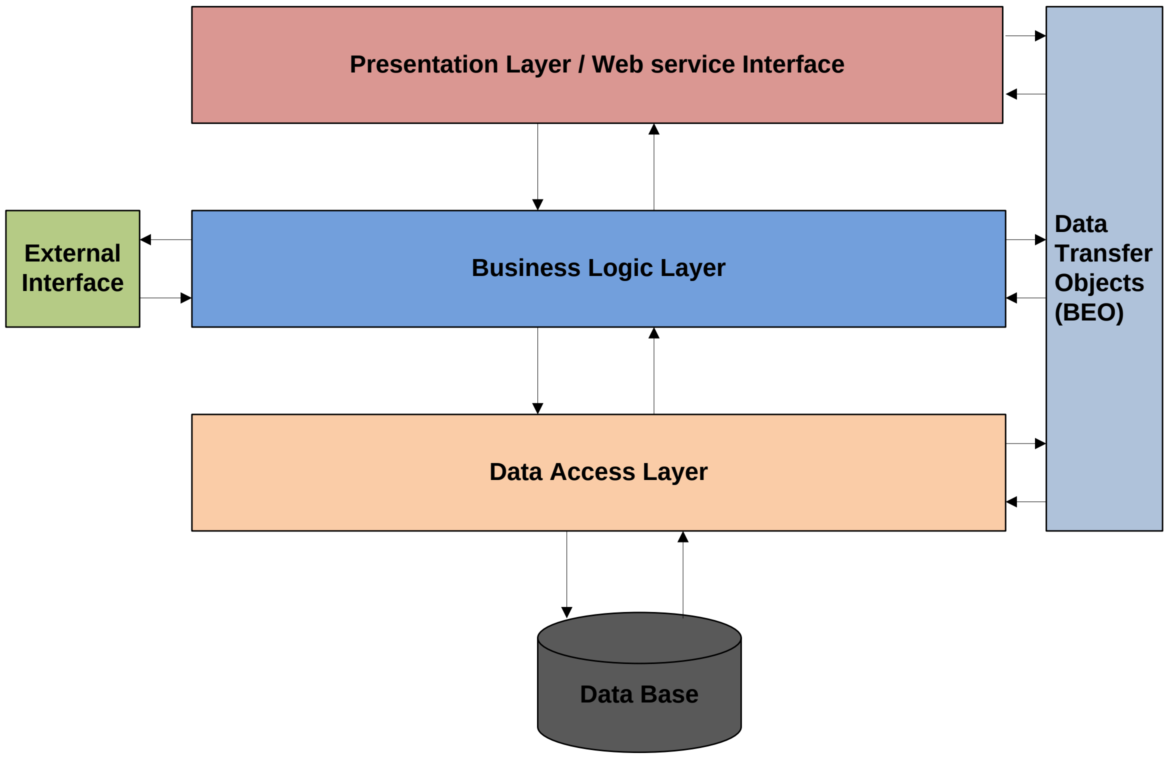
Most of the developers are inclined towards writing code for higher performance, compromising reliability and maintainability. But considering the long term ROI (Return On Investment), efficiency and performance comes below reliability and maintainability. If your code is not reliable and maintainable, you (company) will be spending lot of time to identify issues, trying to understand code etc throughout the life of your application.

2.0 Purpose of coding standards and best practices

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

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

3.0 The Logical Building Blocks


Presentation Layer / Web service Interface
External Interface
Business Logic Layer
Data Transfer Objects (BEO)
Data Access Layer
Data Base

The above diagram describes the logical building blocks of the Application.

1) The Presentation Layer/ Web service Layer : Also called as the client layer comprises of components that are dedicated to presenting the data to the user. For example: Windows/Web Forms or Web service interface

2) The Business Rules Layer: This layer encapsulates the Business rules or the business logic of the encapsulations. To have a separate layer for business logic is of a great advantage. This is because any changes in Business Rules can be easily handled in this layer. As long as the interface between the layers remains the same, any changes to the functionality/processing logic in this layer can be made without impacting the others. A lot of client-server apps failed to implement successfully as changing the business logic was a painful process.

3) The Data Access Layer:

This layer comprises of components that help in accessing the Database. If used in the right way, this layer provides a level of abstraction for the database structures. Simply put changes made to the database, tables, etc do not affect the rest of the application because of the Data Access layer. The different application layers send the data requests to this layer and receive the response from this layer.

The database is not accessed directly from any other layer/component. Hence the table names, field names are not hard coded anywhere else. This layer may also access any other services that may provide it with data, for instance Active Directory, Services etc. Having this layer also provides an additional layer of security for the database. As the other layers do not need to know the database credentials, connect strings and so on.

4) The Database Layer:

This layer comprises of the Database Components such as DB Files, Tables, Views, etc. The Actual database could be created using SQL Server, Oracle, Flat files, etc.

In an n-tier application, the entire application can be implemented in such a way that it is independent of the actual Database. For instance, you could change the Database Location with minimal changes to Data Access Layer. The rest of the Application should remain unaffected.

5) Data Transfer Objects (BEO):

Normally values passing from one layer other should be in the form of Objects.(Avoid passing individual parameters)

4.0 Naming Conventions

Widespread use and understanding of the naming guidelines should eliminate many of the most common developer questions.

Refer to the following sections for a complete list of standards and guidelines for naming conventions:

- Abbreviations
- Capitalization
- Case sensitivity
- Methods
- Punctuation
- Word choice

Refer to the following sections for examples of the naming conventions:

- Classes
- Enumeration types
- Namespaces
- Parameters, variables, and controls
- Properties
- Other

5.1 Abbreviations

To avoid confusion and guarantee cross-language interoperation, follow these rules regarding the use of abbreviations:

1. Avoid using abbreviations unless the full name is excessive.
2. Avoid abbreviations longer than five characters.
3. Any abbreviations must be widely known and accepted.
4. Use uppercase for two-letter abbreviations, and Pascal Case for longer abbreviations.
5. Do not use abbreviations or contractions as parts of identifier names. For example, use GetWindow instead of GetWin.
6. Do not use acronyms that are not generally accepted in the computing field or in the business context of the application.
7. Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use **UI** for User Interface and **Olap** for On-line Analytical Processing.
8. When using acronyms, use pascal case or camel case for acronyms more than two characters long. For example, use HtmlButton or htmlButton. However, you should capitalize acronyms that consist of only two characters, such as System.IO as opposed to System.Io. Well-known or branded acronyms (such as "EGA"), however, may retain their original capitalization if appropriate.
9. Do not use abbreviations in identifiers or parameter names. If you must use abbreviations, use camel case for abbreviations that consist of more than two characters, even if this contradicts the standard abbreviation of the word. For example, persisentEoi, not persistentEOI.

5.2 CAPTALIZATION

Use the following capitalization styles when developing

Pascal Case

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of two or more characters.
Example : BackColor
Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example:

backColor

Uppercase

All letters in the identifier are capitalized. Use this convention only for identifiers that consist of two or fewer letters. For example:

System.IO

System.Web.UI

You might also have to capitalize identifiers to maintain compatibility with existing, unmanaged symbol schemes, where all uppercase characters are often used for enumerations and constant values. In general, these symbols should not be visible outside of the assembly that uses them.

The following table summarizes the capitalization rules and provides examples for the different types of identifiers.

Identifier	Case	Example
Class	Pascal	**AppDomain**
Enum type	Pascal	**ErrorLevel**
Enum values	Pascal	**FatalError**
Event	Pascal	**OnValueChange**
Exception class	Pascal	**WebException** **Note** Always ends with the suffix **Exception**.
Read-only Static field	Pascal	**RedValue**
Interface	Pascal	**IDisposable** **Note** Always begins with the prefix I.
Method	Pascal	**ToString**
Namespace	Pascal	**System.Drawing**
Parameter	Camel	**typeName**
Property	Pascal	**BackColor**
Protected instance field	Camel	**redValue** **Note** Rarely used. A property is preferable to using a

		protected instance field.
Public instance field	Pascal	**RedValue** **Note** Rarely used. A property is preferable to using a public instance field.
Constant Values	UpperCase	**WEEK_DAYS_COUNT**

5.3 Case Sensitivity

To avoid confusion and coding errors, do not use two names within the same context that differ only by case. C# is a case-sensitive language, and will see these two identifiers as separate instances.

The following is an example of an incorrectly named identifier:

strLastName
strlastname

5.4 Punctuation

The naming conventions outlined in this document do not use any whitespace or punctuation other than the period (.) for the namespace.

Underscores (_), dashes (-) and any other punctuation should not be used in class names, methods, namespaces, folders, files, etc.

Punctuation is reserved for system-level objects such as the App_Theme folder and _Default on the class definition.

5.5 Word Choice

Avoid using class names that duplicate commonly used .NET framework namespaces. For example, do not use any of the following names as a class name: System, Collections, Forms or UI.

For a complete list of keywords, visit MSDN at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconWordChoice.asp.

Other conflicts may exist within your presentation and business tier.
The following example illustrates a naming conflict between the presentation and business tiers:

Connections (folder in the presentation layer)
Connections.cs (class file in the business layer)

The following example illustrates a naming conflict between folder names and Web forms (.aspx pages) in the presentation tier:

StyleDesign (folder in the presentation layer)
StyleDesign.aspx (Web form in the presentation layer)

The solution for this potential naming conflict is outlined in the Classes section.

5.6 Classes

- Use a noun or noun phrase to name a class.
- Use Pascal case.
- Use abbreviations sparingly.
- Do not use a type prefix, such as C for class, on a class name.
- Do not use the underscore character.
- Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, ApplicationException is an appropriate name for a class derived from a class named Exception. Use reasonable judgment in applying this rule.
- The file name should reflect the class(es) it contains.
- Classes should be grouped based on logical functionality.

  For example, in the BSU.UCS.Common library, all of the Person classes are in the Person.cs file and all Department classes are in the Department.cs file.

- When creating classes in the presentation layer, they should be prefixed with the following identifier, based on application type:

  Web – Web application
  Win – Windows application

Mobile – Mobile application
For example: WebCommon.cs or MobileBusiness.cs

5.7 Enumeration Types

The enumeration (Enum) value type inherits from the Enum Class. The following rules outline the naming guidelines for enumerations.

- Use Pascal case.
- Use abbreviations sparingly.
- Do not use an Enum suffix on Enum type names.
- Use a singular name for most Enum types.

The following code example illustrates how to define an enumeration type:

```
public enum Status
{
// numbering scheme sets the first element to 0, followed by an n+1 progression
// you may also change this behavior and start the first value at 192
// the other two values will be 193 and 194
Active, // =0
Inactive, // =1
All // =2
}
```

5.8 Methods

The following rules outline the naming guidelines for methods:

- Use verbs or verb phrases to name methods.
- Use Pascal case.

The following are examples of correctly named methods.

```
Delete()
GetPersonList()
Insert()
Update()
IsEnabled()
```

5.9 NameSPaces

Namespaces are a logical (not physical) way to group classes. A single namespace may contain one or more physical class files (classname.cs). When the ASP.NET framework compiles your application, all class files with the same namespace declaration are compiled into a single assembly (.dll file).

When creating namespaces, use the following guideline:

ega.ministryname.servicename

EGA.MOE.Accredation (Company name followed by the department name)

You should use Pascal case for namespaces, and separate logical components with periods, as in EGA.MOE.Accredation

Use plural namespace names if it is semantically appropriate. For example, use System.Collections rather than System.Collection.

5.10 Parameters and controls

The following rules outline the naming guidelines for parameters.

- Use camel case for parameter names.
- Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios. For example, Visual Studio 2008 provides intelligence to the developer as they type.
- Use names that describe a parameter's meaning rather than names that describe a parameter's type. Use type-based parameter names sparingly and only where is appropriate.

  Note: The prefix on the parameter name will serve to identify the type. Refer to the list below for acceptable type prefixes.

- Do not use reserved parameters.

Control/Type	Prefix	Control/Type	Prefix
AdRotator	ad		
Boolean	bln	ImageButton	ibtn
BulletedList	bl	ImageList	ilst
Button	btn	ImageMap	imap
Calendar	cal	Integer	int
Char	chr	Label	lbl
CheckBox	cbx	LinkButton	lbtn

CheckBoxList	cbxl	ListBox	lbx
ComboBox	cmb	ListView	lv
DataColumn	dc	MultiView	mv
DataGrid	dg	Panel	pnl
DataReader	drdr	PlaceHolder	ph
DataGrid	dg	RadioButton	rbtn
DataRow	dr		
DataSet	ds	RadioButtonList	rbtnl
DataTable	dt	RequiredFieldValidator	req
DateTime	date	String	str
Decimal	dec	StringBuilder	strb
Dialog	dlg	Table	tbl
DialogResult	drs	TableCell	tc
Double	dbl	TableFooterRow	tfr
DropDownList	ddl	TableHeaderCell	thc
Exception	ex	TableHeaderRow	thr
FileUpload	fu	TableRow	tr
Form	frm	TextBox	tbx
GridView	gv	Validation Summary	vs
GroupBox	gbx	Wizard	wiz
HashTable	htbl		
HyperLink	lnk		

For ASP.NET controls not listed above, prefix the ID with the uppercase letters of the control name.

5.11 Events

The following rules outline the naming guidelines for events:

- Use Pascal case.
- Use an EventHandler suffix on event handler names.
- Specify two parameters named sender and e. The sender parameter represents the object that raised the event. The sender parameter is always of type object, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named e.
- Name an event argument with the EventArgs suffix.
- Consider naming events with a verb. For example, correctly named event names include Clicked, Painting and DroppedDown.

- Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of a pre-event, and a post-tense verb to present post-event.

  For example, a Close event that can be canceled should have a Closing event and a Closed event. Do not use the BeforeXxx/AfterXxx naming pattern.

- Do not use a prefix or suffix on the event declaration on the type.

  For example: use Close instead of OnClose.

5.12 Interfaces

The following rules outline the naming guidelines for interfaces:

- Name interfaces with nouns or noun phrases, or adjectives that describe behavior. For example, the interface name IComponent uses a descriptive noun. The interface name ICustomAttributeProvider uses a noun phrase. The name IPersistable uses an adjective.
- Use Pascal case.
- Use abbreviations sparingly.
- Prefix interface names with the letter I, to indicate that the type is an interface.
- Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only the letter I prefix on the interface name.
- Do not use the underscore character.

The following code example illustrates how to define the interface IComponent and its standard implementation, the class Component:

```
public interface IComponent
{
// implementation code goes here
}
public class Component: IComponent
{
// implementation code goes here
}
```

5.13 Property Names

- Name properties using nouns or noun phrases
- Use Pascal Casing
- Consider naming a property with the same name as its type

Example : **BackColor, NumberOfItems**

The following code example illustrates correct property naming:

```
public class SampleClass
{
public Status AppStatus
{
// code for get and set accessors goes here
}
}
```

5.14 Variables Naming

The following are some general guidelines for variable names:

- Variable names should be between 8 - 20 characters long. Usually between 8-16 characters.
- Variable names should clearly communicate what the variable stores.
- Use Camel case for variable names.
- Avoid use of the _ underscore character in your variable names if possible. It makes them longer and more work to type.

5.15 ReadOnly and Const Field Names

- Name static fields with nouns, noun phrases or abbreviations for nouns
- Use Capital Case

5.16 Exception Naming

- Use ex and name of the exception as a standard variable name for an Exception object

  Example
  ```
  Catch(FileNotFoundException exFileNotFound)
  {
  //Handle Exception
  }
  ```

5.0 Programming Practices

5.1 Scope of Variables

The word scope in this context means the extent to which a variable can be seen/used inside a program. There are three levels of scope:

- Public this is variables exposed to outside world. Use properties for giving read only or write only access to outside world instate of using public
- Private (the variable is accessible only inside the class)
- Protected(the variable is access only from the class and sub class)
- Internal (Variable can be used inside the namespace)

5.2 Statements

- Compound statements are statements that contain lists of statements enclosed in braces { statements }.
- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the beginning of the line following the line that begins the compound statement and be indented to the beginning of the compound statement. The closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

The following example illustrates a if, if...else, if...else if...else statements:

```
if  (...)
{
// statement
}

// ***********************************************************

if  (...)
{
// statement
}
else
```

```
{
// statement
}

// ***********************************************************

If (...)
{
// statement
}
else if (...)
{
// statement
}
else
{
// statement
}
```

The following example illustrates a for statement:

5.3 Blank lines

Blank lines improve readability by setting off sections of code that are logically related. One blank line should always be used in the following circumstances:

- Between the local variables in a method and its first statement
- Between logical sections inside a method to improve readability
- After the closing brace of a code block that is not followed by another closing brace
- Use one blank line to separate logical groups of code.

Good:	**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 ( ... )
```
	```
	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
	```

```
        {
                // Do something
                // ...

                return false;
        }

        return true;
}
```
	```
	                // ...
	                return false;
	        }
	        return true;
	}
	```

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

5.4 Wrapping lines

When an expression will not fit on a single line, break it up according to these general principles:

- Break after a comma
- Break after an operator
- Prefer higher-level breaks to lower-level breaks
- Align the new line with the beginning of the expression at the same level on the previous line

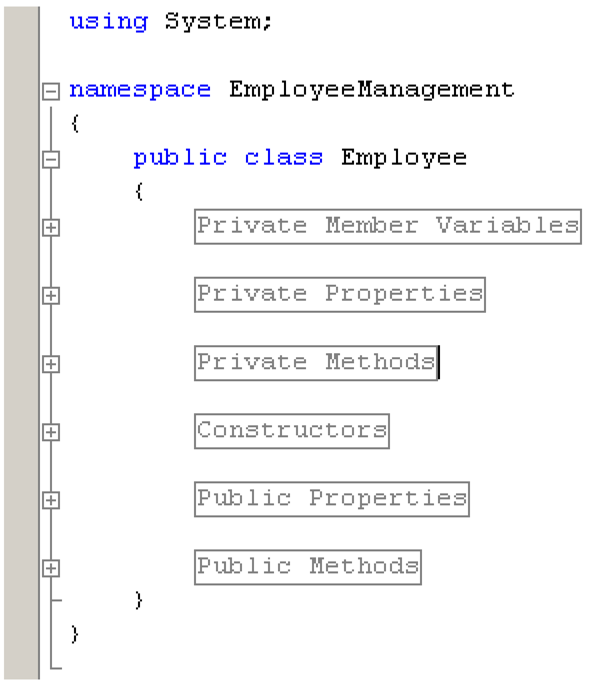
5.5 Regions

As your code gets longer and longer, it becomes more and more difficult to navigate. While you can select classes and methods from the drop-down list above the main editor window within , you can also group your code into logical regions. Specify regions with the #region keyword and a description at the beginning of the code segment. Use a corresponding #endregion keyword at the end of the segment.

The following example illustrates the use of the #region keyword:

```
#region Maintenance Classes
// TODO: Verify business logic for operation
// code block
#endregion
```

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
    }
}
```

5.6 *Indentation and Spacing*

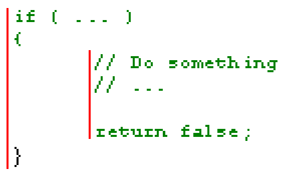
1. Use TAB for indentation. Do not use SPACES. Define the Tab size as 4.

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

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

Good:	Not Good:

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

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

Good:	**Not Good:**

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

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

5.7 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 **/* … */**

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

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

10. Comments will be added at class levels to get a brief understanding to the class/File

11.

Eg:
```
/// <summary>
/// ***********************************************************************
/// Name            : Address
/// Pupose          : (1) Business Entity class
///
/// Author          : XXXXX
/// Created Date      : 03-Feb-09
/// Modification History:
/// ----------------------------------------------------------------------
/// Date :      |Modification:
///           |
///           |
///           |
///           |
/// ***********************************************************************
/// </summary>
```

6.0 Application Development

6.1 Data Access

(1) Always use stored procedure to access database, Don't execute any database queries directly from the application.
(2) Always Database Access to execute procedures

6.2 Exception Management

The Exceptions thrown by .NET runtime have their base class as Exception. There are varieties of system exceptions that are thrown. Please find below some of the Exceptions thrown by the .NET runtime.

- OutOfMemoryException
- NullReferenceException
- InvalidCastException
- ArrayTypeMismatchException
- IndexOutOfRangeException
- ArithmeticException
- DivideByZeroException
- OverFlowException

a. 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.

b. 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.

c. 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 "";
  }
}
```

d. 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'.

e. 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;
}
```

f. 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.

g. 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.

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

6.3 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. 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.

6.4 Returning a DataSet versus single item

When retrieving data from the database, the following guidelines apply for the return type of the method:

- If the method does not return a value, use void
- If you are returning a **single** item (column/row value), such as an ID or a name, the return type of the method would be the data type of that item.

  For example: string, int, or Guid.

- If you are returning **2 or more** items, the return type of the method is a DataSet.

Typically, the return type for a method will be a DataSet. This is done to provide the greatest amount of flexibility when binding the data to a control.

7.0 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:	Not Good:

void SavePhoneNumber (string phoneNumber
)
{
// Save the phone number.
}
	// 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:	**Not 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.
// ...
}
	// Save address and send an email to the supervi 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 supervis 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 **I**nt**16**)
string name; (not **S**tring)
object contactInfo; (not **O**bject)

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 and if you are sure this value never need to be changed

7. You should use the constants in the config file or database so that you can change it later.

8. Do not hardcode strings. Use resource files.

9. 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” )
{
          //…
}
```

10. Use String.Empty instead of “”

Good:

```
If ( name == String.Empty )
{
```

Not Good:

```
If ( name == “” )
{
```

// do something }	// do something }

11. 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.

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

Good:	**Not 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;
    }
}
```
	```
	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;
	    }
	}
	```

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

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

15. 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.

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

17. 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:".

18. 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."

19. 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."

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

21. 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.

22. 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.

23. 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.

24. 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”.

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

26. Logically organize all your files within appropriate folders. Use 2 level folder hierarchies. You can have up to 10 folders in the root folder and each folder can have up to 5 sub folders. If you have too many folders than cannot be accommodated with the above mentioned 2 level hierarchy, you may need re factoring into multiple assemblies.

27. 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.

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

29. 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();
}
```

Appendix

Reserved Words

The following is a partial list of reserved words in VB and C# .NET languages.

AddHandler	AddressOf	Alias	And	Ansi

As	Assembly	Auto	Base	Boolean
ByRef	Byte	ByVal	Call	Case
Catch	CBool	CByte	CChar	CDate
CDec	CDbl	Char	CInt	Class
CLng	CObj	Const	CShort	CSng
CStr	CType	Date	Decimal	Declare
Default	Delegate	Dim	Do	Double
Each	Else	ElseIf	End	Enum
Erase	Error	Event	Exit	ExternalSource
False	Finalize	Finally	Float	For
Friend	Function	Get	GetType	Goto
Handles	If	Implements	Imports	In
Inherits	Integer	Interface	Is	Let
Lib	Like	Long	Loop	Me
Mod	Module	MustInherit	MustOverride	MyBase
MyClass	Namespace	New	Next	Not
Nothing	NotInheritable	NotOverridable	Object	On
Option	Optional	Or	Overloads	Overridable
Overrides	ParamArray	Preserve	Private	Property
Protected	Public	RaiseEvent	ReadOnly	ReDim
Region	REM	RemoveHandler	Resume	Return
Select	Set	Shadows	Shared	Short
Single	Static	Step	Stop	String
Structure	Sub	SyncLock	Then	Throw

To	True	Try	TypeOf	Unicode
Until	volatile	When	While	With
WithEvents	WriteOnly	Xor	eval	extends
instanceof	package	var		

Downloads

FxCop	http://www.gotdotnet.com/team/fxcop/

References

DotNet Spider	http://www.dotnetspider.com/tutorials/BestPractices.aspx
Class Member Usage Guidelines	http://msdn2.microsoft.com/en-us/library/426s83c3(VS.71).aspx
Type Usage Guidelines	http://msdn2.microsoft.com/en-us/library/9cws5bzd(vs.71).aspx
Design Guidelines for Class Library Developers	http://msdn2.microsoft.com/en-us/library/czefa0ke(vs.71).aspx
Microsoft Inductive User Interface Guidelines	http://msdn2.microsoft.com/en-us/library/ms997506.aspx
Guidelines for Keyboard User Interface Design	http://msdn2.microsoft.com/en-us/library/ms971323.aspx
Microsoft UI Portal	http://msdn2.microsoft.com/en-us/library/aa286531.aspx
XML Documentation	http://msdn2.microsoft.com/en-us/library/b2s063f7(vs.71).aspx
XML Documentation Tutorial	http://msdn2.microsoft.com/en-us/library/aa288481(VS.71).aspx
C# and VB.NET Comparison	http://msdn2.microsoft.com/en-us/library/czz35az4(vs.71).aspx http://www.geocities.com/gkmsr007/articles/CSharpVersusVBdotNET_Paper.htm http://www.harding.edu/USER/fmccown/WWW/vbnet_csharp_comparison.html http://www.codeproject.com/dotnet/vbnet_c__difference.asp
Microsoft Internal Coding Guidelines	http://blogs.msdn.com/brada/articles/361363.aspx
IDesign C# Coding Standards (Juval Lowy)	Available under “Resources” at: http://www.idesign.net/