Coding conventions and best practices

Purpose of coding standard and best practices to do it

Coding standards are a set of guidelines used for programming language that recommends programming style and best practices to achieve it. The coding standards generally covers indentation, comments, naming conventions, programming practices, file structure within project, architectural best practices etc. Software developers are highly recommended to follow these guidelines. The coding guidelines have following advantages.

1. Increases the readability of source code written.
2. Will contain less bugs and works more efficiently.
3. Requires less maintenance.
4. Easier for old and new developers to maintain and modify the code.
5. Leads to increase in productivity of developers.
6. Reduces the overall cost for software development.
7. Make the difference between a successful project and a project that is, at worst, dead on delivery.

How to follow coding standards in your teams

1. Make your own coding standard document as per industry standards by sitting together in a team and share across the team.
2. Look at the existing code and decide what you want in yours.
3. Assign someone to do the frequent code review and peers reviews.
4. Involve seniors and get some inspiration from their previous experience.
5. Whenever new fresh guy arrives, introduce projects or products codebase to him or her. Educate that guy to follow coding standards.

Naming Conventions

There are three type of naming conventions generally used while doing C# programming

1.Pascal Convention – First character of all word are in upper case and other characters are in lower case.
Example: HelloWorld
2. Camel Case Convention – The first character of all words, except the first word, is upper case and other characters are lower case.
Example: helloWorld
3. Hungarian Case Convention – The data type as prefix is used to define the variable by developers long ago. This convention is not used anywhere now a day’s except local variable declaration.

Example:

```
string m_sName;
string strName;
int iAge;
```

Let’s generalize the module with correct and incorrect naming conventions.Let’s generalize the module with correct and incorrect naming conventions.

Sr. No	Module	Description	Correct	Wrong

1.	Class	Use Pascal conventions for defining class name	public class HelloWorld
			{
			}
				public class helloWorld
				{
				}

2.	Method	Use Pascal conventions for defining class name	public void AddNumbers(int first, int second)
			{
			}
				public void addNumbers(int first, int second)
				{
				}

3.	Interface	Use Prefix “I” with Camel Casing to define interface	public interface IEmployee
			{
			}
				public interface Iemployee
				{
				}

4.	Local Variables	Use Hungarian Meaningful, descriptive words to name variables. Do not use abbreviations	string firstName int salary	string fName string name string _firstName int sal
5.	Member Variables	Member variables must be prefix with underscore(_) so that they can be identified by other local variables and constants	private IEmployee _employeeService = null private UserRole userRole private UserGroup userGroup	private IEmployee empService = null private UserRole usrRole private UserGroup usrGrp
6.	Boolean variables	Prefix Boolean variables with “is” or some smaller prefixes	private bool _isDirty private bool _isFinished	private bool dirty private bool finished private bool isFinished
7.	Namespace	The namespace should be logical grouping of classes with specific pattern	<CompanyName>.<ProductName>.<ModuleName> Example: ABC.SchoolManagement.BusinessLayer ABC.SchoolManagement.DataAccessLayer ABC.SchoolManagement.WebUI	Example: BusinessLayer DataAccessLayer WebUI

8.	File Name	Filename should match with class name i.e. Pascal name	public class HelloWorld
			{
			}
			The Filename must be: HelloWorld.cs
				public class HelloWorld
				{
				}
				Filename: helloworld.cs

Control Prefix Naming Conventions

The best practice generally used while developing web and window application uses prefixes for UI elements. Since ASP.NET and Windows Form have too many controls available to use, hence summarizing them through tabular chart.

Sr.No	Control	Prefix
1	Label	lbl
2	TextBox	txt
3	DataGrid	grd
4	Button	btn
5	ImageButton	imb
6	Hyperlink	hyp
7	DropDownList	ddl
8	ListBox	lst
9	DataList	dtl
10	Repeater	rep
11	Checkbox	chk
12	CheckBoxList	cbl
13	RadioButton	rdo
14	RadioButtonList	rbl
15	Image	img
16	Panel	pnl
17	PlaceHolder	phd
18	Table	tbl
19	Validators	val

Code Indentation and Comments

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. To achieve this below points would be helpful,

1. Use default code editor setting provided by Microsoft Visual Studio.

2. Write only one statement and declaration per line.
3. Add one blank line space between each method.
4. Use parentheses to understand the code written.
5. Use xml commenting to describe functions, class and constructor.
6. Use Tab for indentation.
7. Use one blank line to separate logical groups of code.
8. Use #region and #endregion to group related piece of code as per below

- a. Private Member
- b. Private Properties
- c. Public Properties
- d. Constructors
- e. Event Handlers/Action Methods
- f. Private Methods
- g. Public Methods

9. Do not write comments for every line of code and every variable declared.
10. Use // or /// for comments avoid using /* … */
11. If you have to use some complex for any reason, document it very well with sufficient comments.
12. If all variables and method names are meaningful, that would make the code very readable and will not need many comments.

Good Programming Practices

1. Avoid writing of long functions. The typical function should have max 40-50 lines of code. If method has more than 50 line of code, you must consider re factoring into separate private methods.
2. Avoid writing of long class files. The typical class file should contain 600-700 lines of code. If the class file has more than 700 line of code, you must create partial class. The partial class combines code into single unit after compilation.
3. Don’t have number of classes in single file. Create separate file for each class.
4. Avoid the use of var in place of dynamic.
5. Add a whitespace around operators, like +, -, ==, etc.
6. Always succeed the keywords if, else, do, while, for and foreach, with opening and closing parentheses, even though the language does not require it.
7. The method name should have meaningful name so that it cannot mislead names. The meaningful method name doesn’t need code comments.

Good:

```
private void SaveAddress(Address address)
{
}
```

Bad:

```
// This method used to save address
private void Save (Address addr)
{
}
```

8. The method or function should have only single responsibility (one job). Don’t try to combine multiple functionality into single function.

Good:

```
public void UpdateAddress(Address address)
{
}
public void InsertAddress(Address address)
{
}
```

Bad:

```
public void SaveAddress(Address address)
{
        if(address.AddressId == 0)
        {
           // Insert address code
        }
        else
        {
           // Update address code
        }
}
```

9. Avoid using common type system. Use the language specific aliases

Good:

```
int age;
string firstName;
object addressInfo;
```

Bad:

```
System.Int32 age;
String firstName;
Object addressInfo;
```

10. Do not hardcode string or numbers instead of it create separate file for constants and put all constants into that or declare constants on top of file and refer these constants into your code.
11. You can also put some constants like database connection, logger file name, SMTP information variables etc. in form of key and value pair in config file.
12. Don’t hardcode strings. Use resource files.
13. While comparing string, convert string variables into Upper or Lowers case

Good:

```
if (firstName.ToLower() == “rohit”)
  {
  }
if (firstName.ToUpper() == “ROHIT”)
  {
  }
```

Bad:

```
if (firstName == “rohit”)
  {
  }
```

14. Use String.Empty instead of “”

Good:

```
if (firstName == String.Empty)
      {
      }
```
Bad:
```
if (firstName == “”)
      {
      }
```
15. Avoid using member variables. Declare local variable in function or method itself and pass it to other method whenever required instead of sharing the member variable between two methods. If you share a member variable between methods, it will be difficult to track which method changed the value and when.

16. Don’t make member variables to public or protected. Keep them as private and expose them as public or protected properties

17. Use enums wherever require. Don’t use numbers or strings to indicate discrete values.

Good:
```
  public enum LoggerType
     {
           Event,
           File,
           Database
   }
  public void LogException(string message, LoggerType loggerType)
     {
         switch (loggerType)
         {
            case LoggerType.Event:
               // Do something
               break;
            case LoggerType.File:
               // Do something
               break;
            case LoggerType.Database:
               // Do something
               break;
            default:
               // Do something
               break;
         }
    }
```
Bad:
```
      public void LogException(string message, LoggerType loggerType)
       {
         switch (loggerType)
         {
            case “Event”:
               // Do something
               break;
```

```
            case “File”:
                // Do something
                break;
            case “Database”:
                // Do something
                break;
            default:
                // Do something
                break;
        }
    }
```

18. The event handler should not contain the code to perform the required action. Instead of call another private or public method from the event handler. Keep event handler or action method clean as possible.

19. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path. Use input or output classes (System.IO) to achieve this.

20. Always do null check for objects and complex objects before accessing them.

Good:

```
 public Contact GetContactDetails(Address address)
    {
        if (address != null && address.Contact != null)
        {
            return address.Contact;
        }
   }
```

Bad:

```
public Contact GetContactDetails(Address address)
    {
        return address.Contact;
   }
```

21. Error message to end use should be user friendly and self explanatory but log the actual exception details using logger. Create constants for this and use them in application.

Good:

“Error occurred while connecting to database. Please contact administrator.”

“Your session has been expired. Please login again.”

Bad:

“Error in Application.”

“There is an error in application.”

22. Avoid public methods and properties to expose, unless they really need to be accessed from outside the class. Use “internal” if they are accessed only within the same assembly and use “private” if used in same class.

23. Avoid passing many parameters to function. If you have more than 4-5 parameters use class or structure to pass it.

Good:

```
public void UpdateAddress(Address address)
    {
   }
```

Bad:

```
public void UpdateAddress(int addressId, string country, string state, string phoneNumber, string pi
nCode, string address1,
string address2)
    {
    }
```

24. While working with collection be aware of below points,

- a. While returning collection return empty collection instead of returning null when you have no data to return.
- b. Always check Any() operator instead of checking count i.e. collection.Count > 0 and checking of null
- c. Use foreach instead of for loop while traversing.
- d. Use IList<T>, IEnumrable<T>, ICollection<T> instead of concrete classes e.g. using List<>

25. Use object initializers to simplify object creation.

Good:

```
var employee = new Employee{
                    FirstName = “ABC”,
                    LastName = “PQR”,
                    Manager = “XYZ”,
                    Salary = 12346.25
                };
```

Bad:

```
var employee = new Employee();
employee.FirstName = “ABC”;
employee.LastName = “PQR”;
employee.Manager = “XYZ”;
employee.Salary = 12346.25;
```

26. Use meaningful names for linq query variables. The following example uses puneCustomers for customers who are located in Pune.

```
var puneCustomers = from cust in customers
                    where cust.City == “Pune”
                select cust.Name;
```

27. Don’t use single character names such as I, j, k instead of use index or counter specific to operation

```
for (int counter = 0; counter < nameArray.length; counter++)
    {
    }

    foreach (var employee in employees)
    {
    }
```

28. The using statements should be sort by framework namespaces first and then application +namespaces in ascending order

```
using System;
using System.Collections.Generic;
```

```
using System.IO;
using System.Text;
using Company.Product.BusinessLayer;
```

29. Make sure that you have good logging and tracing framework. If you configured to log errors, it should log only errors but if you configure to log traces then is should record all errors, warnings, and error details. You can trace or log errors into windows event log, sql server or to a separate file. User these logging or tracing framework extensively through the application.

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. Simplify your code by using the C# using statement. If you have a try-finally statement in which the only code in the finally block is a call to the Dispose method, use a using statement instead.

Good:

```
using (var fileToOpen = new FileInfo(fileName))
{
        // File operation
}
```

Bad:

```
    var fileInfo = new FileInfo(fileName);
         try
         {
            // File operation
         }
         finally
         {
            if (fileInfo != null)
            {
               fileInfo.Delete();
            }
    }
```

32. Always catch only the specific exception instead of catching generic exception.

Good:

```
 void ReadFile(string fileName)
       {
          try
          {
             // read from file.
          }
          catch (System.IO.IOException fileException)
          {
             // log the error. Re-throw exception
             throw fileException;
          }
          finally
          {
             // close file connection
```

```
        }
  }
```

Bad:

```
 void ReadFile(string fileName)
      {
         try
         {
            // read from file.
         }
         catch (Exception ex)
         {
            // catching general exception
         }
         finally
         {
            // close file connection
         }
  }
```

33. 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.

34. Do the brainstorming session within team to make reusable components.

Architecture Design Level Guidelines

1. Always use multi tier (N-Tier) Architecture.
2. Implement loosely coupled architecture using interfaces and abstract class.
3. Use of generics would help you to make reusable classes and functions.

```
public class MyClass<T> where T : SomeOtherClass
   {
      public void SomeMethod(T t)
      {
         SomeOtherClass obj = t;
      }
}
```

4. Separate your application into multiple assemblies. Create separate assemblies for UI, Business Layer, Data Access Layer, Framework, Exception handling and Logging components.
5. Do not access database from UI pages. Use data access layer to perform all task which are related to database.
6. Always use stored procedure instead of writing inline queries in C# code.
7. Always use transaction in database operation like Create, Update, and Delete. This would be helpful to rollback old data again in case of any exception occurred while execution of Sql statement.
8. Don’t put complex logic inside stored procedure instead of it put it into the business layer.
9. Don’t use prefix as “sp” or “sp_” to the user defined stored procedure similarly don’t use prefix as “fn” or “fn_” as the all system level procedure start with “sp” and functions start with “fn” which triggers overload for search of procedures.
10. For installing database on client machine user installer sql scripts.

11. Try to use design pattern, practices and SOLID principles.
12. For same code, create separate utility file or move it to base class.
13. Use try-catch-finally 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 to caller layer so that the application can catch it and show the user specific message on UI.
14. Don’t store large objects into Session. Storing large or complex object into session may consume server memory. Destroy or Dispose such session variable after use.
15. Don’t store large object into view state, this will increase the page load time.
16. Always use style sheet classes to defile style. Avoid use of inline style in html instead create separate class and refer it to html UI.
17. Always refer third party dll, javascripts and css framework through NuGet package so that you can update with latest version whenever required.
18. Always refer minified version of javascript or css files, this will reduce unnecessary overhead to the server.

Revision #4
Created 27 December 2019 04:00:32 by Pranavi Pamireddy
Updated 27 December 2019 05:14:54 by Pranavi Pamireddy