C# Coding Standards

TOLLPLUS

Contents

Acknowledgement

The best practices and/or coding standard mentioned here are based on own experiences, team experiences and information available in books or on the internet. There is always chance of improvements in whatever we do and this document is no exception. This should not be considered as full and complete.

Before and after implementing this, do check yourself and/or with your team lead for suitability and/or performance.

You may find some spelling or grammatical mistakes in this document, please feel free to let your Team Leader know about it.

Please feel free to write comment, feedback or suggestions at snarayan@tollplus.com or let your Team Leader know; to further improve this document.

1. Naming guidelines

1. Members, Parameters, Variables should be named in such a way that
    a. It is easily readable and understandable
    b. Grammatically correct
2. Use following casing for
    a. Class, Struct, Property, Interface, Enumeration, Events, Constant & Static fields, Method Names, – PascalCase
    b. Private field, variables, parameters – camelCase

3. Do not include numbers in variable names
4. Do not prefix fields or variables with m_, g_ etc. For global variables you can prefix with _
5. Do not use abbreviation while writing method names, variable names.
6. Do not repeat the name of the class in its methods or property names

```
class Person
{
      // Wrong
      string GetPerson(int id);

      // Correct
      string Get(int id);
}
```

7. Naming property names
    a. Property names should be named with noun, noun phrases, or adjective
    b. Prefix Boolean field with Is, Has, Can, Allow etc. so that by reading the name, it is evident that what it will do.
8. Namespace name should contain names, layers, verbs, features and should maintain proper hierarchy

```
TollPlus.Ceba.Issuer
TollPlus.Ceba.Acquirer
TollPlus.Ceba.Issuer.Webpi
```

9. Use a verb or verb phrase to name an event. Like Click, Close, MouseOver etc.
10. Prefix an event handler with On. Like OnClose, OnOpen, OnClick etc.

2. Class/Interface design

A class is a construct that enables us to create our own custom types by grouping variables, methods and events. It's a blueprint and defines the data and behavior of a type.

Single Purpose

A class or interface should have a single purpose within the system. It is also called a Single Responsibility Principle – first principle of SOLID principle explained below).

Meaningful Name

A class or interface should have meaningful name, do not combine vaguely related members in the same class or interface.

Avoid Static Class

Except in case of extension method, try to avoid Static classes. It is very difficult to test it in isolation.

Do not create shadow method of the base class

Creating shadow method of the base class breaks the Polymorphism principle and also makes the sub classes difficult to understand.

```
class Manager
 {
     public virtual string GetName()
     {
         return "Ram";
     }
 }

 class Employee : Manager
 {
     public new string GetName()
     {
         return "Mr. Ram";
     }
}

Employee employee = new Employee();
     var name = employee.GetName(); // Mr. Ram
     Console.WriteLine(name);

     name = ((Manager)employee).GetName(); // Ram
     Console.WriteLine(name);

     Console.ReadKey();
```

1. Prefix interface name with I.
2. Use singular names for Enums like Employee.HR, Employee.Engineer

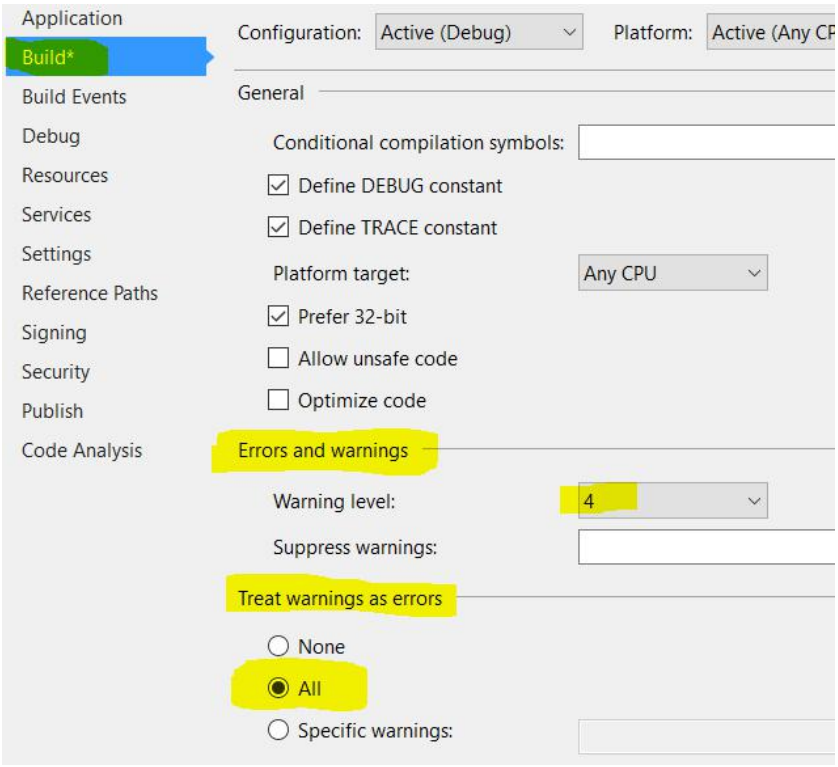
3. Framework guides

1. Use C# predefined type aliases for variable types instead of system type like use
    a. string not String
    b. int not Int32

c. bool not Boolean
d. object not Object

2. Do not hardcode values such as database connectionStrings, flags, and settings into the code. Keep it into app.config or web.config files under ConnectStrings or AppSettings

3. Build project with highest warning level, set following in the project property.



4. Fill the AssemblyInfo.cs file properly and maintain proper company name, copyright, version and other details.

4. Member Design

1. Declare member variables at the top and static variables at the very top of the class.
2. Property should be stateless and not dependent on other property. A property can be set without first setting another property.
3. User method instead of property, if
    a. The work is not only to set the field value
    b. Value conversion is required
    c. The work changes the value of some other fields
    d. It returns different result each time it is called even with same parameters (like generating random numbers)
4. A method or property should be doing only one thing
5. Do not expose state full object through static members

6. To return a collection, do not return collection of concrete classes, instead return IEnumerable<T> or ICollection<T> or IReadOnlyCollection<T> (read only collections)
7. Properties, Methods returning string or collections should never return null
    a. Always return empty string or empty collection instead of null values. This helps in avoiding additional checking of null values in the code and makes the code cleaner
8. Pass specific parameters not the entire object if not using all object properties/field. Remember 'Do not ship the truck if you only need a package'.

5. Miscellaneous Design

1. Returning values of true, false to report success or failure that is prone to have many conditional statements. Instead of that try to throw exception and catch it into calling methods.
2. The exception message should be meaningful and clearly describe what should be done to fix it.
3. Throw most specific exceptions rather than throwing generic exception; like for null throw NullReferenceException rather than just Exception. Generic exceptions should come in the last, follow the hierarchy of the exceptions.
4. Use generic constraint to restrict a class to be used with

```
public class Organization<T> where T: Manager
 {
        public string GetName(T t)
        {
               Manager obj = t;
         return obj.GetName();
        }
 }

Organization<Manager> o = new Organization<Manager>();
     Manager m = new Manager();
     var name = o.GetName(m);
```

5. Always evaluate the result of the LINQ expression by using .ToList() or .ToArray() instead of just returning the query. Unless you use .ToList() or .ToArray() method, the actual query is not executed against the database.

6. Maintainability

1. A method should not be too big to understand. Refractor it to make the chunk of code meaningful, easily understandable and maintainable.
2. Make all methods private and types (class/interface) internal by default. After that based on the need, expose them with appropriate modifiers (public, protected etc.)
3. Name assemblies same as their namespaces
4. Use PascalCase to name the file
5. One file should contain only one type however nested types should contain in the same file.
6. Partial class file name should be named prefixing by its type eg. Employee, Employee.Account, Employee.HR
7. Try to use using statement to refer the namespace instead of using fully qualified names

a. Do not use – `var table = new System.Data.DataTable();`
b. Use –

```
using System.Data;

var table = new DataTable();
```

8. Use var when the type is obvious

```
a. var age = 2; //wrong - is it int, int16, uint or float?
```

use var as the result of the LINQ query or where the type is obvious from the statement.

```
a. var query = db.Employees;
b. var salary = new GetSalary();
c. var items = new List();
```

9. Declare and initialize the variables as late as possible. Define variables wherever it is needed.
10. Assign each variable in a separate statement. Avoid var age = myage = yourage = 30;
11. Initialize object and collection in the same statement rather than doing separately

```
// avoid
var employee = new Employee();
employee.Name = "Ram";
employee.Address = "Hyderabad";

// prefer
var employee = new Employee() {
      Name = "Ram",
      Address = "Hyderabad"
};
```

similarly

```
var weekDays = new List{"Sunday", "Monday"};
```

12. Do not make explicit comparison to Boolean values

```
if (isPdfGenerated == true) // avoid
if (isPdfGenerated) // correct
```

13. Avoid nested loops
14. Always use {} (blocks) after conditional or loop statements even if only one statements are there for them.
15. Always use a default: block in the switch statement
16. If-else-if conditional statement must finish with else part.
17. Avoid multiple return statements from a method.
18. For conditional statements, use conditional assignment rather than using if else conditions

```
bool isTrue = (6 > 7);
return (age > 7) ? "No" : "Yes";
```

19. Write

```
return age ?? -1;
```

instead of

```
int age;
if (age == null)
{
        age  =-1;
}
else
{
        return age.value;
}
```

20. Write

```
return person.Father?.Name;
```

instead of

```
if (person.Father != null)
{
        return person.Father.Name;
}
else
{
        return null;
}
```

21. Encapsulate complex conditional statement or an expression in a method or property

Instead of

```
if (person.Age > 18 && person.Country.Equals("India") &&
person.Height = 156)
{
   // do something
}
```

Write like

```
if (IsPersonEligible(person))
{

}

private bool IsPersonlEligible(person)
{
   return (person.Age > 18 && person.Country.Equals("India") &&
```

```
person.Height = 156);
}
```

22. Use .TryParse method to parse the data type to avoid any error

23. Write comment for any major decision making code or conditions. Write comments for every method by press /// above the method so it automatically builds comment section for method and its parameters.

24. Use string.IsNullOrEmpty or string.IsNullOrWhiteSpace methods to check for empty or null string.

25. Remove any unused variables, Check for all warnings after the build and fix them.

26. In ideal scenario, no methods or constructors should have more than 3 parameters.

27. In most scenarios, try to avoid ref and out parameters in the method; it makes the code difficult to understand.

28. Use is pattern over as pattern while casting.

```
var thisPerson = person as Person; // it may return null
if (thisPerson is Person person) {} // it returns bool parameter
```

29. Do not check in commented code in source safe

30. Build the project successfully before checking in to source safe.

7. Performance guides

1. Values that are not going to change throughout the applications such as flags, settings can be declared as
    a. Runtime constant – using `readonly` keyword (little slower than const/static, however its value can be changed in constructor) or
    b. Compile time constant –using `const` keyword (faster than read only however, it's value can't be changed)
2. Prefer string.Format or StringBuilder for string concatenation.
3. Use conditional Attributes while debugging like
    a. `#if DEBUG and #endif or`
    b. `[Conditional("DEBUG"")`
4. Use foreach loop to enumerate through IEnumerable or Collcection objects

5. Do not use try, catch and finally block for deeper methods created inside BAL or DAL layers or any deeper layer unless you are handling them into those layers itself. Do not just catch the error and throw it.
   Exception: If you have a scenarios, where you want to handle those errors and do certain activities.
6. To dispose any objects, use try and finally block (you can skip catch block). Let the error bubble up to the upper layer if you do not want to handle error in deeper layers.
7. Catch only those exceptions that you can handle. In the outer layer you can catch error specific exceptions first such as NullReferenceException, DivideByZeroException and in the last catch generic Exception.
8. Use using block with those object that inherits IDisposable interface such as SqlConnection, MemoryStream and other reader objects. If you are going to use multiple such objects, instead of using using statement with all of them, use try and finally – instantiate all objects before try and in finally, dispose them all.

   using blocks internally creates try and finally block, so instead of having using for each objects (try and finally for each objects), explicitly write one try and finally.
9. To determine if a particular IEnumerable<T> collection is empty, try to use .Any() method rather than Count(), as count may trigger iterating through each collection objects that may impact the performance. If Length property of the collection is exposed, you can use it instead of .Any().
10. Do not reference unnecessary project or third party assemblies into the project.

8. Threading / Parallel Processing Guides

1. Use async method only for I/O bound operations
2. Use Task.Run for CPU bound operations

```
// try commenting and uncommenting the t.Wait() method and run
public static void Main()
 {
     ShowThreadInfo("Application");

     var t = Task.Run(() => ShowThreadInfo("Task"));
     // t.Wait();
     for (var i = 0; i < 200; i++)
     {
         Console.WriteLine(i);
     }
     t.Wait();

     Console.Read();
 }

 static void ShowThreadInfo(String s)
 {
```

```
        Console.WriteLine("{0} Thread ID: {1}",
                          s, Thread.CurrentThread.ManagedThreadId)
}
```

Task.Wait blocks the main thread and may even cause deadlocks situation if thread for which it is waiting never completes.

```
// example of how more than one task can run and wait for all to
proceed further.

      var list = new ConcurrentBag<string>();
      string[] dirNames = { ".", "../../" };
      List<Task> tasks = new List<Task>();
      foreach (var dirName in dirNames)
      {
          Task t = Task.Run(() =>
          {
              foreach (var path in Directory.GetFiles(dirName))
                  list.Add(path);
          });
          tasks.Add(t);
      }

      Task.WaitAll(tasks.ToArray());
      foreach (Task t in tasks)
      {
          Console.WriteLine("Task {0} Status: {1}", t.Id, t.Status);
      }

      Console.WriteLine("List of files \n\r=================");
      foreach(var file in list)
      {
          Console.WriteLine(file);
      }

      Console.WriteLine("Number of files read: {0}", list.Count);
             Console.Read();
```

Task.WaitAll blocks the main thread and may even cause deadlocks situation if any of the thread is waiting for completion.

3. Parallel processing
    a. Do's
        i. When the application is running on multi core processors
        ii. Scenario where individual iterations of a loop is independent
        iii. Calling a thread-safe method within the parallel loop
    b. Don'ts
        i. Do not overuse of parallel loop (almost never do nested parallel loop)
        ii. Where non-thread safe method is being called within the loop
        iii. Where your processor resource is limited
        iv. Do not use all the cores of your processors, try to keep one for operating system to do other activities on the server

4. Use Parallel.For when we know the number of elements we are going to process

```
        public static void Main()
         {
             // decide the thread count
             var threadCount =
int.Parse(ConfigurationManager.AppSettings["ThreadsToRun"].ToString());

             // start the thread now
             Parallel.For(0, threadCount, i => RunSingleInstance(i));
         }

         static void RunSingleInstance(int instanceNumber)
         {
             // do your work
        }
```

5. Use Parallel.ForEach to iterate through the collection where you never know how many items you are going to get in the collection.

```
        var invoices = new InvoicePDFGeneration().GetCustomerDetails(invoicePDFGenDate);

        Parallel.ForEach(invoices, invoice =>
        {
                 string emailAddress = invoice.EmailAddress;
        });
```

6. To limit the number of cores to be used and cancel the loop in Paralle.For or ForEach loop, use ParallelOptions and CancellationTokenSource.

```
             var cts = new CancellationTokenSource();

             // Use ParallelOptions instance to store the CancellationToken
             ParallelOptions po = new ParallelOptions();
             po.CancellationToken = cts.Token;
             po.MaxDegreeOfParallelism = InvoicePDFGenerationConstants.ThreadCount;

             Console.WriteLine(" Press 'c' to cancel.");
             // Run a task so that we can cancel from another thread.
             Task.Factory.StartNew(() =>
             {
                 if (Console.ReadKey().KeyChar == 'c')
                     cts.Cancel();
                 Console.WriteLine("Press any key to exit");
               });

               try
               {
                  Parallel.ForEach(invoices, po, invoice =>
                  {
```

```
            po.CancellationToken.ThrowIfCancellationRequested();

          }
        }
        catch (OperationCanceledException ex)
        {
          Console.WriteLine(ex.Message);
        }
```

7. To break or stop the loop while it is running after a certain condition is met or exception is thrown, you can use loopState.Break() or loopState.Stop() methods

```
Parallel.ForEach(invoices, po, (invoice, loopState) =>
{
       try
       {
           // Do some work
       }
       catch(Exception ex)
       {
              // Completes all iterations of all threads that are prior to the
              // current iterations and then exit the loop
              loopState.Break();

              // Stop all iterations as soon as convenient
              loopState.Stop();
       }
}
```

8. To handle error inside the Parallel loop, use following pattern

```
// ConcurrentQueue to enable thread safe.
var exceptions = new ConcurrentQueue<Exception>();

Parallel.ForEach(invoices, invoice =>
  {
    try
    {

    }
   catch (Exception ex)
    {
     exceptions.Enqueue(ex);
    }
   });

 // Throw the exceptions after the loop completes.
  if (exceptions.Count > 0) throw new AggregateException(exceptions);
```

9. To lock a block of code so that it can be accessible only one thread at a time, wrap the block of code with lock() statement.

```
object lockObject = new object(); // do not keep this statement inside loop
```

```
lock (lockObject)
{
    GetHtmlToGeneratePDFUsingPDFTable(pdfString);
}
```

10. To invoke multiple operations in parallel that is not part of a loop, use Parallel.Invoke method.

```
  // Perform three tasks in parallel on the source array
Parallel.Invoke(() =>
        {
            Console.WriteLine("Begin first task...");
            Method1();
        },  // close first Action

        () =>
        {
            Console.WriteLine("Begin second task...");
            Method2();
        }, //close second Action

        () =>
        {
            Console.WriteLine("Begin third task...");
            Method3();
        } //close third Action
           ); //close parallel.invoke
```

9. ADO.NET Guides

1. Always call .Close() and .Dispose() methods for SqlConnection

2. Use using blocks or read Point 8 or Performance Guides on how to avoid using blocks with each object by writing only one try and finally block and disposing all objects initialized before try block.

3. Use DataTable instead of DataSet in SqlDataAdapter's Fill method unless SQL statement or Stored procedure returns more than one result set.

4. Use SqlDataReader only for small set of data to quickly populate it or iterate through. Do not use this for large result set. It holds the connection open till all iterations of the object is done.

5. Use CommandBehavior.CloseConnection while calling ExecuteReader. cmd.ExecuteReader(CommandBehavior.CloseConnection), it ensures that the database

connection is closed when the reader is closed.

6. Always close and/or dispose SqlCommand and SqlDataAdapter objects.
7. Open database connection as late as possible and close database connection as early as possible.
8. Do specify connection pooling in the database connection string. Write the min and max pool size value as per your application need.

```
<connectionStrings>

    <add name="DbConnectionString" connectionString="Data
Source=192.168.50.38\CORESQLSERVER1;Initial Catalog=NTTACOREDEV;User ID=devuser;
         Password=devuser;Min Pool Size=5;Max Pool Size=1000;Pooling=true;Connect
Timeout=7200;"/>
            </connectionStrings>
```

9. Mention SqlCommand timeout for long running queries.
10. Avoid using SqlCommand(cmd).Parameters.AddWithValue method. Always explicitly define the SqlParameter and its type, set the data and use it.

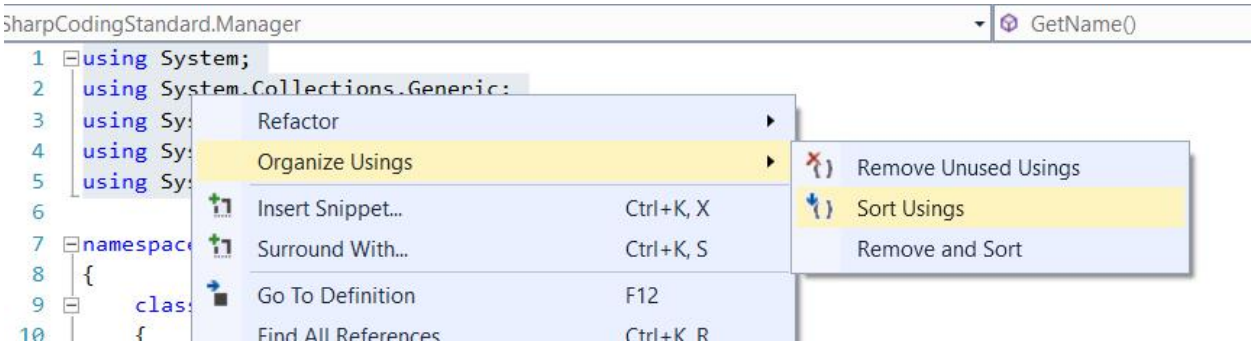
```
SqlParameter[] parameters = new SqlParameter[10];
parameters[0] = new SqlParameter("@parameterName",
 SqlDbType.VarChar, 30);

      parameters[0].Value = "some value";
      cmd.Parameters.AddRange(parameters);

           SqlParameter param = new SqlParameter("@parameterName",
           SqlDbType.VarChar, 30);
           param.Value = "Some value";
          cmd.Parameters.Add(param);
```

11. Always close the SqlDataReader object.
12. Use Transactions in ADO.NET very cautiously

10. Layout guides

1. Try to restrict the line width under 130 characters
2. Keep one line space between keywords like if, for, while
3. Add a space around any operators like +, - etc.
4. Prefer using opening and closing braces in a new line
5. Press Ctrl+K+D to auto format the code as per Visual Studio guidelines

6. Use Visual Studio context menu to sort, remove unused namespaces like below
s



7. Restrict use of #region for following

    a. Private fields and constants
    b. Nested class
    c. Interface implementation