C# Coding Rules

Author:	Renat Fakhrutdinov
Approved by:	Alexey Metelev

1. Table of contents

1. Table of contents 2
2. Preface 3
3. Naming Conventions and Style 4
4. Indention 6
5. Coding Practices 7
6. Project Settings and Project Structure 14
7. Framework Specific Guidelines 15
   7.1 Data Access 15
   7.2 ASP.NET and Web Services 15
   7.3 Multithreading 17
   7.4 Serialization 18
   7.5 Remoting 19
   7.6 Security 19
   7.7 System.Transactions 21
   7.8 Enterprise Services 21
8. Resources 23

2. Preface

This coding standard is mostly based on Microsoft Design Guidelines for Class Library Developers and other sources that are listed in section Resources. Sections 3 – 6 are **obligatory** while section 7 is advisable and can be regarded as the best practices. There might be situations where good design requires that you violate these guidelines. Such cases should be rare, and it is necessary that you provide a solid justification for your decision.

3. Naming Conventions and Style

1. Use Pascal casing for type and method names and constants:

```
public class SomeClass
{
   const int DefaultSize = 100;
   public SomeMethod()
   {}
}
```

2. Use camel casing for local variable names and method arguments:

```
   int number;
   void MyMethod(int someNumber)
   {}
```

3. Prefix interface names with I:

```
   interface IMyInterface
   {...}
```

4. Prefix private member variables with _. Use Camel casing for the rest of a member variable name following the _.

```
   public class SomeClass
   {
      private int _number;
   }
```

5. Suffix custom attribute types with **`Attribute`**.
6. Suffix custom exception types with **`Exception`**.
7. Suffix custom event argument types with **`EventArgs`**.
8. Suffix custom event handlers with **`EventHandler`**.
9. Do not use suffix **`Delegate`**.
10. Name methods using verb-object pair, such as **`ShowDialog()`**.
11. Methods with return values should have a name describing the value returned, such as **`GetObjectState()`**.
12. Use the C# aliases rather than types in the `System` namespace. For example:

```
object NOT Object
string NOT String
int NOT  Int32
```

13. With generics, use capital letters for types. Reserve suffix Type when dealing with the .NET type Type.

```
//Correct:
public class LinkedList<K,T>
{...}
```

14. Use meaningful namespaces such as the product name or the company name.
15. Try to avoid fully qualified type names. Use the `using` statement instead.
16. Avoid putting a `using` statement inside a namespace.
17. Group all framework namespaces together and put custom or third-party namespaces underneath separated by a line.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;

using MyCompany;
using MyControls;
```

18. Use delegate inference instead of explicit delegate instantiation.

```
public delegate void MyEventHandler();
public void SomeMethod()
{...}
MyEventHandler myHandler = SomeMethod;
```

19. Declare a local variable as close as possible to its first use.
20. A file name should reflect the class it contains.
21. When using partial types and allocating a part per file, name each file after the logical part that part plays. For example:

```
//In MyClass.cs
public partial class MyClass
{...}

//In MyClass.Designer.cs
public partial class MyClass
{...}
```

4. Indention

1. Maintain strict indentation. Do not use non-standard indentation, such as one space. Recommended values are 4 (four) spaces.
2. Indent comment at the same level of indentation as the code you are documenting.
3. All comments should pass spell checking.
4. All member variables should be declared at the top, with one line separating them from the properties or methods.

```
public class MyClass
{
   int    _number;
   string _name;

   public void SomeMethod1(int index, string message)
   {}

   public void SomeMethod2()
   {}
}
```

5. Always place an open curly brace (`{`) in a new line.
6. Always use a curly brace scope in an `if` statement, even if it conditions a single statement.
7. All classes in a file and methods and properties in a class shall be separated by a line.
8. All statements inside a `switch` statement shall be aligned with the beginning of constant expressions. `Case` and `break` shall be separated by an empty line:

```
int index = SomeMethod();

switch(index)
{
   case 1:
         DoCase1();
         break;

   case 2:
         DoCase2();
         break;

   default:
         DoDefault();
         break;
}
```

9. Lines shall not exceed 100 characters.

10. The following regions must exist in type definitions:

```
public class MyClass()
{
    #region Consts
    #enedregion

    #region Fields
    #enedregion

    #region Constructors
    #enedregion

    #region Properties
    #endregion

    #region Methods
    #endregion
}
```

5. Coding Practices

1. Avoid putting multiple classes in a single file.
2. A single file should contribute types to only a single namespace. Avoid having multiple namespaces in the same file.
3. Avoid methods with more than 5 arguments. Use structures for passing multiple arguments.
4. Do not manually edit any machine-generated code.
5. Avoid comments that explain the obvious.
6. Comment only operational assumptions, algorithm insights and so on.
7. Use method-level comments only as tool tips for other developers.

8. Never hard-code a numeric value; always declare a constant instead.
9. Use the `const` directive only on natural constants such as the number of days of the week.
10. Avoid using `const` on read-only variables. For that, use the `readonly` directive.

```
public class MyClass
{
   public const int DaysInWeek = 7;
   public readonly int Number;

   public MyClass(int someValue)
   {
      Number = someValue;
   }
}
```

11. Assert every assumption.

```
using System.Diagnostics;

object GetObject()
{...}

object someObject = GetObject();
Debug.Assert(someObject != null);
```

12. Every line of code should be walked through in a "white box" testing manner.
13. Catch only exceptions for which you have explicit handling.
14. In a `catch` statement that throws an exception, always throw the original exception (or another exception constructed from the original exception) to maintain the stack location of the original error:

```
catch(Exception exception)
{
   MessageBox.Show(exception.Message);
   throw; //Same as throw exception
}
```

15. Avoid using error code as method return values.
16. Use FCL exception types.
17. When defining custom exceptions derive them from `Exception and` provide custom serialization.
18. Avoid multiple `Main()` methods in a single assembly.
19. Make only the most necessary types `public`, mark others as `internal`.
20. Avoid friend assemblies, as they increase inter-assembly coupling.
21. Avoid code that relies on an assembly running from a particular location.

22. Minimize code in application assemblies (EXE client assemblies). Use class libraries instead to contain business logic.
23. Avoid providing explicit values for enums.

```
//Correct
public enum Color
{
   Red,
   Green,
   Blue
}
```

24. Always use zero-based arrays.
25. Do not provide `public` or `protected` member variables. Use properties instead.
26. You should mark class as `sealed` if it should not be derived. Otherwise mark `public` and `protected` methods and properties as `virtual`.
27. Never use unsafe code, except when using interop.
28. Use explicit casting. Use the `as` operator only if you check the result of casting operation:

```
MyType obj1 = new MyType();

ISomeInterface obj2 = obj1 as ISomeInterface;

if(obj2 != null)
{
   obj2.Method1();
}
else
{
   //Handle casting error
}
```

29. Always check a delegate for null before invoking it.
30. Do not provide public event member variables. Use event accessors instead.

```
public class MyPublisher
{
   private MyEventHandler _someEvent;

   public event MyEventHandler SomeEvent
   {
      add
      {
         _someEvent += value;
      }
      remove
      {
         _someEvent -= value;
      }
   }
}
```

31. Never return `null` when return type is a collection. Return empty collection instead.
32. Classes and interfaces should have at least 2:1 ratio of methods to properties.
33. Avoid interfaces with one member.
34. Do not have more than 20 members per interface.
35. Avoid events as interface members.
36. Expose interfaces on class hierarchies.
37. Prefer using explicit interface implementation.
38. Never hardcode strings that will be presented to end users. Use resources instead.
39. Never hardcode strings that might change based on deployment such as connection strings.
40. Use `string.Empty` instead of “”:

```
//Avoid
string name = "";
//Use
string name = string.Empty;
```

41. When dynamically building a string from several strings, use `StringBuilder`, not concatenations.
42. Avoid providing methods on structures.
43. Always provide a static constructor when providing static member variables.
44. Do not use late-binding invocation when early-binding is possible.
45. Never use `goto` unless in a switch statement fall-through.
46. Always have a `default` case in a `switch` statement. If you do not have default handling place assert statement there:

```
int index = SomeMethod();

switch(index)
{
   case 1:
         DoCase1();
         break;
   case 2:
         DoCase2();
         break;
   default:
         //Either
         DoDefault();
         //or
         Debug.Assert(false);
         break;
}
```

47. Do not use the `this` reference unless invoking another constructor from within a constructor.

```
//Example of proper use of "this"
public class MyClass
{
   public MyClass(String message)
   {}

   public MyClass()  : this("Hello")
   {}
}
```

48. Do not use the base word to access base class members unless you wish to resolve a conflict with a subclasses member of the same name or when invoking a base class constructor.

```
//Example of proper use of "base"
public class Cat
{
   public Cat(String name)
   {}

   virtual public void Meow(int howLong)
   {}
}

public class Angora : Cat
{
   public Angora(String name): base(name)
   {}

   override public void Meow(int howLong)
   {
      base. Meow(howLong);
   }
}
```

49. Do not use `GC.AddMemoryPressure()`.

50. Implement Dispose() and Finalize() methods based on the following template:

```
public class MyClass : IDisposable
{
   ~MyClass ()
   {
      Dispose(false);
   }

   // This public method can be called to deterministically
   // release an unmanaged resource.
   public void Dispose()
   {
      // Because the object is explicitly cleaned up, stop the
      // garbage collector from calling the Finalize method.
      GC.SuppressFinalize(this);

      // Call the method that actually does the cleanup.
      Dispose(true);
   }

   // The common method that does the actual cleanup.
   // Finalize, Dispose, and Close call this method.
   protected virtual void Dispose(bool disposing)
   {
      // Synchronize threads calling Dispose/Close simultaneously
      lock(this)
      {
         if(disposing)
            {
               // The object is being explicitly disposed of/closed, not
               // finalized. It is therefore safe for code in this if
               // statement to access fields that reference other
               // objects because the Finalize method of these other objects
               // hasn't yet been called.
            }

         // The object is being disposed of/closed or finalized.
         if(IsValid)
            {
               // If the resource is valid, close the unmanaged resource.
               FreeResource();

               // Set the flag to false. This precaution
               // prevents the possibility of calling FreeResource twice.
               IsValid = false;
            }
      }
   }
}
```

51. Always run code unchecked by default (for the sake of performance), but explicitly in checked mode for overflow- or underflow-prone operations:

```
int CalcPower(int number, int power)
   {
      int result = 1;
      for(int count = 1;count <= power;count++)
         {
            checked
            {
               result *= number;
            }
        }
      return result;
   }
```

52. Avoid explicit code exclusion of method calls (`#if…#endif`). Use conditional methods instead:

```
public class MyClass
{
   [Conditional("MySpecialCondition")]
   public void MyMethod()
   {}
}
```

53. Avoid casting to and from `System.Object` in code that uses generics. Use constraints or the as operator instead:

```
class SomeClass
{}
//Avoid:
class MyClass<T>
{
   void SomeMethod(T t)
   {
      Object temp = t;
      SomeClass obj = (SomeClass)temp;
   }
}

//Correct:
class MyClass<T> where T : SomeClass
{
   void SomeMethod(T t)
   {
      SomeClass obj = t;
   }
}
```

54. Do not define constraints in generic interfaces. Interface level-constraint can often be replaced by strong-typing.

```
public class Employee
{...}

//Avoid:
public interface IList<T> where T : Employee
{...}

//Correct:
public interface IEmployeeList : IList<Employee>
{...}
```

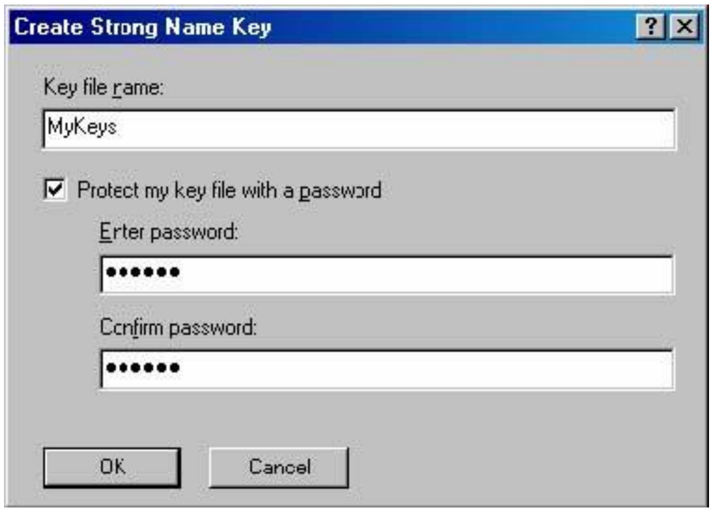
55. Do not define method-specific constraints in interfaces.
56. Do not define constraints in delegates.
57. If a class or a method offers both generic and non generic flavours, always prefer using the generics flavour.

6. Project Settings and Project Structure

1. Always build your project with warning level 4
2. Treat warnings as errors in the Release build (note that this is not the default of Visual Studio).
3. Always explicitly state your supported runtime versions in the application configuration file.

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v2.0.5500.0"/>
    <supportedRuntime version="v1.1.5000.0"/>
  </startup>
</configuration>
```

4. Avoid explicit pre-processor definitions `(#define`). Use the project settings for defining conditional compilation constants.
5. Do not put any logic inside `AssemblyInfo.cs`.
6. Do not put any assembly attributes in any file besides `AssemblyInfo.cs`.
7. Populate all fields in `AssemblyInfo.cs` such as company name, description, and copyright notice.
8. All assembly references in the same solution should use relative path.
9. Avoid cyclic references between assemblies.
10. Use password -protected keys.


Create Strong Name Key

Key file name:

MyKeys

☑ Protect my key file with a password

Enter password:

••••••

Confirm password:

••••••

7. Framework Specific Guidelines

7.1 Data Access

1. Always use typed data sets or data tables . Avoid raw ADO.NET.
2. Always use transactions when accessing a database.

   a) Always use Enterprise Services or `System.Transactions` transactions.

   b) Do not use ADO.NET transactions by enlisting the database explicitly.
3. Always use transaction isolation level set to `Serializable`. Management decision is required to use anything else.
4. Do not use the Data Source window to drop connections on windows forms, ASP.NET forms or web services. Doing so couples the presentation tier to the data tier.
5. Run components accessing SQL Server under separate identity from that of the calling client.
6. Always wrap your stored procedures in a high level, type safe class. Only that class invokes the stored procedures. Let Visual Studio 2005 type-safe data adaptors automate as much of that as possible.

7.2 ASP.NET and Web Services

1. Avoid putting code in ASPX files of ASP.NET. All code should be in the code-beside partial class.
2. Code in code beside partial class of ASP.NET should call other components rather than contain direct business logic.
3. Always check a session variable for null before accessing it.

4. In transactional pages or web services, always store session in SQL server.
5. Avoid setting the Auto-Postback property of server controls in ASP.NET to True.
6. Turn on Smart Navigation for ASP.NET pages.
7. Provide interfaces for web services.
8. Always provide namespace and service description for web services.
9. Always provide a description for web methods.
10. When adding a web service reference, provide meaningful name for the location.
11. In both ASP.NET pages and web services, wrap a session variables in a local property. Only that property is allowed to access the session variable, and the rest of the code uses the property, not the session variable:

```
public class Calculator : WebService
{
   int Memory
   {
      get
      {
         int memory = 0;
         Object state = Session["Memory"];

         if(state != null)
            {
               memory = (int)state;
            }

         return memory;
      }
      set
      {
         Session["Memory"] = value;
      }
   }

   [WebMethod(EnableSession=true)]
   public void MemoryReset()
   {
      Memory = 0;
   }
}
```

12. Always modify client-side web service wrapper class to support cookies, since you have no way of knowing whether the service uses Session state or not.

```
public class Calculator : SoapHttpClientProtocol
{
   public Calculator()
   {
      CookieContainer = new System.Net.CookieContainer();
      Url = ...;
   }
}
```

7.3 Multithreading

1. Manage asynchronous call completion on a callback method. Do not wait, poll, or block for completion.
2. Always name your threads. The name is traced in the debugger Threads window, making debug sessions more productive.

```
Thread currentThread = Thread.CurrentThread;
string threadName = "Main UI Thread";
currentThread.Name = threadName;
```

3. Do not call `Suspend()` or `Resume()` on a thread.
4. Do not call `Thread.Sleep()`, except in the following conditions:
5. a) `Thread.Sleep(0)` is acceptable optimization technique to force a context switch.
   b) `Thread.Sleep()` is acceptable in testing or simulation code.
6. Do not call `Thread.SpinWait()`.
7. Do not call `Thread.Abort()` to terminate threads. Use a synchronization object instead to signal the thread to terminate.
8. Avoid explicitly setting thread priority to control execution. You can set thread priority based on task semantic, such as `ThreadPriority.BellowNormal` below normal for a screen saver.
9. Do not read the value of the `ThreadState` property. Use `Thread.IsAlive()` to determine whether the thread is dead or alive.
10. Do not rely on setting the thread type to background thread for application shutdown. Use a watchdog or other monitoring entity to deterministically kill threads.
11. Do not use thread local storage unless thread affinity is guaranteed.
12. Do not call `Thread.MemoryBarrier()`.
13. Never call `Thread.Join()` without checking that you are not joining your own thread.

```
void WaitForThreadToDie(Thread thread)
{
   Debug.Assert(Thread.CurrentThread.GetHashCode() != thread.GetHashCode());
   thread.Join();
}
```

14. Always use the `lock()` statement rather than explicit Monitor manipulation.
15. Always encapsulate the `lock()` statement inside the object it protects.

```
public class MyClass
{
   public void DoSomething()
   {
      lock(this)
      {...}
   }
}
```

16. You can use synchronized methods instead of writing the `lock()` statement yourself.
17. Avoid increasing the maximum number of threads in the thread pool.
18. Never stack lock statements because that does not provide atomic locking. Use `WaitHandle.WaitAll()` instead.

```
MyClass obj1 = new MyClass();
MyClass obj2 = new MyClass();
MyClass obj3 = new MyClass();

//Do not stack lock statements
lock(obj1)
lock(obj2)
lock(obj3)
{
   obj1.DoSomething();
   obj2.DoSomething();
   obj3.DoSomething();
}
```

7.4 Serialization

1. Prefer the binary formatter.
2. Mark serialization event handling methods as private.
3. Always mark non-sealed classes as serializable.
4. When implementing `IDeserializationCallback` on a non-sealed class, make sure to do so in a way that allowed subclasses to call the base class implementation of `OnDeserialization()`.
5. Always mark un-serializable member variables as non serializable.
6. Always mark delegates on a serialized class as non-serializable fields:

```
[Serializable]
public class MyClass
{
   [field:NonSerialized]
   public event EventHandler MyEvent;
}
```

7.5 Remoting

1. Prefer administrative configuration to programmatic configuration.
2. Always implement `IDisposable` on single call objects.
3. Always prefer a TCP channel and a binary format when using remoting, unless a firewall is present.
4. Always provide a `null` lease for a singleton object.

```
public class MySingleton : MarshalByRefObject
{
   public override Object InitializeLifetimeService()
   {
      return null;
   }
}
```

5. Always provide a sponsor for a client activated object. The sponsor should return the initial lease time.
6. Always unregister the sponsor on client application shutdown.
7. Always put remote objects in class libraries.
8. Avoid using SoapSuds.
9. Avoid hosting in IIS.
10. Avoid using uni-directional channels.
11. Always load a remoting configuration file in Main() even if the file is empty, and the application does not use remoting:

```
static void Main()
{
   RemotingConfiguration.Configure("MyApp.exe.config");

   /* Rest of Main() */
}
```

12. Avoid using `Activator.GetObject()` and `Activator.CreateInstance()` for remote objects activation. Use `new` instead.
13. Always register port 0 on the client side, to allow callbacks.
14. Always elevate type filtering to full on both client and host to allow callbacks.

7.6 Security

1. Always demand your own strong name on assemblies and components that are private to the application, but are public (so that only you can use them):

```
public class PublicKeys
{
   public const String MyCompany =
   "1234567894800000940000000602000000240000" +
   "52534131000400000100010007D1FA57C4AED9F0"+
   "A32E84AA0FAEFD0DE9E8FD6AEC8F87FB03766C83"+
   "4C99921EB23BE79AD9D5DCC1DD9AD23613210290"+
   "0B723CF980957FC4E177108FC607774F29E8320E"+
   "92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99"+
   "285D622CAA652C1DFAD63D745D6F2DE5F17E5EAF"+
   "0FC4963D261C8A12436518206DC093344D5AD293";
}

[StrongNameIdentityPermission(SecurityAction.LinkDemand, PublicKey =
                                    PublicKeys.MyCompany)]
public class MyClass
{...}
```

2. Apply encryption and security protection on application configuration files.
3. When importing an interop method, assert unmanaged code permission, and demand appropriate permission instead.

```
[DllImport("user32",EntryPoint="MessageBoxA")]
private static extern int Show(IntPtr handle,string text,string caption,
                                      int msgType);

[SecurityPermission(SecurityAction.Assert,UnmanagedCode = true)]
[UIPermission(SecurityAction.Demand,
   Window = UIPermissionWindow.SafeTopLevelWindows)]
public static void Show(string text,string caption)
{
   Show(IntPtr.Zero,text,caption,0);
}
```

4. Do not suppress unmanaged code access via the SuppressUnmanagedCodeSecurity attribute.
5. Do not use the /unsafe switch of TlbImp .exe. Wrap the RCW in managed code so that you could assert and demand permissions declaratively on the wrapper.
6. On server machines, deploy a code access security policy that grants only Microsoft, ECMA, and self (identified by a strong name) full trust. Code originating from anywhere else is implicitly granted nothing.
7. On client machines, deploy a security policy which grants client application only the permissions to execute, to call back the server and to potentially display user interface. When not using ClickOnce, client application should be identified by a strong name in the code groups.
8. To counter a luring attack, always refuse at the assembly level all permissions not required to perform the task at hand.

```
[assembly:UIPermission(SecurityAction.RequestRefuse,
 Window=UIPermissionWindow.AllWindows)]
```

9. Always set the principal policy in every `Main()` method to Windows

```
public class MyClass
{
   static void Main()
   {
      AppDomain currentDomain = Thread.GetDomain();
      currentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
   }
//other methods
}
```

10. Never assert a permission without demanding a different permission in its place.

7.7 System.Transactions

1. Always dispose of a `TransactionScope` object.
2. Inside a transaction scope, do not put any code after the call to `Complete()`.
3. Do not call `Complete()` multiple times.
4. When setting the ambient transaction, always save the old ambient transaction and restore it when you are done.
5. In Release builds, never set the transaction timeout to zero (infinite timeout).
6. When cloning a transaction, always use `DependentCloneOption.BlockCommitUntilComplete`.
7. Create a new dependent clone for each worker thread. Never pass the same dependent clone to multiple threads.
8. Do not pass a transaction clone to the `TransactionScope`'s constructor.

7.8 Enterprise Services

1. Do not catch exceptions in a transactional method. Use the `AutoComplete` attribute.
2. Do not call `SetComplete()`, `SetAbort()`, and the like. Use the `AutoComplete` attribute.

```
[Transaction]
public class MyComponent : ServicedComponent
{
   [AutoComplete]
   public void MyMethod(Int64 objectIdentifier)
   {...}
}
```

3. Always override `CanBePooled` and return true (unless you have a good reason not to return to pool)

```
public class MyComponent : ServicedComponent
{
   protected override Boolean CanBePooled()
   {
      return true;
   }
}
```

4. Always call `Dispose()` explicitly on a pooled objects unless the component is configured to use JITA as well. Never call `Dispose()` when the component uses JITA. Always set authorization level to application and component. Set authentication level to privacy on all applications.

```
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(
              true, //Authorization
              AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
              Authentication=AuthenticationOption.Privacy,
              ImpersonationLevel=ImpersonationLevelOption.Identify)]
```

5. Set impersonation level on client assemblies to `Identity`.
6. Always set `ComponentAccessControl` attribute on serviced components to `true` (the default is `true`):

```
[ComponentAccessControl]
public class MyComponent : ServicedComponent
{...}
```

7. Always add to the `Marshaler` role the Everyone user:

```
[assembly: SecurityRole("Marshaler",SetEveryoneAccess = true)]
```

8. Apply `SecureMethod` attribute to all classes requiring authentication:

```
[SecureMethod]
public class MyComponent : ServicedComponent
{...}
```

8. Prefixes for UI Controls

8.1 DevExpress Controls

UI Control	**Prefix**
System.Windows.Forms.Label	lbl
System.Windows.Forms.Button	btn
System.Windows.Forms.BindingSource	bindSrc
System.Windows.Forms.DataGridViewTextBoxColumn	dgrvTxtCol
System.Windows.Forms.Panel	pnl
DevExpress.XtraEditors.LookUpEdit	lookUp
DevExpress.XtraEditors.SimpleButton	btn
DevExpress.XtraEditors.TimeEdit	time
DevExpress.XtraEditors.TextEdit	txt
DevExpress.XtraEditors.RadioGroup	rdgr
DevExpress.XtraEditors.DXErrorProvider.DXErrorProvider	errorProvider
DevExpress.XtraEditors.DXErrorProvider.DXValidationProvider	validationProvider
DevExpress.XtraGrid.GridControl	grd
DevExpress.XtraGrid.Views.Grid.GridView	grdv
DevExpress.XtraGrid.Columns.GridColumn	col
DevExpress.XtraEditors.Repository.RepositoryItemButtonEdit	rpiBtn
DevExpress.XtraEditors.Repository.RepositoryItemLookUpEdit	rpiLookUp
DevExpress.XtraEditors.Repository.RepositoryItemTextEdit	rpiTxt
DevExpress.XtraEditors.Repository.RepositoryItemCheckEdit	rpiChk
DevExpress.XtraEditors.SpinEdit	spin
DevExpress.XtraEditors.CheckEdit	chk
System.Windows.Forms.ToolStrip	ts for example: ToolStripLabel – tsLbl ToolStripTextBox – tsTxt etc
DevExpress.XtraEditors.DateEdit	de

9. Resources

1. The IDesign C# Coding Standard (www.idesign.net)
2. Jeffrey Richter, "Applied Microsoft .NET Framework Programming"

3. FxCop rules (www.gotdotnet.com/team/fxcop)