Introduction

This document serves as a definition of Tecuity's coding standards for source code in the C# Programming Language. A source file may only be said to be in the *Tecuity Style* if it adheres to the standards contained herein.

Notes

- Although this coding standard attempts to outline all of the standards desirable for source code adhering to the *Tecuity Style*, a majority of the standards contained herein are contained in Microsoft's Framework Design Guidelines.
    - Additional coding standards not contained in this document may be sought out there
    - If a standard cannot be identified either in this document or the *Framework Design Guidelines*, you can request that a standard be created.

Terminology

- **Pascal Case** is a casing strategy where a name consists of one or more words, and each word (including the first character) is uppercase.
    - **Example:** Connection
    - **Example:** MyConnection
- **Camel Case** is two casing strategies. The first strategy, named *Upper Camel Case* is identical to *Pascal Case*. The second strategy, named *Lower Camel Case* is similar to *Pascal Case* with the exception of the first character of the name being lowercase. For the purposes of this standard, all references to *Camel Case* will refer to *Lower Camel Case*.
    - **Example:** connection
    - **Example:** myConnection
- **Member** is a part of a class. These include:
    - Property
    - Field
    - Constructor
    - Method
    - Nested Class

TLDR;

- Do:
    - Do use *Pascal Case* when naming *classes* and *interfaces*
    - Do use *Pascal Case* when naming *class members*
    - Do use *Pascal Case* when naming namespaces
    - Do use acronyms with three (3) or more characters in length as words and the appropriate casing
    - Do use *Upper Case* when using an acronym two (2) characters in length
    - Do name classes such that they are the same name as their containing file
    - Do use *Camel Case* when naming local variables and method parameters
    - Do prefix interface names with the uppercase letter 'I'
    - Do name interfaces using nouns or adjectives

    - Do suffix attribute class names with the 'Attribute' identifier
    - Do use meaningful names as identifiers
    - Do use singular nouns for enumeration names
    - Do prefix all generic type arguments with the letter 'T'
    - Do name namespaces using the project name followed by the folders containing the source files
    - Do always use curly braces ('{}') when creating a scoped block. No single line blocks without curly braces
    - Do declare class properties and fields at the top of the class
    - Do provide comments for lines and processes which cannot be quickly understood
    - Do split method chaining over several lines
    - Do separate single line comment start characters ('//') with a space before the comment
    - Do place multiple line comment characters ('/*', '*/') on their own lines with text indented
    - Do declare type when creating variables
    - Do use predefined type primitives
    - Do contain only one (1) class per file
    - Do maybe Specify enumeration values
    - Do suffix exception class names with the 'Exception' identifier
    - Do use tabs to indent lines
    - Do provide a single, empty line between each statement and group of statements
    - Do separate methods with a single blank line
    - Do use each class and method for only a single responsibility

- Do Not:

    - Do not use All Upper Case For Identifiers, even constants or readonly
    - Do not use Hungarian notation
    - Do not use underscores when defining a name
    - Do not use C# keywords as identifiers
    - Do not use abbreviations or contractions when naming
    - Do not suffix an enumeration with the 'Enum' or 'Enumeration' identifier
    - Do not suffix a flag enumeration with the 'Flag' or 'Flags' identifier
    - Do not use the 'var' keyword
    - Do not create static classes
    - Do not use class properties or fields when a scoped variable will suffice
    - Do not specify the numeric type of enumerations
    - Do not use jump statement: continue, goto

Naming

> # DO
>
> **Do use *Pascal Case* when naming *classes* and *interfaces***
>
> **Reasoning:** This is consistent with the *Framework Design Guidelines*
>
> **Example:**

```
public class MyClass
public interface IServiceProvider
```

DO

Do use *Pascal Case* when naming *class members*

Reasoning: This is, in part, consistent with the *Framework Design Guidelines*. In addition to the *Framework Design Guidelines*, using *Pascal Case* on all class members helps to immediately identify the members of a class, instead of causing any confusion when using a casing strategy which is not specific to classes.

Example:

```
public static void Main(string[] args)
public string MyString { get; set; }
private string MyString { get; set; }
private string MyString;
```

Exception

When using a class member field as a backing value for a property, use the same name as the property as the name of the backing field prefixed with an underscore ('_'). Note: don't use backing field unless absolutely needed (e.g. additional logic needed for getting and/or setting variable members).

Reasoning: This helps to immediately identify the property which is backed by the field as well as immediately identify the field which backs the property.

Example:

```
private string _MyString;
public string MyString
{
        get
        {
                return _MyString;
        }
        set
        {
                _MyString = value;
        }
}
```

DO NOT

Do not use All Upper Case For Identifiers, even constants or readonly

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
// Correct
public readonly double GravitationalConstant;

// Not Correct
public readonly double GRAVITATIONALCONSTANT;
```

DO

Do use *Pascal Case* when naming namespaces

Reasoning: This is consistent with the *Framework Design Guidelines*

Example:

```
namespace Project.Services.Forms
```

DO

Do use acronyms with three (3) or more characters in length as words and the appropriate casing

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
// Correct
public class XmlDocument
public string XmlString { get; }
string xmlString = "";
```

```
// Not Correct
public class XMLDocument
public string XMLString { get; }
string xMLString = "";
```

DO

Do use *Upper Case* when using an acronym two (2) characters in length

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
// Correct
public class UIControl

// Not Correct
public class UiControl
```

DO

Do name classes such that they are the same name as their containing file

Reasoning: This is consistent with the *Framework Design Guidelines*, and prevent losing classes which are in files named something other than the class name.

Example:

```
// Correct
Filename: SystemService.cs
public class SystemService

// Not Correct
Filename: Classes.cs
public class SystemService
```

DO

Do use *Camel Case* when naming local variables and method parameters

Reasoning: This is consistent with the *Framework Design Guidelines*, and helps immediately identify that a variable is scoped, and not a member of a class.

Example:

```
string myString = "";
```

Example:

```
public void HandleString(string myString)
```

DO NOT

Do not use Hungarian notation

Reasoning: This is consistent with the *Framework Design Guidelines*.

Reasoning: Very little additional meaning is provided by such notation.

Reasoning: In the event a variable's type changes, variable must be renamed to match new type.

Example:

```
// Correct
string name = "";
int counter = 0;

// Not Correct
string strName = "";
int iCounter = 0;
```

DO

Do prefix interface names with the uppercase letter 'I'

Reasoning: This is consistent with the *Framework Design Guidelines*, and helps immediately identify an interface.

Example:

```
public interface IService
```

DO

Do name interfaces using nouns or adjectives

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
public interface IShape
{

}

public interface IGroupable
{

}
```

DO

Do suffix attribute class names with the 'Attribute' identifier

Reasoning: This is consistent with the *Framework Design Guidelines*, and helps immediately identify an attribute class.

Example:

```
public class MyAttribute : Attribute
```

DO NOT

Do not use underscores when defining a name

Reasoning: This prevents having multiple identical names differed only by an underscore.

Example:

```
// Not Correct
string _myString = "";
string my_other_string = "";
```

Exception

When using a class member field as a backing value for a property, using the same name as the property prefixed with an underscore ('_') helps identify the one instance where the same name is desired.

DO NOT

Do not use C# keywords as identifiers

Reasoning: This prevents confusion as keywords will not accurately describe the value being stored.

Example:

```
// Correct
string firstName = "";

// Not Correct
string @string = "";
```

DO NOT

Do not use abbreviations or contractions when naming

Reasoning: This prevents any confusion caused by multiple possible meanings for abbreviations and contractions.

Example:

```
// Correct
public class Window
public Window Window { get; }
```

```
// Not Correct
public class UserGrp
public UserGroup UserGrp { get; }
```

Exception

When the abbreviation is commonly used and well known -- primarily only 'Id'.

Example:

```
public int RecordId { get; }
```

DO

Do use meaningful names as identifiers

Reasoning: This is consistent with the *Framework Design Guidelines*, and immediately identifies the value(s) stored.

Example:

```
// Correct
IEnumerable<Customer> seattleCustomers = customers.Where(customer =>
customer.City == "Seattle");

// Not Correct
IEnumerable<Customer> someCustomers = customers.Where(customer =>
customer.City == "Seattle");
```

DO NOT

Do not suffix an enumeration with the 'Enum' or 'Enumeration' identifier

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
// Correct
public enum Item
{
```

```
        One,
        Two
}

// Not Correct
public enum ItemEnum
{
        One,
        Two
}
```

DO NOT

Do not suffix a flag enumeration with the 'Flag' or 'Flags' identifier

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
using System;

// Correct
[Flags]
public enum ColorHues
{
   None = 0,
   Black = 1,
   Red = 2,
   Green = 4,
   Blue = 8
}

// Not Correct
[Flags]
public enum ColorHueFlags
{
   None = 0,
   Black = 1,
   Red = 2,
   Green = 4,
   Blue = 8
}
```

DO

Do use singular nouns for enumeration names

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
public enum Item
{
        One,
        Two
}
```

Exception

When creating an enumeration for use as a flag, use a plural noun for the enumeration name.

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
using System;

[Flags]
public enum ColorHues
{
   None = 0,
   Black = 1,
   Red = 2,
   Green = 4,
   Blue = 8
}
```

DO

Do prefix all generic type arguments with the letter 'T'

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
public class MyClass<TArgument, TResult>
```

Example:

```
public void DoStuff<TArgument, TResult>()
```

DO

Do name namespaces using the project name followed by the folders containing the source files

Reasoning: This is consistent with the *Framework Design Guidelines*, and helps immediately locate source files with only the namespace available

Example:

```
Path: Project/Services/Forms

namespace Project.Services.Forms
```

Exception

When desiring additional organization on the filesystem but not when coding, namespaces may stop short of the full path.

Reasoning: Keeping source code well organized is ideal, but having many different namespaces which must be used is not always ideal.

Example:

```
Path: Project/Services/Forms
Path: Project/Services/Actions
Path: Project/Services/Data

namespace Project.Services
```

Layout

DO

Do always use curly braces ('{}') when creating a scoped block. No single line blocks without curly braces

Reasoning: This explicitly identifies where the block begins and ends.

Note: This includes single line if/else if/else blocks.

Example:

```
// Correct
if (true)
{
        // Process condition check result
}

// Not Correct
if (true)
  // Process condition check result
```

Example:

```
using (IResource resource = new Resource())
{
        // Use resource
}
```

DO

Do declare class properties and fields at the top of the class

Reasoning: This is a generally accepted practice, and helps keep classes organized.

Example:

```
public class MyClass
{
        public string MyString { get; set; }
        public string MyOtherString { get; set; }

        private string OtherValue;

        public MyClass()
        {

        }
}
```

Exception

When entering a scoped block, a blank line is not required after the opening curly brace ('{').

DO

Do provide comments for lines and processes which cannot be quickly understood

Reasoning: This helps your colleages understand the code at hand without having to carefully read through the entire operation. Several lines of comments are permitted and encouraged, if needed.

Example:

```
// Get Complex Results
// Then Check Them For 'X'
// Get Only 'Y' From Results
// Convert 'Y' Into Usable Form
GetComplicateResultSet()
        .PerformAnotherCheck()
        .PerformFiltering()
        .DoMoreStuff();
```

DO

Do split method chaining over several lines

Reasoning: This cleans the chained operation and makes it easier to see where one part of the chain ends and another begins.

Example:

```
GetComplicateResultSet()
        .PerformAnotherCheck()
        .PerformFiltering(item =>
                item.DoItemStuff())
        .DoMoreStuff();
```

DO

Do separate single line comment start characters ('//') with a space before the comment

Reasoning: This makes it easier to see the start of the comment text.

Example:

```
// This is a comment
```

DO

Do place multiple line comment characters ('/*', '*/') on their own lines with text indented

Reasoning: This makes it easier to see the start and end of the comment text.

Example:

```
/*
        This is a comment
*/
```

Usage

DO NOT

Do not use the 'var' keyword

Reasoning: This abstracts away the type, and makes code harder to follow and understand.

Example:

```
var result = GetResult();
```

DO

Do declare type when creating variables

Reasoning: This immediately identifies the type and makes code easier to follow and understand.

Example:

```
string result = GetResult();
```

DO NOT

Do not create static classes

Reasoning: This is considered an anti-pattern. Instead of static classes, using the singleton pattern or dependency injection is recommended.

Example:

```
static class CoolClass
{
  // Member methods, variables, etc...
}
```

Alternative Example (Singleton):

```
// NOTE: This is not thread safe
class Singleton
{
      private static Singleton _Instance;

      protected Singleton()
      {
          // ...
      }

      public static CityBO Instance
      {
          get
          {
              if (_Instance == null)
              {
                  _Instance = new Singleton();
              }

              return _Instance;
          }
      }
```

```
        // ...
}
```

DO NOT

Do not use class properties or fields when a scoped variable will suffice

Reasoning: This keeps the class clutter free when only one method or other scope requires the use of a variable.

DO

Do use predefined type primitives

Reasoning: This is consistent with the *Framework Design Guidelines*.

Example:

```
// Correct
string result;
int count;
string countString = string.Join(", ", names);
int index = int.Parse(input);

// Not Correct
String result;
Int32 count;
string countString = String.Join(", ", names);
int index = Int32.Parse(input);
```

DO

Do contain only one (1) class per file

Reasoning: This is consistent with the *Framework Design Guidelines* and prevent losing track of classes not in their own files.

Exception

Nested classes

Example:

```
class Person
{
      private class NestedClass
      {
          // ...
      }
}
```

DO MAYBE

Do maybe Specify enumeration values

Reasoning: Enumerations do not need specific numeric values to function properly. In general values should not be provided, and should be inferred by the order of the enumeration's values. Providing numeric values is acceptable when absolutely necessary.

Example:

```
// Correct
public enum Direction
{
        North,
        East,
        South,
        West
}

// Acceptable, But Not Recommended
public enum Direction
{
        North = 1,
        East = 2,
        South = 3,
        West = 5
}
```

Exception

Flag enumerations are required to have specific numeric values (32 values maximum).

Example:

```
using System;

[Flags]
public enum ColorHues
{
   None = 0,
   Black = 1,
   Red = 2,
   Green = 4,
   Blue = 8
}
```

DO NOT

Do not specify the numeric type of enumerations

Reasoning: Enumeration will use integers internally automatically. If you need to specify a larger data type, your enumeration likely has too many values and should be reconsidered.

Example:

```
// Correct
public enum Direction
{
        North,
        East,
        South,
        West
}

// Not Correct
public enum Direction : long
{
        North,
        East,
        South,
        West
}

// Not Correct
public enum Direction : long
{
        North = 1,
        East = 2,
        South = 3,
        West = 5
}
```

DO

Do suffix exception class names with the 'Exception' identifier

Reasoning: This is consistent with the *Framework Design Guidelines*, and immediately helps identify exception classes.

Example:

```
using System;

public class BarcodeReadException : Exception
```

DO NOT

Do not use jump statement: continue, goto

Reasoning: These jump statements interrupt the natural flow of an application. The use of 'goto' is considered bad industry practice, and should be completely avoided. The use of 'continue' interrupt the natural cycle of a loop doing what should have been accomplished by regular flow statements. It is much more difficult to keep 'continue' straight than it is to use normal flow control that is used everywhere else in the application.

DO

Do use tabs to indent lines

Reasoning: Using tabs instead of spaces for line indentation provides a number of benefits: they are easier and quicker to navigate when moving the cursor through the start of the line; they may be adjusted based on personal preference for either larger or smaller indentation; they replace the preferred number of spaces with only a single character ('\t').

DO

Do provide a single, empty line between each statement and group of statements

Reasoning: Splitting up statements and groups of statements provides significantly better readability and maintainability as statements are split apart by operation. Several very closely related statements may be grouped together, yet once that similarity ends, so too should the group, and a single, empty line be entered before proceeding.

Example:

```
// Correct
string fullName = customer.FirstName + customer.LastName;

IEnumerable<Order> orders = GetOrders(fullName);

ProcessOrders(orders);

// Correct
string firstName = customer.FirstName;
string lastName = customer.LastName;

IEnumerable<Order> orders = GetOrders(firstName + lastName);

ProcessOrders(orders);

// Not Correct
string firstName = customer.FirstName;
string lastName = customer.LastName;
string fullName = customer.FirstName + customer.LastName;
IEnumerable<Order> orders = GetOrders(fullName);
ProcessOrders(orders);
```

DO

Do separate methods with a single blank line

Reasoning: This helps identify where methods begin and end.

Example:

```
public void MyMethod()
{

}

public void MyOtherMethod()
{

}
```

DO

Do use each class and method for only a single responsibility

Reasoning: This is an industry best practice. It keeps the source code manageable as each piece is only performing a single function. As issues arise, this separation of concerns causes troubleshooting to be significantly easier as there will likely be a limited number of places (ideally only one) where a particular bug could exist, making it simpler to locate and resolve.