Travel n Study Coding Policy

Version 1.0

Revision History

Date	Version	Description	Author
2012-10-21	1.00	First version	Javier Hualpa

Contents

1. Introduction

1.1 Purpose of this document

The purpose of this document is to communicate a common practice that will affect how easy is to understand the code, review it and revise it months after you write it. It will also affect how easy is for others to read, understand, and modify code once any team member is not present or available.
It is recommended that all team members get familiar with the coding style policy and decide whether to apply it manually or by using the code style enforcer tool that automatically suggests mismatching with the policy and ways to correct them.
The project will apply a subset of the IDesign Coding Standard and the Code Style Enforcer tool by Joel Fjordén that detects mismatches against the standard mentioned.

1.2 Document organization

The document is organized as follows:

- Section 1, *Introduction*, describes the contents of this policy and its purpose.
- Section 2, *Naming Conventions and Style*, explains rules to represent the logical structure of the code in an accurately and consistently.
- Section 3, *Coding Practices*, explains techniques for achieving maximum results for code development.

1.3 Intended Audience

The intended audience is:

- All team members of the Travel n Study project.

1.4 Scope

This document addresses the coding style, conventions and practices for the project Travel n Study in the C# programming language. The conventions discussed can be applied to any other average C# project. It does not cover specific framework design guidelines neither source code settings or project structure. However it forces some code layout and file structure that contributes to distributed development.

1.5 References

[1] IDesign coding Standard: http://www.idesign.net/Downloads/GetDownload/1985
[2] Code Style Enforcer: http://joel.fjorden.se/static.php?page=CodeStyleEnforcer

2. Naming Conventions and Style

- Use Pascal casing for type and method names and constants:

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

- Use camel casing for local variable names and method arguments.

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

- Prefix interface names with **I**

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

- Prefix private member variables with **m_**. Use Pascal casing for the rest of a member variable name following the **m_**.

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

- Suffix custom attribute classes with **Attribute**.

- Suffix custom exception classes with **Exception**.

- Name methods using verb-object pair, such as **ShowDialog()**.

- Methods with return values should have a name describing the value returned, such as **GetObjectState()**.

- Use descriptive variable names.
  a) Avoid single character variable names, such as **i** or **t**. Use **index** or **temp** instead.
  b) Avoid using Hungarian notation for public or protected members.
  c) Do not abbreviate words (such as **num** instead of **number**).

- Always use C# predefined types rather than the aliases in the **System** namespace. For example:

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

- With generics, use capital letters for types. Reserve suffixing **Type** when dealing with the .NET type Type.

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

//Avoid:
public class LinkedList<KeyType,DataType>
{...}
```

- Use meaningful namespaces such as the product name or the company name.

- Avoid fully qualified type names. Use the **using** statement instead.
- Avoid putting a **using** statement inside a namespace.
- Group all framework namespaces together and put custom or third-party namespaces underneath.

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

- Use delegate inference instead of explicit delegate instantiation.

```
delegate void SomeDelegate();
public void SomeMethod()
{...}
SomeDelegate someDelegate = SomeMethod;
```

- Maintain strict indentation. Do not use tabs or nonstandard indentation, such as one space. Recommended values are three or four spaces, and the value should be uniform across.
- Indent comments at the same level of indentation as the code you are documenting.
- All comments should pass spell checking. Misspelled comments indicate sloppy development.
- All member variables should be declared at the top, with one line separating them from the properties or methods.

```
public class MyClass
{
   int m_Number;
   string m_Name;
   public void SomeMethod1()
   {}
   public void SomeMethod2()
   {}
}
```

- Declare a local variable as close as possible to its first use.
- A file name should reflect the class it contains.
- 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
{...}
```

- Always place an open curly brace (**{**) in a new line.
- With anonymous methods, mimic the code layout of a regular method, aligned with the delegate declaration. (complies with placing an open curly brace in a new line):

```
delegate void SomeDelegate(string someString);
//Correct:
void InvokeMethod()
```

```
{
   SomeDelegate someDelegate = delegate(string name)
                               {
                                     MessageBox.Show(name);
                               };
   someDelegate("Juval");
}
//Avoid
void InvokeMethod()
{
   SomeDelegate someDelegate = delegate(string name){MessageBox.Show(name);};
   someDelegate("Juval");
}
```

- Use empty parentheses on parameter-less anonymous methods. Omit the parentheses only if the anonymous method could have been used on any delegate:

```
delegate void SomeDelegate();
//Correct
SomeDelegate someDelegate1 = delegate()
                             {
                                 MessageBox.Show("Hello");
                             };
//Avoid
SomeDelegate someDelegate1 = delegate
{
   MessageBox.Show("Hello");
};
```

- With Lambda expressions, mimic the code layout of a regular method, aligned with the delegate declaration. Omit the variable type and rely on type inference, yet use parentheses:

```
delegate void SomeDelegate(string someString);
SomeDelegate someDelegate = (name)=>
                             {
                                 Trace.WriteLine(name);
                                 MessageBox.Show(name);
                             };
```

- Only use in-line Lambda expressions when they contain a single simple statement. Avoid multiple statements that require a curly brace or a **`return`** statement with inline expressions. Omit parentheses:

```
delegate void SomeDelegate(string someString);
void MyMethod(SomeDelegate someDelegate)
{...}

//Correct:
MyMethod(name=>MessageBox.Show(name));

//Avoid
MyMethod((name)=>{Trace.WriteLine(name);MessageBox.Show(name);});
```

3. Coding Practices

- Avoid putting multiple classes in a single file.

- A single file should contribute types to only a single namespace. Avoid having multiple namespaces in the same file.

- Avoid files with more than 500 lines (excluding machine-generated code).

- Avoid methods with more than 200 lines.
- Avoid methods with more than 5 arguments. Use structures for passing multiple arguments.
- Lines should not exceed 120 characters.
- Do not manually edit any machine-generated code.
  a) If modifying machine generated code, modify the format and style to match this coding standard.
  b) Use partial classes whenever possible to factor out the maintained portions.
- Avoid comments that explain the obvious. Code should be self-explanatory. Good code with readable variable and method names should not require comments.
- Document only operational assumptions, algorithm insights and so on.
- Avoid method-level documentation.
  a) Use extensive external documentation for API documentation.
  b) Use method-level comments only as tool tips for other developers.
- With the exception of zero and one, never hard-code a numeric value; always declare a constant instead.
- Use the **const** directive only on natural constants such as the number of days of the week.
- 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;
   }
}
```

- In general, prefer overloading to default parameters:

```
//Avoid:
class MyClass
{
   void MyMethod(int number = 123)
   {...}
}
//Correct:
class MyClass
{
   void MyMethod()
   {
   MyMethod(123);
   }
   void MyMethod(int number)
   {...}
}
```

- When using default parameters, restrict them to natural immutable constants such as null, false, or 0:

```
void MyMethod(int number = 0)
{...}
void MyMethod(string name = null)
{...}
```

```
void MyMethod(bool flag = false)
{...}
```

- Assert every assumption. On average, every 6th line is an assertion.

```
using System.Diagnostics;

object GetObject()
{...}

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

- Every line of code should be walked through in a "white box" testing manner.
- Catch only exceptions for which you have explicit handling.
- 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;
}
```

- Avoid error codes as method return values.
- Avoid defining custom exception classes.
- When defining custom exceptions:
  a) Derive the custom exception from **Exception**.
  b) Provide custom serialization.
- Avoid multiple **Main()** methods in a single assembly.
- Make only the most necessary types public, mark others as **internal**.
- Avoid friend assemblies, as they increase inter-assembly coupling.
- Avoid code that relies on an assembly running from a particular location.
- Minimize code in application assemblies (EXE client assemblies). Use class libraries instead to contain business logic.
- Avoid providing explicit values for enums unless they are integer powers of 2:

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

//Avoid
public enum Color
{
   Red = 1,
   Green = 2,
   Blue = 3
}
```

- Avoid specifying a type for an enum.

```
//Avoid
public enum Color : long
{
   Red,Green,Blue
}
```

- Always use a curly brace scope in an **if** statement, even if it conditions a single statement.

- Avoid using the ternary conditional operator.

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

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

- Avoid function calls in Boolean conditional statements. Assign into local variables and check on them.

```
bool IsEverythingOK()
{...}
//Avoid:
if(IsEverythingOK())
{...}
//Correct:
bool ok = IsEverythingOK();
if(ok)
{...}
```

- Always use zero-based arrays.

- With indexed collection, use zero-based indexes

- Always explicitly initialize an array of reference types using a **for** loop.

```
public class MyClass
{}
const int ArraySize = 100;
MyClass[] array = new MyClass[ArraySize];
for(int index = 0; index < array.Length; index++)
{
   array[index] = new MyClass();
}
```

- Do not provide public or protected member variables. Use properties instead.

- Avoid explicit properties that do nothing except access a member variable. Use automatic properties instead:

```
//Avoid:
class MyClass
{
   int m_Number;
   public int Number
   {
      get
      {
          return m_Number;
      }
      set
      {
          m_Number = value;
      }
   }
```

```
}
//Correct:
class MyClass
{
   public int Number
   {
      get;set;
   }
}
```

- Avoid using the **new** inheritance qualifier. Use **override** instead.
- Always mark public and protected methods as **virtual** in a non-sealed class.
- Never use unsafe code, except when using interop.
- Avoid explicit casting. Use the **as** operator to defensively cast to a type.

```
Dog dog = new GermanShepherd();
GermanShepherd shepherd = dog as GermanShepherd;
if(shepherd != null)
{...}
```

- Always check a delegate for **null** before invoking it.
- Classes and interfaces should have at least 2:1 ratio of methods to properties.
- Avoid interfaces with one member.
- Strive to have three to five members per interface.
- Do not have more than 20 members per interface. Twelve is probably the practical limit.
- Avoid events as interface members.
- When using abstract classes, offer an interface as well.
- Expose interfaces on class hierarchies.
- Prefer using explicit interface implementation.
- Never assume a type supports an interface. Defensively query for that interface.

```
SomeType obj1;
IMyInterface obj2;
/* Some code to initialize obj1, then: */
obj2 = obj1 as IMyInterface;
if(obj2 != null)
{
   obj2.Method1();
}
else
{
//Handle error in expected interface
}
```

- Never hardcode strings that will be presented to end users. Use resources instead.
- Never hardcode strings that might change based on deployment such as connection strings.

- Use **String.Empty** instead of "":

```
//Avoid
string name = "";
//Correct
string name = String.Empty;
```

- When building a long string, use **StringBuilder**, not **string**.

- Avoid providing methods on structures.
  a) Parameterized constructors are encouraged.
  b) Can overload operators.

- Always provide a static constructor when providing static member variables.

- Do not use late-binding invocation when early-binding is possible.

- Use application logging and tracing.

- Never use **goto** unless in a **switch** statement fall-through.

- Always have a **default** case in a **switch** statement that asserts.

```
int number = SomeMethod();
switch(number)
{
   case 1:
      Trace.WriteLine("Case 1:");
      break;
   case 2:
      Trace.WriteLine("Case 2:");
      break;
   default:
      Debug.Assert(false);
      break;
}
```

- 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")
   {}
}
```

- 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 Dog
{
   public Dog(string name)
   {}
   virtual public void Bark(int howLong)
   {}
}
public class GermanShepherd : Dog
{
   public GermanShepherd(string name): base(name)
   {}
   override public void Bark(int howLong)
```

```
    {
        base.Bark(howLong);
    }
}
```