.NET Coding Standards for SDL Implementations

SDL Professional Services

SDL Amsterdam

Hoogoorddreef 60
1101 BE Amsterdam
Netherlands
+31 (0)20 20 10 500

www.sdl.com

Table of contents

Version Management

The owner of the document is SDL – Content and Analytics. You can request a copy of this document from your nominated contact person at SDL.

Version	Date	Author	Distribution
0.1 Draft	28-10-2015	Jarno Henneman	Internal
0.2 Recommendations	25-11-2015	Jarno Henneman	Internal
1.0 Final	3-12-2015	Jarno Henneman	Available for external

Reference

The main references used whilst preparing this document:

- https://msdn.microsoft.com/en-us/library/ff926074.aspx
  by Microsoft

1. Introduction

This document describes the SDL standards .NET (in particular C#) code development for SDL implementations as recommended and carried out by SDL Professional Services.

The broader definition of maintainable code is that it is easily readable and understandable by developers other than the original developer. Code that does not meet this requirement is likely to contain more bugs than maintainable code.

Please note that these guidelines were not invented by SDL; there are some excellent sources of .NET coding guidelines and we just collected the ones we found most valuable and tailored them to the context of SDL implementations.

Dave Thomas, founder of the Eclipse strategy, described clean code as follows:

"Clean code can be read, and enhanced by a developer other than its original author. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone."

The view of what is and what is not clean maintainable code may differ from one group of developers to another. A professional developer must be flexible to adapt his coding style to meet the requirements of the wider group.

2. Naming convention

A standardized naming convention is widely considered the greatest aid that a developer can give to his or her peers who later need to understand the code.

2.1. Capitalization Conventions

To differentiate words in an identifier, capitalize the first letter of each word in the identifier. Do not use underscores to differentiate words, or for that matter, anywhere in identifiers. There are two appropriate ways to capitalize identifiers, depending on the use of the identifier:

- PascalCasing
- camelCasing

A special case is made for two-letter acronyms in which both letters are capitalized, as shown in the following identifier:

- IOStream

The camelCasing convention, used only for parameter names, capitalizes the first character of each word except the first word, as shown in the following examples. As the example also shows, two-letter acronyms that begin a camel-cased identifier are both lowercase.

- propertyDescriptor
- ioStream
- htmlTag

The following table describes the capitalization rules for different types of identifiers.

Identifier	Casing	Example
Namespace	Pascal	namespace **System.Security** { ... }
Type	Pascal	public class **StreamReader** { ... }
Interface	Pascal	public interface **IEnumerable** { ... }

Method	Pascal	public class Object {
		  public virtual string **ToString**();
		}

Property	Pascal	public class String {
		  public int Length { **get;** }
		}

Event	Pascal	public class Process {
		  public event EventHandler Exited;
		}

Field	Pascal	public class MessageQueue {
		  public static **TimeSpan** InfiniteTimeout;
		
		  public const **Min** = 0;
		}

Field (Private)	Camel	private string _licenseFileName;*
Field (Constants)	PascalCase	private const int MaxFileCounter = 10;

Enum value	Pascal	public enum **FileMode** {
		  Append,
		}

Parameter	Camel	public class Convert {
		  public static int ToInt32(string **value**);
		}

2.2. Use US English for identifier names

Use only alphanumerical characters for identifiers, and use US English.

Also make sure identifiers are spelled correctly, particularly in public APIs.

```
enum Color {
   LightGray,
   ….
}
```

2.3. Name an identifier according to its meaning, not its type

Names of identifiers should be as specific as possible. A name that only reveals the type is in most cases not specific enough.

It may be tempting to give a local variable a more generic name (in extreme: a name that only mentions its type) in order to be able to re-use it for multiple purposes. Resist this temptation and simply define multiple, separate variables.

2.4. Specify types and keywords

Avoid the use of the "var"; specify types explicitly - it improves readability tremendously!

```
public string websiteUrl = “http://www.sdl.com”;
```

2.5. Use C# predefined type names

When creating new variables, please use the proper C# definitions as described below.

sbyte	byte	(u)short
(u)int	(u)long	Float
double	bool	Char
string	object	

3. Comments

3.1. XML Documentation Comments

In .NET you can create documentation for your code by including XML elements in special comment fields in the source code. You indicate these by triple slashes and you place them directly before the code block to which the comments refer.

For example:

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass{}
```

When you compile with the /doc option, the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool or use a tool such as Sandcastle.

For more information about which tags are available for the /doc option, see:
https://msdn.microsoft.com/en-us/library/5ast78ax.aspx

3.2. Use US English for comments

Use US English for your comments, and make sure that the comments are spelled correctly. This applies to all comments being made.

Lifetime of comments is the same as that of the source code. Just as code requires maintenance, so do the comments!

3.3. Comment in functions

It is not advisable to see comments in the XML Documentation comments when creating comments within the code to clarify decisions, unclear or TODO code. In these cases use the double slashes //.

Don’t state the obvious

In your comments don’t state the obvious. Do not repeat what is already clear from the code itself (“Loop over all the entries in the list…”). Rather “fill in the gaps” by providing background information on assumptions made, algorithms applied, optimization performed, etc.

4. Source code structure and layout

Creating a familiar structure within your code helps the readability and usability of the code.
This keeps the code cleaner especially when multiple people are working on the same code.

4.1. Indentation: use spaces instead of tabs

The indentation can become permanently garbled when different people use different settings (spaces vs. tabs, tab size). Therefore, it is important that everybody uses the same settings.

Spaces (4) is the most sustainable and recommended approach.

4.2. Source code lines

Very long lines are hard to read. Although with expanding monitor sizes and resolutions more and more characters fit on the screen, we recommend that a source line should not be longer than a 100 to max 150 characters. If your code line is longer, break it up in multiple lines.

Acceptable ways of breaking up long statements (typically a method call or declaration with many parameters):

1. Fit as much as possible on a single line, parameters that do not fit have to go on the next line, which is indented – preferred: 'hanging indent'.
2. Put each parameter on its own line 'indented'.

4.3. Compound statements (Curly brackets)

When using a compound statement (curly brackets): place them always on the next line.

4.4. Flow control

All flow control primitives (if, else, while, for, do, switch) should be followed by a block, even if it is empty or if it contains only one statement.

```
If (b1)
{
        If (b2)
        {
        …
```

5. Coding Practices

Below are some good coding practices and guidelines to how you should program.

5.1. The use of ‘this’ keyword

Using the ‘this’ keyword should be avoid and only used when necessary like,

- Indexers,
- Invoke other constructors,
- Pass the instance as an argument

‘this’ should not be used to construct to dereference members.

```
public tcmUri (int itemId)
{
        this.test = “Don’t use this. here!”;
}
```

5.2. Keep methods small

Similar to keeping ‘source code lines’ small, it’s in general recommended to keep methods smaller. We recommend a method holds around a 100 lines maximum.

If the method is longer, it becomes too complex to grasp and you lose overview. Use functional decomposition to split up the method is smaller sub-methods.

5.3. Declaring, Initializing variables

Methods shouldn’t be very large and they should keep the ‘distance’ between declaring a variable and using it small. For simplifying the structure of a method it’s advisable to declare variables at the top of the method.

Also put the initialization of a variable near its declaration (if possible, on the same line).

5.4. Use of ‘Const’

If an assembly references a Const field from another assembly, the actual value is embedded in the referring assembly. In other words, if assembly A uses a Const that is defined in assembly B, the computed value is embedded in assembly A.

This means that if the value for the Const in assembly B is changed, assembly A will still be using the original value until it is recompiled against the new B.

Therefore, Const should only be used for “real” constants like the value of Pi, the number of hours per day, or the maximum value for a 32 bit integer. For “less constant” constants that you might want to change over time, use a static read-only field instead.

Note that this only applies to Const fields that are public or protected. If a field is private, it’s perfectly OK to mark it as Const even though you expect that its value will change in a few months’ time.

5.5. Use of public static read-only fields for predefined instances

If a type has predefined instances, declare them as public static read-only fields.

5.6. Implementing ‘IDisposable’

To prevent resource leaks (for unmanaged resources) or holding on to expensive resources when they are no longer needed, be sure to implement the Dispose pattern when working with such resources.

Use the following template code:

```
public class ResourceHolder : IDisposable
{
        public void Dispose()
        {
                Dispose(true);
                GC.SuppressFinalize(this);
         }

        protected virtual void Dispose(bool disposing)
        {
                if (disposing)
                {
                        // Release managed resources here
                }

                // Release unmanaged resources here
        }
…
```

5.7. Changing loop variables inside a for loop

It is confusing to change the loop variable of a ‘for’ loop inside the loop body. If you need to break out of a ‘for’ loop mid-way, use ‘break’ or set some flag to signal ending of the loop.

5.8. Add ‘default’ to every switch-statement

Every ‘switch’ statement should have a default case. In addition, if the default should be unreachable it should assert or throw.

5.9. Avoid multiple ‘return’ statements

In general avoid having more than a single ‘return’ statement, as having multiple exit points in a method increases complexity. The most common exception to this rule is when you’re checking preconditions. It is fine to return early from a method if the preconditions aren’t met.

5.10. Be concise when using Booleans

When you are only setting a Boolean to true or false based on some criterion, don’t be too wordy.

```
isPositive = (value > 0);
```

5.11. Declare all fields as private

Fields (data members or member variables) should be marked with private. Use properties to provide access from another class (include or subclass)/

Note that it is allowed to have public events.

5.12. Utility classes as Static

If a class only contains static properties and methods (a typical ‘utility’ class) mark the class as static as well.

5.13. Avoid using ‘new’

It is possible to hide the implementation from the base class by using the ‘new’ modifier on a class member. This is hardly ever needed. Alternatively, a virtual method can be overridden (and the new implementation may or may not call the base class implementation).

5.14. Check the result of defensive cast

Using defensive casting via the ‘as’ operator is a good way to prevent invalid cast exception. Though remember: the result still needs to be checked for null!

```
private void Publish(OrganizationalItem item)
{
        Folder folder = item as Folder;
        if (folder != null)
        {
                ...
```

5.15. Using parameter syntax for Boolean

When validating a Boolean expression it’s recommended to name the parameter of the method. This increases the human readability of the code.

So instead of using:

```
WebRequestContext.Localization.Refresh(true);
```

Please use the following:

```
WebRequestContext.Localization.Refresh(allSiteLocalizations: true);
```

5.16. Avoid (direct) use of literal values

Do not use literal values in your code other than defined symbolic constants.
Use the constant String.Empty instead of the literal empty string (“”).

About SDL

SDL (LSE: SDL) allows companies to optimize their customers' experience across the entire buyer journey. Through its web content management, analytics, social intelligence, campaign management and translation services, SDL helps organizations leverage data-driven insights to understand what their customers want, orchestrate relevant content and communications, and deliver engaging and contextual experiences across languages, cultures, channels and devices.

SDL has over 1,500 enterprise customers, over 400 partners and a global infrastructure of 70 offices in 38 countries. We also work with 72 of the top 100 global brands.