LogixHealth Coding Standards and Guidelines

Terminology and Definitions

The following terminology is referenced throughout this document	
Coloring & Emphasis	
Blue	Text colored blue indicates a C# keyword or .NET type
Bold	Text with additional emphasis to make it stand-out
Keywords	
Always	Emphasizes this rule must be enforced
Never	Emphasizes this action must not happen
Do Not	Emphasizes this action must not happen
Avoid	Emphasizes that the action should be prevented, but some exceptions may exist
Try	Emphasizes that the rule should be attempted whenever possible and appropriate
Example	Precedes text used to illustrate a rule or recommendation
Reason	Explains the thoughts and purpose behind a rule or recommendation

Design/Implementation Rules – C#

DESIGN/IMPLEMENTATION RULES

DO`s and DON`Ts

- Implement or Design a module/class which should have responsibility over a single part of the functionality provided/consumed by the application and that responsibility should be entirely encapsulated by it. (Single Responsibility Principle)

  Example – Consider this class which has Actions/Methods related to Carriers, Financial Class and Chart Status

```
public class MasterDictionariesController : BaseController
{
        #region Carriers
                public ActionResult MdCarriersView(...) { }
                public void GetMdCarriersExcelExport(...) { }
                public ActionResult MdCarriersCrud(...) { }
        #endregion Carriers

        #region Financial Class
                public ActionResult SaveMdFinancialClass(...) { }
                public ActionResult CheckDuplicateMdFinancialClass(...) { }
        #endregion Financial Class

        #region Chart Status
                public JsonResult GetMdChartStatusListByName(...) { }
                public ActionResult GetMdChartStatusBySortOrder(...) { }
        #endregion Chart Status
}
```

  The above implementation can be split into multiple class/module based on the feature/functionality. Advantage it`s easy to scale-out the application

Design/Implementation Rules

DO`s and DON`Ts

- Always segregate the Business Classes with Interfaces (Interface Segregation Principle) with respective functionalities. So that, no client code are forced to use an interface which is irrelevant to it.
- Do not implement any tightly coupled code. Instead use Dependency Inversion Principle (Dependency Injection)
- Do not create PUBLIC CONSTRUCTOR for abstract types/classes. Because you cannot create instances of abstract types
- Do not create empty interfaces
- Avoid excessive parameters on methods (maximum 3 or 4 parameters). Instead create a type and use as parameter
- Avoid out parameters
- Do not catch general exception types. Catching generic exception types can hide run-time problems from the library user, and can complicate debugging

DESIGN/IMPLEMENTATION RULES

DO`s and DON`Ts

- Do not expose generic lists. Do not expose List<T> in object models. Use Collection<T>, ReadOnlyCollection<T> or KeyedCollection<K,V> instead. List<T> is meant to be used from implementation, not in object model API. List<T> is optimized for performance at the cost of long term versioning. For example, if you return List<T> to the client code, you will not ever be able to receive notifications when client code modifies the collection
- Do not pass types by reference
- Implement IDisposable correctly. All IDisposable types should implement the Dispose pattern correctly
- Do not explicitly raise exceptions from unexpected locations. There are some methods, such as Equals and GetHashCode, which users do not expect to raise exceptions. Therefore calls to these methods are not commonly wrapped in try catch blocks
- Write-only properties usually indicate a flawed design

DESIGN/IMPLEMENTATION RULES

DO`s and DON`Ts

- The ObsoleteAttribute.Message property provides the text message displayed when an obsolete type or member is compiled. This message should provide information on the replacement for the obsolete element.
- Do not combine your models/entities and data access implementation together
- Always keep Models/Entities as separate library
- Do not add DTO to the models as a suffix
- Do not add DataAccess to the data access implementation classes as a suffix
- Always design your models coincide with UI
- Always use fully qualified type while decorating the members/classes

  Example –

```
[System.Runtime.Serialization.DataMemberAttribute()]
public string ErrorMessage { get; set; }
```

Design/Implementation Rules

DO`s and DON`Ts

- Do not manually edit any machine-generated code. If modifying machine-generated code, modify the format and style to match the coding standard
- Controllers (MVC Controller and API Controller) should have only payload validations. That’s, NO business related validation should be present
- Implement switch statement instead of IF statement wherever necessary
- Use properties where appropriate. Properties should be used instead of Get/Set methods in most situations. Methods are preferable to properties in the following situations:
  - The operation is a conversion, is expensive or has an observable side-effect
  - The order of execution is important
  - Calling the member twice in succession creates different results
  - A member is static but returns a mutable value or the member returns an array
- Do not cast unnecessarily. Avoid duplicate casts where possible, since there is a cost associated with them
- Properties should not return arrays. Properties that return arrays are prone to code inefficiencies. Consider using a collection or making this a method
- Avoid uncalled private code

Design/Implementation Rules

DO`s and DON`Ts – ASP.NET MVC/API Guidelines

- “Thin Controllers, Functional ViewModels, Fat Models“ – Controllers are nothing more than a director showing the way on where to go on how to process data. Your controller should be extremely thin (< 5 lines of code)
- Always decorate action methods with HTTP GET/POST/DELETE/PUT attributes
- DO NOT include conditional code in your Views - All of your processing should happen in your controller or models, not in your Views. Your Views are meant to receive and display data
- It's best to move the conditional code into an HtmlHelper (see HtmlHelper Guidelines below) or process it in the model
- Use Strong-Typed ViewModels where applicable
- It's always a chore using ViewBag, ViewData, or TempData. Over the years, my perspective is to send POCOs (Plain old CLR objects) to your Views. It makes handling your data a lot easier and provides a better Intellisense experience for designers and developers on the HTML side
- Make your ViewModels a little more functional - While passing data through your ViewModels requires a simple POCO, it doesn't mean that your ViewModels can't help you modify the data for a View

DESIGN GUIDELINES & CODING STANDARDS – C#

NAMING CONVENTIONS

The goal of this standard presents the naming conventions for the applications developed in C# language in order to enforce consistency. These standards provide number of benefits to the development process.

- Code is more readable and less complex
- Code is easier to maintain and correct
- Code has consistent structure across the organizatio

NAMING CONVENTIONS

The following terminology is referenced throughout this document

Access Modifier

C# keywords public, protected, internal, and private declare the allowed code-accessibility of types and their members. Although default access modifiers vary, classes and most other members use the default of private. Notable exceptions are interfaces and enums which both default to public

Camel Case

A word with the first letter lowercase, and the first letter of each subsequent word-part capitalized
Example – `string userName;`

Pascal Case

A word with the first letter capitalized, and the first letter of each subsequent word-part capitalized
Example – `string UserName;`

Identifier

A developer defined token used to uniquely name a declared object or object instance
Example – `public class PatientType`

Common Type System

The .NET Framework common type system (CTS) defines how types are declared, used, and managed. All native C# types are based upon the CTS to ensure support for cross-language integration

Naming Conventions

Identifier	Public	Protected	Internal	Private	Notes
Solution File	Pascal Case	Φ	Φ	Φ	Always Match suite of applications
Project File	Pascal Case	Φ	Φ	Φ	Always match Assembly Name and Root Namespace
Namespace	Pascal Case	Φ	Φ	Φ	Partial Project/Assembly match
Source File	Pascal Case	Φ	Φ	Φ	Match contained class
Other Files	Pascal Case	Φ	Φ	Φ	Apply where possible
Class or Struct	Pascal Case	Pascal Case	Pascal Case	Pascal Case	
Interface	Pascal Case	Pascal Case	Pascal Case	Pascal Case	Prefix with a capital "**I**" Example – `interface IStandards { }`
Method	Pascal Case	Pascal Case	Pascal Case	Pascal Case	Use a Verb or Verb-Object pair or an Action
Property	Pascal Case	Pascal Case	Pascal Case	Pascal Case	Do not prefix with Get or Set

NAMING CONVENTIONS

Identifier	Public	Protected	Internal	Private	Notes
Field	Pascal Case	Pascal Case	Pascal Case	camel Case prefix with underscore	Only use UNDERSCORE for Private fields. No Hungarian Notation
Constant	Pascal Case	Pascal Case	Pascal Case	camel Case prefix with underscore	Only use UNDERSCORE for Private fields. No Hungarian Notation
Static Field	Pascal Case	Pascal Case	Pascal Case	camel Case prefix with underscore	Only use UNDERSCORE for Private fields. No Hungarian Notation
Enum	Pascal Case	Pascal Case	Pascal Case	Pascal Case	
Parameter	Φ	Φ	Φ	camel Case	

NAMING CONVENTIONS

Syntax and Usage	
Identifier	**Rules with example**
Solution File	Use Pascal Case Always Match suite of applications **Example –** Solution Name » LogixHealth.ICER Solution File Name » LogixHealth.ICER.sln
Project File	Use Pascal Case Always match Assembly Name and Root Namespace **Example –** Project Name » LogixHealth.ICER.Codify.UI Namespace Name » LogixHealth.ICER.Codify.UI Assembly Name » LogixHealth.ICER.Codify.UI.dll

Naming Conventions

Syntax and Usage	
Identifier	**Rules with example**
Namespace	Use Pascal Case Always match Assembly Name and Root Namespace [CompanyName].[SuiteOfAppName].[AppName].[ComponentName] **Example –** **namespace** LogixHealth.ICER.Codify.UI
Source File	Use Pascal case Always match Class Name and file name Avoid including more than one **Class**, **Enum** (global), or **Delegate** (global) per file **Example –** Chart.cs -> **public class** Chart When using a partial types and allocating a part file, name each file after the logical part that part plays **Example –** Chart.cs -> **public class** Chart **// In PartialTypes.cs** **public partial class** PartialTypes

Naming Conventions

Syntax and Usage	
Identifier	**Rules with example**
Resource or Embedded File	Use Pascal Case Use a name describing the file contents
Class or Struct	Use Pascal case Use a noun or noun phrase Do not add any suffix or prefix like Struct, Class, C or S Use a compound word to name a derived class where appropriate. The second part of the derived class's name should be the name of the base class Use reasonable judgment in applying this rule **Example –** **private class** MyClass {…} **internal class** SpecializedAttribute : Attribute {…} **private struct** ApplicationSettings {…}

Naming Conventions

Syntax and Usage	
Identifier	**Rules with example**
Custom Attribute Classes	Use Pascal case Add the suffix 'Attribute' **Example –** `public class ObsoleteAttribute`
Custom Exception Classes	Use Pascal case Add the suffix 'Exception' **Example –** `public class BusinessException`

Method	Use Pascal case
	Use verbs or verb phrases to name methods
	Methods with return values should have a name describing the value returned
	Example –
	`public void ShowDialog()`
	`public string GetAssemblyVersion()`

NAMING CONVENTIONS

Syntax and Usage	
Identifier	**Rules with example**
Interface	Use Pascal case Always prefix with a letter ‘I’ Use a noun, noun phrase, or an adjective **Example –** **// Noun** **public interface** IComponent **// Noun Phrase** **public interface** ICustomAttributeProvider **// Adjective** **public interface** IPersistable Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter I prefix on the interface name **Example –** **// Interface definition** **public interface** IComponentProvider { } **// Interface implementation** **public class** ComponentProvider : IComponentProvider { }

Naming Conventions

Syntax and Usage	
Identifier	**Rules with example**
Property	Use Pascal case Do not include the parent class name within a property name **Example –** **// Bad Code** Customer.CustomerName **// Good Code** Customer.Name Property name should represent the entity it returns. Never prefix property names with “Get” or “Set” Consider creating a property with the same name as its underlying type **Example –** **public string** Name { **get**; **set**; }
Field (public, protected, or internal)	Use Pascal case Do not use abbreviated words such as num instead of number **Example –** **public string** Name; **protected ICollection<**Customer**>** Customers;

NAMING CONVENTIONS

Syntax and Usage	
Identifier	**Rules with example**
Field (public, protected, or internal)	Use Pascal case Do not use abbreviated words such as num instead of number **Example –** **public string** Name; **protected ICollection<**Customer**>** Customers;
Field (private)	Use cascal case suffix with underscore Do not use abbreviated words such as num instead of number **Example –** **private string** _name; **private ICollection<**Customer**>** _customers;

Constant/Static	Use Pascal case
	Example –
	const int DefaultFileSize = 100;
	static int DefaultFileSize = 100;

NAMING CONVENTIONS

Syntax and Usage	
Identifier	**Rules with example**

Enum Type and options	Use Pascal case
	Do not use an Enum suffix/prefix on Enum type names
	Always add the FlagsAttribute to a bit field Enum type
	Use a singular name for most Enum types, but use a plural name for Enum
	Example –
	// Bad Code
	public enum ColorsEnum
	{
	RedColor,
	BlueColor
	}
	// Good Code
	public enum Colors
	{
	Red,
	Blue
	}

NAMING CONVENTIONS

Syntax and Usage

Identifier	Rules with example

Method Parameters	Use Camel Case
	Use descriptive parameter names which describe parameters meaning rather than its type
	Do not prefix parameter names with Hungarian type notations
	Avoid excess parameters. That is, if a method has more than 3 parameters try making it as class or structure
	Example –
	// Bad Code
	public **Patient** GetPatientDemographic(Guid patientID, string socialSecurityNumber, DateTime dateOfBirth, DateTime dateOfVisit, string diagnosticCode, Guid providerID)
	{…}
	
	// Good Code
	Public class PatientDemographicFilter
	{
	public Guid patientID { get; set; }
	public string socialSecurityNumber { get; set; }
	public DateTime dateOfBirth
	public DateTime dateOfVisit
	public string diagnosticCode
	public Guid providerID
	}
	public **Patient** GetPatientDemographic(**PatientDemographicFilter** filter)
	{…}

NAMING CONVENTIONS

DO`s and DON`Ts

Always use camel Case or Pascal Case names

Avoid ALL CAPS and all lowercase names. Single lowercase words or 2/3 letters are acceptable. Any Abbreviations must be widely known and accepted

Avoid abbreviations longer than 3 characters

Do Not create declarations of the same type (namespace, class, method, property, field, or parameter) and access modifier (protected, public, private, internal) that vary only by capitalization

Do Not use names that begin with a numeric character

Do add numeric suffixes to identifier names

Always choose meaningful and specific names

Variables and Properties should describe an entity not the type or size

Do Not use Hungarian Notation

Example –

Bad Code: `string strName; or int iCount; or User objUser; or User userObject;`

Good Code: `string name; or int count; or User userModel;`

Naming Conventions

DO`s and DON`Ts

Avoid using abbreviations unless the full name is excessive

Do Not use C# reserved words as names

Avoid naming conflicts with existing .NET Framework namespaces, or types

Avoid adding redundant or meaningless prefixes and suffixes to identifiers

Example –

```
// Bad Code
public enum ColorsEnum
public class CVehicle
public struct RectangleStruct
```

Do Not include the parent class name within a property name

Example –

Bad Code: Customer.CustomerName

Good Code: Customer.Name

NAMING CONVENTIONS

DO`s and DON`Ts

Try to prefix Boolean variables and properties with “**Can**”, “**Is**” or “**Has**” or sufix “**Exists**”

Append computational qualifiers to variable names like **Average**, **Count**, **Sum**, **Min**, and **Max** where appropriate

STYLES AND FORMATTING

Coding style causes the most inconsistency and controversy between developers. Each developer has a preference, and rarely are two the same. However, consistent layout, format, and organization are key to creating maintainable code. The following sections describe the preferred way to implement C# source code in order to create readable, clear, and consistent code that is easy to understand and maintain.

Styles and Formatting

Formatting – General Guidelines

Avoid having multiple namespaces in the same file. Never declare more than 1 namespace per file

Avoid putting multiple classes in the same file

Always place curly braces ({ and }) on a new line

Always use curly braces ({ and }) in conditional statements

Always use a Tab & Indention size of 4

Declare each variable independently – not in the same statement

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

Avoid method with more than 25 lines. Consider re-factoring the method

Avoid method with more than 5 arguments. Use structures for passing multiple arguments

Avoid Line not which exceeds 120 characters

Place namespace "using" statements together at the top of file

STYLES AND FORMATTING

Formatting – General Guidelines

Remove unused “using” statements

Group internal class implementation by type in the following order

- Member variables
- Constructors & Finalizers
- Nested Enums, Structs, and Classes
- Properties
- Methods

Sequence declarations within type groups based upon access modifier and visibility

- Public
- Protected
- Internal
- Private

Segregate interface Implementation by using #region statements

Styles and Formatting

Formatting – General Guidelines

Only declare related attribute declarations on a single line, otherwise stack each attribute as a separate declaration

Example –

```
// Bad Code
[Attrbute1, Attrbute2, Attrbute3]
public class MyClass
{…}

// Good Code
[Attrbute1, RelatedAttribute2]
[Attrbute3]
public class MyClass
{…}
```

Styles and Formatting

Comments and Embedded Documentation – Guidelines

All comments should be written in the same language (English), spell checked, be grammatically correct, and should contain appropriate punctuation

Use // or /// but never /* … */

Do not “flowerbox” the comment block like below

Example –

```
// *********************************
// Bad Comment Block
// *********************************
```

Each file shall contain a header a block and header block must consist of a #region block containing the copyright statement and the name of the file

Example –

```
#region Copyright LogixHealth 2019
// All rights are reserved. Reproduction or transmission in whole or in part, in any form or by any means,
// electronic, mechanical or otherwise, is prohibited without the prior written consent of the
// copyright owner.
// Filename: CodeAdministration.cs
#endregion
```

STYLES AND FORMATTING

Comments and Embedded Documentation – Guidelines

Use inline comments to explain assumptions, algorithms insight, and known issues

Avoid comments that explain the obvious. Code should be self-explanatory

Only use comments for bad code to say “fix this code” – otherwise remove, or rewrite the code

To prevent recurring problems, always use comments on bug fixes and workaround code, especially in a team environment. When writing comments for bug fixes, use the following format near the code being modified

Example –

```
// Bug Fix: Short description of the bug and bug number if available
```

Include comments using Task-List keyword flags to allow comment-filtering

Example –

```
// TODO: Place Database Code Here
// UNDONE: Removed P\Invoke Call due to errors
// HACK: Temporary fix until able to re-factor
```

Use XML tags for documenting all public, protected and internal declarations. Using these tags allows IntelliSense to provide useful details while using the types

Always include <**summary**> comments. Include <**param**>, <**return**>, and <**exception**> comment where applicable

Include <**see cref**=""/> and <**seeAlso cref**=""/> where possible