Software Coding Guidelines for the Regional Integrated Corridor Management System

Version: *1.0*

Date: July 6, 2018

Approval date: *July 5, 2018*

DOCUMENT CONTROL PANEL		
File Name:	R-ICMS-CG-1.0.docx	
File Location:	\\dyn.datasys.swri.edu\Shares\Projects\10-23368_FDOT_D5_ICMS\Deliverables\Software Coding Guidelines\R-ICMS-CG-1.0.docx	
Version Number:	1.0	
	Name	**Date**
Created By:	Clay Westin, SwRI	5/14/2018
	Robert Heller, SwRI	5/14/2018
Reviewed By:	Clay Packard, FDOT/VHB	July 5, 2018
Modified By:	Robert Heller, SwRI	July 6, 2018
Approved By:	Clay Packard, FDOT/VHB	July 5, 2018

Table of Contents

List of Acronyms and Abbreviations

COTS ..... Commercial Off the Shelf
DFE ..... Data Fusion Environment
DSS ..... Decision Support System
FDOT ..... Florida Department of Transportation
IEN ..... Information Exchange Network
LRU ..... Lowest Replaceable Unit
R-ICMS ..... Regional Integrated Corridor Management System
SRS ..... Software Requirements Specification
SwRI ..... Southwest Research Institute
TSM&O ..... Transportation Systems Management and Operation

1 Overview

The purpose of a Coding Guidelines is to produce code that is easy to read and understand quickly and accurately. Conventions assure that the appearance of the code does not distract from understanding. The goal is to reduce the reading (and writing) of code to a highly mechanical process that minimizes creative approaches to appearance; all creativity is thereby reserved for the difficult tasks of understanding the needs of the user and designing a computerized system that meets those needs effectively.

1.1 Document Overview

This document is organized into sections, each representing guidelines for a particular computer programming language that will be used to implement some portion of the R-ICMS system. For those guidelines that will follow established "industry standards" a reference to where the guideline can be found is provided.

1.2 Project Overview

1.2.1 Project Identification

Project Name: *Central Florida Regional Integrated Corridor Management System*.
Agreement Number: BE521
Financial Project Identification: 436328-1-82-01
Federal Aid Project Number: *Not Applicable*.

1.2.2 Purpose and Scope

The R-ICMS will consist of, but not be limited to; commercial off-the-shelf (COTS) modeling software (provided by the DEPARTMENT), a custom-built Decision Support System (DSS), a custom- built Information Exchange Network (IEN) subsystem that includes dashboards and other user interfaces to the system, and a Data Fusion Environment (DFE) to host data sources for both the R-ICMS and other external users and applications.

1.3 Referenced Documents

The following documents, of the exact issue shown, form a part of this document to the extent specified herein. In the event of a conflict between the contents of the documents referenced herein and the contents of this document, this document shall be considered the superseding document.

Standard Written Agreement, Agreement Number BE521	*FDOT District 5 Procurement. A copy is maintained on the Project SharePoint Site.*
ITN-DOT-16-17-5004-ICMS	*FDOT District 5 Procurement. A copy is maintained on the Project SharePoint Site.*

Systems Engineering and ITS Architecture (Topic No 750-040-003)	*http://www.dot.state.fl.us/proceduraldocuments/procedures.shtm*
R-ICMS-PSEMP-1.0.docx	*FDOT District 5*
Python Coding Standard	*https://www.python.org/dev/peps/pep-0008/*
Python Docstring Standards	*https://www.python.org/dev/peps/pep-0257/*
Scala Coding Standards	*https://docs.scala-lang.org/style/*
Shell Scripting	*https://google.github.io/styleguide/shell.xml*
JavaScript -- Angular JS	*https://angular.io/guide/styleguide*

2 Coding Standards

SwRI will follow coding standards when developing code for the R-ICMS project. In general, the team shall establish any standard / guideline as deemed appropriate that is not available from the coding styles referenced below and shall be updated in the coding standards document. Similarly, any deviations from the standards shall also be added to the document after obtaining proper approvals from the project manager.

2.1 C# Coding Standard

The SwRI coding standard in use during the development of the SunGuide system is in Attachment A of this document. By using the same standard as that used by the SunGuide effort, the code developed for the R-ICMS project will have the same style as that code base.

In addition to the coding standards, SwRI uses ReSharper plug-in for Visual Studio. ReSharper assists development of C# projects with continuous code analysis for errors and suggestions for code optimizations.

- ReSharper also allows for customizations to the suggestions to include parts of the coding standard. For instance,
- ReSharper can look at method names to ensure they use camel case (or another method consistent with the coding standard) and flag the method if the method name does not meet the standard.
- ReSharper also checks comments of methods to make sure they exist and also spell checks all comments. Suggestions for changes are made known through a vertical bar next to the scroll bar that contains tick marks where suggestions are made in the file. These marks are color coded so that a user can open a file and easily identify critical comments (e.g. compilation errors) from suggestions (e.g. comment had spelling mistake).
- ReSharper also helps clean up code by flagging unused methods and segments of code that are unreachable or unnecessary (i.e. a function has an embedded return statement so the rest of the code will never be executed).

2.2 Python Coding Standard

SwRI will use the Python coding standard as described in "**PEP 8 – Style Guide for Python Code**" that may be found at the following URL:

https://www.python.org/dev/peps/pep-0008/

This standard is the de-facto code style guide for Python. A high quality, easy-to-read version of PEP 8 is also available at pep8.org. In addition, the project shall also use the "**PEP 257 – Docstring Conventions**" for docstring conventions which is available at the following URL:

https://www.python.org/dev/peps/pep-0257/

Docstrings are string literals that occur as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

2.3 Scala Coding Standard

SwRI will use the coding style guidelines published by Scala Project at the following URL:

https://docs.scala-lang.org/style/

Along with the coding style guidelines quoted above, the project shall also adopt best practices described at the following URL.

https://github.com/databricks/scala-style-guide

2.4 Shell Scripting

There are various shell scripting standards available depending upon the tasks/projects types, so SwRI will be using google shell style guides at the following URL.

https://google.github.io/styleguide/shell.xml

2.5 JavaScript – Angular JS

JavaScript programming is different than a procedural programing languages like C#. Java, etc., so it is important to implement JavaScript code using the suggested style guidelines below. SwRI will be using Angular JS style guide that may be found at the following URL

https://angular.io/guide/styleguide

Attachment A
Coding Standard for C Sharp (C#)

1. Introduction

1.1. Purpose

This document details the standards and practices for the development of software in the C Sharp (C#) language. The standards are designed to accomplish several goals:

- Provide a uniform look and feel to all software developed by a group.
- Make it easier for developers to review each other's code.
- Prevent the introduction of defects through the use of best practice structural and syntactic conventions.
- Make it easier to find and fix defects.
- Make it more efficient for developers to switch between projects when resource reallocations are required.
- Improve software portability and reusability.
- Improve long term maintenance.

The majority of the time and money spent on production software is spent on maintenance. Since maintenance is a future activity, the software written today is a means of communicating ideas and information to some future programmer, perhaps even the original author. Therefore, the primary goal when writing software must be to produce code that is ***correct, easy to read*** and ***easy to maintain***.

1.2. Scope

This standard is applicable to all software developed in C#, regardless of whether the result is for a Windows desktop application, Web Service, Web site, etc.

1.3. Variances

For a specific project, there may be special reasons that require deviation from these standards. In such cases, it is the project manager's responsibility to define the deviations and to inform all project team members. This must be documented in a memo or other written format that can become part of the project records.

2. Naming Conventions

The naming scheme is one of the most influential aids to understanding the logical flow of an application. A name should tell "what" rather than "how." By avoiding names that expose the underlying implementation, which can change, a layer of abstraction that simplifies the complexity is preserved. For example, use "getNextStudent()" instead of "getNextArrayElement()". A tenet of naming is that difficulty in selecting a proper name may indicate a need to further analyze or define the purpose of an item. Make names long enough to be meaningful, but short enough to avoid verbosity. Programmatically, a unique name serves only to differentiate one item from another. Expressive names function as an aid to a human reader; therefore, it makes sense to provide a name that a human reader can comprehend.

However, be certain that the chosen names are in compliance with the C# language's rules and standards.

2.1 Method Names

The common code based used by projects utilizes camel casing to allow a programmer to easily distinguish between code from Microsoft libraries and internal code. Individual processes may either use camel casing ("getNextStudent") or the Microsoft standard of capitalizing each letter of words ("GetNextStudent"). Naming must be consistent within a process. When maintaining code, continue the naming style already in use.

Do:

- Use the verb-noun method for naming routines that perform some operation on a given object, such as "calculateInvoiceTotal()".
- Ensure method overloads perform similar operations.
- When naming methods, include a description of the value being returned, such as "getCurrentWindowName()".
- When naming methods whose operation includes specific units of measure, include the type of units, such as "computeDistanceInMeters()".
- Make names descriptive without excessive length.

Don't:

- Use elusive names that are open to subjective interpretation, such as "AnalyzeThis()". Such names contribute to ambiguity more than abstraction.
- Include class names in the name of class properties, such as "Book.BookTitle". Instead, use "Book.Title".

2.2 Variable Names

Do:

- Append computation qualifiers (Avg, Sum, Min, Max, Index) to the end of a variable name where appropriate.
- Use complimentary pairs in variable names, such as min/max, begin/end, and open/close.
- Use camel casing ("documentFormatType") where the first letter of each word except the first is capitalized.
- Use boolean variable names containing Is which implies Yes/No or True/False values, such as "fileIsFound".
- Use a meaningful name even for short-lived variables.
- Include units of measure for variables with a specific unit of measure, such as "distanceInMeters".

Don't:

- Use mysterious names that are open to subjective interpretation, such as "xxK8."
- Use terms such as Flag when naming status variables, which differ from Boolean variables in that they may have more than two possible values. Instead of "documentFlag", use a more descriptive name such as "documentFormatType".

- Use single-letter variables names, other than for short (less than three lines) loop indexing.
- Use literal numbers or literal strings, such as for (i = 1; i < 7; i++), where 7 means nothing to someone reading the code. Instead, use named constants, such as for (i = 1; i <
- NUM_DAYS_IN_WEEK; i++) for ease of maintenance and understanding.
    - There are exceptions, the values 0, 1 and null can nearly always be used safely.
    - Often 2 and -1 can also be used.
    - Strings intended for logging or tracing are exempt from this rule.

2.3 Graphical User Interface (GUI) Widgets

GUI controls should be named to reflect their type. This makes the code easier to understand and promotes future maintainability. The following standard is recommended for naming controls. Controls not expressly cited in this table should be named in a similar fashion.

Control Prefix Example

```
TextBox txt:          txtName, txtPassword
Label lbl:            lblErrorMsg
ListBox list:         listPrescriptions
ComboBox combo:       comboNames
ListView lvw:         lvwRxData
TreeView tree:        treePatients
Button btn:           btnPrint
```

Controls that are static (i.e., never referenced in the code), such as a GroupBox label that never changes, need not follow this convention.

2.4 Class Names

Class names should be Pascal-cased and should be descriptive of the class' intended purpose.

- CacheManager
- XmlSerializer

When using Visual Studio, always change the default name assigned by the Integrated Development Environment (IDE). For example, change ***Form1*** to ***ConfigurationForm*** and ***Service1*** to ***SvcEmr***.

In many cases, the .NET development environment allows the declaration of instances of remote classes in the same manner as a local instance of a "normal" class. This is a powerful feature; however, it should be made apparent that the instance is "not normal." Additionally, an "interface" is a special type of object. Use the following prefixes for special types of classes:

"I" for interfaces **IAsyncResult**

"Svc" for web service classes **SvcEmr, SvcPharmacy**

"RemObj" for remotable objects **RemObjWaveManager**

Miscellaneous class usage information:

- Base classes are a useful way to group objects that share a common set of functionality. Base classes can provide a default set of functionality, while allowing customization through extension.
- Minimize the use of abbreviations, but use those that are created consistently. An abbreviation should have only one meaning. For example, if “min” is used to abbreviate minimum, do so everywhere and do not use “min” to also abbreviate minute.
- File and folder names, like procedure names, should accurately describe their purpose.
- Avoid reusing names for different elements, such as a routine called “ProcessSales()” and a variable called “iProcessSales”.
- Avoid homonyms, such as write and right, when naming elements to prevent confusion during code reviews.
- When naming elements, avoid commonly misspelled words. Also, be aware of differences that exist between regional spellings, such as color/colour and check/cheque.
- Avoid typographical marks to identify data types, such as “$” for strings or “%” for integers.

3. Code Format

3.1 Namespaces

Namespaces should be in lower case and according to the following pattern:

gov.its.<process>.<folder>.<subfolder>

For example, for LCS, the base classes are contained in the “gov.its.lcs” namespace. Folders can be within that namespace such as “gov.its.lcs.handlers”. For classes in these types of namespaces, the project should contain corresponding folders.

3.2 Methods

Since code tends to be viewed one screen at a time, a single method should fit on one screen, if possible. Very rarely should a method size exceed two printed pages (including comments). Every C# method shall have a standard method header. This leverages the eXtensible Mark-up Language (XML) Documentation feature in Visual Studio for creating code comment reports as in this example:

```
/// <summary>
/// Clean up any resources being used. Stops and releases timer resources
/// </summary>
/// <param name=”disposing”></param>
/// <returns>True if no error occured or False if an error occured.</returns>
protected override bool Dispose( bool disposing )
```

When calling a method, do not put each of the arguments on a separate line. Where possible, the procedure call should occupy a single line. For procedure calls with a lot of arguments, it is acceptable to split the line.

3.3 Class Format

- Classes must have class documentation and a copyright header at the top of the file.
- Sample copyright headers should be provided by the project manager.

- All fields for a class should be at the top of the class.
- Use regions to encapsulate variables, constructors and methods.
- Typical use of regions would be:

```
public class MyClass
{
        #region private member variables
        #endregion private member variables

        #region protected member variables
        #endregion

        #region constructors
        public MyClass()
        {
        }
        #endregion constructors

        #region public methods
        #endregion

        #region protected methods
        #endregion

        #region public methods
        #endregion
}
```

- For methods, regions may be used differently in order to group by functionality. For instance:

```
#region travel time calculations
#endregion travel time calculations

#region aggregators of data
#endregion aggregators of data

#region configuration of links
#endregion configuration of links
```

- Adding the region name to the "#endregion" is useful when scanning through a code file.

3.4 Indentation and Braces

Indentation, spacing, and other formatting rules are enforced using **StyleCop** (http://code.msdn.microsoft.com/sourceanalysis). Settings can vary on a per project basis. The project should provide a sample *Settings.SourceAnalysis* file containing the appropriate settings which can be ignored. The settings file only provides the exceptions to rules set in **StyleCop**.

- The standard indent level is four spaces. Make sure editors insert ***spaces*** when indenting, not tab characters.
- For switch statements, the CASE entries shall be at the same level as the switch.
- As a general rule, methods should contain no more than four levels of indentation.
- Line up braces on the left. Always use braces, even for single lines of code following if, else, for, or while constructs. This saves confusion and mistakes, and it does not cause the compiler to generate any extra code.

  WRONG:

```
while( ... ) {
```

```
...
}
```

RIGHT:

```
while( ... )
{
        ...
}
```

- Even though C# allows a looping or conditional construct within a single body line to forego the brackets {}, brackets should always be used. This reduces the potential for errors if another line of code is added later and makes the code more readable.

WRONG:

```
while ( true )
DoWork();
```

RIGHT:

```
while ( true )
{
        DoWork()
}
```

3.5 Line Length

Comment and code lines should not extend too long. Previously, this was 80 characters, but with C# more verbose and screens having higher resolutions, this can vary. Now, this is more subjective.

3.6 Comments

- Use double slashes for comments. This precludes having to use an ending */.

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

- Use // TODO to mark places where code still needs to be implemented, tested or modified.
- Indent comments to the same level as the code to which it applies.
- Under most conditions, do NOT put comments at the end of a code line. The comment should immediately precede the lines to which it applies.
- Do not use stars at end of comments (box format).
- Use XML tags for documenting types and members. Example:

```
/// <summary>
/// Initializes a new instance of the <see cref="DaHandler"/> class.
/// </summary>
```

The following Commenting Checklist was taken from *Code Complete: A Practical Handbook of Software Construction, 1st Edition* by Steven C. McConnell (1993).

- Does the source listing contain most of the information about the program?
- Can someone pick up the code and immediately begin to understand it?
- Do the comments explain the codes intent or summarize what the code does, rather than just repeating the code?
- Has tricky code been re-written rather than commented?
- Are the comments up to date?
- Are comments clear and correct?
- Does the commenting style allow comments to be easily modified?

- Do the comments focus on *why* rather than *how*?
- Do the comments prepare the reader for the code to follow?
- Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
- Are surprises documented?
- Have abbreviations been avoided?
- Is the distinction between major and minor comments clear?
- Is code that works around an error or undocumented feature commented?
- Are units on data declarations commented?
- Are the ranges of values on numeric data commented?
- Are coded data meanings commented?
- Are limitations on input data commented?
- Are flags documented to the bit level?

3.7 Spacing

Use spaces for clarity. If necessary, sacrifice space for readability. Again, these settings can be set per project using **StyleCop**.

OK:

```
for( i = 1; i < 5; i++ );
```

NOT:

```
for(i=1;i<5;i++);
```

3.8 Miscellaneous

- Only use “this.” to prevent name clashing (if the method parameter matches a class field).
- Avoid multiple or conditional return statements.
- If implementing one of the Object methods (e.g., “Equals”, “GetHashCode”), it is required that both must be implemented. Also override when implementing the “IComparable” interface.
- Avoid the use of exceptions as flow control. Exceptions should be thrown in exceptional circumstances.
- Always log that an exception is thrown.
- Code reviews are completed using the checklist in Appendix C. When developing code, the checklist should be gone over to ensure compliance.

4. Coding Practices

4.1 Clarity

Write code for clarity and understanding. Leave the optimization for the compiler. Do not try to guess where performance “bottlenecks” will occur. If it is later decided that the code runs too slowly, performance tools can help identify the true “bottlenecks.” Use parenthesis to make code clearer (but do not get carried away). If the order of operation is not intuitively obvious, use parenthesis.

4.2 Object Scope

In a class definition, all member variables shall be private or protected. Access to member variables by anyone other than a derived class shall be through either member methods or the get set accessors.

```
public UddiKeyCollection Keys
{
        get
        {
                return this.keys;
        }
        set
        {
                this.keys = value;
        }
}
```

4.3 Constants

A constant is an expression that can be fully evaluated at compile time. Constants should be all upper case with underbars:

```
public const int MAX_QUEUE_ITEMS = 100;
```

The same holds true for constant enumerations.

4.4 Flags

Flags are only to be used to mark events or options. Not counting the flag initialization, there should only be one place the flag is set, and one place the flag is cleared. It can be checked any number of places. If multiple flags are required to control the logical flow of a process, and these flags are set or cleared in more than one place, then what is really present is state processing. Define a state variable, constants for the various states, procedures for entering (and/or exiting if necessary) the defined states, and the program logic for each event in each state.

4.5 Data Types

The system namespace is the root namespace for fundamental types in the .NET Framework. This namespace includes classes that represent the base data types used by all applications: “Object” (the root of the inheritance hierarchy), “Byte”, “Char”, “Array”, “Int32”, “String”, and so on. Many of these types correspond to the primitive data types that C# uses. When writing code using .NET Framework types, use the C# corresponding keyword when a .NET Framework base data type is expected. The following table lists some of the value types the .NET Framework supplies, briefly describes each type, and indicates the corresponding type in C#. The table also includes entries for the Object and String classes, for which C# has corresponding keywords.

Category	Class Name	Description	C# Data Type
Integer	Byte	An 8-bit unsigned integer.	byte
	SByte	An 8-bit signed integer. Not CLS compliant.	sbyte
	Int16	A 16-bit signed integer.	short

	Int32	A 32-bit signed integer.	int
	Int64	A 64-bit signed integer.	long
	UInt16	A 16-bit unsigned integer. Not CLS compliant.	Ushort
	UInt32	A 32-bit unsigned integer. Not CLS compliant.	Uint
	UInt64	A 64-bit unsigned integer. Not CLS compliant.	ulong
Floating point	Single	A single-precision (32-bit) floatingpoint number.	float
	Double	A double-precision (64-bit) floating-point number.	double
Logical	Boolean	A Boolean value (true or false).	bool
Other	Char	A Unicode (16-bit) character.	char
	Decimal	A 96-bit decimal value.	decimal
	IntPtr	A signed integer whose size depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform).	IntPtr No built-in type.
	UIntPtr	An unsigned integer whose size depends on the underlying platform (a 32- bit value on a 32-bit platform and a 64-bit value on a 64-bit platform). Not CLS compliant.	UIntPtr No built-in type.
Class objects	Object	The root of the object hierarchy.	object
	String	An immutable, fixed-length string of Unicode characters.	string

4.6 GUIs

C# GUIs will be consistent with either those of the SunGuide system. Guidance for use of and controls can be found in the MSDN. Furthermore, GUIs will be presented to the FDOT during the appropriate design reviews for final approval.

4.7 Generated Code

Code that is automatically generated from the Visual Studio development environment, or other tools such as the XML Schema Definition Language (XSD) utility, does not have to comply with this coding standard. Do not attempt to change generated code, in most cases modifying the code breaks it. StyleCop can be made to ignore these files by adding a tag at the top of the file.

4.8 XML

Since XML is tightly integrated into C# and the .NET framework, it is briefly addressed here.

- When designing XML schema, XSD should be used. Document Type Definitions (DTD) will not be used.
- XML tags should be all lower case where possible. This convention does not apply when generating schema using automated tools. For instance, generating a DataSet schema from an existing database table.
- XML documents should be indented for readability. Where possible, build a hierarchical structure of data types by including multiple schema of simpler types.
- When creating XML for transmission, create in an appropriately indented format. This will allow the text to be human readable without needing to be reformatted.