Coding Standards & Guidelines

iInterchange System Pvt. Ltd.

Version 1.0

Date : 06-May-2011		Code : CSG
PREPARED BY	**REVIEWED BY**	**APPROVED BY**
V. Jayakrishnan	S.Thiruppathi	V. Balaji

iInterchange™

Your Indian Technology Partner

www.iInterchange.com

REVISION HISTORY

VERSION NO	AUTHOR	RELEASE DATE	CHANGE DETAILS		APPROVED BY
			SECTION	DESCRIPTION	
1.0	V.Jayakrishnan				
1.1	S.Thiruppathi	09-Jun-11		Documentation	
1.2	V. Jayakrishnan	19-Sep-11		Javascript Rules added	
1.3	G.Govindarajan	11-MAR-13		**Browser Compatability Rules**	

Formatted: Indent: Left: 0", Hanging: 0.25", No bullets or numbering

1. Document Overview

This document is a tool to aid in the implementation of solid development practices to ensure the creation of consist, readable, and functional code. The development rules guidelines, and specification outlined in this document will benefit developers in the maintenance, bug fixing, upgrades, and performance of existing and continued coding efforts. Although this document does outline many rules and guidelines for development it still allows for developer independence when creating algorithm, code flow, and code implementation. But, as one developer touches another developer's code, the bases of the code should look as if one developer had created it. Many of these rules and guidelines are probably being implemented. There may be rules that are used but no seen in the list. If this is the case, please notify iInterchange so that they may be added.

2. Specifications

2.1. Variables

All variables should be declared with a combination of both the scope prefix and declaration type prefix. A minimum of one single uppercase letter should be used to separate words of a variable name. Uppercase letters should never be used for the scope or declaration type of a variable declaration. The following is a format for variable declaration:

<scope prefix>_<declaration type prefix><Name> AS <datatype>
Examples:

```
Private pvt_intRecordCount As Integer
Public pub_strUserName As String
```

2.2. Scope Prefix

All variable scopes are 3 letter combinations in lowercase letters followed by an underscore.

~~**2.2.1. Module**~~

~~The following prefixes to be followed in all the modules.~~

~~Private pvt_~~
~~Public pub_~~
~~Friend fnd_~~

~~2.2.2. Classes and Forms~~

~~The following prefixes to be followed in all the Classes and Forms.~~

```
~~Private pvt_~~
~~Protected pro_~~
~~Friend fnd_~~
~~Public no prefix~~
```

~~2.2.3.~~2.2.1. Methods

There is no need to add prefix in case of local variables.

2.3. Declaration Type Prefix

All variable declarations types are 3 letter combinations in lowercase letters.

int Integer	**typ** Type	**txt** iTextBox	**rptr** Repeater
i16 Int16	**col** Collection	**dat** iDate	**pnl** Panel
i32 Int32	**ds** DataSet	**lkp** iLookup	**rv** ReportViewer
i64 Int64	**dt** DataTable	**ifg** iFlexGrid	**mnu** Menu
dbl Double	**dr** DataRow	**ddl** Dropdownlist	**tv** TreeView
sng Single	**dc** DataColumn	**lb** ListBox	**tmr** Timer
bln Boolean	**astr** String()	**chk** Checkbox	
byt Byte	**aint** Int()	**btn** Button	
vnt Variant	**ex** Exception	**div** Div	
str String	**bv** ByVal	**tbl** Table	
dat Date	**br** ByRef	**hdn** HiddenField	
chr Char	**adr** DataRow()	**img** Image	
dec Decimal		**fm** iFrame	
obj Object		**fl** InputFile	
sbr StringBuilder		**rbtn** RadioButtion	
hsh Hashtable		**lv** ListView	
arr Array		**gv** GridView	
als ArrayList		**fv** FormView	

If a variable is of an array, then the above declaration types should be prefixed with “a”.

Example:

```
Public pub_aintScores(4) As Integer
```

2.4. Name

All variable names should be no fewer than 3 characters in length. Variable names should be meaningful and intuitive in their use. The use of x or I is not allowed as a name.

2.5. Constants

All variable names should be no fewer than 3 characters in length. Variable names should be meaningful and intuitive in their use. The use of x or I is not allowed as a name.

Constants are declared in all uppercase letters with the exception of the scope and declaration type prefixes. Constants do adhere to the rules of variable definition followed by the Name.

Example:

```
Public Const pub_strHOME_URL As String = "http://URL"
```

3. Classes

All classes should be well thought out, compact in design, reusable, and well documented. ~~The following is the format for class definition;~~

~~<MODULE prefix>_<Name>~~

~~When a type is defined in the name of a class, refer to the variable definition for the valid values~~.

3.1. Definition

Class name should be meaningful and intuitive to use and should be at least 8 characters in length.

3.2. Inheritance

Classes should be built as compact and streamlined as possible and thought of as building blocks to be reused. Although all classes will not be inherited or bases classes for other classes, building reusable objects should always be the forethought in designing classes.

3.3. Polymorphism

The use of polymorphism is encouraged. Polymorphism by nature breaks objects down into independent entities, streamlines code, makes object smaller, helps to isolate bugs, and limit the time necessary for maintenance. ~~The base class for any interface that is surrounded by a namespace should use the name of clsBase.~~

3.4. Design Patterns

The understanding and use of Design Patterns is beneficial in the development and implementation of solid class structures. Wherever possible, Envision suggests that the Design Pattern practices be utilized.

3.5. Error Classes

Application Error Classes are available in iInterchange Framework component. The use of Application Error Classes is to resolve the issues in the live environment quickly. The Errors should be thrown from Data Access layers, Business Logic layers and Presentation Layers. Although they are very similar in their implementation and functionality, there will be cases where the error messages thrown will be unique to the specific environment for which they belong.

4. Methods

Method names should be meaningful and intuitive to use and should be at least 6 characters in length. Avoid the use of the words Get and Set at the beginning of method name to separate methods. Use overloading instead.

Example:

```
Public Function pub_GetUserName() as String
Public Function pub_SetUserName(ByVal bv_Name as String)
```

4.1. Passed Parameters

Parameters should be passed in a manner for which they are being used. If the value of a parameter is to be changed within the method then it should be passed by reference otherwise passed by value. When declaring parameters, they should be explicitly defined as either passed by reference or by value. A parameter declaration is as follows:

<Usage> <prefix>_<declaration type><Name> AS <datatype>

Examples:

```
ByRef br_strName as String
ByVal bv_strName as String
```

4.2. Indentation

Tab size and Indent size should be set to 4 spaces. This must be set the same for all developers who will be working on or sharing code.

4.3. Regions

Regions are very useful when it comes to easy navigation and finding particular method quickly. This must be used for every method with keyword (CREATE, UPDATE, GET, VALIDATION) for identification.

Examples:

```
#Region "CREATE : pub_IMPIGMCreateIMP_IGM()"

#End Region

#Region "UPDATE : pub_IMPIGMUpdateIMP_IGM()"

#End Region

#Region "GET: pub_IMPIGMGetIMP_IGM()"

#End Region

#Region "VALIDATION: pub_IMPIGMValidateContainer()"

#End Region
```

4.4. White Space

Liberal use of white space is highly encouraged. This provides enhanced readability and is extremely helpful during debugging and code reviews. The indentation example above shows an example of the appropriate level of white space.

Note:

Blank lines should be used to separate logical blocks of code in much the way a writer separates prose using headings and paragraphs. Note the clean separation between logical sections in the previous code example via the leading comments and the blank lines immediately following.

4.5. Long lines of code

Comments and statements that extend beyond 80 columns in a single line can be broken up and indented for readability. Care should be taken to ensure readability and proper representation of the scope of the information in the broken lines. When passing large numbers of parameters, it is acceptable to group related parameters on the same line.

Example:

```
Private Function FindRedCansByPrice( _
         ByVal price As Decimal, _
         ByRef canListToPopulate As Integer, _
         ByRef numberOfCansFound As Integer) As Boolean
```

4.6. Namespaces

Namespaces should be used to group a collection of related classes, forms, or other objects together. Using namespaces will help in understanding the relationships of objects that support each other.

4.7. Error Trapping

All Functions, Subroutines, and Properties with more than one line of code should have a TRY CATCH Block.

Example:

```
Try
<Statements>
Catch Exception
<Collect and Assign Inner Exception Information>
End Try
```

Once an error has been trapped, the follow is the sequence for collecting the error information;

1. The ConstMethodName should be assigned to the error object.
2. Any error message should be assigned to the error object.
3. The appropriate throw statement should be executed when necessary.

The throw is done to send the error information up the calling stack to the highest level where it will be logged. An error should never be logged unless the object logging it is known to be the highest level object in the call stack

5. Code Documentation

Documenting code is just as important as the code itself. Code that is well documented saves money, time, aggravation, and helps the maintenance of future upgrades and bug fixes.

5.1. Statements

Although not every coded statement needs to be documented and good solid code can be self-documented, it is still important to document individual coded statements. The important key elements to keep in mind are lines of code that play a key role, if blocks, activity within a loop, and those lines of code that are not self-documenting.

5.2. Variable Declarations

When declaring variables each variable should have at least one line of comment describing the use of the variable. If the variable is relied upon by or reliant upon another variable this should be noted.

5.3. Classes, Forms, and Modules

The following documentation comments should go at the top of every class, form, or module that is created.

```
''' <summary>
~~''' Name:~~
~~''' NAME OF THE CLASS, FORM OR MODULE~~
~~''' Purpose:~~
''' CREATE A GOOD SOLID DESCRIPTION OF WHAT PURPOSE THE
''' OBJECT SERVES
''' </summary>
''' <remarks>
''' ~~Changes and Change Date~~Add any validations or remarks to be noted
~~''' NOTE ANY CHANGES TO THE OBJECT LIKE ADDED METHODS OR~~
''' ~~VARIABLES OR BUG FIXES~~</remarks>
```

5.4. Methods

The following documentation comments should go at the beginning of every method.

```
''' <summary>
~~''' Name:~~
~~''' NAME OF THE FUNCTION, SUBROUTINE OR PROPERTY~~
~~''' Purpose:~~
''' CREATE A GOOD SOLID DESCRIPTION OF WHAT PURPOSE THE
''' METHOD SERVES AND ANY RULES THAT GOVERNS THE USE OF THE METHOD
''' </summary>
''' <param name="param1">PURPOSE OF THE PARAMETER AND ANY VALUE RESTRICTIONS</param>
''' <returns>WHAT THE RETURN TYPE IS AND ANY VALUE RESTRICTIONS</returns>
''' <remarks>
''' Add any validations or remarks to be noted
~~Changes and Change Date~~
~~''' NOTE ANY CHANGES TO THE OBJECT LIKE ADDED~~
''' ~~VARIABLES OR BUG FIXES~~</remarks>
```

5.5. Commenting

Commenting code is also as important as the code itself. A Business Logic that is well documented saves time, aggravation, and helps the maintenance of future upgrades and bug fixes.

Note:
Commenting business logic code blocks are not encouraged. This will help to maintain clarity in the business logics.

Example:

```
'Dim stateSalesTax As Decimal
'Dim citySalesTax As Decimal
```

2.2.1. End-Of-Line Comments

Use End-Of-Line comments only with variable and member field declarations. Use them to document the purpose of the variable being declared.

Example:

```
Private name As String = String.Empty ‘ User-visible label for control
Private htmlName As String = String.Empty ‘ HTML name attribute value
```

2.2.2. Single Line Comments

Use single line comments above each block of code relating to a particular task within a method that performs a significant operation or when a significant condition is reached.

Example:

```
‘ Compute total price including all taxes
Dim stateSalesTax As Decimal  = CalculateStateSalesTax(amount, Customer.State)
Dim citySalesTax  As Decimal  = CalculateCitySalesTax(amount, Customer.City)
Dim localSalesTax As Decimal  = CalculateLocalSalesTax(amount, Customer.Zipcode)
Dim totalPrice   As Decimal  = amount + stateSalesTax + citySalesTax + localSalesTax
```

Note:

- ✓ Comments should always begin with a single quote, followed by a space.
- ✓ Comments should document intent, not merely repeat the statements made by the code.
- ✓ Use an imperative voice so that comments match the tone of the commands being given in code.

6. Rules

The following rule set has been collected over the years through various sources and developers experiences. It should be considered the norm with very few if any exceptions.

6.1. Single line “if...then” statements must not be used

In the interest of making source code more readable, single line “if…then” statements are to be avoided i.e. the ‘End If’ keyword must be used for all ‘if’ statements.

6.2. The : should not be used to separate statements

Each statement should be on its own line.

6.3. Variable declaration should be ordered in the following manner

Constants, Private, Protected, Public. When declared in a method the order should be Constants, Dim.

6.4. Constant declarations should be ordered in the following manner:

Private, Protected, Public.

6.5.Method bodies, whether functions, procedures must not exceed 50 lines

For readability and possible reuse, if a method becomes more than 50 lines of code, break the method up into smaller methods.

6.6. Property procedures must not exceed 5 lines

Property procedures should be short and quick executing code bits and generally limited to accessing private member variables.

6.7. Constants declaration type must be explicitly specified using "As" keyword

Although a constant can't be changed, it's still a good idea to define the constant as the data type it is using.

6.8.Constants must be correctly formed

<scope prefix><declaration type prefix><body> The body of a constant's name must always be uppercase and underscore characters are used to separate individual words.

6.9.Shared constants and enumerators should be placed in grouped module files

Creating large module files of constants and enumerators is not permitted. Break up the constants and enumerators into smaller module files. It is better to have to include many module files than to include a module file with many constants or enumerators that will not be used within a program.

6.10. All function, subroutines, and property methods must use "Try…Catch" blocks for error trapping

To ensure good error trapping, a "Try…Catch" block must exist in all methods. Multiple catch statements are preferred if multiple type of error can be generated.

6.11. Using "Catch Exception" should be the last catch statement in a "Try…Catch" block

Envision prefers that specific classes are created for catching errors. These classes will inherit the Exception class as a base class. The specific classes should be listed first in the catch statements with the Catch Exception last.

6.12. Re-throw errors from called methods to the calling method

If an error is caught, it should be re-thrown to propagate up through the calling stack.

6.13. Allocate a unique DLL base address

To minimize memory usage and achieve optimal loading of DLLs, allocate a unique base address when building the component.

6.14. Always use "Option Explicit"
Use Option Explicit to force the declaration of variables.

6.15. Never use "Option Base"
The "Option Base" statement determines the default base index for an array. By default, the base index is 0 and should not be altered.

6.16. Early binding is preferred over late binding
By default, use early binding on components unless it's absolutely necessary to use late binding.

6.17. Explicitly declare variables using the object type. Do not use the object variable type
When creating variables of other object types like class, form, etc… use the explicit object name instead of the object data type and the CType() function.

6.18. Avoid "Lbound()," "Ubound," and "Count property" in "For…Next" loops
When iterating an array variable using a For…Next loop, assign a variable the value returned from a call to Lbound(), Ubound, or Count prior to commencement of the loop.

6.19. Avoid null string comparisons
When testing to see if a string is empty, use the Len() function instead of comparing a variable to ""

6.20. Use string character functions instead of variant character functions
Avoid the use of the variant "chr()", "left()", "mid()", etc… functions; use "chr$()", "left$()", "mid$()", etc… function instead.

6.21. Avoid the use of "+" operator for string concatenation
The "+" operator must not be used for string concatenation. Always use the "&." All parameters to a method must be explicitly defined. When creating a parameter for a method, it must be explicitly defined with byval or byref, and use the "As" keyword to define the declaration type.

6.22. Return data types for function must be specified
Each function that returns a value should be explicitly defined even if a variant is going to be returned.

6.23. All functions, subroutines, and properties must have a single exit point
Having a single exit point simplifies debugging and can help to ensure that clean-up code is executed prior to exiting the method.

6.24. Optional parameters must always be assigned a default value
When creating optional parameters, be sure to assign them a default value to insure that they are not null or empty.

6.25. "IsMissing()" function should never be used
Since all optional parameters are assigned a default value, the "IsMissing()" function is not necessary.

6.26. The “Public” keyword for variables in classes should be used carefully

A private variable with a property for access is the preferred method of exposed variables in classes.

6.27. Put it on paper first

Throughout the design of a projects objects, data model, and algorithms should be thought out and designed, even if hand drawn, on paper first. Talk about issues with a peer to gain another perspective.

6.28. Avoid including “using” statements or references that are not necessary

Only include Using statements for items that are actually being used all others should be removed.

6.29. All classes must have a “New” and “Finalize” subroutine defined even if they are empty

Since the complier will put these subroutines in the code automatically it’s a good idea to go ahead and explicitly implement them in the editor.

6.30. All variable declarations are to be explicitly defined

When defining a variable there should always be a data type defined. Explicitly define the data type for which the variable will use even if it’s a variant, which is the default data type.

6.31. Never use “Redim” to change the declaration type of a variable

Redim should only be used to change the size of an array; but never change the data type of a variable.

6.32. Use “Redim” and “Preserver” sparingly

Redim/Preserves are very expensive due to the shifting of memory is required. If using Redim/Preserve is necessary, do it in large blocks and use a counter to keep track of the actual size and shrink the size as the last step.

6.33. All property declarations must use the “As” keyword to define the type expected to be used

Properties are the interface to private variables. As such, properties should use the AS keyword in the SET block and declaration and be defined with the same data type as the variable for which they are governing.

6.34. All functions, subroutines, and properties should have a “Try Catch” block

Error trapping is one of the most important parts of any successful development effort. Every function, subroutine, and property that has more than one of code, should have a “Try…Catch” block. Refer to the Error Trapping Section of this document for details.

~~6.35. All functions, subroutines, and properties need a constant with the method and class name~~

~~In order to create a proper audit trail for fixing bugs and identifying errors, the first line of every method will be as follows CONST [[[DLL Name], [EXE Name]], [COMPONENT Name], [METHOD Name]]~~

~~Example:~~

~~constMETHODNAME = "[Logging Controller.clsLogger.Write]"~~

~~The preceding tells us that something happened in the Logging Controller DLL in a class called clsLogger with a method called Write. Refer to the Error Trapping section (3.4) for usage.~~

~~6.36.~~6.35. Minimize the use of "." separators to enhance readability and performance

When referencing an objects methods, properties, or operations the "." should not be used more than twice. If the "." is being used more than twice, Envision prefers that an assignment to a variable of the referenced item be used. This will allow for more readable code and the code will execute faster as segment/offset references are expensive in terms of performance.

Example:

Here a XML stream has several nodes that have a firstchild reference.

xmlDoc.firstchild.firstchild.firstchild.selectSingleNode("SOMENODE")

The preferred syntax would be

xmlNode = xmlDoc.firstchild.firstchild

xmlNode = xmlNode.firstchild.selectSingleNode("SOMENODE")

~~6.37.~~6.36. Use FOREACH when iterating through collections

When iterating collections use the FOREACH interface instead of a standard FOR loop with counter variable control. FOREACH is more intuitive and executes faster than a standard FOR loop.

7. Javascript Rules

- Comment should be added in every javascript method.

  Ex:

```
//Initialize page while loading the page based on page mode
function initPage( mode) {
}
```

- Method name prefix "fn" should not be used.
- The method name has to indicate word boundaries using camelCase. thus rendering "two words" as either "twoWords" or "TwoWords".
- Try catch should not be used.
- Variable names should have datatype prefix with single letter.

  Ex:

  var sName;//For data type string

var iAge;//For data type integer

var dEstimateDate;//For data type date

var oCallback = new Callback() // For object type

- Global variable names should have data type prefixed with g and with data type single letter.

  Ex:

  var gsName;//For data type string

- Any object used in the method. It should disposed at the end of method.

Ex:

```
        oCallback = null;
```

8. Browser Compatability Rules

Formatted: Heading 2

8.1 Aspx Pages And CSS

Formatted: Font: 12 pt, Bold

Formatted: Normal, Indent: Left: 0.25"

- "expressions" should not be used in page/CSS for height/width, Instead min-height,max-height can be used.

Formatted: Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

  Ex:

Formatted: Indent: Left: 0.5"

```
Instead of this,
          width: expression(document.body.offsetWidth - 5);
Use,

                max-width:995px;
                min-width:600px;
```

Formatted: Indent: First line: 0.5"

Formatted: Indent: Left: 0.5", First line: 0.5"

Formatted: Indent: Left: 0.5"

Formatted: Indent: First line: 0.5"

- Should use CSS Class "tblstd" for all the table markups.

Formatted: Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

- For Range validation both minimum/maximum values has to be given.
- Multiple Row Tabs Should Not be Used.
- Static header height should be 10px lower than the height in IE for list and other grid master pages.

8.2 JavaScript

Formatted: Font: 12 pt, Bold

Formatted: Normal, Indent: Left: 0.25"

- Value for elements other than textbox and Lookup has to be done using wrapper function for all javaScript statement.

Formatted: Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

- window.showModalDialog, window.confirm should not be used instead custom modal dialog can to be used.

- Should not handle event object in javascript.
- Accessing DOM object should be done using corresponding wrapper functions.
- Accessing iframe's variables,scripts,document will be done through wrapper functions.

Formatted: Indent: Left: 0.5"

Ex:

Instead of ,

```
        document.frames["FrameID"].Script.loadPageData( framedoc.URLUnencoded, sQryStr, sMode, iItemno);
```

Formatted: Indent: Left: 0.5"

Use,

```
       getIFrameObj("FrameID ").loadPageData(framedoc.URLUnencoded, sQryStr, sMode, iItemno);
```

Formatted: Indent: Left: 0.5"

Formatted: Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

- Controlling the Browser's addressbar,maximize,minimize button will not be supported.
- JavaScript will not Close any browser window unless it is opened by the same window.
- Should not call any events or attach any events to Controls manually.
- Use setText(obj,value) to set the Text value of any element..

Formatted: Indent: Left: 0.5"

Ex:

Instead of,

```
el("messagetitle").innerText=" INFORMATION :";
```

Use,

```
setText(el("messagetitle"),"INFORMATION :");
```

Formatted: Indent: First line: 0.5"

Formatted: Indent: Left: 0.5"

- Use getText(obj) to set the Text value of any element..

Ex:

Instead of,

```
if (getText(el('btnFetch')) == "Fetch")
```

Formatted: Indent: First line: 0.5"

Use,

```
if (el('btnFetch').innerText == "Fetch")
```

- Use DBC() wrapper function insted of calling function document.body.click() directly.\
- To set properties such as Value,Readonly,Clear Values,Show/hide,Use Corresponding wrapper functions.

Ex:

For Textboxes Instead of ,

```
el("ContractWith").value="";//To Clear values
el('lkpYear').readOnly=true;//To Set/Reset Readonly
el('sampleDiv').style.display="none";//To hide element
el('sampleDiv').style.display="block";//To show element
el("txtUserID").focus();//To Set focus

Use,
clearTextValues("lkpContractWith");//To Clear values
setReadOnly('lkpYear',true); //To Set/Reset Readonly
hideDiv(sElementID); //To hide element
showDiv(sElementID); //To show element
setFocusToField("txtUserID ");//To Set focus

For Lookups,
Use,
clearLookupValues("lkpYear");
```

- , Use Corresponding element's Value for CSS property display.

Ex:

For displaying any TR element,display:"table-row" should be used instead of "block",

Similarly for TABLE "table" should be used and for TD use "table-cell".

Formatted: Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5"

Formatted: Indent: Left: 0.5"

Formatted: Indent: First line: 0.5"

Formatted: Indent: First line: 0.5"

Formatted: Indent: First line: 0.5"

Formatted: Indent: First line: 0.5"

Formatted: Font: (Default) Arial, 10 pt

Formatted: CSG_Paragraph, Bulleted + Level: 1 + Aligned at: 0.25" + Indent at: 0.5", Adjust space between Latin and Asian text, Adjust space between Asian text and numbers

Formatted: Font: Arial, 10 pt, Font color: Auto, Pattern: Clear

Formatted: Font: (Default) Arial, 10 pt

Formatted: CSG_Paragraph, Indent: Left: 0.5", Adjust space between Latin and Asian text, Adjust space between Asian text and numbers

Formatted: Indent: First line: 0.5"

Formatted: Indent: Left: 0.5"