



.NET CODING STANDARDS & BEST PRACTICES

Version 1.1
November 7, 2007

CHANGE HISTORY

Date	Version	Author	Change Description
6/27/06	.1	Hy Tran	Initial Draft
8/8/06	.2	Doug Swingle	Updated to reflect JUSTIS style and coding standards book excerpts.
8/17/06	.3	David Mullings	Revised & Reorganized
3/24/07	.4	David Mullings	Additional guidelines added and document revised.
04/02/07	.5	David Mullings	Expanded section dealing with structuring source code.
06/01/07	.6	Doug Swingle	Added practices for working with SQL Reporting.
06/11/07	.7	Bashir Mohammed	Added Team Foundation Server best practices.
06/14/07	1.0	David Mullings	Initial Release
11/07/07	1.1	David Mullings	Modified Source Code Organization section.

Table of Contents

1	Introduction	6
2	Programming Standards	7
2.1	Primary Programming Language.....	7
2.2	Web Platform.....	7
2.3	Windows Application Platform	7
2.4	Database Operations.....	7
2.5	Coding Style	7
2.6	Keep it Simple, Stupid.....	8
3	Naming Convention and Standards	9
3.1	Use meaningful, descriptive words to name types.....	9
3.2	Use Pascal Case when naming types	9
3.3	Do not use using Hungarian notation	9
3.4	Do not use Camel Casing	9
3.5	Delineate class members from method members	10
3.6	Prefix the name of an Interface type with the letter “I”	10
3.7	Custom exception types should end with the word, Exception	10
3.8	Abstract types should end with the word, Base	10
3.9	Prefix Boolean variables and properties with the words “Can”, “Is” or “Has”	10
3.10	Do not include the parent class name within a property name	10
3.11	Common generic variable names.....	11
3.12	Common prefixes for Win Form controls.....	11
3.13	Common prefixes for Web Form controls.....	12
4	Comments.....	14
4.1	General Rules for Comments.....	14
4.2	Spaces and Indentation	14
4.3	File Header Comments	14
4.4	Method-Level Comments	15
4.5	Block-Level Comments	15
4.6	Statement -Level Comments.....	16
4.7	Code Change Comments	16
5	Source Code Organization.....	18
5.1	Consider using the secondary drive for all your work.....	18
5.2	Follow the standard project file structure	18
5.3	How To: Create a new application projects.....	20
5.3.1	Step 1 – Create Root Folder.....	20
5.3.2	Step 2 – Create a Blank Solution	20
5.3.3	Step 3 – Add a Web Site to Your Solution	20
5.3.4	Step 4 – Add a Class Library to Your Solution	21
5.3.5	Step 5 – Add a Windows Form Project to Your Solution.....	21
5.4	Code one class definition per source file.....	21
5.5	Name the file after the class.....	21
5.6	Organize types under the DCPretrial root namespace	22
5.7	Organize members within a class	22
6	Exception Handling Best Practices.....	24
6.1	Derive custom exceptions from the PSAException type.....	24
6.2	Use Try blocks instead of On Error statements	24
6.3	Do not use exceptions to control execution flow.....	24
6.4	Throw Exceptions only when you need to.....	24

6.5	Order multiple Catch statements carefully	24
6.6	Re-throw exceptions caught in class libraries.....	25
6.7	Write User Friendly Error Message.....	25
6.8	Protect again revealing sensitive data.....	25
6.9	Keep unhandled exceptions from being displayed to end-users	26
7	Security Best Practices	28
7.1	Craft with security in mind.....	28
7.2	Never trust what you've been given	28
7.3	Be careful how and where you store connection strings.....	28
7.4	Use your own credentials when connecting to databases	28
7.5	Use stored procedures to execute queries	28
7.6	Protect SQL Code from injection attacks	28
7.7	Use HtmlEncode to protect against Cross-site Scripting	29
8	Unit Testing Best Practices.....	30
9	Team Foundation Server Source Control Best Practices & Guidance.....	31
9.1	Getting the latest version of source code	31
9.2	Files kept and not kept in source control	31
9.3	Checking out code for editing.....	32
9.4	Checking in modified code.....	33
9.5	Deleting code from Source Control	33
9.6	Sharing shelved code	34
9.6.1	Shelving working code	34
9.6.2	Un-shelving working code.....	34
9.6.3	Deleting shelved code.....	35
10	General Coding Best Practices	36
10.1	Turn on Option Strict, Option Explicit and Option Compare for all projects.....	36
10.2	Don't declare multiple variables on the same line.....	36
10.3	Never define public instance fields.....	36
10.4	Avoid having multiple statements on one line.....	37
10.5	Don't code single-statement If blocks	37
10.6	Use the AndAlso and OrElse instead of the And and Or operators	37
10.7	Don't compare Boolean types to true/false values	38
10.8	Use direct assignments to set Boolean types	38
10.9	Avoid the IIF function	38
10.10	Testing and comparing object types	39
10.11	Favor Select Case blocks to test for multiple values	39
10.12	Never code multiple statements on a Case block line.....	39
10.13	Put most likely to occur condition at top of Select Case statement	40
10.14	Declare controlling variable in For and For each loops.....	40
10.15	Don't use an array's length property to control For loops	41
10.16	When to use For ... Next and For Each ...Next loops.....	41
10.17	Doing For loops when the controlling variable are not sequential	41
10.18	Use the & operator to concatenate strings	42
10.19	When to use char variables	42
10.20	Initialize string types to String.Empty	42
10.21	How to split long string expressions.....	42
10.22	Looping over all the characters of a string.....	42
10.23	Avoid passing types by reference	43
10.24	Use method overloading to reduce boxing	44
10.25	Use method overloading to reduce the number of arguments.....	45
10.26	Use method overloading rather than relying on optional parameters	45
10.27	Validate all arguments before using them	46
10.28	Have a single exit point for all methods	46
10.29	Use the Return keyword to return values from a method or property	47

10.30	Methods and properties that return strings	47
10.31	Use the Get-prefixed to name methods that return values	48
10.32	Get-prefixed methods matching property names	48
10.33	Set-prefixed methods	48
10.34	Consider using parentheses to return the results of an expression	49
10.35	Don't use properties as methods or methods as properties	49
10.36	Use Enums to return result status codes	50
10.37	Return Zero-element Arrays over Un-initialized arrays	50
11	ASP.Net Best Practices.....	51
11.1	Use Model-View-Presenter Pattern	51
11.2	Validate input on both the client and server	51
11.3	Set the ViewStateUserKey property on all web pages	51
11.4	Redirect using Server.transfer or Server.Execute	51
11.5	Favor the ViewState dictionary over hidden fields.....	51
12	Data Access Best Practices.....	52
12.1	Access the database through the Agency's Data Access Layer.....	52
12.2	Don't use Datasets to pass data between application tiers.....	52
12.3	Data Reader vs. DataSet	52
13	SQL Reporting Best Practices	53
13.1	Report Standard Fonts	53
13.2	Report Margins	53
13.3	Naming conventions	53
13.4	Using Reporting Services Embedded Code.....	54
13.5	Report Readability (for Summary Reports).....	54
13.6	Naming Data Sources and Data Sets	55
13.7	Deploying Reports.....	55
13.8	Use Visual SourceSafe	55
13.9	Use Shared Data Sources.....	56
13.10	Use Views and Stored Procedures	56
13.11	Create a backup of the Encryption Key	56
13.12	Review Reports Before Deploying	56
13.13	Use folders and Descriptions to Organize Reports	56
13.14	Assign Security at the Folder Level.....	56

1 INTRODUCTION

The coding standards and best practices outlined in this document reflect the current practices and standards used by the Software Development Team to build and maintain applications in support of PSA's mission.

These standards and best practices are based upon industry recognized standards and are intended to provide developers with practical guidance, dos and don'ts, pitfalls, naming conventions and styles, and recommendations for writing source code. This document, however, is not a definitive, all inclusive guide, but a general introduction to some of the more common rules to follow when constructing .Net source code. Software development, after all, is both a skill and an art form. This document is intended to help developers learn the techniques needed to practice this art form to the highest skill level possible.

This document is based upon *Practical Guidelines and Best Practices for Microsoft Visual Basic and Visual C# Developers* by Francesco Balena & Giuseppe Dimauro (Microsoft Press, 2005). All agency developers should read the book in its entirety because of its great wealth of practical information on how to write clean, efficient and easily maintainable .Net source code.

Several of the recommended practices and standards made by Microsoft's Pattern and Practices Group have been adopted by the Agency and have been included in this document as well.

It is expected that all code written by the Software Development Team and contractors utilized by this agency conforms to the standards and best practices outlined within this document. Because of such, this document also serves as the basis for conducting periodic code reviews of source code to insure that what is developed for this Agency represents the highest quality of software construction achievable.

2 PROGRAMMING STANDARDS

2.1 *Primary Programming Language*

Visual Basic.Net has been designated as the primary programming language for building custom applications within PSA. Unless otherwise directed by the Software Architect, all classes, components, code libraries, scripts, and server side code must be written in Visual Basic.Net.

2.2 *Web Platform*

ASP.Net has been designated as the primary platform for building web-based applications for PSA. All ASP.Net applications should use XHTML 1.0 Transitional as their markup language, JavaScript/VBScript for client-side code, and Visual Basic.Net on the server-side.

2.3 *Windows Application Platform*

Windows Form.Net has been designated as the primary platform for building windows-based applications for PSA.

2.4 *Database Operations*

Stored procedures have been designated as the primary mechanism for storing, retrieving, updating or deleting data contained within an agency database. Do not write In-Line or Dynamic SQL code within classes, components, code libraries or scripts unless the Software Architect has granted you permission to do so.

2.5 *Coding Style*

Source code should be written in a clear, concise and easy to follow manner. Large tasks should be broken up into smaller, more modular and reusable units of work. Do not use Spaghetti coding¹ as this is evidence of a poor programming style and will make future maintenance and debugging tasks harder.

¹ Spaghetti code is described as "...a program's source code that is difficult to read and/or follow by a human because of how the original programmer wrote the code. Spaghetti code is often not organized and has portions of code that may belong at the bottom of the code at the top of the code or vice versa. Spaghetti code may also skip to other portions of the code numerous times making it hard to track down issues within the code. Finally, some users consider programs that contain several goto statements spaghetti code because they have to try to follow each of the goto statements throughout the whole program. Spaghetti code is considered bad practice because a program may be more prone to experience errors, and if errors are experienced, it is more difficult to locate what is causing the error to occur. Spaghetti code can be reduced by keeping your program organized, always commenting your code, and if possible, breaking your code into sections." Definition taken from <http://www.computerhope.com/jargon/s/spaghatt.htm>.

2.6 Keep it Simple, Stupid

All things being equal, the simplest solution to a problem is the better one to utilize. In other words, do not overly complicate a procedure by combining several coding statements into one hard to decipher statement, or choose a solution just because it's novel, unique or demonstrates how smart you think you are. Also, do not incorporate more technology than is truly needed to solve the problem. Do not add more layers, call more routines, or otherwise complicate a task that can be solved rather simply without all the extra overhead. The goal should always be to create the simplest and most elegant solution to a problem. These types of routines always demonstrate great programming ability and are the easiest to maintain and correct. Any code that appears overly complicated will be rejected with instructions to simplify the code.

3 NAMING CONVENTION AND STANDARDS

3.1 Use meaningful, descriptive words to name types²

When naming classes, method, parameters and variables, select names that clearly reflect the value that the variable/parameter is going to contain, the operation a given method is going to perform or the object that a class represents. Do not use abbreviations unless the abbreviation is universally recognizable. Also, do not use names that begin with numeric or special characters.

3.2 Use Pascal Case³ when naming types

```
Public Class HelloWorld
    Public Sub SayHello(ByVal UserName As String)
        Dim HelloMessage As String = "Hello, " & UserName & ".
        Welcome to my world!"
        ...
    End Sub
End Class
```

3.3 Do not use using Hungarian notation

The use of special prefix characters to identify the data type of a variable is no longer considered good programming style and should not be used when developing code libraries. The Agency will continue to support its use when naming Win and Web Form controls as indicated in Sections 1.12 and 1.13 below.

```
'*** Wrong
Dim sClientName As String
Dim nClientID As Integer

'*** Right
Dim ClientName As String
Dim ClientID As Integer
```

3.4 Do not use Camel Casing⁴

While Microsoft is currently actively promoting the use of Camel Casing within .Net source code, we have elected to not utilize this programming style within our source code. The use of a single programming style and convention makes source code debugging and maintenance simpler and easier.

² In .Net, classes, methods, parameters, members and variables are all considered types.

³ Pascal Case – The first letter of all words are capitalized and all subsequent letters in the word appear in lower case, i.e. ClientFirstName.

⁴ Camel Case is a style of programming where the first word of a type's name appears solely in lower case letters and every subsequent word begins with an upper case letter, i.e. clientFirstName.

3.5 *Delineate class members from method members*

Prefix class members with *m_* to distinguish them from method members.

```
    *** Wrong  
    Dim ClientName As String  
    Dim ClientID As Integer  
  
    *** Right  
    Dim m_ClientName As String  
    Dim m_ClientID As Integer
```

3.6 *Prefix the name of an Interface type with the letter “I”*

```
Public Interface IWidget  
...  
End Interface
```

3.7 *Custom exception types should end with the word, Exception*

```
Public Class DataAccessException  
    Inherits PSAException  
...  
End Class
```

3.8 *Abstract types should end with the word, Base*

```
Public MustInherit Class PrismBase  
...  
End Class
```

3.9 *Prefix Boolean variables and properties with the words “Can”, “Is” or “Has”*

```
Dim IsTrueName As Boolean  
Dim CanUpdatePage As Boolean  
Dim HasDrugTestRecords As Boolean
```

3.10 *Do not include the parent class name within a property name*

```
    *** Wrong  
    Client.ClientFirstName()
```

```

'*** Right
Client.FirstName()

```

3.11 Common generic variable names

Unless a variable represent key a data item, use the following naming convention when you create generic variables involving the following types.

Object Type	Variable Name
Controlling index in For loops	indx, indy, indz
DB Connection	Conn
DbCommand	Cmd
DataReader	Rdr
DataSet	Ds
DataTable	Dt
SqlParameter	SqlParams

```

Public Overrides Function List(ByVal ClientImageID As Integer) As
DataTable
    Dim conn as new SqlConnection(ConnectionString)
    Dim cmd as new SqlCommand(cmd)
    Dim dt As new DataTable
    Dim rdr As new SqlDataReader
    Try
        cmd.Connection = connection
        cmd.CommandText = "gspClientPhotoList"
        cmd.CommandType = Data.CommandType.StoredProcedure
        cmd
        rdr = cmd.ExecuteReader()
        dt.Load(rdr)
    Catch ex As SqlException
        ...
    Catch ex As Exception
        ...
    End Try
    Return dt
End Function

```

3.12 Common prefixes for Win Form controls

Use the following prefix when naming objects of the indicated Win form control.

Control	Prefix
Button	cmd
Calender	cal
Checkbox	chk
CheckedListBox	clt
DataGrid	grd

Control	Prefix
ComboBox (dropdownlistbox)	cb
Hyperlink	hyp
Image	img
Label	lbl
ListBox	lst
RadioButton	rb
RadioButtonlist	rlst
Table	tbl
TextBox	txt
ToolBar	tb
DataView	gv
DetailView	dv

3.13 Common prefixes for Web Form controls

Use the following prefix when naming objects of the indicated Web form control.

Prefix	Type	Prefix	Type
adrot	AdRotator	mpag	MultiPage
cmd	Command/Button	pnl	Panel
cal	Calendar	phld	PlaceHolder
chk	CheckBox	rb	RadioButton
clst	CheckedListBox	rlst	RadioButtonList
vacmp	CompareValidator	varng	RangeValidator
ctrl	Control	vareg	RegularExpressionValidator
crv	CrystalReportViewer	rep	Repeater
dg	DataGrid	repi	RepeaterItem
dgrc	DataGridColumn	vareq	RequiredValidator
dgri	DataGridItem	tbl	Table
dls	DataList	tcel	TableCell
dlsi	DataListItem	trow	TableRow
cbo	ComboBox	tab	TabStrip
hyp	HyperLink	txt	TextBox
img	Image	tba	ToolBar
ibtn	ImageButton	tvw	TreeView
lbl	Label	vasum	ValidationSummary
lbtn	LinkButton	xml	XML
lst	ListBox	xpwin	XpWindow
lit	Literal	gv	Gridview

3.14 Do not use .NET keywords as identifiers

The table below lists the keywords that should be avoided as identifiers.

abstract	AddHandler	AddressOf	Alias	And
AndAlso	Ansi	As	Assembly	Auto
Base	bool	Boolean	break	ByRef
Byte	ByVal	Call	Case	Catch
CBool	CByte	CChar	CDate	CDec

Cdbl	Char	checked	CInt	Class
CLng	CObj	Const	continue	CSByte
CShort	CSng	CStr	CType	CUInteger
Declare	Default	Delegate	Dim	Do
Double	Each	Else	ElseIf	End
Enum	Erase	Error	Eval	Event
Exit	explicit	extends	extern	ExternalSource
False	Finalize	Finally	fixed	float
For	foreach	Friend	Function	Get
GetType	Global	Goto	Handles	If
Implements	implicit	Imports	In	Inherits
Instanceof	int	Integer	Interface	internal
Is	IsFalse	IsNot	IsTrue	Let
Lib	Like	lock	Long	Loop
Me	Mod	Module	MustInherit	MustOverride
My	MyBase	MyClass	Namespace	Narrowing
New	Next	Not	Nothing	NotInheritaable
NotOverridable	null	Object	Of	On
operator	Option	Optional	Or	OrElse
Out	Overloads	Overridable	override	Overrides
package	ParamArray	Params	Partial	Preserve
Private	Property	Protected	Public	RaiseEvent
ReadOnly	ReDim	ref	Region	Rem
Removehandler	Resume	Return	sbyte	sealed
Select	Set	Shadows	Shared	Short
Single	sizeof	stackalloc	Static	Step
Stop	String	struct	Structure	Sub
switch	SyncLock	Then	This	Throw
To	True	Try	TryCast	TypeOf
UInt	UInteger	ulong	ushort	unchecked
Unicode	unsafe	Until	using	var
virtual	void	volatile	When	While
Widening	With	WithEvents	WriteOnly	Xor
Yield				

4 COMMENTS

4.1 General Rules for Comments

All comments should be written in U.S. English using concise, but complete sentences. Comments should also use the same level of indentation as the code block it is related to and should immediately precede that code without any intervening blank lines.

```
For i As Integer = 0 To NameList.Length - 1
    'Initialize the array elements.
    NameList(i) = i
Next
```

4.2 Spaces and Indentation

- Use tabs instead of spaces to indent code.
- Use four spaces for each indenting level.
- Avoid two or more consecutive blank lines.
- Manually wrap lines that are longer than 80 characters.
- Break a long line after the comma but before an operator, if possible.
- Avoid wrapping a line in the middle of an expression, especially if in the middle of a parenthesized expression.
- Indent all wrapped lines by one tab from the start of the first line.
- Leave one space before and after binary operators such as + and *.

Note: Typing CTRL+K, CTRL+D will automatically format your current document for you.

4.3 File Header Comments

Each source file should contain a header comment block appropriate for the file type⁵ (sql, aspx, vb, etc...). The header comment block should include a copyright notice, author's name, purpose/description, creation date, \$History\$ VSS tag. Where appropriate wrap the header comments block inside of **#region** tags so it can be easily collapsed.

```
#Region "File Header"
'-----
'Name: Lockup.vb
```

⁵ Standard headers for common file types can be found in Visual Source Safe.

```
'Author: John Doe
'Created: June 15, 2006
'Description: DAL Component to encapsulate accessing the Lockup
              table in PRISM.
'
'© 1999-2006 DC Pretrial Services Agency. All rights reserved.
'-----
'$History: $
'-----
#End Region
```

4.4 Method-Level Comments

Use XML comments to document all major methods inside of .Net code libraries.

```
''' <summary>
''' Determine if the answer provided matches a given application
user's stored answer.
''' </summary>
''' <param name="ApplicationToken">Application's token.</param>
''' <param name="UserID">UserID of the user.</param>
''' <param name="Answer">User's answer to the security question
posed.</param>
''' <returns>True if the answer matches, false
otherwise</returns>
''' <remarks></remarks>
Public Shared Function ValidUserSecurityAnswer(ByVal
ApplicationToken As Guid, ByVal UserID As String, ByVal Answer As
String) As Boolean
    ...
    Dim RetValue As Boolean = False
    ...
    Return RetValue
End Function
```

4.5 Block-Level Comments

Always describe what a block of statements does, but avoid obvious remarks. Ideally, there should be a line of comments every four (4) to six (6) executable statements.

```
If RetValue = LogOutStatus.Success AndAlso WebContext() Then
    'calling from http (asp.net) context, destroy forms
    'authentication cookie.
    FormsAuthentication.SignOut()
    ...
Else
    'calling from a windows context - Do Nothing
End If
```

4.6 Statement-Level Comments

Avoid adding statement-level comments (that is, comments on the same line and to the right of executable code), unless you are explaining a variable or argument.

```
**** Wrong
newD = newD.AddTicks(myTicks) 'To get real date, we need to add
the number of ticks since 1/1/1601

**** Right
Private _Ticket As FormsAuthenticationTicket 'ASP Generated
Authentication Ticket for user.

Private dbFirstName As String 'First name of Application User
retrieved from database.
```

4.7 Code Change Comments

Comments should be added to source code to indicate changes being made to the original. The comments should indicate who is making the change, the date the change is being made, W.O. #, a brief description of the change, and a start, obsolete, and end indicator. Here the format to be followed:

```
' <Changer's Name> - <Date of Change>: W.O. #, <brief description> - START6
' <Changer's Name> - <Date of Change>: W.O. #, <brief description> - OBSOLETE7
' <Changer's Name> - <Date of Change>: W.O. #, <brief description> - END
```

```
'JDOE - 08/12/2006: W.O. 1756 - Perform ActiveDirectory check on
internal users - START
If User.IsInternal AndAlso UserInActiveDirectory(User.Name) Then
    ...
Else
    ...
End If
'JDOE - 08/12/2006: W.O. 1756 - Perform ActiveDirectory check on
internal users - END
```

⁶ The start and end tags may be omitted when the change involves three (3) or less lines of code. In these cases replace the START tag with “the following x number of lines”

⁷ OBSOLETE should only be used when commenting out four (4) or more lines of code. Lines of codes should be deleted from a source file unless you are absolutely certain that they will not need to be restored at some point in the future.

NOTE: Code Change Comments along with any obsolete (commented out) code that is more than 2 years old should be deleted from source code.

5 SOURCE CODE ORGANIZATION

5.1 Consider using the secondary drive for all your work

If your PC comes equipped with a secondary drive, consider using it to store your documents and work files (source code, word documents, Visio diagrams, etc). This ensures that all your work is saved whenever you re-image your PC or if the C drive ever crashes.

5.2 Follow the standard project file structure

To ensure a consistent approach across all of our development efforts and to keep source code and project files transportable across PC's and developers, all development work should be grouped together beneath a single root folder entitled *D:\PSAProjects*⁸. Under this single root folder you should then create one subfolder per application solution⁹ as shown here:

D:\PSAProjects	- Root container for all of PSA's development work
\AppSolution1	- Container folder for AppSolution1 project files
\AppSolution2	- Container folder for AppSolution2 project files

For example:

D:\PSAProjects	
\PRISM	- All PRISM 2.x project files would go under here
\PRISMJ	- All PRISMJ project files would go under here
\DMTS.NET	- All DTMS project files would go under here
\PRISM.NET	- All PRISM 3 project files would go under here

Files pertaining to each application solution should then be structured according to the following pattern:

- *Source* – container for various source code files that make up application solution.
 - *SQLScripts* - container for all SQL scripts related to this solution.

⁸ Please note the standard file structure organization may be altered to maintain backwards compatibility with the existing PRISM 2.x application base library.

⁹ An application solution represents all the components used to build an application, i.e. ASP.NET files, code libraries, and SQL Scripts. An Application project refers to the individual components within a solution, i.e. an ASP.NET Website, Class Library, Windows Form App, etc.

- *Tables* – container for all table related objects. Each file should relate to one table and contain all objects (keys, indices, constraints, except triggers) belonging to that table.
- *StoredProcs* – container for all stored procedures definitions
- *Functions* – container for all function definitions
- *Triggers* – container for all triggers
- *Libs* – container for class libraries
 - *ClassLibrary1* – source code for ClassLibrary1
 - *ClassLibrary2* – source code for ClassLibrary2
 - *ClassLibrary3* – source code for ClassLibrary3
- *Websites* – container for asp.net applications
 - *WebApp1* – source code for WebApp1
 - *WebApp2* – source code for WebApp2
- *WinApps* – container for windows/wpf applications
 - *WinApp1* – source code for WinApp1
 - *WinApp2* – source code for WinApp2
- *Reports* – container for SQL Reporting Server rpt files
- *Unit Tests* - container for unit tests
- *Bin* - container for third-party binaries needed to execute application.
- *Docs* - container for project documentation. (Optional if docs are being stored in SharePoint on the Project Team’s website).
- *Installer* - container for installer source code and binaries (optional)
- *Builds* - container for team build scripts (optional)
- *Tests* - container for test team test cases
 - *FunctionalTests*
 - *PerformanceTests*
 - *SecurityTest*

Additional folders may be added or removed depending upon the nature and scope of the development project.

5.3 *How To: Create a new application projects.*

The steps in this section outline how to create application projects that are optimized for team development and for use with Team Foundation Server Source Control.

5.3.1 Step 1 – Create Root Folder

If you have not done so already, open Windows Explorer and create a root folder to hold all of your development work. As noted in Section 5.2 above, this folder should be called **PSAProjects**. For this walk through we will create it on the C: drive so that root folder becomes C:\PSAProjects.

5.3.2 Step 2 – Create a Blank Solution

Next we create a blank solution that will be the container for all our source code.

1. On the **File** menu, point to **New** and then click **Project**.
2. Expand **Other Project Types** and select **Visual Studio Solutions**.
3. Select **Blank Solution**.
4. Give a name to your solution. Typically this will be the Application Project name mentioned above, i.e. PRISM, DMTS, etc. For purposes of this How To we will use *AppSolution* as the solution name.
5. Set the **Location** to C:\PSAProjects and then click **OK**.

This creates a C:\PSAProjects\AppSolution folder on your computer. Visual Studio adds the solution (.sln) file and solution user options (.suo) file to this folder.

5.3.3 Step 3 – Add a Web Site to Your Solution

In this step, you add an ASP.NET Web site to your solution. If you're not building a Web application then you may skip this section.

1. In **Solution Explorer**, right-click **AppSolution**, point to **Add**, and then click **New Web Site**.
2. In the **Add New Web Site** dialog box, leave the **Location** as **File System**, and the **Language** as **Visual Basic**.
3. Set the **Location** directory to C:\PSAProjects\<<Solution Name>>\Source\Websites\<<WebSite Name>>. For example, C:\PSAProjects\AppSolution\source\SolutionWeb.
4. Click **OK** to close the **Add New Web Site** dialog box.

The new projects are added beneath the **Source\WinApps** folder.

5.3.4 Step 4 – Add a Class Library to Your Solution

If your application requires class libraries, add them as follows:

1. In **Solution Explorer**, right-click your **AppSolution** solution, point to **Add**, and then click **New Project**.
2. Select **Visual Basic** as the project type and **Class Library** as the template.
3. Provide a name for your class library, set the **Location** to **C:\PSAProjects\AppSolution\Source\Libs**, and then click **OK**.

The new projects are added beneath the **Source\Lib** folder. Repeat these steps for each class library being added to your solution.

5.3.5 Step 5 – Add a Windows Form Project to Your Solution

In this step, you add a new Windows Forms project to your solution. If you're not building a Windows Application then you may skip this section.

1. In Solution Explorer, right-click Solution **AppSolution**, point to Add, and then click New Project....
2. In the Add New Project dialog box, select **Visual Basic** as the project type and **Windows Application** as the template.
3. Set the Location to **C:\PSAProjects\MyApp\Source\WinApps** directory.
4. Name your project.
5. Click OK to close the Add New Project dialog box and add your project.

The new projects are added beneath the **Source\WinApps** folder.

It is important that when adding new application projects to your solution that you set its folder location to the appropriate folder location indicated in Section 5.2 above.

5.4 *Code one class definition per source file*

Each source file should only contain one class per file. Do not declare multiple classes within the same source file unless the class is a subclass within the larger class or is a support class that is used solely by the main class of the file.

5.5 *Name the file after the class*

The name of the source file should match the name of class with a .vb extension added to the end.

5.6 Organize types under the DCPretrial root namespace

All types should belong to the DCPretrial root namespace using the following format:
DCPretrial.<Project Name>.TypeName.

```
Namespace DCPretrial.Prism
    Public Class ClientPhoto
        Inherits PrismBase
        Public Sub New()
            ...
        End Sub
    End Class
End Namespace
```

The following table list several common namespaces that have already been designated for use.

Namespace	Description
DCPretrial.Common	Contain common types that are common to all of the Agency's applications.
DCPretrial.Data	Contains types used to connect to, execute queries against, and return data from Agency databases.
DCPretrial.Security.Management	Contains types used to administer the Enterprise Security Management System.
DCPretrial.Security	Contains types used to performed enterprise authentication and identification of users.
DCPretrial.Utilities	Contains utility routines that are reused by several applications.
DCPretrial.Prism	Contains type used for the Adult PRISM application.
DCPretrial.Logging	Contains types used to handle application logging.

5.7 Organize members within a class

Group member definition according to their category and alphabetize them within their respective groups and use a **#region** directive to collapse the member groups easily.

Always adopt the same order when defining members:

1. Event and delegate definition
2. Private and public fields declarations
3. Constructors
4. Instance public methods

5. Static public methods and properties
6. Methods in interfaces
7. Instance public properties
8. Private (helper) methods

6 EXCEPTION HANDLING BEST PRACTICES

6.1 *Derive custom exceptions from the PSAException type*

The PSAException class has been designed to provide central exception handling and logging to a computer's Event Log. Deriving all of PSA's custom exception classes from it will simplify exception handling routines and allow for future changes to be easily propagated to all our applications.

```
Public Class DataAccessException
    Inherits PSAException
    ...
End Class
```

6.2 *Use Try blocks instead of On Error statements*

Use the Try-Catch-Finally block to handle exceptions in .Net source code instead using the old VB6 style of *On Error Goto* or *On Error Resume Next*.

6.3 *Do not use exceptions to control execution flow*

Do not use exceptions as a means of controlling the flow of source code because they are extremely time and resource-consuming. Consider having method return status code (e.g. negative numbers or a null object reference) if you want control the flow of code based upon the action of a particular method.

6.4 *Throw Exceptions only when you need to*

Throw an exception only if you want to be sure that client code doesn't miss an error condition. And in this case, always throw the most specific exception to the actual error situation that has arisen.

6.5 *Order multiple Catch statements carefully*

When catching multiple exceptions within a try-catch block, always order the exceptions being caught so that the most likely ones occur first and that specific exceptions are caught before generic exceptions. Placing generic exceptions before more specific ones will result in the more specific exceptions never being caught.

```
**** Wrong
Try
    ...
Catch ex As Exception
    ...
```

```

Catch ex As SQLException '*** exception will never be caught.
...
End Try

'*** Correct
Try
...
Catch ex As SqlException
...
Catch ex As Exception
...
End Try

```

6.6 *Re-throw exceptions caught in class libraries*

In general, do not catch exceptions inside of class libraries unless you are planning on re-throwing them or throwing a library specific exception that will let client code know that an application exception has occurred.

6.7 *Write User Friendly Error Message*

Use complete, user friendly sentences in exception messages, include a trailing period, and provide enough information for a client to solve the problem.

```

'*** Wrong
If width <=0 Then
    Throw New ArgumentException("An application error has
occurred.", "Width")
End If

'*** Correct
If width <=0 Then
    Throw New ArgumentException("Width can't be zero. Please
enter a value greater than zero.", "Width")
End If

```

6.8 *Protect again revealing sensitive data*

Be careful to not expose information that should be regarded as private or sensitive in exception messages. Do not reveal internal system or applications details, such as stack traces, SQL statement fragments, user credentials, files names, or database information to end users. This information can be used by hackers to compromise the security of the system.

6.9 *Keep unhandled exceptions from being displayed to end-users*

Include a global exception handler in your Windows Form/WPF or ASP.NET applications that will catch any exceptions not being handled by else where in code. You

should log these events to the window event log and display a generic user friendly error message to end-users.

Window Applications

The following steps will add a global exception handler to a Windows application.

1. Add a new class to your Windows project.
2. Give the class a meaningful name such as “Main”.
3. Replace the code within in the class with the following code snippet

```
Imports System.Threading
Imports System.Windows.Forms
Imports Microsoft.SqlServer.MessageBox
Imports DCPretrial.Logging

Friend Class Main

    Public Sub New()
    End Sub

    <STAThread(> _
    Shared Sub Main()
        AddHandler Application.ThreadException, AddressOf
GlobalExceptionHandler
        Dim Debugger As New DebugLogger()
        Dim EvtLogger As New EventLogLogger()
        LogManager.Logger.AddLogger(Debugger)
        LogManager.Logger.AddLogger(EvtLogger)
        Application.Run(New frmLogon)
    End Sub

    Private Shared Sub GlobalExceptionHandler(ByVal Sender As
Object, ByVal e As ThreadExceptionEventArgs)
        Dim MsgBox As ExceptionMessageBox = New
ExceptionMessageBox(e.Exception.Message)
        MsgBox.Show(frmLogon)
    End Sub
End Class
```

4. Replace the code within the Sub Main() main to add as many application loggers as are required by your application and to also add any application start-up code that will be needed.
5. Replace the form being instantiated in the Application.Run line with the name of the main form within your application.
6. Modify the code within the GlobalExceptionHandler to process the exception being caught.

Web Applications

Add the following script block the global.asax file inside of your web application to create a global exception handler.

```
<%@ Import Namespace="System.Diagnostics" %>

<script language="VB" runat="server">
Sub Application_Error(sender As Object, e As EventArgs)
    'get reference to the source of the exception chain
    Dim ex As Exception = Server.GetLastError().GetBaseException()

    'log the details of the exception and page state to the
    'Windows 2000 Event Log
    EventLog.WriteEntry("Test Web", _
        "MESSAGE: " & ex.Message & _
        "\nSOURCE: " & ex.Source & _
        "\nFORM: " & Request.Form.ToString() & _
        "\nQUERYSTRING: " & Request.QueryString.ToString() & _
        "\nTARGETSITE: " & ex.TargetSite & _
        "\nSTACKTRACE: " & ex.StackTrace, _
        EventLogEntryType.Error)
End Sub
</script>
```

7 SECURITY BEST PRACTICES

7.1 Craft with security in mind

Always think about security when writing source code. Don't assume that only authorized users will call your routine or use your application. Always think how can I prevent someone from using my code in ways that I did not intend for it to be used.

7.2 Never trust what you've been given

Assume all input is malicious and its source is not trustworthy. Constrain input by validating it for type, length, format and range and reject everything that does not fit into that constraint. This should be done at each layer of the application.

7.3 Be careful how and where you store connection strings

Do not store plain text versions of connection strings in any of your source code files, application configuration files or registry settings. Always encrypt connection strings and restrict access to them using ACL or some other method of limiting access to authorized personnel only.

7.4 Use your own credentials when connecting to databases

Do not use application user credentials when connecting to a database, production or otherwise, unless you are verifying the accessibility and permissions of those accounts.

7.5 Use stored procedures to execute queries

Use stored procedure rather than in-line SQL statements. Besides the performance benefits of stored procedure over in-line SQL statements, the use of stored procedures also ensures that input values are checked for type and length. A value outside of the acceptable range automatically triggers an exception. Parameters are also treated as safe literal values and not executable code within the database. This protects our system from SQL Injection attacks.

7.6 Protect SQL Code from injection attacks

If the Software Architect has granted permission to use In-line SQL within your application, you must filter all string data flowing into and out of your application using the `DCPretrial.Utilities.PSAUtility.RenderSQLStringSafe()` method. This method will strip out any questionable query syntax from your string variables.

7.7 Use *HtmlEncode* to protect against Cross-site Scripting

Use the *HttpUtility.HtmlEncode* method to encode output if it contains input from the user, such as input from fields, query strings, and cookies or from other sources, such as database. Never just echo input back to the user without validating and/or encoding the data.

```
Response.Write(HttpUtility.HtmlEncode("Request.Form["txtLastName"
])
```

Note: By default, the *DataAccess.GetString()* method *Html* encodes all string values retrieved from a database field for you.

8 UNIT TESTING BEST PRACTICES

<< This section will be completed in version 1.5 >>

9 TEAM FOUNDATION SERVER SOURCE CONTROL BEST PRACTICES & GUIDANCE

9.1 *Getting the latest version of source code*

From **Source Control Explorer**, select the file, right-click and click **Get Latest Version**. This downloads a read-only copy of the latest version of the file into the workspace on your computer.

Note: The **Get Latest Version** operation does not check out the file and the check out for edit operation does not do a get. You need to perform both steps individually. This behavior is different from VSS behavior.

9.2 *Files kept and not kept in source control*

Files kept in source control:

The following list identifies the key file types that you should add to source control. These are also the file types that are added when you click **Add Solution to Source Control**.

- **Solution files (*.sln).** Solution files maintain a list of constituent projects, dependencies information, build configuration details, and source control provider details.
- **Project files (*.csproj or *.vbproj).** Project files include assembly build settings, referenced assemblies (by name and path), and a file inventory.
- **Visual Studio Source Control Project Metadata (*.vspssc).** These files maintain project bindings, exclusion lists, source control provider names and other source control metadata.
- **Application configuration files (*.config).** XML configuration files contain project and application specific details used to control your application's run time behavior. Web applications use files called Web.config. Non-Web applications use files called App.config.

Note: At run time, the Visual Studio build system copies App.config to your project's Bin folder and renames it as <YourAppName>.exe.config. For non-web applications, a configuration file is not automatically added to a new project. If you require one, add it manually. Make sure you call it App.config and locate it within the project folder.

- **Source files (*.aspx, *.asmx, *.cs, *.vb, ...).** Source code files, depending on application type and language.

- **Binary dependencies (*.dll).** If your project relies on binary dependencies such as third party DLLs, you should also add these to your project within source control.

Files that should not be added to source control:

The following files are specific to each developer and should therefore not be added to version control:

- **Solution user option files (*.suo).** These contain personalized customizations made to the Visual Studio IDE by an individual developer.
- **Project user option files (*.csproj.user or *.vbproj.user).** These files contain developer specific project options and an optional reference path that is used by the Visual Studio to locate referenced assemblies.
- **WebInfo files (*.csproj.webinfo or *.vbproj.webinfo).** This file keeps track of a project's virtual root location. This is not added to source control to allow individual developers to specify different virtual roots for their own working copy of the project. While this capability exists, you and all team members are recommended to use a consistent (local) virtual root location when you develop Web applications.
- **Build outputs.** These include assembly dynamic-link libraries (DLLs), interop assembly DLLs and executable files (EXEs). (Note though that assemblies such as third-party binaries that are not built as part of the build process should be placed under version control as described above).

9.3 Checking out code for editing

- From **Source Control Explorer**, select the file, right-click and click **Get Latest Version**. This downloads a read-only copy of the latest version of the file into the workspace on your computer.
- Right-click the file and click **Check Out for Edit**.
- Choose your required lock type. Select **None** to allow other users to check the file in and out during the period of time you are working on the file. This type of shared check out is generally recommended because most conflicts should they arise can be resolved automatically.

Note: That the get latest version operation does not check out the file and the check out for edit operation does not do a get. You need to perform both steps individually. This behavior is different from VSS behavior.

When selecting your lock type, consider the following:

- A shared checkout (**None**) avoids introducing potential bottlenecks into your development process by preventing someone else working in the same file.
- You should only lock a file while editing it if you are concerned that there will be a conflict resulting in a complicated manual merge operation.
- Select the **Check Out** lock type to prevent other users from checking out and checking in the file. This option prevents other people from editing the file which could represent a potential bottleneck to your development process. This option ensures that you can apply your changes back to the source control database without the possibility of other changes having been made to the file by other people.
- Select the **Check In** lock type to allow other users to check out the file but prevent them from checking it in. Again, this option ensures that you will be able to check-in your edits without conflicts.

9.4 *Checking in modified code*

- In **Source Control Explorer**, navigate in the **Folders** list to the folder associated with the items that you want to check in.
- In the lists of items to the right of the **Folders** section, right-click the items that you wish to check in, and choose **Check In Pending Changes**. The **Check In - Source Files** dialog box appears.
- In the **Source Files** channel, select the items that you wish to check in, and type any applicable comments in the **Comment** text box. This should include work order number if the change is related to work order number
- If these items are associated with a Team Foundation work item, click the **Work Items** channel, and select the items that you are checking in.

9.5 *Deleting code from Source Control*

Do not delete code out of Source Control without first checking with Team Foundation Source Control Server administrator.

9.6 *Sharing shelved code*

To shelve a source code for sharing code with a team member, perform a **Get Latest** operation to synchronize your workspace with the latest server version and then build your application to ensure that it compiles. Shelf the source using the **Source Control Explorer**. The team member who has been handed off the code then needs to unshelve the code.

Shelving is useful when you have work in progress that is to be completed by another team member. You can then shelve your changes to make a handoff easier. By synchronizing the latest code, you get an opportunity to incorporate changes to source files that have been made outside of your workspace.

9.6.1 Shelving working code

- In Source Control Explorer, right-click, and choose **Shelve Pending Changes**.
- In the **Shelve - Source Files** dialog box, type the shelve set name, for example **shelvetest** in the **Shelve set name** box.
- In the **Comment** box, type **Testing my shelve set**, and then click **Shelve**.
- The files and folders are copied to the source control server and are available for other team members to unshelve.

When the other team member unshelve a shelve set, Team Foundation restores each shelved revision into the destination workspace as a pending change as long as the revision does not conflict with a change that is already pending in the workspace.

9.6.2 Un-shelving working code

- In Visual Studio 2005 Team System, click **File**, point to **Source Control**, and then click **Unshelve**.
- In the **Owner name** box, type the shelve set creator's name (for example, ADVENTUREWORKS\JuanGo or simply juango) and then click **Find**.
- In the **Results** pane, select the shelve set you want to unshelve into your workspace, and then click **Details**.
- If you want to delete the shelve set from the Team Foundation source control server, deselect the **Preserve shelve set on server** option.

- Optionally deselect the **Restore work items and check-in notes** option if you do not want to have the work items and check-in notes associated with the shelve set restored.
- When the **Details** dialog box appears, select the shelve set or shelve set items you want to unshelve into your workspace, and then click **Unshelve**.

9.6.3 Deleting shelved code

If you want to delete the shelve set from the Team Foundation source control server, deselect the **Preserve shelve set on server** option.

10 GENERAL CODING BEST PRACTICES

10.1 Turn on *Option Strict*, *Option Explicit* and *Option Compare* for all projects

Setting these compiler options help to ensure that source code functions as expected and difficult-to-debug run-time errors are caught while you are still developing code.

10.2 Don't declare multiple variables on the same line

Declare one variable per line of code instead of combining multiple declarations on a single line of code.

```
*** Wrong
Dim x As Integer, Name As String, Salary As Decimal

*** Correct
Dim x As Integer
Dim Name As String
Dim Salary As Currency
```

10.3 Never define public instance fields

Never define public instance fields. Instead, use a private field and wrap it inside a public property so that you can validate the incoming value before it's assigned to the private field. Unless the enclosing type is sealed, consider whether the property should be marked as virtual.

```
***Wrong
Public Name as String

***Correct
Private m_Name as String

Public Property Name() As String
    Get
        Return m_Name
    End Get
    Set (ByVal Value as String)
        If String.IsNullOrEmpty(Value) Then
            Throw New ArgumentException("Invalid
Name")
        End If
        m_Name = Value
    End Set
End Property
```

10.4 Avoid having multiple statements on one line

Don't include multiple statements on a single line because lines containing single statements make the code more readable and enable you to add a comment for each individual variable.

```
***Wrong
x = 1: y = 1

***Correct
x = 1      '(comment here)
y = 1      '(comment here)
```

10.5 Don't code single-statement If blocks

Avoid single-line *If* statements. Always close an *If* statement with an *End If* keyword. By doing so, it's easier to add remarks and other statements in the *If* block.

```
***OK
If x > 0 Then y = 0

***Better
If x > 0 Then
    y = 0
End If
```

10.6 Use the *AndAlso* and *OrElse* instead of the *And* and *Or* operators

Use *AndAlso* and *OrElse* operators when combining Boolean conditions because these operators perform better than *And* and *Or* operators, which should be reserved for bit-field manipulation.

```
***Wrong
If x > 0 And y < 10 Then Console.WriteLine("ok")
***Correct
If x > 0 AndAlso <10 Then Console.WriteLine("ok")

'Use Or instead of OrElse because Increment function modifies its
'argument.
If x > 0 Or Increment(y) > 10 Then Consold.WriteLine ("ok")
...
Function Increment(ByRef n As Integer) As Integer
    n += 1
    Return n
End Function
```

10.7 Don't compare Boolean types to true/false values

Avoid comparisons with true and false Boolean values.

```
***Wrong
Sub PerformTask(ByVal condition1 As Boolean, ByVal condition2 As
Boolean)
    If condition1 = True then
        ...
    ElseIf condition2 = False Then
        ...
    End If

***Correct
Sub PerformTask(ByVal condition1 As Boolean, ByVal condition2 As
Boolean)
    If condition1 Then
        ...
    ElseIf Not condition2 Then
        ...
    End If
End Sub
```

10.8 Use direct assignments to set Boolean types

Use a direct assignment of the result of a Boolean expression instead of an *If...Then...Else* block.

```
***Wrong: too verbose
Dim ok As Boolean
If x > 0 Then
    ok = True
Else
    ok = False
End if

***OK
Dim ok as Boolean = (x > 0)
```

10.9 Avoid the IIF function

Avoid using the IIF function in time-critical code or if one of its operands is a function call. Instead use an explicit *If...Then...Else* block.

```
***Wrong
Dim s As String = IIf(Name <> "", Name, getUsername())

***OK
Dim s As String = IIf(Name <> "", Name, "Unknown")
```

```

'(Next code assumes that names is an array of strings.)
'***Wrong:  uses IIf inside a loop
For I As Integer = 0 To names.length - 1
    Console.WriteLine(IIf(names(i) <> "", names(i), "Unknown"))
Next

'Correct
For I As Integer = 0 To names.Length - 1
    If names(i) <> "" Then
        Console.WriteLine(names(i))
    Else
        Console.WriteLine("Unknown")
    End If
Next

```

10.10 Testing and comparing object types

Call the `DCPretrial.Utilities.PSAUtilities.AreSameType()` when you want to determine if two types are actually, really of the same type. The `TypeOf` function will return true if object1 is of the same type as object2 or any object that descends from that object2. The `AreSameType()` will return true when the two objects are exactly of the same type.

```

If PSAUtilities.AreSameType(Client1, Client2) Then
    ...
End Sub

```

10.11 Favor Select Case blocks to test for multiple values

Favor a *Select Case* block instead of an *If* statement when testing the same variable against four or more values or ranges of values.

```

'***Wrong
If x = 1 OrElse x = 3 OrElse (x >=6 AndAlso x <= 9) Then
    Console.WriteLine("OK")
End If

'***Correct
Select Case x
    Case 1, 3, 6 To 9
        Console.WriteLine("OK")
End Select

```

10.12 Never code multiple statements on a Case block line

Never include multiple statements on the same line as the *Case* keyword. It is acceptable to have a *Case* clause followed by a single statement on the same line, but only if all the *Case* clauses in the *Select Case* block are followed by a single statement.

```

***Wrong
Select Case x
    Case 1: y = 2
    Case 2: y = 5: z = 8
End Select

***Correct
Select Case x
    Case 1
        y = 2
    Case 2
        y = 4
        z = 8
End Select

***Also correct: all Case blocks contain one statement.
Select Case x
    Case 1: y = 2
    Case 2: y = 5
    Case 3: y = 8
End Select

```

10.13 Put most likely to occur condition at top of Select Case statement

Always check the most frequent values near the top of a *Select Case* statement because this arrangement improves execution speed as values in the block are usually tested in the order they appear.

10.14 Declare controlling variable in For and For each loops

Always declare the controlling variable of a *For* and *For Each* loop inside the loop. This syntax is more concise and, above all, ensures that the variable isn't accidentally used after the last iteration of the loop.

```

***Wrong
Dim I As Integer
For i = 1 To 100
    ...
Next
Dim p As Person
For each p In colPersons
    ...
Next

***Correct
For I As Integer = 1 to 100
    ...
Next
For Each p As Person in colPersons
    ...

```

Next

10.15 Don't use an array's length property to control For loops

Always explicitly test the controlling variable in *For* loops against the array's length property. Don't attempt to optimize code by storing the Length property's value in temporary variable

```
***Wrong: uses a function in a Microsoft.VisualBasic library.
Dim arr(99) As Integer
For i As Integer = 0 To Ubound(arr)
    ...
Next
***Wrong: caches upper limit in a variable
Dim arr(99) As Integer
Dim maxIndex As Integer = arr.Length - 1
For I As Integer = 0 To maxIndex
    ...
Next
***Correct
Dim arr(99) As Integer
For I As Integer = 0 To arr.Length - 1
    ...
Next
```

10.16 When to use For ... Next and For Each ...Next loops

For ... Next loops are usually faster when working with regular arrays. For Each ... Next are usually faster when with collections.

10.17 Doing For loops when the controlling variable are not sequential

By coupling For Each loops with the ability to create arrays on the fly, you can execute a block of statements with values for a controlling variables that aren't necessarily in sequence.

```
Dim isPrime as Boolean = True

For each var as Integer in New Integer() {2, 3, 5, 7, 11, 17, 19}
    If (number Mod var) = 0 isPrime = false: exit For
Next
```

10.18 Use the & operator to concatenate strings

Use the & operator to concatenate strings instead of the + operator. The + operator is only supported for historical reasons. This guidance makes for more readable and less ambiguous code.

10.19 When to use char variables

Use Char variables rather than String variables if you are sure you need to store only individual characters. Char variables are more efficient because they take less memory than a 1-char String object and because they are value types rather than reference types.

10.20 Initialize string types to String.Empty

Explicitly initialize string fields and variables to a String.Empty. Explicit assignment avoids NullreferenceException errors when referencing the string and simplifies code because the string doesn't have to be tested against null.

```
***Wrong
Dim x As String
...
If Not X Is Nothing Then
    'Must check for Nothing before processing the string
End If

***Correct
Dim x As String = ""
*** BEST !!!
Dim x As String = String.Empty
...
'Process the string (no need to check it first).
```

10.21 How to split long string expressions

When splitting a long string expression over multiple lines, have each line after the first on begin with the concatenation operator. This coding style makes it more evident that the statement is split over more than one line even if the code is too long to fit in the editor window.

```
Dim s As String = "A long string expression " _
    & "that is split over two lines"
```

10.22 Looping over all the characters of a string

Use the Chars property in a For loop instead of a more elegant, but slower For Each loop.

```

Dim companyName As String = "Code Architects Srl"

'***OK
For Each c As Char In companyName
    Console.Write( c)
Next

'***Better: nearly twice as fast
For index As Integer = 0 To companyName.Length -1
    Console.Write(companyname.Chars(index))
Next

```

10.23 Avoid passing types by reference

Avoid defining methods that take arguments passed by reference – that is, by means of the *ByRef* keyword – if the argument is a reference type. The only exception to this rule is when the method must be allowed to set the parameter to a null object reference or to make it point to a different object.

To help you better understand when to pass an argument by reference, it is essential that you understand the difference between passing an argument by value or by references as well as the implications of passing reference types rather than value types. There are four cases in total:

- a. **Passing value types by value** – A copy of the value type is created and passed to the method. This argument-passing style is commonly used and exhibits no side effects because the called method can't change any member of the original value type.
- b. **Passing reference types by value** – A copy of the pointer to the object is created and passed to the method. In most cases, this is the correct way to pass a reference type: the method can use the pointer to access and change the fields and properties of the object. If the method changes the pointer (for example, it assigns it a null value or makes it point to a different instance), the original object variable isn't affected.
- c. **Passing value types by reference** – A pointer to the original value is passed to the method. When a value type argument is passed by reference, the code in the method can modify the properties in the value type. This case is less common than case a, and it's mostly used in functions when you want to return additional information to the caller or in Pinvoke/COM Interop scenarios.
- d. **The address of the pointer to the object is passed to the method** – This is the least common scenario: the called method can change the object's fields and properties, as in case b, plus all the changes to the pointer are reflected in the original object variable. For example, if the method sets the argument to a null object reference, the original object variable is set to a null reference as well.

```

    ***Wrong:  the method doesn't need to modify the pointer.
Sub ClearProperties(ByRef p As Person)
    p.Firstname = Nothing
    p.LastName = Nothing
End Sub

    ***Correct:  the object is passed by value.
Sub ClearProperties(ByVal p As Person)
    p.Firstname = Nothing
    p.LastName = Nothing
End Sub

    ***Correct:  the method assigns a new object reference
Sub ClearProperties(ByRef p As Person)
    p = New Person
End Sub

```

10.24 Use method overloading to reduce boxing

Provide overloaded versions of the same method if the method can take arguments of different types rather than using a single method that takes Object arguments. This guideline helps to avoid the boxing operation that would occur if a value-type argument were passed to an Object parameter.

It is essential that the overloaded methods that serve value types don't delegate to the more generic method that takes an object; otherwise, you would have a boxing operation anyway.

```

Public Sub WriteLine(ByVal arg as Long)
    'This version serves Short, Integer and Long Arguments.
End Sub
Public Sub WriteLine(ByVal arg as Double)
    'This version serves Single and Double arguments.
End Sub
Public Sub WriteLine(ByVal arg as Decimal)
    'This version serves Decimal arguments.
End Sub
Public Sub WriteLine(ByVal arg as DateTime)
    'This version serves DateTime arguments.
End Sub
Public Sub WriteLine(ByVal arg as String)
    'This version serves String arguments.
End Sub
Public Sub WriteLine(ByVal arg as Object)
    'This version serves arguments of any other type.
End Sub

```

10.25 Use method overloading to reduce the number of arguments

Provide overloaded versions of the same method if the method can take a variable number of arguments to reduce the number of arguments that clients have to pass in the most common scenarios.

```
Public Sub FormatText(ByVal text As String)
    FormatText(text, 0, Nothing)
End Sub

Public Sub FormatText(ByVal text As String, ByVal indent As Integer)
    FormatText(txt, indent, Nothing)
End Sub

Public Sub FormatText(ByVal text As String, ByVal indent As Integer, ByVal formatter As Object)
    'Process the format text request.
    ...
End Sub
```

10.26 Use method overloading rather than relying on optional parameters

Use method overloading to support methods with different numbers of arguments rather than relying on optional parameters for public methods in public types. If the method really requires a variable number of arguments and using overloading isn't practical, use a *ParamArray* parameter.

```
'**Wrong: a public method with an optional argument.
Public Class SampleType
    Public Sub PerformTask(ByVal name as String, _
        Optional ByVal pwd as String = Nothing)
        ...
    End Sub
End Class

'**Correct: method overloading is used instead.
Public Class SampleType
    Public Sub Performtask(ByVal name as String)
        'Invoke the more complete overload
        PerformTask(name,Nothing)
    End Sub
    Public Sub PerformTask(ByVal name as String, ByVal pwd as String)
        ...
    End Sub
End Class
```

10.27 Validate all arguments before using them

Validate all arguments being passed in before utilizing them in a method. This should occur at the top of a method before any local variables are declared and instantiated. Throw an `ArgumentNullException` or `ArgumentException` for each argument that is null or not within an acceptable range. Use the `Message` property to explain why the argument has been rejected.

```
Sub PerformTask (ByVal text as String, ByVal width as Integer)
    'Validate all arguments before proceeding
    If text Is Nothing Then
        Throw New ArgumentNullException("text")
    ElseIf width <=0 Then
        Throw New ArgumentException("Width can't be Zero.
Please supply a value greater than Zero.", "Width")
    End If
    ...
End Sub
```

10.28 Have a single exit point for all methods

Methods should have a single exit point. `Sub` procedures (VB) should have no `Exit Sub` statement. Methods that return values should contain only one `Return` statement. The reason for this practice is because having a single exit point is considered to be good design and enables you to add tracing statements or assertions easily. However, if special cases can be dealt with at the top of the method, it is OK to exit earlier by means of an additional `Exit Sub` or `Return` (VB) statement.

```
Public Function LoadClientNames(ByVal ClientID As Integer) As
Datatable
    If IsNothing(ClientID) OrElse ClientID <= 0 Then
        Throw New ArgumentException("ClientID must be greater than
Zero", "ClientID")
    End If

    Dim RetValue As DataTable = Nothing
    Dim Clients As New Prism.Client
    Try
        RetValue = Clients.GetNames(ClientID)
    Catch ex As PSAException
        ...
    Catch ex As Exception
        ..
    End Try

    Return RetValue
End Function
```

10.29 Use the *Return* keyword to return values from a method or property

Return a value to the caller by means of the *Return* keyword instead of assigning the return value to the implicit local variable named after the current property or method, because the return keyword often enables the computer to optimize your code more efficiently.

```
***Wrong
Function GetUserName() As String
    GetUsername = m_UserName
End Function

***Correct
Function GetUserName() As String
    Return m_UserName
End Function
```

In addition, you can also use the *Return* keyword (without any argument) to replace an *Exit Sub* statement.

```
Sub PerformTask()
    ***OK
    If x = 0 Then Exit Sub

    ***Better
    If x = 0 Then Return
End Sub
```

10.30 Methods and properties that return strings

When defining a method or property that returns a string, return an empty string rather than a null object reference when the result string has no characters. This simplifies the task of the caller code, which can be used to return the string without having to test it first for a *Nothing*.

```
Function Spaces(ByVal n as Integer) As String
    If n <=0 Then
        Return String.Empty 'Empty string rather than nothing
    Else
        Return new String ("c, n)
    End If
End Function
```

10.31 Use the Get-prefixed to name methods that return values

Use the Get prefix for methods whose primary function is to evaluate and return a value; the portion of the name following Get should describe what the method returns.

```
GetClientName()  
GetClientDrugTestResults()  
GetUserSecurityProfile()
```

10.32 Get-prefixed methods matching property names

Don't have a type expose a method named *GetPropertyName*, where *PropertyName* is the name of a property exposed by the same name. The reason for this is because exposing a property and a Get-prefixed method that apparently returns the same information can be very confusing to users and should be avoided.

```
*** Wrong  
Public Function GetDrugTestStatus() As Integer  
    ...  
End Function  
  
Private _DrugTestStatus As Integer  
Public Property DrugTestStaus() As Integer  
    Get  
        ...  
    End Get  
    Set(ByVal value As Integer)  
        ...  
    End Set  
End Property  
  
*** Right  
Private _DrugTestStatus As Integer  
Public Property DrugTestStaus() As Integer  
    Get  
        ...  
    End Get  
    Set(ByVal value As Integer)  
        ...  
    End Set  
End Property
```

10.33 Set-prefixed methods

Consider the opportunity to define one or more Set-prefixed methods that take arguments and that assign multiple properties in one shot.

```
Public Sub SetCustomerData(ByVal id As Integer, ByVal name As  
String, _
```

```
        ByVal address as String, ByVal city As String, ByVal  
country As String)  
        ...  
End Sub
```

10.34 Consider using parentheses to return the results of an expression

When returning the results of an expression from a method, optionally enclose the expression in parentheses if it helps readability. Don't use parentheses when returning a single value.

```
***OK  
Return (123)  
Return x = 0  
  
***Better  
Return 123  
Return (x = 0)
```

10.35 Don't use properties as methods or methods as properties.

Properties set or return values, methods perform actions. Don't use a property as a method and don't use a method as property. In other words, don't use a property to perform an action or task other than setting or retrieving values. In other all other cases use a method (either function or sub).

```
*** Wrong  
Public Property BlankOutValues()  
    ...  
End Sub  
  
Public Sub SetClientName(ByVal NewName as String(  
    ...  
End Sub  
  
***Right  
Public Sub Clear()  
    ...  
End Sub  
  
Public Property Name() As String  
    Get  
        ...  
    End Get  
    Set(ByVal value As String)  
        ...  
    End Set  
End Property
```

10.36 Use Enums to return result status codes

Use Enums to pass the result status to callers rather a result property.

```
***Wrong: Don't use Property to return result statuses
Customer.Update()
If Customer.ResultCode = -1 then
    ...
Elseif Customer.ResultCode = 1 then
    ...
End if

***Right: Use an Enum to return the status code
Dim Result as Customer.UpdateStatus
Result = Customer.Update()
If Result = NoRecordsUpdate then
    ...
Elseif result = TooManyRecords then
    ...
Elseif result = UpdateSuccessful then
    ...
End if
```

10.37 Return Zero-element Arrays over Un-initialized arrays

Visual Basic.Net has two types of empty arrays: un-initialized arrays and arrays that contain zero elements. In the first case an array variable is set to Nothing and in the second, the array variable is initialized to an array that has no elements. The main advantage of returning zero-element arrays over un-initialized arrays is that with zero-element arrays callers do not test the array variable for being nothing before referencing it. The code example below shows how to create a zero-element array.

```
*** declaring an array of no elements.
Dim MyArray(-1) as Integer
```

11 ASP.NET BEST PRACTICES

11.1 Use Model-View-Presenter Pattern

To promote the use of the Model-View-Presenter pattern, only presentation markup and JavaScript/VBScript should be placed within a form's .aspx page. All server side code should be placed in a code-behind file. Neither the form's .aspx page nor its code-behind file should make calls in the data access layer or directly into the database.

11.2 Validate input on both the client and server

Do not rely on client-side validation as your only input validation mechanism because it can be easily bypassed by hackers into your system. You should validate all input both client-side and server-side. Use client-side validation only to reduce round trips and to improve the user experience.

11.3 Set the ViewStateUserKey property on all web pages

Place the following line of code inside of the Page.Init method of all web pages you develop. Setting this property increases the security of the ViewState mechanism and prevents one-click attacks from malicious users¹⁰.

```
Me.ViewStateUserKey = Session.SessionID
```

11.4 Redirect using Server.Transfer or Server.Execute

Use the Server.Transfer or Server.Execute methods to redirect execution to another page in the same ASP.Net Application. Use the Response.Redirect method to redirect execution to a page belonging to another web application. The difference between the two is that the Server performs the redirection entirely on the server without having to go back to the client for a roundtrip and the Response does not.

11.5 Favor the ViewState dictionary over hidden fields

Use the ViewState dictionary to persist variable values between postbacks to the same page. The ViewState dictionary is a hashed and encrypted hidden field that works with all browsers if the end user has disabled cookie support. Only use hidden fields when posting data between different pages.

¹⁰ In this kind of attack, hackers manually build an HTML page or use the HTML page received by another user to submit invalid data to the server.

12 DATA ACCESS BEST PRACTICES

12.1 Access the database through the Agency's Data Access Layer

Use the Agency's standard data access layer (DCPretrial.DataAccess) to perform all of your database access operations. The data access layer contains the code needed to execute stored procedure against any of the Agency's databases and to return data in object format support by ADO.NET.

12.2 Don't use Datasets to pass data between application tiers.

Because Datasets are heavy components to marshal you should not use them to pass data between application tiers. Use either DataTables (type or un-typed), ArrayLists, XML or serialized data classes (aka Data Transfer Objects) since they are generally lighter to marshal than Datasets.

12.3 Data Reader vs. DataSet

Read database data into a DataSet in the following cases

- a. You must update the database and you want to implement an optimistic update strategy
- b. You must account for relations existing in different tables.
- c. You are binding data to one or more Windows Forms controls.
- d. You are binding data to two or more Web forms controls.
- e. You need to store data between consecutive postbacks in an ASP.NET application
- f. You need to pass data between layers in a multitiered application
- g. You want to cache data, for example, to reduce traffic on a slow network and improve database performance and scalability.

In all other cases, use a DataReader for processing data coming from a database. In general, processing data by means of a DataReader is faster than using a Data-Adapter to fill a DataTable in a DataSet.

13 SQL REPORTING BEST PRACTICES

13.1 Report Standard Fonts

The table below outlines standard fonts used in specific sections of PSA reports.

Section	Sub Section	Font Type	Font Size	Sample
Header	Agency Title	Times New Roman; Bold;All CAPS	14	SAMPLE
Header	Division Title	Times New Roman	12	Sample
Header	Agency Information	Times New Roman	10	Sample
Header	Report Title	Times New Roman;Bold	14	Sample
Body	Column Header	Times New Roman;Bold	10	Sample
Body	Body Label	Times New Roman;Bold	10	Sample
Body	Body Text (Form Letter)	Times New Roman	11 or 12*	Sample
Body	Body Text (Client and Summary Reports)	Times New Roman	10	Sample
Body	Group Heading	Times New Roman;Bold	11	Sample
Footer	Footnote/Page Number	Times New Roman	8	Sample

*Font size for Form Letters will depend on space available. When possible, 12 point font should be used unless that will cause the report to expand over 1 page.

13.2 Report Margins

Standard margins for all reports will be: 0.5 inch top; 0.5 inch left; 0.5 inch right; 0.5 inch bottom. To set this property in SQL Reporting Services, go to Menu > Report > Report Properties > Layout Tab.

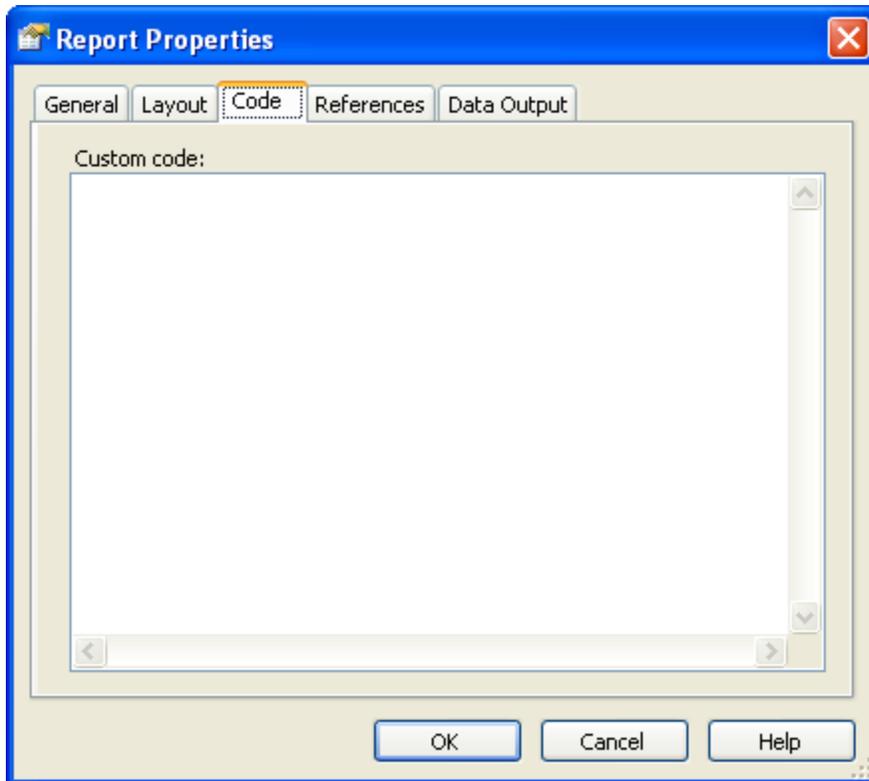
13.3 Naming conventions

Report names should reflect the title of the report as close as possible. For example, the file for “Address Verification” should be AddressVerification.rdl, “Contact Report” should be Contact.rdl, etc.

When creating database stored procedures for use only in reports, use the prefix “Rpt” to name the stored procedure. For example, a stored procedure used to generate Appointment Slips can be named sp**Rpt**GetAppointmentSlip.

13.4 Using Reporting Services Embedded Code

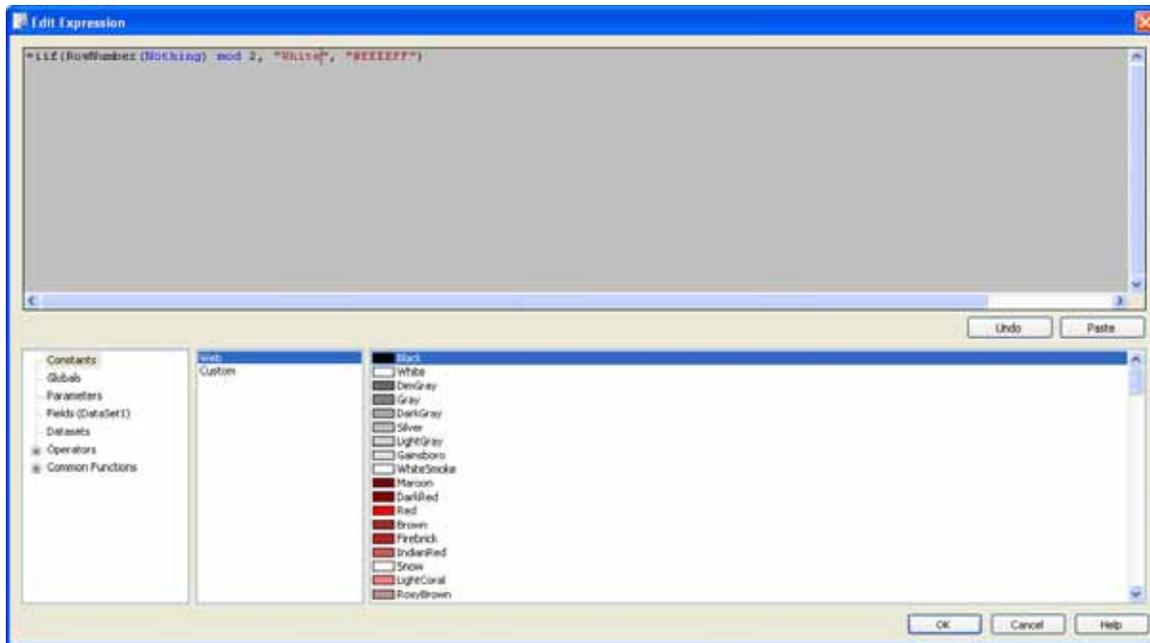
It is possible to create custom functions in the Report Properties>Code Window shown below.



Code displayed in this window can be used in a variety of ways to display data and calculated fields. When writing code in this area of Reporting Services, follow coding standards discussed earlier in this document.

13.5 Report Readability (for Summary Reports)

In an effort to make Summary Reports more readable, it is useful to implement background shading for alternate rows. This can be accomplished by clicking on the row in the Data Table that contains the details of the report, and setting its BackgroundColor property as follows using an expression:



13.6 Naming Data Sources and Data Sets

When creating a Data Source to be used in a PRISM report, the Data Source name will be “PRISMDataSource”. If creating more than one Data Set for a report, each Data Set should have a meaningful name to reflect the data being retrieved. For example, when the stored procedure spRptUserInfoHeader is used to bring back report header info, name the Data Set “HeaderInfo”.

13.7 Deploying Reports

Three important properties should be properly set when deploying reports:

- 1) TargetServerURL = <http://psa-sqlrpting/ReportServer>
- 2) TargetReportFolder = PRISMReports/PRISM
 - a. The failure to set this report property correctly will create more effort to verify reports are being referenced properly in the code where they are called.
- 3) TargetDataSourceFolder = DataSources/PRISM
 - a. The failure to set this report property correctly may require additional effort to confirm deployed reports are referencing the correct data source.

13.8 Use Visual SourceSafe

Visual SourceSafe provides seamless versioning support for the development environment and is easy to integrate with SQL Server Reporting Services.

13.9 Use Shared Data Sources

The use of shared data sources will provide for a centralized storage of database credentials. This will decrease the amount of work required to deploy reports throughout deployment, testing and production environments.

In addition, setting the “Overwrite Data Sources” option in the Report Properties tab to “False” will prevent a development data source from overwriting a production data source.

13.10 Use Views and Stored Procedures

Using views and stored procedures for data sets reduces effort in maintaining the reports should query selection or field selection criteria change.

13.11 Create a backup of the Encryption Key

In order to protect all of the reports and data sources that have been deployed to the Report Server, it helps to create a backup of the Encryption Key because the sensitive information stored in the Report Server can become corrupt and no longer accessible. This leads to several problems including the inability to view or decrypt report credentials.

13.12 Review Reports Before Deploying

Before deploying reports, it is important to confirm several tasks:

- 1) Confirm the queries are optimized.
- 2) Confirm reports look as expected when rendered to a different format. The same report can look different when rendered as a .pdf, .tif or .html file.
- 3) Confirm data is formatted properly and grouping is functioning where applicable.

13.13 Use folders and Descriptions to Organize Reports

Placing the reports in a directory structure can alleviate the burden of organizing reports as the quantity grows and becomes more involved to maintain. For PRISM, separate directories are created for Reports and Data Sources, and subdirectories for PRISM and PRISM Juvenile elements under each.

13.14 Assign Security at the Folder Level

Assigning security at the folder level and letting the folder contents inherit security permissions will allow for easier maintenance of security features as the quantity of reports continues to grow. In addition, it is strongly recommended to assign security rights to domain groups instead of individual users.