Roche

Pharmaceuticals

Warsaw CoE

Warsaw CoE Local Supporting Document

Guide to Coding Standards and Naming Conventions

Document ID: Warsaw CoE SPT 9.1.1

Version: 2.0

Effective Date: 10 Mar 2006

PL id: 1144748/3

Document Information

Document Owner	Warsaw CoE Site Head.
Document Location	Physically: Warsaw CoE Document Archive, Warsaw, Poland. Electronically: Approved version in Project Library in PGL SOP Repository (http://projectlibrary.bas.roche.com/projectlibrary/livelink?func=ll&objId=907821&objAction=browse&sort=name).
Geographical Scope	Local: Warsaw CoE.
Associated Documentation	None.

Review

Role	Name	Dept.	Date	Signature
Author	Adrian Czabanowski	Warsaw CoE	In Project Library	Signed electronically
Author	Marek Marszałek	Warsaw CoE	In Project Library	Signed electronically
Author	Leszek Popko	Warsaw CoE	In Project Library	Signed electronically
Author	Paweł Postupalski	Warsaw CoE	In Project Library	Signed electronically
Author	Marek Półtorak	Warsaw CoE	In Project Library	Signed electronically
Author	Marek Suski	Warsaw CoE	In Project Library	Signed electronically
Author	Daniel Szokalski	Warsaw CoE	In Project Library	Signed electronically
Author	Zbigniew Zawadzki	Warsaw CoE	In Project Library	Signed electronically
Technical Review	Piotr Czarnas	Warsaw CoE	In Project Library	Signed electronically
Technical Review	Mirosław Majczak	Warsaw CoE	In Project Library	Signed electronically
PGDQ Review	Wolfgang Schumacher	PGDQ / Basel	In Project Library	Signed electronically

Approval

Role	Name	Dept.	Date	Signature
Document Owner	John W. Hill	PGD	In Project Library	Signed electronically

Document History

Version	Changes	Effective Date
1.0	First Approved versions by PGLM (authors: A. Czabanowski, M. Marszałek, L. Popko, P. Postupalski, M. Suski, Z. Zawadzki, M. Majczak) and PGLI (author: P. Czarnas).	5 Feb 2005
2.0	PGLM (JAVA, Visual Basic, ORACLE SQL, PL/SQL) and PGLI (.Net and C#) documents merged and contents standardized by D. Szokalski. Document adjusted to new PG documentation standard; Objective, Scope and Definitions sections added by D. Szokalski. CVS Directory Structure and Naming Conventions section added by M. Półtorak.	10 Mar 2006

Notes: This document is published without signatures. The electronic signatures for this document are included in Project Library.

Table of Contents

List of Figures

1 Objective

The objective of this supporting document is to give a detailed overview to coding standards and naming conventions corresponding with technologies used at Warsaw CoE site in order to ensure consistent structure and dependencies of programs developed in-house.

Programs developed adhering to the rules contained within presented standards should:

- become more easily understandable and transparent,
- become easier to maintain,
- become easier to enhance,
- allow code reuse.

This supporting document, as an overview, presents many ideas and rules that have been stated in other coding standards existing in software development industry, such as, for instance, Code Conventions for the Java Programming Language from SUN Microsystems.

This supporting document also presents Concurrent Versioning System (CVS) directory structure and naming conventions.

2 Scope

2.1 Applicability

This document is applicable whenever Warsaw CoE staff members need a detailed guideline to specific programming technology standard and naming convention covered in this document. This Guide to coding standards and naming conventions will cover the coding standards and naming conventions (including CVS directory structure and naming conventions) of the following technologies used as programming languages at Warsaw CoE site:

- Visual Basic,
- JAVA,
- ORACLE SQL Language,
- PL/SQL Language,
- .Net and C#.

Organizationally this supporting document applies to Warsaw CoE staff members dealing with software development activities in their daily work.

Geographical scope of this supporting document is limited to Warsaw CoE located in Poland.

2.2 Out of scope

Coding standards different than 5 presented above are not covered in this supporting document. Whenever new programming technology is introduced at Warsaw CoE site this guide shall be revised in order to add a chapter concerning the specific technology or a separate guide shall be drawn up for a new technology.

3 Definitions

3.1 Acronyms

The following acronyms are used in this document:

Acronym	Description
CVS	Concurrent Versioning System
GPF	General Protection Fault
IDE	Integrated Development Environment
IM	Information Management
INS	Instruction
IT	Information Technology
MDI	Multi Document Interface
PL	Procedural Language
SDLC	Software Development Lifecycle
SPT	Supporting Document
SQL	Structured Query Language
UI	User Interface
VB	Visual Basic
Warsaw CoE	Warsaw Center of Excellence
CVS	Concurrent Versions System
SW	Software

4 Visual Basic 6.0 language

4.1 Tools setup

4.1.1 Visual Studio 6.0 settings

Within your Visual Basic IDE go to Tools/Options and ensure your environment is using the following settings:

Tab	Option	Setting	Comment
Editor	Code Settings -Require Variable Declaration	Checked	When this option is set then editor requires you to declare all variables prior to usage. Setting this option to true automatically adds the statement "Option Explicit" into the source code.
Editor	Code Settings -Tab Width	4	Allows for easier reading and maintainability of code.
General	Error Trapping -Break in class module	True	Any unhandled error produced in a class module causes the project to enter break mode at the line of code in the class module, which produced the error.
General	Compile -Compile On Demand	False	Determines whether a project is fully compiled before it starts, or whether code is compiled as needed, allowing the application to start sooner. If you choose the Start With Full Compile command on the Run menu, Visual Basic ignores the Compile on Demand setting and performs a full compile.
Environment	When a program starts -Save Changes	True	This option will automatically save any changes you have made before the program starts. This is done to assure no work is lost due to General Protection Faults (GPF) or other unexpected errors. If you have a new file, the "Save as…" common dialog box appears so you can give a name and location for your project file.

4.1.2 Screen layouts for MDI Forms

Main Forms

1. Set the Main form as an MDIForm.
2. Use the MSComctLib.toolbar object to display shortcuts to menu items.
3. Use the MSComctlLib.StatusBar object to display messages and the date.
4. The main form should display maximized and be able to be minimized and resized.

Child Forms

1. Fonts
    a. MS Sans Serif, size 8, for all standard input fields and labels. (VB Default)

b. Black color for standard text descriptions. Different colors for emphasis.

2. Child forms should be children of the main MDI Parent Form. MDIChild property set to true.
3. Toolbar and menu items should be made enabled/disabled when form is activated.
4. Forms should be able to be minimized, maximized and resized.
5. The Edit Frame should be resized to reflect the size of the form.
6. Controls on the screen should be resized within the form when appropriate.
7. All form operations should exist on the menus and toolbar unless it is inappropriate, then use buttons on the form.
8. Controls should be read left to right, top to bottom, and the tab order should reflect this.
9. Radio buttons should be grouped within a frame.
10. Controls should be no bigger than necessary.
11. Controls needs to placed on the form so that they are ordered into rows and columns. This means that as much as possible controls going down the screen have the same value set for the Left property and controls going across the screen share the same Height property value.
12. If a Textbox control is going to be used to display read only data then this control should be disabled so that the users will know that they cannot enter data.
13. MSFlexgrid should be used to display data in a grid.
14. The escape key should close the current form.
15. All child forms in the application should behave in the same way. They should all have the same look and feel.

4.2 Source code readability rules

4.2.1 Indention

Indent code in:

- Between an "`If`" statement and its "`End If`", "`Else`" or "`ElseIf`".
- Between an "`Else`" and its "`End If`".
- Between an "`ElseIf`" and its "`Else`" or "`End If`".
- Between a "`Select`" statement and its "`End Select`" statement.
- Between each "`Case`" statement in a "`Select`".
- Between a "`Do`" statement and its "`Loop`".
- Between a "`With`" statement and its "`End With`".
- Between a "`For`" statement and its "`Next`".
- Between an "`Edit`" or "`AddNew`" method and its "`Update`" or "`CancelUpdate`".
- Between the start and end of a transaction.
- Within the declaration section to show subordination, for instance between a "`Type`" and its "`End Type`".

4.2.2 Line length

Avoid lines longer than 80 characters.

4.2.3 Wrapping lines

When assigning SQL statement to string variable split SQL keywords into new line of code.

Example:

```
strSQL =  "SELECT NAME, LASTNAME" & _
                "FROM EMPLOYES" & _
                "WHERE ID = 10"
```

4.2.4 Blank lines

Blank lines improve readability by setting off sections of code that are logically related.
Two blank lines should always be used between sections of a source file.
One blank line should always be used in the following circumstances:

- between methods,
- between the local variables in a method and its first statement,
- before a block or single-line comment,
- between logical sections inside a method to improve readability.

4.2.5 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space.

  Example:

  ```
  While (True)
      ...
  Loop
  ```

  Note: Blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary and arithmetic operators should be separated from their operands by spaces.

  Example 1:

  ```
  a = c + d
  ```

```
a = (a + b) / (c * d)
```

Example 2:

```
While (iIndex < 1)

  iIndex = iIndex + 1

Loop
```

4.3 Source files

4.3.1 Naming rules

Type	Prefix
Form	frm
Class	cls
Module	mod
User control	uc
Log file	log

4.4 Comments

4.4.1 File header comments

Below and example is presented of the comment block in the header of the file that describes the main purpose of the file.

```
'************************************************
' Module:      clsGenericDataItems
' Filename:    clsGenericDataItems.cls
' Copyright:   Copyright 2004 Roche. All Rights Reserved.
' Description:
'
'
' Module history:
' 1.0       10/Jul/2004
'           John Smith
'           Initial Version
'************************************************
```

4.4.2 Methods header comments

Below and example is presented of the comment block in the header of the procedure that describes the main purpose of the procedure.

```
'************************************************
' Procedure Name:  HelloWorld
' Parameters:
'              vstrUserName - String - User name
' Return Value:
'              String - Hello message
' Description:
'               Create hello message for specified user name
'
' Procedure history:
' 1.0       10/Jul/2004
'           John Smith
'           Initial Version
'************************************************
Private Function HelloWorld(ByVal vstrUserName As String) As String
```

A property does not need to have two headers for get and let\set procedures. One header explaining the property is enough.

4.4.3 Code comments

Part of the code where variables are declared, initialized, cleaned up or destroyed must be commented and separated from rest of the code.

Every "`If`", "`Select Case`", any loops or other statements must have explanations about its purpose.

Place a comment against any piece of code which could be ambiguous.

4.5 Declarations

4.5.1 Variable naming rules

Variable names should be prefixed with to reflect: the scope, data type and then the variable name.

Scope	Prefix
Global (example: Global giCount As Integer)	g
Module or form level (example: Dim mstrName As String)	m
Reference (example: ByRef rstrName As String)	r
Value (example: ByVal viJobCode As Integer)	v

Local (example: Dim iCount as Integer)	i

Constants

Name of constant is always upper case. When name consist from two or more words split them with the "_" character.

Example:

```
"SORT_CODE_PREFIX"
```

Enumerations

Name of the enumeration should have as a prefix name of the company.

Example:

```
Public Enum rocheDaysOfWorkingWeek
    Monday = 1
    Tuesday = 2
    Wednesday = 3
    Thursday = 4
    Friday = 5
End Enum
```

User Types

Name of the user-defined type should have 'typ' as prefix.

Example:

```
Type typEmployee
     iId as String
     strFirstName As String
     strLastName As String
End Type
```

Standard data types prefixes (Hungarian notation)

Data type	Prefix
Boolean	b
String	str
Date	dt

Currency	cur
Double	dbl
Array	arr
Single	sng
Handle	h
Long	l
Integer	i
Variant	vnt
Object	o
Collection	col
Enumeration	e
Byte	b
User-defined type	typ

ADO data types prefixes

Data type	**Prefix**
Connection	cnn
Command	cmd
Parameter	par
Transaction	trn
Recordset	rs

Custom controls types prefixes

Data type	**Prefix**
Animation button	ani
Combo Box	cbo
Checkbox	chk
Picture clip	clp
Command Button	cmd
Communications	com
Data control	dat
Data aware combo box	dbc
Data aware grid	dbg

Data aware list box	dbl
Directory list box	dir
Common dialog control	dlg
Drive list box	drv
File list box	fil
Frame	fra
Form	frm
Gauge	gau
Group push button	gpb
Grid	grd
Horizontal scroll bar	hsb
Image	img
Image list	iml
Keyboard key state	key
Label	lbl
Line	lin
List box	lst
List view	lsv
Multimedia control	mci
MDI child form	mdi
Menu	mnu
MAPI message	mpm
MAPI session	mps
Masked edit	msk
OLE control	ole
Option Button	opt
Outline control	out
Progress Bar	pbr
Picture	pic
3D panel	pnl
Report control	rpt
Rich text box	rtb

Status Bar	sbr
Shape controls	shp
Slider	sld
Spread	spd
Spin control	spn
Tab control	tab
Tool bar	tbr
Tab strip	tbs
Timer	tmr
Tree view	trv
Text Box	txt
Vertical scroll bar	vsb

Menus

When naming menus, use the prefix „mnu" followed by the the caption that the menu item belongs to. For example for "File -> Send -> Fax" create "mnuFSendFax" name.

4.5.2 Building elements naming convention

Function can not have more then 50 lines of code (comments and blank lines does not count). Move and split large block of code into smaller functions. Some of the functions will have generic purpose and those functions should be moved to modules for re-use.

4.6 Statements

4.6.1 Loop statements

Do Loop

```
Do [{While | Until} condition]

    [statements]
    [Exit Do]
    [statements]

Loop
```

Do Loop While\Until

```
Do
```

```
        [statements]
        [Exit Do]
        [statements]

    Loop [{While | Until} condition]
```

For Each

```
    For Each element In group

        [statements]
        [Exit For]
        [statements]

    Next [element]
```

For Next

```
    For counter = start To end [Step step]

        [statements]
        [Exit For]
        [statements]

    Next [counter]
```

While Wend

```
    While condition

        [statements]

    Wend
```

4.6.2 Logical statements

```
If condition Then [statements] [Else elsestatements]

If condition Then

    [statements]

[ElseIf condition-n Then

    [elseifstatements] ...
```

```
[Else

      [elsestatements]]

End If
```

4.6.3 Choice statements

Select Case

```
Select Case testexpression

      [Case expressionlist-n

            [statements-n]] ...
      [Case Else

            [elsestatements]]

End Select
```

Switch

```
Switch(expr-1, value-1[, expr-2, value-2…[, expr-n,value-n]])
```

Choose

```
Choose(index, choice-1[, choice-2, ... [, choice-n]])
```

4.6.4 Return value statements

Value that must be returned from function as a result, must be passed from local variable instead of directly assign return value to function name.

Example:

```
Public Function Test As Boolean
   Dim bResult As Boolean

    bResult = True

    Test  = bResult
End Function
```

4.7 Packages

4.7.1 Naming rules

Because of the limitation of Visual Basic libraries names should be made as short as possible. Name of the library should be created by taking first letters of VB project that is used to create a library.

Example:

VB project name: SampleLibraryToMakeOrders.vbp
Name of library: SLTMO.dll

4.8 Error handling

Each function must have error handling and clean up code.
When error was caught, clean up code must be also executed.
Use '`ErrHandler`' and '`ExitHandler`' labels for error handling and clean up code.

Example:

```
Public Function Test As Boolean
    On Error GoTo ErrHandler

          [statements]

ExitHandler:
    ' Write cleanup code here

    Exit Function
ErrHandler:
    ' Write error handling code here
    Select Case err.Number
        Case Else
            ' Unexpected error
            ErrorHandler Me, "Test", err.Number, err.Description
    End Select

    Resume ExitHandler
End Function
```

4.9 Programming practices

1. In a form, module, or class, make all internal functions and subroutines private.
2. Make public only the routines that will be called from outside objects.
3. When a form closes, it should check to see if the data has changed and needs to be saved.
4. Operations should be disabled/enabled appropriately at all times.
5. When the focus is on a text box, the data in the box should be selected when appropriate. The text in remarks/comments fields should not be selected normally.

6. Use global variables sparingly. Pass parameters to functions and subroutines instead of creating global variables.
7. When declaring variable use full package\library name with the variable type (for example "`ADODB.Recordset`").
8. Use comments in the code not to repeat what code do but what is the purpose of it.
9. When creating a function with generic purpose move it to the 'Common'\'Helpers' module for re-use.
10. When naming function/procedure a good practice is to use a combination of Nouns and Verbs. For example `InvoiceAdd`, `InvoiceDelete`, `InvoiceCalcTax`. Discuss as a team and use similar naming conventions for functions\procedures as other developers. This will make code easier to read for other colleagues in your team. If you are not sure is the name is correct, ask other developer for advice.
11. Before implementing a common function check maybe this function already exists but has different name than you might expect.
12. Each file where code can be written must starts with 'Option Explicit'. See Visual Studio Standards above.
13. Use interfaces classes, when you are creating classes with the same sets of functions, procedures or properties.
14. Use '`DoEvents`' calls in part of the code where application will do extensive operations (loops, getting data from database). This will prevent UI to freeze but will lower the performance of the application.
15. Any object should be first declared and then initialized. If you declare an object as new this will slow down performance. Use two lines as in Dim rs as recordset: set rs = new recordset
16. First line in the procedure code must be declaration of the error handler label.
17. Create a module (for example "Global.bas") for storing all global variables or constants.
18. Local variables must be declared at the top of the procedure and not in the body.
19. When naming Boolean variables the positive form should be use (for example `bIsValid` not `bIsNotValid`).
20. Always include "`Case Else`" in "`Select Case`" structures.
21. Never compare a Boolean for equality with True or False.
22. Do not use default properties of objects.

4.10 CVS Directory Structure and Naming Conventions

The directory structure of the CVS repository and naming conventions that should be used for each project is described in this chapter.

For details regarding CVS please refer to "PGL, Source Code Handling and Release Management (Using CVS - Concurrent Versioning System), Global SOP" document.

The directory structure must be clear, consistent and contain separate subdirectories needed for storing in order all the files required to build, compile and unit test the entire application. The directory structure meant to help with keeping order within the different application files in the CVS repository is shown in Figure 1.

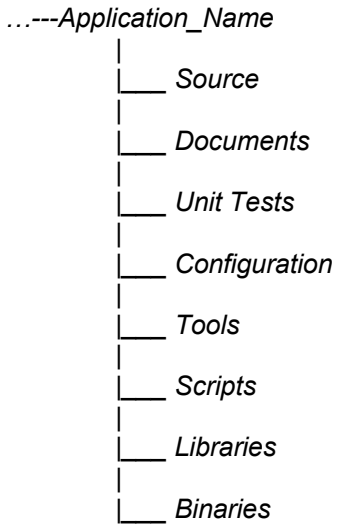


Figure 1: The possible inner structure of "Application_Name" folder for VB language.

"Application_Name" it is the folder containing all the source, documents, tests, configuration, tools, scripts, libraries, binaries and other files belonging to or related to the SW application. The naming conventions for this folder are the following:

APP_FullName

Where:

- APP : Abbreviation for SW Application.
- FullName : Full name of the application.
- If abbreviation for SW application together with the FullName of the application exceeds 44 characters then we are leaving only the abbreviation.

e.g. PSS_Patient_Summary_System

The purpose of the sub-folders within the *"Application_Name"* folder is the following:

- *"Source"* – In this folder all the source code, graphics and other files being integral part of the Application shall be stored.
- *"Documents"* – In this folder shall be stored all the files documenting the developed application. If all the files documenting the application are already in PLv2, listed in README file within the *"Application_Name"* folder and no other documents exist that should be stored here this folder is not necessary.
- *"Unit Tests"* – In this folder all shall be stored all the source code, binaries and data files related to Unit Test prepared for the developed application.
- *"Configuration"* – This is the folder where all configuration data/scripts/documents should be stored.
- *"Tools"* – This is the folder where all supporting tools used during the application development should be stored. In case of the self-developed tools, the source and binaries should be located here. In case when developer have used external tools the installation files, other binaries and source files (if available) should be stored in this directory.
- *"Scripts"* – This is the folder where all the additional supporting scripts (UNIX scripts, SQL scripts, Shell scripts) should be stored.

- *"Libraries"* – This is the folder where the binary, installation and source (if available) versions of the libraries used during the development should be stored. This folder should also be used as a reference for the application during the development in case when binary version of the library is used.
- *"Binaries"* – This is the folder where the binary and preferably installation versions of the official production releases of the application should be stored. All the binaries for production release must be packed into one zip archive. The naming conventions for this file are the following:
  APP_X_Y_Z_A.zip
  Where:
  - APP : Abbreviation for SW Application.
  - X: Version number Major.
  - Y: Version number Minor.
  - Z: Release number.
  - A: Optional (e.g. Built number).

  e.g. PSS_3_1_5_23.zip

Folders *"Source"*, *"Tests"*, *"Binaries"* and *"Tools"* are mandatory for each application. Other folders should be created depending on the developer needs.

Note: Every *"Application_Name"* directory should contain a README file that covers:

- short description of the Application,
- system versions, e.g. OC version, Windows version, where the SW application will work,
- build and install directions,
- direct people to all the related resources:
  - directories of source,
  - online documentation,
  - paper documentation,
  - design documentation,
- anything else that might help someone.

It is preferred that a README file will be in text format but it can also be MS Word or PDF file.

It is important to update the README file with every production release.

Every software release must be properly tagged in CVS according to guidelines given in the "PGL, Release, Tag, and Branch Management, Global Instruction" document.

5 Java language

5.1 Tools setup

Tool setup depends on tool which is used for Java development.

5.1.1 For IntelliJ IDEA:

File > Settings > Project Settings > Global Code Style

Here you can change code convention elements. Default settings are compatible with SUN java coding standards.

5.1.2 For Eclipse:

Window > Preferences, and then Java > Editor

Here you can change code convention elements. Default settings are compatible with SUN java coding standards.

5.2 Source code readability rules

5.2.1 Indention

Tabs should be always replaced by four spaces.

5.2.2 Line length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length-generally no more than 70 characters.

5.2.3 Wrapping lines

When an expression will not fit on a single line, break it according to the following general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are two examples of breaking method calls:

Example 1:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);
```

Example 2:

```
var = someMethod1(longExpression1,
                 someMethod2(longExpression2,
                         longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

Example 1:

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6; // PREFER
```

Example 2

```
longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longname6; // AVOID
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

Example 1:

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}
```

Example 2

```
//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

Line wrapping for `if` statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult.

Example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();            //MAKE THIS LINE EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions.

Example 1:

```
alpha = (aLongBooleanExpression) ? beta : gamma;
```

Example 2:

```
alpha = (aLongBooleanExpression) ? beta
                                 : gamma;
```

Example 3:

```
alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

5.2.4 Blank lines

Blank lines improve readability by setting off sections of code that are logically related.
Two blank lines should always be used in the following circumstances:

- between sections of a source file,
- between class and interface definitions.

One blank line should always be used in the following circumstances:

- between methods,
- between the local variables in a method and its first statement,
- before a block or single-line comment,
- between logical sections inside a method to improve readability.

5.2.5 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space.

  Example:

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

  Note: A blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except "." should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.

  Example 1:

```
a += c + d;
a = (a + b) / (c * d);
```

  Example 2:

```
while (d++ = s++) {
    n++;
}
printSize("size is " + foo + "\n");
```

- The expressions in a `for` statement should be separated by blank spaces.

  Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank space.

  Example 1:

```
myMethod((byte) aNum, (Object) x);
```

Example 2:

```
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

5.3 Source files

5.3.1 Naming rules

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

5.3.2 Organization rules

Java source files have the following ordering:

- beginning comments,
- package and Import statements,
- class and interface declarations.

5.3.2.1 Beginning Comments

All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

5.3.2.2 Package and Import Statements

The first non-comment line of most Java source files is a `package` statement. After that, `import` statements can follow.

Example:

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

Note: The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.

5.3.2.3 Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear.

	Part of Class/Interface Declaration	Notes
1	Class/interface documentation comment (`/**...*/`)	None.
2	`class` or `interface` statement	None.
3	Class/interface implementation comment (`/*...*/`), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
4	Class (`static`) variables	First the `public` class variables, then the `protected`, then package level (no access modifier), and then the `private`.
5	Instance variables	First `public`, then `protected`, then package level (no access modifier), and then `private`.
6	Constructors	None.
7	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

5.4 Comments

Java programs can have two kinds of comments: implementation comments and documentation comments.

Implementation comments are mean for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Implementation comments are those found in C++, which are delimited by /*...*/, and //.

Documentation comments (known as "doc comments") are Java-only, and are delimited by /**...*/. Documentation comments can be extracted to HTML files using the `javadoc` tool.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to

reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or unobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.
Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

5.4.1 Implementation Comment Formats

Programs can have four styles of implementation comments:

- block,
- single-line,
- trailing,
- end-of-line.

5.4.1.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

Example:

```
/*
 * Here is a block comment.
 */
```

Block comments can start with `/*-`, which is recognized by indent(1) as the beginning of a block comment that should not be reformatted.

Example:

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 *    one
 *        two
 *            three
```

```
 */
```

Note: If you don't use indent(1), you don't have to use `/*-` in your code or make any other concessions to the possibility that someone else might run indent(1) on your code.

5.4.1.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line.

Example:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

5.4.1.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

Example:

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
}
```

5.4.1.4 End-Of-Line Comments

The `//` comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code.

Example:

```
if (foo > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;          // Explain why here.
}
//if (bar > 1) {
```

```
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

5.4.2 Documentation Comments

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters `/**...*/`, with one comment per class, interface, or member. This comment should appear just before the declaration.

Example:

```
/**
 * The Example class provides ...
 */
public class Example { ...
```

Note: Top-level classes and interfaces are not indented, while their members are. The first line of doc comment (/**) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately *after* the declaration. For example, details about the implementation of a class should go in in such an implementation block comment *following* the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration *after* the comment.

For further details, see "How to Write Doc Comments for Javadoc" which includes information on the doc comment tags (`@return, @param, @see`).

5.4.3 File header comments

For file headers use block comments and documentation comments.

5.4.4 Methods header comments

For methods headers use block comments and documentation comments.

5.4.5 Code comments

For code use line comments trailing comments or end of line comments.

5.5 Declarations

5.5.1 Variable naming rules

Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore "_" or dollar sign "$" characters, even though both are allowed.

Variable names should be short yet meaningful. The choice of a variable name should be mnemonic, i.e. designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.

The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores "_" (ANSI constants should be avoided, for ease of debugging).

5.5.2 Building elements naming convention

5.5.2.1 Number Per Line

One declaration per line is recommended since it encourages commenting.

Example of preferred layout:

```
int level; // indentation level
int size;  // size of table
```

Example of not preferred layout:

```
int level, size;
```

Do not put different types on the same line.

Example:

```
int foo,  fooarray[]; //WRONG!
```

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs.

Example:

```
int    level;          // indentation level
int    size;            // size of table
Object currentEntry;  // currently selected table entry
```

5.5.2.2 Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

5.5.2.3 Placement

Put declarations only at the beginning of blocks (a block is any code surrounded by curly braces "{" and "}").

Example:

```
void myMethod() {
    int int1 = 0;          // beginning of method block

    if (condition) {
        int int2 = 0;      // beginning of "if" block
        ...
    }
}
```

Don't wait to declare variables until their first use as it can confuse the unwary programmer and hamper code portability within the scope.

The one exception to the rule is indexes of `for` loops, which in Java can be declared in the `for` statement.

Example:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block.

Example:

```
int count;
...
myMethod() {
```

```
    if (condition) {
        int count = 0;      // AVOID!
        ...
    }
    ...
}
```

5.5.3 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list.
- Open brace "{" appears at the end of the same line as the declaration statement.
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{".

  Example:

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}

    ...
}
```

- Methods are separated by a blank line.

5.6 Statements

5.6.1 Loop statements

5.6.1.1 "`for`" Statements

A `for` statement should have the following form:

```
for (initialization; condition; update) {
    statements;
}
```

An empty `for` statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a `for` statement, avoid the complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialization clause) or at the end of the loop (for the update clause).

5.6.1.2 "`while`" Statements

A `while` statement should have the following form:

```
while (condition) {
    statements;
}
```

An empty `while` statement should have the following form:

```
while (condition);
```

5.6.1.3 "`do-while`" Statements

A `do-while` statement should have the following form:

```
do {
    statements;
} while (condition);
```

5.6.2 Logical statements

5.6.2.1 "`if`", "`if-else`", "`if else-if else`" Statements

The `if-else` class of statements should have the following form:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
```

```
    statements;
} else{
    statements;
}
```

Note: `if` statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

5.6.3 Choice statements

5.6.3.1 "`switch`" Statements

A `switch` statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

Every time a case falls through (doesn't include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

5.6.3.2 "`try-catch`" Statements

A `try-catch` statement should have the following format:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
```

```
}
```

A `try-catch` statement may also be followed by `finally`, which executes regardless of whether or not the `try` block has completed successfully.

Example:

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

5.6.4 Return value statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way.

Example:

```
return;

return myDisk.size();

return (size ? size : defaultSize);
```

5.7 Packages

5.7.1 Naming rules

The first non-comment line of most Java source files is a `package` statement. After that, `import` statements can follow.

Example:

```
package java.awt;

import java.awt.peer.CanvasPeer;
```

Note: The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.

5.8 Error handling

The names of Exceptions must end with the word "Exception".

Example:

```
NoMoreNumbersException
```

Java distinguishes between two types of exception:

Checked exceptions extend `java.lang.Exception`, and the compiler insists that they are caught or explicitly rethrown.

Unchecked or **runtime** exceptions extend `java.lang.RuntimeException`, and don't need to be caught.

If a method throws checked exceptions, it should be declared on a separate row directly after the method declaration.

Example:

```
public int getNextValue(int previousValue)
    throws NoMoreValuesException {
    ...
}
```

Whether we used checked or unchecked exceptions, there is still need to address the issue of "nesting" exceptions. Typically this happens when we're forced to catch a checked exception we can't deal with, but want to rethrow it, respecting the interface of the current method. This means that we must wrap the original, "nested" exception within a new exception.

Java 1.4-aware exceptions should implement a constructor taking a throwable nested exception and invoking the new Exception constructor. This means that we can always create and throw them in a single line of code as follows.

Example:

```
catch (RootCauseException ex) {
     throw new MyJava14Exception("Detailed message", ex);
}
```

5.9 Logging

It is recommended to use one of following libraries for logging:

- Log4j
- JDK 1.4 logging API
- Commons logging

The use of these libraries depend on system architecture and project specific needs.

5.10 Programming practices

5.10.1 Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten-often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a `struct` instead of a class (if Java supported `struct`), then it's appropriate to make the class's instance variables public.

5.10.2 Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead.

Example:

```
classMethod();              //OK
AClass.classMethod();       //OK
anObject.classMethod();     //AVOID!
```

5.10.3 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

5.10.4 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read.

Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

Don’t use the assignment operator in a place where it can be easily confused with the equality operator.

Example:

```
if (c++ = d++) {          // AVOID! (Java disallows)
    ...
}
```

should be written as:

```
if ((c++ = d++) != 0) {
```

```
    ...
}
```

Don't use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler.

Example:

```
d = (a = b + c) + r;         // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

5.10.5 Miscellaneous Practices

5.10.5.1 Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

Example:

```
if (a == b && c == d)      // AVOID!
if ((a == b) && (c == d)) // RIGHT
```

5.10.5.2 Returning Values

Try to make the structure of your program match the intent.

Example 1:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

should instead be written as:

```
return booleanExpression;
```

Example 2:

```
if (condition) {
    return x;
}
return y;
```

should be written as:

```
return (condition ? x : y);
```

5.10.5.3 Expressions before `?' in the Conditional Operator

If an expression containing a binary operator appears before the `?` in the ternary `?:` operator, it should be parenthesized.

Example:

```
(x >= 0) ? x : -x;
```

5.10.5.4 Special Comments

Use `XXX` in a comment to flag something that is bogus but works. Use `FIXME` to flag something that is bogus and broken.

5.11 CVS Directory Structure and Naming Conventions

The directory structure of the CVS repository and naming conventions that should be used for each project is described in this chapter.

For details regarding CVS please refer to "PGL, Source Code Handling and Release Management (Using CVS - Concurrent Versioning System), Global SOP" document.

The directory structure must be clear, consistent and contain separate subdirectories needed for storing in order all the files required to build, compile and unit test the entire application. The directory structure meant to help with keeping order within the different application files in the CVS repository is shown in Figure 2.

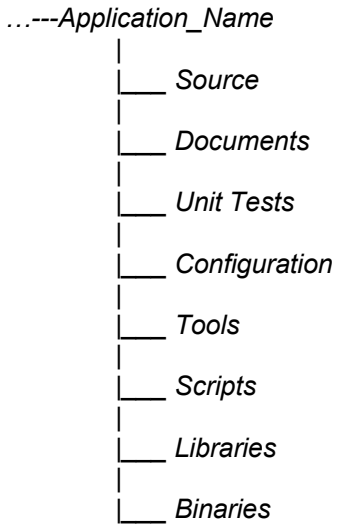


Figure 2: The possible inner structure of "Application_Name" folder for Java language.

"Application_Name" it is the folder containing all the source, documents, tests, configuration, tools, scripts, libraries, binaries and other files belonging to or related to the SW application. The naming conventions for this folder are the following:

APP_FullName

Where:

- APP : Abbreviation for SW Application.
- FullName : Full name of the application.
- If abbreviation for SW application together with the FullName of the application exceeds 44 characters then we are leaving only the abbreviation.

e.g. PSS_Patient_Summary_System

The purpose of the sub-folders within the *"Application_Name"* folder is the following:

- *"Source"* – In this folder all the source code, graphics and other files being integral part of the Application shall be stored.
- *"Documents"* – In this folder shall be stored all the files documenting the developed application. If all the files documenting the application are already in PLv2, listed in README file within the *"Application_Name"* folder and no other documents exist that should be stored here this folder is not necessary.
- *"Unit Tests"* – In this folder all shall be stored all the source code, binaries and data files related to Unit Test prepared for the developed application.
- *"Configuration"* – This is the folder where all configuration data/scripts/documents should be stored.
- *"Tools"* – This is the folder where all supporting tools used during the application development should be stored. In case of the self-developed tools, the source and binaries

should be located here. In case when developer have used external tools the installation files, other binaries and source files (if available) should be stored in this directory.

- *"Scripts"* – This is the folder where all the additional supporting scripts (UNIX scripts, SQL scripts, Shell scripts) should be stored.
- *"Libraries"* – This is the folder where the binary, installation and source (if available) versions of the libraries used during the development should be stored. This folder should also be used as a reference for the application during the development in case when binary version of the library is used.
- *"Binaries"* – This is the folder where the binary and preferably installation versions of the official production releases of the application should be stored. All the binaries for production release must be packed into one zip archive. The naming conventions for this file are the following:
  APP_X_Y_Z_A.zip
  Where:
  - APP : Abbreviation for SW Application.
  - X: Version number Major.
  - Y: Version number Minor.
  - Z: Release number.
  - A: Optional (e.g. Built number).

  e.g. PSS_3_1_5_23.zip

Folders *"Source"*, *"Tests"*, *"Binaries"* and *"Tools"* are mandatory for each application. Other folders should be created depending on the developer needs.

Note: Every *"Application_Name"* directory should contain a README file that covers:

- short description of the Application,
- system versions, e.g. OC version, Windows version, where the SW application will work,
- build and install directions,
- direct people to all the related resources:
  - directories of source,
  - online documentation,
  - paper documentation,
  - design documentation,
- anything else that might help someone.

It is preferred that a README file will be in text format but it can also be MS Word or PDF file.

It is important to update the README file with every production release.

Every software release must be properly tagged in CVS according to guidelines given in the "PGL, Release, Tag, and Branch Management, Global Instruction" document.

6 Oracle SQL language

6.1 Layout and Documentation

6.1.1 Layout

Write SQL statements in lowercase characters. Use uppercase characters when attention should be drawn to a specific part of the SQL statement. Note that comparisons to literal constants might require uppercase characters.

Start every clause that contains a column name, a table name on a new line.

Place commas that separate column or table names on the first position of a new line. This rule also applies to data definition statements.

Example:

```
select          crp_crtn.crtn
,          crp_crtn.crtn_code
from       crp_crtn
where      to_char(crp_crtn.crtn) like '33%'
AND        CRP_CRTN.CRTN != 'GOB'          --exclude Goblet CRTNs
```

6.1.2 Inline Documentation

Use inline documentation only to document or clarify individual parts of a SQL statement.

Place comments within SQL statements on the right-hand side of the SQL text, at the end of the line. Right-align the *end of comment* markers. Use complete lines within the SQL statement if the comment lines become too large.

Example:

```
select          crp_crtn.crtn
,          crp_crtn.crtn_code
from       crp_crtn
,          mon_report_tot
where      crp_crtn.crtn = mon_report_tot.crtn       /*join condition        */
and        mon_report_tot.mon_type = 'ICR30'         /*PD30 exists           */
and        crp_crtn.crtn != 'GOB'              /*exclude Goblet CRTNs  */
and        crp_crtn.status_flag in ('PLA', 'ACT', 'COM', 'PRT')
/*take only Planned, Active, Completed and Prematurely Terminated CRTNs    */
```

6.2 Datatype Conversion

Specify an explicit datatype conversion function where Oracle would otherwise perform implicit datatype conversion.

Example:

```
where to_char(mon_report_tot.mon_date, 'DD-MON-YYYY') like '%JAN%'
not
where mon_report_tot.mon_date like '%JAN%'
```

When you compare an indexed column to a constant, apply any necessary datatype conversion to the constant.

Example:

```
where mon_report_tot.mon_type = upper(p_mon_type)
not
where lower(mon_report_tot.mon_type) = p_mon_type
```

Note: Do not store graphic images in a LONG column; the resulting implicit RAW_TO_HEX conversion takes up a lot of resources (memory and disk space). Instead use a column of datatype LONG RAW.'

6.3 Functions and Operators

6.3.1 Functions

Avoid the DECODE expressions that contain return arguments of more than one datatype. If you cannot avoid this, explicitly convert all return arguments to a single datatype.

Example:

```
select            decode(crp_crtn.status_flag, 'PLA', 'Planned', '0')   indicator
from        crp_crtn
not
select            decode(crp_crtn.status_flag, 'PLA', 'Planned', 0)
      indicator
from        crp_crtn
```

Note: Do not use the MAX function to retrieve a new highest value for a column; for example, for key generation. Instead, use a sequence generator or a separate table that contains the highest value(s).

6.3.2 Operators

Always specify parentheses around the arguments of an OR operator.

Be critical of specifying the IS NULL operator in a SQL statement; try to prevent unnecessary full table scans.

Avoid the use of the LIKE operator with columns of DATE and NUMBER datatype; implicit datatype conversion may disable index use.

6.4 SQL Statements

6.4.1 SELECT

Specify a full column list in a SELECT statement. Do not use the asterisk (*) shorthand in SELECT statements in application programs.

Only select those columns that you actually use in your program.

Prefix each column in the SELECT list with a table alias, even in a statement that selects rows from a single table or view.

Specify a column alias for expressions and for columns that are modified by a function.

Example:

```
select          max(mon_report_tot.mon_date) last_visit
from       mon_report_tot
```

Note: Never use the original column name for a column alias.

6.4.2 FROM

Don't prefix table names in the FROM clause with usernames. Use public synonyms instead.

Example:

```
select          crp_crtn.crtn
,          crp_crtn.status_flag
from       crp_crtn
not
select          crp_crtn.crtn
,          crp_crtn.status_flag
from       aims.crp_crtn
```

6.4.3 WHERE

Write any join conditions directly following the reserved word WHERE.

Example:

```
select          crp_crtn.crtn
,          crp_crtn.status_flag
,          mon_report_tot.mon_date
,          mon_report_tot.mon_type
from       crp_crtn
,          mon_report_tot
where      crp_crtn.crtn = mon_report_tot.crtn
and        mon_report_tot.mon_type = 'ICR30'
```

not

```
select          crp_crtn.crtn
,          crp_crtn.status_flag
,          mon_report_tot.mon_date
,          mon_report_tot.mon_type
from       crp_crtn
,          mon_report_tot
where      mon_report_tot.mon_type = 'ICR30'
and        crp_crtn.crtn = mon_report_tot.crtn
```

6.4.4 GROUP BY

Don't group on columns modified by other expressions or functions than those stated in the SELECT list.

Place as many conditions as possible in the WHERE clause instead of in the HAVING clause, to limit the number of rows that need to be sorted.

Example:

```
select          max(mon_report_tot.mon_date) last_visit
,          mon_report_tot.crtn
from       mon_report_tot
where      mon_report_tot.crtn = 33035
group by   mon_report_tot.crtn
```

not

```
select          max(mon_report_tot.mon_date) last_visit
,          mon_report_tot.crtn
from       mon_report_tot
group by   mon_report_tot.crtn
having          mon_report_tot.crtn = 33035
```

6.4.5 ORDER BY

Use the ORDER BY clause only when your data really needs to be presented in a sorted order.

Don't rely on sorting that occurs by index use without specifying an ORDER BY clause.

Use numbers instead of column names in the ORDER BY list only with the set operators UNION, MINUS, and INTERSECT.

Example:

```
select          crp_protocol.prot_no
,          crp_protocol.pla_enr_start
,          crp_protocol.pla_enr_end
,          crp_protocol.pla_cases
from       crp_protocol
order by   crp_protocol.prot_no asc
```

```
,          crp_protocol.pla_enr_start desc
```

Example:

```
select          crp_protocol.prot_no
,          crp_protocol.pla_enr_start
,          crp_protocol.pla_enr_end
,          crp_protocol.pla_cases
from       crp_protocol
union
select          crp_protocol.prot_no
,          crp_protocol.rev_enr_start
,          crp_protocol.rev_enr_end
,          crp_protocol.rev_cases
from       crp_protocol
order by   1 asc
,          2 desc
```

Use the abbreviations ASC and DESC for ASCENDING and DESCENDING, respectively.

Use the DESC keyword only if one or several columns have to be sorted in descending order. Remember that the default ordering is ascending.

Note: If you use mixed ascending and descending sorting, specify both the ASC and DESC keywords for all column names in the ORDER BY list.

6.4.6 FOR UPDATE

Limit the number of rows that you reserve for future update by always specifying a restrictive WHERE clause.

Always specify a column list in the FOR UPDATE clause. Omitting the column list in a join statement will result in placement of row locks on all rows of all tables selected by the statement.

Specify the primary key column(s) of a table, prefixed with the table alias, in the FOR UPDATE clause of a SELECT statement, to indicate from which table(s) rows must be locked.

Only specify NOWAIT in a FOR UPDATE clause if the resulting error can be properly handled by the programming environment.

6.4.7 Subqueries

If possible, rewrite subqueries into joins.

Example:

```
select          crp_crtn.crtn
,          crp_crtn.status_flag
from       crp_crtn
```

```
,           mon_report_tot
where       crp_crtn.crtn = mon_report_tot.crtn
and         mon_report_tot.mon_type = 'ICR30'
not
select           crp_crtn.crtn
,           crp_crtn.status_flag
from        crp_crtn
where       exists
                 (select    1
                       from      mon_report_tot
                       where          mon_report_tot.crtn = crp_crtn.crtn
                       and       mon_report_tot.mon_type = 'ICR30'
                    )
```

If possible, write correlated subqueries into non-correlated subqueries by replacing column references in the subquery with bind variables.

Example:

```
select           crpprt1.prot_no
,           crpprt1.pla_cases
from        crp_protocol crpprt1
where       crpprt1.crdp_no = p_crdp_no
and         crpprt1.pla_cases >
            (select    avg(crpprt2.pla_cases)
             from      crp_protocol crpprt2
             where          crpprt2.crdp_no = p_crdp_no
            )
not
select           crpprt1.prot_no
,           crpprt1.pla_cases
from        crp_protocol crpprt1
where       crpprt1.crdp_no = p_crdp_no
and         crpprt1.pla_cases >
            (select    avg(crpprt2.pla_cases)
             from      crp_protocol crpprt2
             where          crpprt2.crdp_no = crpprt1.crdp_no
            )
```

Don’t use the concatenation operator when comparing a column list from a subquery to a column list from the main query.

Example:

```
select           mon_per_all.firstname||' '||mon_per_all.lastname mon_name
from        mon_per_all
where       (mon_per_all.firstname
            ,mon_per_all.lastname) not in
            (select    cmg_pres_resource_all.firstname
             ,         cmg_pres_resource_all.lastname
```

```
            from      cmg_pres_resource_all
           )
```

not

```
select     mon_per_all.firstname||' '||mon_per_all.lastname mon_name
from       mon_per_all
where      mon_per_all.firstname||' '||mon_per_all.lastname not in
           (select    cmg_pres_resource_all.firstname||' '||
                      cmg_pres_resource_all.lastname
            from      cmg_pres_resource_all
           )
```

Note: When comparing a column list from a subquery to a column list from the main query, only perform any necessary datatype conversion in the SELECT list of the subquery.

6.4.8 Outer Join

Don't use outer-join columns in predicates that are branches of an OR.

Limit outer join columns to the top level of the WHERE clause, or to (nested) AND predicates only.

6.4.9 INSERT

Specify a full column list for the target table in INSERT statements.

Example:

```
insert into susar_fax_numbers
           (number_id
           ,description_text
           ,fax_no
           )
values
           (1
           ,'Welwyn dummy fax no.'
           ,'44-01707-363359'
           )
```

not

```
insert into susar_fax_numbers
values
           (1
           ,'Welwyn dummy fax no.'
           ,'44-01707-363359'
           )
```

Specify NULL values for unknown values in INSERT statements with the VALUES or SELECT FROM construct. Don't rely on the implicit insertion of NULL values in non-specified columns.

Example:

```
insert into susar_fax_numbers
           (number_id
           ,description_text
           ,fax_no
           )
values
           (null
           ,'Welwyn dummy fax no.'
           ,'44-01707-363359'
           )
```

not

```
insert into susar_fax_numbers
values
           (
           ,'Welwyn dummy fax no.'
           ,'44-01707-363359'
           )
```

6.4.10 DELETE

Always specify the FROM keyword in DELETE statements.

6.5 CVS Directory Structure and Naming Conventions

The directory structure of the CVS repository and naming conventions that should be used for each project is described in this chapter.

For details regarding CVS please refer to "PGL, Source Code Handling and Release Management (Using CVS - Concurrent Versioning System), Global SOP" document.

The directory structure must be clear, consistent and contain separate subdirectories needed for storing in order all the files required to build, compile and unit test the entire application. The directory structure meant to help with keeping order within the different application files in the CVS repository is shown in Figure 3.

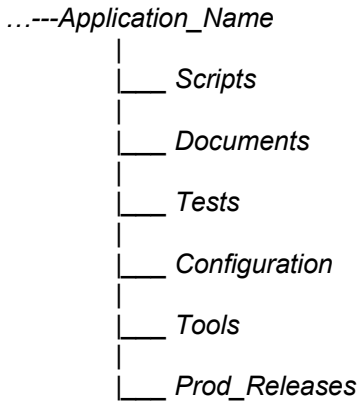


Figure 3: "The possible inner structure of "Application_Name" folder for Oracle SQL scripts"

"Application_Name" it is the folder containing all the source, documents, tests, configuration, tools, scripts, libraries, binaries and other files belonging to or related to the SW application. The naming conventions for this folder are the following:

APP_FullName

Where:

- APP : Abbreviation for SW Application.
- FullName : Full name of the application.
- If abbreviation for SW application together with the FullName of the application exceeds 44 characters then we are leaving only the abbreviation.

e.g. PSS_Patient_Summary_System

The purpose of the sub-folders within the *"Application_Name"* folder is the following:

- *"Scripts"* – This is the folder where all the developed Oracle SQL scripts should be stored.
- *"Documents"* – In this folder shall be stored all the files documenting the developed application. If all the files documenting the application are already in PLv2, listed in README file within the *"Application_Name"* folder and no other documents exist that should be stored here this folder is not necessary.
- *"Tests"* – In this folder all shall be stored all the source code, scripts, data files and other supporting documents related to Unit Test prepared for the developed application.
- *"Configuration"* – This is the folder where all configuration data/scripts/documents should be stored.
- *"Tools"* – This is the folder where all supporting tools used during the scripts development should be stored. In case of the self-developed tools the scripts and other related files should be located here. In case when developer have used external tools the installation files, other binaries and source files (if available) should be stored in this directory.
- *"Prod_Releases"* – This is the folder where the binary and preferably installation versions of the official production releases of the application should be stored. All the production binaries for production release must be packed into one zip archive. The naming conventions for this file are the following:
  APP_X_Y_Z_A.zip

Where:

- APP : Abbreviation for SW Application.
- X: Version number Major.
- Y: Version number Minor.
- Z: Release number.
- A: Optional (e.g. Built number).

e.g. PSS_3_1_5_23.zip

Folders *"Source"*, *"Tests", "Binaries"* and *"Tools"* are mandatory for each application. Other folders should be created depending on the developer needs.

Note: Every *"Application_Name"* directory should contain a README file that covers:

- short description of the Application,
- system versions, e.g. OC version, Windows version, where the SW application will work,
- build and install directions,
- direct people to all the related resources:
  - directories of source,
  - online documentation,
  - paper documentation,
  - design documentation,
- anything else that might help someone.

It is preferred that a README file will be in text format but it can also be MS Word or PDF file.

It is important to update the README file with every production release.

Every software release must be properly tagged in CVS according to guidelines given in the "PGL, Release, Tag, and Branch Management, Global Instruction" document.

7 PL/SQL language

Coding style has a large impact on system maintenance and enhancement. If all applications have a similar coding style regardless of the number of developers then the cost of system maintenance and enhancement are generally reduced.

7.1 General Coding Style

7.1.1 Coding Principles

The coding principles can be summarized under the following heads:

- **Control Flow -** Control flow will enter a procedure at the top and will always leave it at the bottom (except when signaling an error condition). No other entry or exit points will be used.

- **Procedure and Function Size** - A procedure will be kept as short as possible. Wherever possible the developer will keep the procedures and functions to below two physical US Letter pages in length.

- **UPPERCASE Versus lowercase** - In general, code should be in lowercase unless indicated otherwise in this section. Instances where upper or lower case will be used are as follows:
  - Reserved Statements. Reserved words that help define the PL/SQL block will be in uppercase. Attribute type declarations will also be in upper case.

    Example:

    ```
    PROCEDURE, FUNCTION, IS, BEGIN, EXCEPTION, END, VARCHAR2, DATE, NUMBER
    ```

    Complete list is in the chapter List of all reserved words

    Other reserved words that help define the body of the PL/SQL shall be in lowercase.

    Example:

    ```
    select, into, from, if, then, else, while,...
    ```

    Refer to the examples in the following sections for more details.

  - Non-Reserved Statements. Non-reserved statements will be in lowercase.

    Example:

    ```
    table names, variable name, column names ...
    ```

7.1.2 Header

Each PL/SQL block of code will contain header information that assists in maintenance and general understanding of the program.

The header information should contain the following items:

- *Name* - The name of the procedure or function etc.
- *Purpose* - A reasonably detailed description of the purpose of the procedure/function etc. It is important to go into detail if the procedure's purpose is not very clear.
- *Revisions* - A log of all changes made to the procedure/function etc. after implementation. The log consists of modification date, author, and description of change.
  - *Ver.* - The Proposed Release that this modification will be included in. The format should be VX.X
  - *Date* - The date of the modification. The format should be DD-Mon-YYYY.
  - *Author* - The author of the modification. The author's first name and last name should be entered in full.
  - *Description* - A description of the change made
- *Parameters* – The names, types, description and specification whether the values passed is IN, OUT or IN-OUT
- *Returned value* – The type and description of returned value (only if function)
- *Example use* – The example of use e.g. `BOOLEAN := GetBooleanValue('Y');`
- *Limitations* – notes about limitations.
- *Algorithm* – Description or document name which describes algorithm used
- *Notes* – Additional information about procedure/function etc.

7.1.3 Code comments

Use comments within PL/SQL code segments to document individual statements or small groups of statements. Comments or documentation with respect to the complete PL/SQL code segment should be implemented using the documentation facilities of the tool embedding the PL/SQL code segment.

Comments will be relevant, clear, and concise. They will be sufficiently detailed such that when read in total, any further reference to external program documentation is not required. The objective is for the program to 'stand alone' with regard to its purpose and functionality.

Place comment markers within PL/SQL blocks at the right hand side of the PL/SQL text. If the comment lines become too large, use complete lines within the PL/SQL block to specify your comment.

Example:

```
end if; -- comment
```

Each line of a comment will be clearly indicated as a comment. This is to avoid confusion with commented code. Both single line comments and comments that go over a line (with the exception of the procedure header) will be commented using the '--' notation.

Example:

```
--    comment
--    comment
--    comment
```

Note: Remove all comment markers implemented for debugging purposes before you deliver a PL/SQL module.

7.2 Procedure Statement

7.2.1 Naming rules

Procedures must be given meaningful, preferably pronounceable, names. If procedures are defined to support other procedures (for example, value checking), a short name for the supported procedure can be part of that procedure's name.

7.2.2 Procedure header

```
/*************************************************************************
   NAME:       ProcedureName
   PURPOSE:    To calculate the desired information.

   REVISIONS:
   Ver        Date        Author           Description
   ---------  ----------  ---------------  ----------------------------------
   1.0        08-Feb-05                    1. Created this procedure.

   PARAMETERS:
   EXAMPLE USE:     ProcedureName;
   LIMITATIONS:
   ALGORITHM:
   NOTES:

*************************************************************************/
```

7.2.3 Statement Layout

- The PROCEDURE label will commence at the first allowable column.
- The PROCEDURE label and name will appear on one line.
- Parameters will appear on separate lines starting in the fifth column.
- The left parenthesis will be to the immediate left of the first parameter.
- The right parenthesis will be to the immediate right of the last parameter.
- All parameters will specify whether the value passed is IN, OUT or IN-OUT.

Example:

```
IN          Values passed in only
OUT         Values passed out only
IN OUT           Values passed in and out
```

- All parameters will be prefixed with 'p_'.
- The IS label will appear immediately after the procedure name.

Example 1:

```
PROCEDURE {procedure_name} IS
...
...
```

Example 2:

```
PROCEDURE {procedure_name}
   ({parameter1} IN    NUMBER,
    {parameter2} OUT   NUMBER,
    {parameter3} IN OUT     DATE) IS
...
...
```

7.3 Function statement

7.3.1 Naming rules

Functions must be given a meaningful, preferably pronounceable, name. If functions are defined to support other functions (for example, conversion), a short name for the supported function can be part of that function's name.

7.3.2 Function header

```
/*****************************************************************************
   NAME:       FunctionName
   PURPOSE:    To calculate the desired information.

   REVISIONS:
   Ver        Date        Author           Description
   ---------  ----------  ---------------  ----------------------------------
   1.0        08-Feb-05                    1. Created this function.

   PARAMETERS:
   RETURNED VALUE:  NUMBER
   EXAMPLE USE:     NUMBER := FunctionName();
   ASSUMPTIONS:
```

```
   ALGORITHM:
   NOTES:
*******************************************************************************/
```

7.3.3 Statement Layout

- The FUNCTION label will commence at the first allowable column.
- The FUNCTION label and name will appear on one line.
- Parameters will appear on separate lines starting in the fifth column.
- The left parenthesis will be to the immediate left of the first parameter.
- The right parenthesis will be to the immediate right of the last parameter.
- All parameters will specify whether the value passed is IN, OUT or IN-OUT.

Example:

```
IN          Values passed in only
OUT         Values passed out only
IN OUT           Values passed in and out
```

- All parameters will be prefixed with 'p_'.
- The RETURN label and return value type will appear on separate lines starting in the second column.
- The IS label will appear immediately after the return label.

Example 1:

```
FUNCTION {function_name}
 RETURN NUMBER IS
...
...
```

Example 2:

```
FUNCTION {function_name}
   ({parameter1} IN    NUMBER,
    {parameter2} OUT   NUMBER,
    {parameter3} IN OUT     DATE)
 RETURN NUMBER IS
...
...
```

7.4 Package Statement

7.4.1 Naming rules

The name of a package should reflect the function(s) that are stored within that package. If, for example, a package contains procedures and functions specifically supporting an application, then that package's name could be "*application_short_name*_functions".

7.4.2 Package header

```
/*****************************************************************************
   NAME:       PackageName
   PURPOSE:    To calculate the desired information.

   REVISIONS:
   Ver        Date        Author           Description
   ---------  ----------  ---------------  ------------------------------------
   1.0        08-Feb-05                    1. Created this package.

   EXAMPLE USE:     NUMBER := PackageName.MyFuncName(Number);
                    PackageName.MyProcName(Number, Varchar2);
   LIMITATIONS:
   ALGORITHM:
   NOTES:

*****************************************************************************/
```

7.4.3 Statement Layout

Package statement layout contains the following features:

- Package label commences at the first allowable column.
- Package label and package name appears on the first line.
- The AS label is coded on the second line.
- OR REPLACE option is always included.
- END keyword that terminates the package statement is always suffixed with the package name.

Example (Package Definition):

```
CREATE OR REPLACE PACKAGE {package_name} AS
    {procedure specification};
    ...
    ...
END {package_name};
```

Example (Package Body):

```
CREATE OR REPLACE PACKAGE BODY {package_name} AS
    {procedure declaration};
    ...
    ...
END {package_name};
```

Definitions of functions and procedures must be identical in the Package Header and Body definitions. In particular in:

- Procedure and function name.
- Parameters names.
- Parameters passed type IN, OUT, IN-OUT.
- Parameter default values.
- Functions return value type.

7.5 Trigger statement

7.5.1 Naming rules

The trigger name consists of:

- Abbreviation of system name .
- Table name.
- Depending on trigger option: before / after, letter “B” or “A” will be added.
- Depending on trigger option: insert / update / delete, letters I and/or U and/or D will be added.
- Depending on trigger option: for each row/per statement, letter R or S will be added.

Example 1:

```
APD_SOAMATRIX_B_IUD_R
                    APD – system name,
                    SOAMATRIX – table name,
                    Before insert update delete for each row.
```

Example 2:

```
APD_DRUGS_A_D_S
                     APD – system name
                     DRUGS – table name
                     After delete per statement.
```

The name of triggers generated form Oracle Designer as a part of Table API can not be: changed due to rules above
It should be created one trigger per triggering event instead of creating multiple for the same event.

7.5.2 Trigger header

```
/*****************************************************************************
   NAME:       TriggerName
   PURPOSE:    To perform work as each row is inserted or updated.

   REVISIONS:
   Ver        Date        Author           Description
   ---------  ----------  ---------------  ----------------------------------
   1.0        08-Feb-05                    1. Created this trigger.

   PARAMETERS:
   LIMITATIONS:
   ALGORITHM:
   NOTES:

*****************************************************************************/
```

7.6 Stored Procedures and Functions

Be careful that you grant a procedure's or function's owner just those privileges that are needed to manipulate the procedure's underlying database objects.

Store procedures and functions in packages whenever possible, since you can modify a package's definition without causing Oracle to compile calling procedures or functions.

Always manually recompile stored procedures or functions when remote overlying objects have been changed.

Keep track of the persistent state of package variables and package cursors.

Upon calling a package construct qualify its name with the name of the package.

7.7 Declarations

7.7.1 Grouping by type

The declarations will be grouped by type, as follows:

- Types,
- Constants,
- Variables,
- Flags,
- Cursors,
- Records / Tables,
- Exceptions.

Declarations will start in the fifth column.

Declares and attributes will be laid out in a consistent manner.

In procedures, the declaration will be grouped together after the PROCEDURE statement and prior to the BEGIN statement.

Example:

```
PROCEDURE {procedure_name}
   ({parameter1} IN    NUMBER,
    {parameter2} OUT   NUMBER,
    {parameter3} IN OUT     DATE) IS

    {variable1}  CHAR(10);
    {variable2}  NUMBER(9,2);
    {variable3}  VARCHAR2

BEGIN
```

Variables for similar items will be grouped together for readability.

Declarations within a group will be declared in alphabetical order if no other logical grouping exists.

Declaration will be used for initialisation only for constants values.

7.7.2 Variables and Fields

All variables declared in the procedure will be referenced.

When variables that are not constants are declared, each declare statement will end with a semicolon with no intervening space between the attribute and the semicolon.

Example:

```
{variable1}          CHAR(10);
{variable2}          NUMBER(6,2);
```

A PL/SQL variable, except for fields in record structures, should not have a name identical to a column name. Try to assign each variable in a PL/SQL code segment a unique name. However, if necessary, use block names as a prefix to make variable names unique.

All variables shall be prefixed with ‘v_’.

If a variable must always have a value, declare the variable as NOT NULL and assign it a (starting) value at declaration.

Use the %TYPE attribute to declare a temporary variable that is used to store a value related to a database column.

If a variable contains a known constant value throughout the program, declare it as a constant.

7.7.3 Constants

When variables that are constants are declared, each declared statement will end with a semicolon with no intervening space between the constant value and the semicolon. The assigned values will align vertically as shown below:

```
{variable1}          CONSTANT CHAR(10)     := 'PRO';
{variable2}          CONSTANT NUMBER(6,2)  := 456;
```

All constants will be prefixed with ‘c_’.

All constant names will be named in the context of the variable they are being assigned or compare to.

Example:

```
if  (duration_schedule < 1440) then
    {statements};
end if;
```

The integer 1440 should be named ‘c_duration_schedule_max_minutes’. Names such as ‘c_number_minutes_in_day’ or ‘c_duration_1440’ are inappropriate.

7.7.4 Attributes

A number type variable will explicitly declare scales and precision. A character variable will explicitly declare a size.

When declaring variables that correspond to a column in a table, the developer will use %TYPE to minimise maintenance.

7.7.5 Cursor Declaration

Declare explicit cursors for all SQL statements that return more than one row or no rows at all.

Begin each cursor name with “cur_”, followed by the short name of the most important table in the SELECT statement. If a table short name is used more than once in a PL/SQL code segment, add a number as a suffix to make the cursor name unique.

Specify a full column list in each SELECT statement you declare. As an exception, you are only allowed to use an asterisk when more than 75% of all columns of a table have been selected.

If the cursor is not to receive parameters, then the following format will be adopted:

```
CURSOR {cursor_name} IS
    select ...
```

The SELECT statement will appear on the following line starting in the fifth column.

If the SELECT statement requires bind variables, declare these bind variables as cursor parameters.

The parameters will be laid out with each parameter aligned on the fifth column on separate lines.

Example:

```
CURSOR {cursor_name}
   ({parameter1} NUMBER,
    {parameter2} NUMBER
    {parameter3} DATE) IS
    select ...
```

The name of a cursor parameter should start with “b_”.

Use the %TYPE attribute referencing a column in the table(s) used in the SELECT statement to declare the datatype for a cursor parameter.

7.7.6 Records Declaration

The name of a record variable should be prefixed with “rec_”.

Use the %ROWTYPE attribute to declare a record structure for each explicitly defined cursor. Do not use the implicit record declaration mechanism. The only exception to this rule is when you use a cursor FOR loop.

The name of a table variable should be prefixed with “tab_”. The corresponding index should be prefixed with “i_”.

7.7.7 Exceptions Declaration

Don’t declare any exception handler with the same name as a standard Oracle exception handler.

The name of an exception should start with “e_” and must be a meaningful name (never the same as a standard Oracle exception).

Place all user-defined exceptions together in the declarative part of the PL/SQL block. First specify all user-defined Oracle exceptions. Then specify all non-Oracle-related exceptions.

Explicitly declare exceptions for each of the Oracle error conditions you can expect to occur in normal program execution. In the comment you specify the text of the Oracle error message.

Example:

```
declare
...
e_snapshot_too_old exception;
pragma exception_init(e_snapshot_too_old,-1555);
/************************************************************
ORA-01555: Snapshot too old (Rollback segment too small)
************************************************************/
...
begin
...
exception
when e_snapshot_too_old then
...
end;
```

Define exceptions and exception handlers for functional errors or warning conditions. In the comment you must specify the condition(s) that raise the exception.

Example:

```
declare
e_out_of_seats exception;
/************************************************************
No more seats available to re-assign passengers for this flight
************************************************************/
...
begin
...
raise e_out_of_seats;
...
exception
when e_out_of_seats then
...
end;
```

Customize exception handlers that use raise_application_error so that only error number -20000 is used. Customize the message text explicitly to pass module-relevant messages.

7.8 Begin Statement

7.8.1 Statements

There will be only one executable statement per line.

There will be consistent indenting level throughout the entire PL/SQL block. Each level of indentation will consist of four columns.

There will be blank lines breaking up code into more readable blocks.

There will be at least one space between the assignment operator and a variable, or parenthesis.

Example:

```
{variable1} := function({variable2});
```

There will be at least one space between an operator and a variable or parenthesis.

Example:

```
{variable1} := {variable2} * ({variable3} + {variable4});
```

There will be a semicolon at the end of each statement with no intervening space between the statement and the terminator(;).

There will be a variable initialised, before it is used in a PL/SQL statement.

There will be multiple rows of assignments aligned to variables such that all declarations are aligned to the declaration with the maximum indentation.

Example:

```
{variable1}     := {variable2} + {variable3};
{variable99999} := {variable2} * {variable3};
```

7.8.2 Assignments that go over a line

Any assignment, too long to be written on one line will be broken at an assignment operator(:=), an operator (+,*,etc.), or a qualifying period(.). The first non-blank character on the new line will be equals sign, operator, or qualifying period.

The assignment operator will be indented at the beginning of the statement. All lines commencing with an assignment operator will be placed consistently within a group of statements with emphasis on readability.

Example:

```
{extremely_long_variable_name}
     := {variable1};
```

The operator will be aligned beneath the assignment operator or the first operator in the expression. It is neater to vertically align all operators if there are more than one in a statement. Hence all sub-expressions will appear on a new line as shown below:

```
{extremely_long_variable_name}
     := {variable1}
     +  {variable2}
     +  {variable3};

{shorter_variable_name} := {variable1}
                        +  {variable2}
                        +  {variable3};
```

Parenthesis will be aligned or indented according to its hierarchy in the statement. This aids readability of complex computational statements.

Example 1:

```
{variable1} := {variable2}
            + ({variable3}
            *  {variable4}
              );
```

Example 2:

```
{variable1} := {variable1}
            + ({variable2}
            *  {variable3}
              )
            +  {variable4}
            / ({variable5}
            -   ({variable6}
            /    {variable7}
                )
              );
```

7.8.3 Arguments

If all arguments can not fit in one line, they will be laid out such that each argument will be aligned on the same column on separate lines as shown below:

```
{procedure_name} ({parameter1},
                  {parameter2},
                  {parameter3});

{variable1} := substr ({parameter1},
                       {parameter2},
                       {parameter3});
```

7.8.4 Block Structure Statements

The following sub-sections are examples of block structure statements:

7.8.4.1 IF THEN ELSIF ELSE END IF Construct

An expression will have only one coded condition to each line. Conditions that are too long to fit in one line will be broken at the operator or qualifying period('.');

The THEN key word will be located immediately after its related condition. ELSIF, ELSE, and END IF key words will be aligned under the IF to which they belong. Both conditions and statements will be indented and aligned as shown below:

```
if    {condition1} then
      {statements};
elsif {condition2} then
      {statements};
else
      {statements};
end if;
```

Conditions will be enclosed in parentheses only when the following statements are true:

- When they are required to force the sequence of testing
- When they will make the order of testing clearer in the case of compound conditions
- When there are mixed Boolean operators (AND, OR, NOT)

Parentheses will be aligned or indented according to their hierarchy in the statement. This aids readability of complex computational statements.

Example:

```
if    ({condition1}
and    {condition2})
```

```
or    {condition3}  then
      {statements};
end if;
```

The developer will write the Boolean operators in full.

Example:

Operator	Correct	Incorrect
AND	AND	&
OR	OR	|

IF statements should not exceed three levels of nesting. The most likely event will be coded first for efficiency reasons. Nested statements will be indented and aligned to indicate the level of nesting.

Example:

```
if     {condition1} then
       {statement1};
       if     {condition2} then
              {statement2};
       else
              {statement3};
       end if;
elsif {condition3} then
       {statement4}
else
       {statement5};
end if;
```

A variable will not be tested repeatedly using nested IF statements. It will be coded using ELSIF.

Example:

```
if     {variable1} = {constant1} then
       {statement1};
elsif {variable1} = {constant2} then
       {statement2};
elsif {variable1} = {constant3} then
       {statement3};
else
       {statement4};
end if;
```

7.8.4.2 WHILE LOOP Construct

The WHILE LOOP statement will be used for conditional loops. WHILE LOOPS save testing the same condition within the loop when reading from a table.

Each condition will be coded on a new line. Conditions that are too long to fit in one line will be broken at the operator or qualifying period.

The END LOOP label will be aligned under the WHILE to which it belongs.

Example:

```
while {condition1}
or    {condition2} loop
      {statements};
end loop;
```

7.8.4.3 LOOP Construct

Do not use the LOOP statement. Instead, use a WHILE or a FOR loop.

7.8.5 Procedure Calls

Parameters in a procedure call will be either:

- laid out in one line OR,
- laid out with each parameter aligned on the same column on separate lines, as shown below.

```
edit_field (x_coordinate,
            y_coordinate,
            width,
            height );
```

7.9 Exception Statement

All application code required exception handling. It's not allowed to write a code without exception handling unless it is strongly necessary.

Exception handling will be located after the last executable statement in the procedure/trigger.

The EXCEPTION label will be on a line by itself aligned with the BEGIN statement.

Each condition will be coded on a new line and will be indented four columns from the exception label as follows:

```
EXCEPTION
    when {condition1} then
         {statements};
    when {condition2} then
```

```
        {statements};
```

7.10 End Statement

The END statement will be on a line by itself. It will align to the BEGIN statement of the block to which it belongs. It will end with a semicolon with no intervening space between END and the semicolon as shown below:

```
END;
```

7.11 List of all reserved words

Reserved words which will be in uppercase:

- BEGIN
- BINARY_INTEGER
- BODY
- BOOLEAN
- CHAR
- DATE
- DECLARE
- END
- EXCEPTION
- FUNCTION
- IS
- NATURAL
- NUMBER
- PACKAGE
- PLS_INTEGER
- PROCEDURE
- REAL
- VARCHAR*
- VARCHAR2*

7.12 CVS Directory Structure and Naming Conventions

The directory structure of the CVS repository and naming conventions that should be used for each project is described in this chapter.

For details regarding CVS please refer to "PGL, Source Code Handling and Release Management (Using CVS - Concurrent Versioning System), Global SOP" document.

The directory structure must be clear, consistent and contain separate subdirectories needed for storing in order all the files required to build, compile and unit test the entire application. The directory structure meant to help with keeping order within the different application files in the CVS repository is shown in Figure 4.

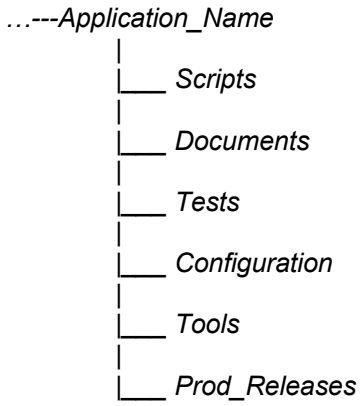


Figure 4: "The possible inner structure of "Application_Name" folder for PL/SQL scripts"

"Application_Name" it is the folder containing all the source, documents, tests, configuration, tools, scripts, libraries, binaries and other files belonging to or related to the SW application. The naming conventions for this folder are the following:

APP_FullName

Where:

- APP : Abbreviation for SW Application.
- FullName : Full name of the application.
- If abbreviation for SW application together with the FullName of the application exceeds 44 characters then we are leaving only the abbreviation.

e.g. PSS_Patient_Summary_System

The purpose of the sub-folders within the *"Application_Name"* folder is the following:

- *"Scripts"* – This is the folder where all the developed PL/SQL scripts should be stored.
- *"Documents"* – In this folder shall be stored all the files documenting the developed application. If all the files documenting the application are already in PLv2, listed in README file within the *"Application_Name"* folder and no other documents exist that should be stored here this folder is not necessary.
- *"Tests"* – In this folder all shall be stored all the source code, scripts, data files and other supporting documents related to Unit Test prepared for the developed application.
- *"Configuration"* – This is the folder where all configuration data/scripts/documents should be stored.
- *"Tools"* – This is the folder where all supporting tools used during the scripts development should be stored. In case of the self-developed tools the scripts and other related files should be located here. In case when developer have used external tools the installation files, other binaries and source files (if available) should be stored in this directory.
- *"Prod_Releases"* – This is the folder where the binary and preferably installation versions of the official production releases of the application should be stored. All the production binaries for production release must be packed into one zip archive. The naming conventions for this file are the following:
  APP_X_Y_Z_A.zip

Where:

- APP : Abbreviation for SW Application.
- X: Version number Major.
- Y: Version number Minor.
- Z: Release number.
- A: Optional (e.g. Built number).

e.g. PSS_3_1_5_23.zip

Folders *"Source"*, *"Tests", "Binaries"* and *"Tools"* are mandatory for each application. Other folders should be created depending on the developer needs.

Note: Every *"Application_Name"* directory should contain a README file that covers:

- short description of the Application,
- system versions, e.g. OC version, Windows version, where the SW application will work,
- build and install directions,
- direct people to all the related resources:
  - directories of source,
  - online documentation,
  - paper documentation,
  - design documentation,
- anything else that might help someone.

It is preferred that a README file will be in text format but it can also be MS Word or PDF file.

It is important to update the README file with every production release.

Every software release must be properly tagged in CVS according to guidelines given in the "PGL, Release, Tag, and Branch Management, Global Instruction" document.

8 .Net and C# language

8.1 .NET Naming Convention

8.1.1 Naming Style and Capitalization

There are two major recommended naming styles for .NET: Pascal case and Camel case. Another convention is Uppercase. Previously Hungarian notation was very popular in the past but now is not recommended.

8.1.1.1 Pascal case

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters.

Example:

BackColor

8.1.1.2 Camel case

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized.

Example:

backColor

8.1.1.3 Uppercase

All letters in the identifier are capitalized. General .NET naming convention recommends using this convention only for identifiers that consist of two or fewer letters.

Example:

System.IO
System.Web.UI

8.1.1.4 Hungarian notation

This notation uses abbreviation of the type name, variable type or containing module name. Naming recommendation for .NET state that „name should describe variable or parameter meaning, not its type or usage”.

Example (identifiers that do not follow .NET naming conventions):

m_name
mbDone
frmMain

WSAConnect

8.1.2 C# Naming Convention

The following table summarizes the capitalization rules and provides examples for the different types of identifiers.

Identifier	Case	Example
Class	Pascal	**AppDomain**
Enum type	Pascal	**ErrorLevel**
Enum values	Pascal	**FatalError**
Event	Pascal	**ValueChange**
Exception class	Pascal	**WebException** **Note** : Always ends with the suffix **Exception**.
Attribute class	Pascal	**SerializableAttribute** **Note :** Always ends with the suffix **Attribute**
Read-only Static field	Pascal	**DateOffset**
Interface	Pascal	**IDisposable** **Note:** Always begins with the prefix I.
Method	Pascal	**ToString**
Private Method	Camel	**calculateFormula** **Note**: Microsoft recommends using Pascal case for naming all methods but using Camel case seems a better choice for private instance methods.
Namespace	Pascal	**System.Drawing**
Parameter	Camel	**typeName**
Property	Pascal	**BackColor**
Private instance field	Camel or Camel prefixed with _	**password** **_password** **Note:** Prefixing the name of a private instance field is recommended because it helps to find fields while using IntelliSense feature of Visual Studio.
Protected instance field	Camel	**redValue** **Note**: A pair of a protected property and a private field is preferable to using a protected instance field.
Public instance field	Pascal	**RedValue** **Note**: A property is preferable to using a public instance field.

8.1.3 Case Sensitivity

To avoid confusion and guarantee cross-language interoperation, follow these rules regarding the use of case sensitivity:

- Do not use names that require case sensitivity. Classes must be fully usable from both case-sensitive and case-insensitive languages. Case-insensitive languages cannot distinguish

between two names within the same context that differ only by case. Therefore, you must avoid this situation in the components or classes that you create.

- Do not create two namespaces with names that differ only by case. For example, a case insensitive language cannot distinguish between the following two namespace declarations:

  namespace ee.cummings;
  namespace Ee.Cummings;

- Do not create a function with parameter names that differ only by case. The following example is incorrect:

  void MyFunction(string a, string A)

- Do not create a namespace with type names that differ only by case. In the following example, *Point p* and *POINT p* are inappropriate type names because they differ only by case:

  System.Windows.Forms.Point p
  System.Windows.Forms.POINT p

- Do not create a type with property names that differ only by case. In the following example, int Color and int COLOR are inappropriate property names because they differ only by case:

  int Color {*get, set*}
  int COLOR {*get, set*}

- Do not create a type with method names that differ only by case. In the following example, *calculate* and *Calculate* are inappropriate method names because they differ only by case:

  void calculate()
  void Calculate()

8.1.4 Abbreviations

To avoid confusion and guarantee cross-language interoperation, follow these rules regarding the use of abbreviations:

- Do not use abbreviations or contractions as parts of identifier names. For example, use *GetWindow* instead of *GetWin*.
- Do not use acronyms that are not generally accepted in the computing field.
- Where appropriate, use well-known acronyms to replace lengthy phrase names. For example, use *UI* for User Interface and *OLAP* for On-line Analytical Processing.
- When using acronyms, use Pascal case or camel case for acronyms more than two characters long. For example, use *HtmlButton* or *htmlButton*. However, you should capitalize acronyms that consist of only two characters, such as *System.IO* instead of *System.Io*.
- Do not use abbreviations in identifiers or parameter names. If you must use abbreviations, use camel case for abbreviations that consist of more than two characters, even if this contradicts the standard abbreviation of the word.

8.1.5 Namespace Naming Guidelines

Namespace names should describe the company, department that implemented the software, project name and project component (.dll). Creating a namespace hierarchy within a component (.dll, .exe) is also recommended. Namespace parts should use Pascal case. Using two letter uppercase abbreviations is also recommended like *System.IO*.

Example:

Roche.Pgdw.WebReports.ReportMVC[.Controllers]

Where:
Roche – is a company name,
Pgdw – department name,
WebReport – project name,
ReportMVC – component,
Controllers – optional name of a sub namespace within the component.

8.1.6 Class Naming Guidelines

The following rules outline the guidelines for naming classes:

- Use a noun or noun phrase to name a class.
- Use Pascal case.
- Use abbreviations sparingly.
- Do not use a type prefix, such as C for class, on a class name. For example, use the class name *FileStream* rather than *CFileStream*.
- Do not use the underscore character (_).
- Occasionally, it is necessary to provide a class name that begins with the letter I, even though the class is not an interface. This is appropriate as long as I is the first letter of an entire word that is a part of the class name. For example, the class name *IdentityStore* is appropriate.
- Where appropriate, use a compound word to name a derived class. The second part of the derived class's name should be the name of the base class. For example, *ApplicationException* is an appropriate name for a class derived from a class named *Exception*, because *ApplicationException* is a kind of *Exception*. Use reasonable judgment in applying this rule. For example, *Button* is an appropriate name for a class derived from *Control*. Although a button is a kind of control, making *Control* a part of the class name would lengthen the name unnecessarily.
- Use a *Base* prefix for base class names – mostly when this is a base layer type.

The following are examples of correctly named classes:

*public class **FileStream***
*public class **Button***
*public class **String***
*public class **BaseOracleDAL***

8.1.7 Interface Naming Guidelines

The following rules outline the naming guidelines for interfaces: name interfaces with nouns or noun phrases, or adjectives that describe behavior. For example, the interface name *IComponent* uses a descriptive noun. The interface name *ICustomAttributeProvider* uses a noun phrase. The name *IPersistable* uses an adjective.

- Use Pascal case.
- Use abbreviations sparingly.
- Prefix interface names with the letter I, to indicate that the type is an interface.
- Use similar names when you define a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the letter I prefix on the interface name.
- Do not use the underscore character (_).

The following are examples of correctly named interfaces:

```
public interface IServiceProvider
public interface IFormatable
```

The following code example illustrates how to define the interface IComponent and its standard implementation, the class Component:

```
public interface IComponent
{
  // Implementation code goes here.
}
public class Component: IComponent
{
  // Implementation code goes here.
}
```

8.1.8 Attribute Naming Guidelines

You should always add the suffix Attribute to custom attribute classes. The following is an example of a correctly named attribute class.

```
public class ObsoleteAttribute
{
}
```

8.1.9 Enumeration Type Naming Guidelines

The enumeration (**Enum**) value type inherits from the Enum Class. The following rules outline the naming guidelines for enumerations:

- Use Pascal case for **Enum** types and value names.
- Use abbreviations sparingly.
- Avoid using an Enum suffix on **Enum** type names.
- Use a singular name for most **Enum** types, but use a plural name for **Enum** types that are bit fields.

- Always add the **FlagsAttribute** to a bit field **Enum** type.

8.1.10 Static Fields Naming Guidelines

8.1.10.1 Static Read-Write Fields and Singleton Accessors

The following rules outline the naming guidelines for static read-write fields and singleton accessors:

- Use nouns, noun phrases, or abbreviations of nouns to name static fields.
- Use Pascal case.
- Do not use a Hungarian notation prefix on static field names.
- It is recommended that you use static properties instead of public static fields whenever possible.

Example 1:

```
// Singleton class and using a static property to access a static field
public class MySingleton
{
  private static readonly MySingleton instance;

  public static MySingleton
  {
    get
    {
      return instance;
    }
  }
}
```

Example 2:

```
public static string ConfigFileName;
```

8.1.10.2 Constants

Constant values are similar to static fields but they are read-only. C# uses also „static readonly" fields when a constant value is not known at compilation time and must be assigned when a class is loaded by class loader Use Pascal case for constants. When declaring constants create separate classes that are sealed (not inheritable), have a private constructor (not instanciable) and only have public const fields.

Example:

```
public sealed class HttpParameters
{
  private HttpParameters()
  {
  }

  public const string ParamName = „name";
}
```

8.1.11 Parameter Naming Guidelines

It is important to carefully follow these parameter naming guidelines because visual design tools that provide context sensitive help and class browsing functionality display method parameter names to users in the designer.

The following rules outline the naming guidelines for parameters:

- Use camel case for parameter names.
- Use descriptive parameter names. Parameter names should be descriptive enough that the name of the parameter and its type can be used to determine its meaning in most scenarios. For example, visual design tools that provide context sensitive help display method parameters to the developer as they type. The parameter names should be descriptive enough in this scenario to allow the developer to supply the correct parameters.
- Use names that describe a parameter's meaning rather than names that describe a parameter's type. Development tools should provide meaningful information about a parameter's type. Therefore, a parameter's name can be put to better use by describing meaning. Use type-based parameter names sparingly and only where it is appropriate.
- Do not use reserved parameters. Reserved parameters are private parameters that might be exposed in a future version if they are needed. Instead, if more data is needed in a future version of your class library, add a new overload for a method.
- Do not prefix parameter names with Hungarian type notation.

The following are examples of correctly named parameters:

*Type GetType(string **typeName**)*
*string Format(string **format**, object[] **args**)*

8.1.12 Method Naming Guidelines

The following rules outline the naming guidelines for methods:

- Use verbs or verb phrases to name methods.
- Use Pascal case for public, protected and internal (assembly visible) methods.
- User Camel case for private methods.

The following are examples of correctly named methods:

public void RemoveAll()
public char[] GetCharArray()
public void Invoke()
private void loadData()

8.1.13 Property Naming Guidelines

The following rules outline the naming guidelines for properties:

- Use a noun or noun phrase to name properties.
- Use Pascal case.
- Do not use Hungarian notation.

Consider creating a property with the same name as its underlying type. For example, if you declare a property named Color, the type of the property should likewise be Color. See the example later in this topic.

The following code example illustrates correct property naming:

```
public class SampleClass
{
  public Color BackColor
  {
    // Code for Get and Set accessors goes here.
  }
}
```

The following code example illustrates providing a property with the same name as a type:

```
public enum Color
{
  // Insert code for Enum here.
}

public class Control
{
  public Color Color
  {
    get {// Insert code here.}
    set {// Insert code here.}
  }
}
```

The following code example is incorrect because the property Color is of type Integer:

```
public enum Color {// Insert code for Enum here.}
public class Control
{
  public int Color
  {
    get {// Insert code here.}
    set {// Insert code here.}
  }
}
```

In the incorrect example, it is not possible to refer to the members of the Color enumeration. Color.Xxx will be interpreted as accessing a member that first gets the value of the **Color** property (type **Integer** in Visual Basic or type **int** in C#) and then accesses a member of that value (which would have to be an instance member of **System.Int32**).

8.1.14 Event Naming Guidelines

The following rules outline the naming guidelines for events:

- Use Pascal case.
- Do not use Hungarian notation.
- Use an EventHandler suffix on event handler names (delegate types).
- Specify two parameters named sender and e. The sender parameter represents the object that raised the event. The sender parameter is always of type object, even if it is possible to use a more specific type. The state associated with the event is encapsulated in an instance of an event class named e. Use an appropriate and specific event class for the e parameter type.
- Name an event argument class with the EventArgs suffix.
- Consider naming events with a verb. For example, correctly named event names include Clicked, Painting, and DroppedDown.
- Use a gerund (the "ing" form of a verb) to create an event name that expresses the concept of pre-event, and a past-tense verb to represent post-event. For example, a Close event that can be canceled should have a Closing event and a Closed event. Do not use the BeforeXxx/AfterXxx naming pattern.
- Do not use a prefix or suffix on the event declaration on the type. For example, use Close instead of OnClose.
- In general, you should provide a protected method called OnXxx on types with events that can be overridden in a derived class. This method should only have the event parameter e, because the sender is always the instance of the type.

The following example illustrates an event handler with an appropriate name and parameters:

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

The following example illustrates a correctly named event argument class:

```
public class MouseEventArgs : EventArgs
{
   int x;
   int y;
   public MouseEventArgs(int x, int y)
      { this.x = x; this.y = y; }
   public int X { get { return x; } }
   public int Y { get { return y; } }
}
```

8.2 Class Member Usage Guidelines

8.2.1 Property Usage Guidelines

Determine whether a property or a method is more appropriate for your needs. For details on choosing between properties and methods, see Properties vs. Methods.
Choose a name for your property based on the recommended Property Naming Guidelines.
When accessing a property using the set accessor, preserve the value of the property before you change it. This will ensure that data is not lost if the set accessor throws an exception.

8.2.1.1 Property State Issues

Allow properties to be set in any order. Properties should be stateless with respect to other properties. It is often the case that a particular feature of an object will not take effect until the developer specifies a particular set of properties, or until an object has a particular state. Until the object is in the correct state, the feature is not active. When the object is in the correct state, the feature automatically activates itself without requiring an explicit call. The semantics are the same regardless of the order in which the developer sets the property values or how the developer gets the object into the active state.
For example, a TextBox control might have two related properties: DataSource and DataField. DataSource specifies the table name, and DataField specifies the column name. Once both properties are specified, the control can automatically bind data from the table into the Text property of the control. The following code example illustrates properties that can be set in any order.

```
TextBox t = new TextBox();
t.DataSource = "Publishers";
t.DataField = "AuthorID";
// The data-binding feature is now active.
```

You can set the DataSource and DataField properties in any order. Therefore, the preceding code is equivalent to the following:

```
TextBox t = new TextBox();
t.DataField = "AuthorID";
t.DataSource = "Publishers";
// The data-binding feature is now active.
```

You can also set a property to null (Nothing in Visual Basic) to indicate that the value is not specified.

Example:

```
TextBox t = new TextBox();
t.DataField = "AuthorID";
t.DataSource = "Publishers";
// The data-binding feature is now active.
t.DataSource = null;
// The data-binding feature is now inactive.
```

The following code example illustrates how to track the state of the data binding feature and automatically activate or deactivate it at the appropriate times:

```
public class TextBox
{
  string dataSource;
  string dataField;
  bool active;

  public string DataSource
  {
    get
```

```
    {
      return dataSource;
    }
    set
    {
      if (value != dataSource)
      {
        // Update active state.
        SetActive(value != null && dataField != null);
        dataSource = value;
      }
    }
    }

  public string DataField
  {
    get
    {
      return dataField;
    }
    set
    {
      if (value != dataField)
      {
        // Update active state.
        SetActive(dataSource != null && dataField != null);
        dataField = value;
      }
    }
  }
  void SetActive(Boolean value)
  {
    if (value != active)
    {
      if (value)
      {
        Activate();
        Text = dataBase.Value(dataField);
      }
      else
      {
        Deactivate();
        Text = "";
      }
      // Set active only if successful.
      active = value;
    }
  }
  void Activate()
  {
```

```
    // Open database.
  }

  void Deactivate()
  {
    // Close database.
  }
}
```

In the preceding example, the following expression determines whether the object is in a state in which the data-binding feature can activate itself.
value != null && *dataField != null*

You make activation automatic by creating a method that determines whether the object can be activated given its current state, and then activates it as necessary.

Example:

```
void UpdateActive(string dataSource, string dataField)
{
  SetActive(dataSource != null && dataField != null);
}
```

If you do have related properties, such as DataSource and DataMember, you should consider implementing the ISupportInitialize Interface. This will allow the designer (or user) to call the ISupportInitialize.BeginInit and ISupportInitialize.EndInit methods when setting multiple properties to allow the component to provide optimizations. In the above example, ISupportInitialize could prevent unnecessary attempts to access the database until setup is correctly completed.
The expression that appears in this method indicates the parts of the object model that need to be examined in order to enforce these state transitions. In this case, the DataSource and DataField properties are affected. For more information on choosing between properties and methods, see Properties vs. Methods.

8.2.1.2 Raising Property-Changed Events

Components should raise property-changed events if they want to notify consumers when the component's property changes programmatically. This is important for also for classes that could be used as a source for data binding. The naming convention for a property-changed event is to add the *Changed* suffix to the property name, such as *TextChanged*. For example, a control might raise a *TextChanged* event when its text property changes. You can use a protected helper routine *Raise<Property>Changed*, to raise this event. However, it is probably not worth the overhead to raise a property-changed event for a hash table item addition. The setter should check if the new value differs from the original one and only then raise the event, this prevents circularity. The following code example illustrates the implementation of a helper routine on a property-changed event:

```
class Control : Component
{
  string text;
  public string Text
  {
```

```
    get
    {
      return text;
    }
    set
    {
      if (!text.Equals(value))
      {
        text = value;
        RaiseTextChanged();
      }
    }
  }
}
```

Data binding uses this pattern to allow two-way binding of the property. Without *<Property>Changed* and *Raise<Property>Changed* events, data binding works in one direction; if the database changes, the property is updated. Each property that raises the *<Property>Changed* event should provide metadata to indicate that the property supports data binding.

It is recommended that you raise changing/changed events if the value of a property changes as a result of external forces. These events indicate to the developer that the value of a property is changing or has changed as a result of an operation, rather than by calling methods on the object.
A good example is the Text property of an Edit control. As a user types information into the control, the property value automatically changes. An event is raised before the value of the property has changed. It does not pass the old or new value, and the developer can cancel the event by throwing an exception. The name of the event is the name of the property followed by the suffix Changing. The following code example illustrates a changing event:

```
class Edit : Control
{
  public string Text
  {
    get
    {
      return text;
    }
    set
    {
      if (text != value)
      {
        OnTextChanging(Event.Empty);
        text = value;
      }
    }
  }
}
```

An event is also raised after the value of the property has changed. This event cannot be canceled. The name of the event is the name of the property followed by the suffix Changed. The generic

PropertyChanged event should also be raised. The pattern for raising both of these events is to raise the specific event from the OnPropertyChanged method. The following example illustrates the use of the OnPropertyChanged method:

```
class Edit : Control
{
  public string Text
  {
    get
    {
      return text;
    }
    set
    {
      if (text != value)
      {
        OnTextChanging(Event.Empty);
        text = value;
        RaisePropertyChangedEvent(Edit.ClassInfo.text);
      }
    }
  }

  protected void OnPropertyChanged(PropertyChangedEventArgs e)
  {
    if (e.PropertyChanged.Equals(Edit.ClassInfo.text))
      OnTextChanged(Event.Empty);
    if (onPropertyChangedHandler != null)
      onPropertyChangedHandler(this, e);
  }
}
```

There are cases when the underlying value of a property is not stored as a field, making it difficult to track changes to the value. When raising the changing event, find all the places that the property value can change and provide the ability to cancel the event. For example, the previous Edit control example is not entirely accurate because the Text value is actually stored in the window handle (HWND). In order to raise the TextChanging event, you must examine Windows messages to determine when the text might change, and allow for an exception thrown in OnTextChanging to cancel the event. If it is too difficult to provide a changing event, it is reasonable to support only the changed event.

8.2.1.3 Properties vs. Methods

Class library designers often must decide between implementing a class member as a property or a method. In general, methods represent actions and properties represent data. Use the following guidelines to help you choose between these options.

Use a property when the member is a logical data member. In the following member declarations, Name is a property because it is a logical member of the class.

Example:

```
public string Name
  get
  {
    return name;
  }
  set
  {
    name = value;
  }
```

Use a method when:

- The operation is a conversion, such as Object.ToString.
- The operation is expensive enough that you want to communicate to the user that they should consider caching the result.
- Obtaining a property value using the get accessor would have an observable side effect.
- Calling the member twice in succession produces different results.
- The order of execution is important. Note that a type's properties should be able to be set and retrieved in any order.
- The member is static but returns a value that can be changed.
- The member returns an array. Properties that return arrays can be very misleading. Usually it is necessary to return a copy of the internal array so that the user cannot change internal state. This, coupled with the fact that a user can easily assume it is an indexed property, leads to inefficient code. In the following code example, each call to the Methods property creates a copy of the array. As a result, 2n+1 copies of the array will be created in the following loop.

```
Type type = // Get a type.
for (int i = 0; i < type.Methods.Length; i++)
{
  if (type.Methods[i].Name.Equals ("text"))
  {
    // Perform some operation.
  }
}
```

The following example illustrates the correct use of properties and methods:

```
class Connection
{
  // The following three members should be properties
  // because they can be set in any order.
  string DNSName {get{};set{};}
  string UserName {get{};set{};}
  string Password {get{};set{};}

  // The following member should be a method
  // because the order of execution is important.
```

```
   // This method cannot be executed until after the
   // properties have been set.
   bool Execute ();
}
```

8.2.1.4 Read-Only and Write-Only Properties

You should use a read-only property when the user cannot change the property's logical data member. Do not use write-only properties.

8.2.1.5 Indexed Property Usage

Note: An indexed property can also be referred to as an indexer.

The following rules outline guidelines for using indexed properties:

- Use an indexed property when the property's logical data member is an array.
- Consider using only integral values or strings for an indexed property. If the design requires other types for the indexed property, reconsider whether it represents a logical data member. If not, use a method.
- Consider using only one index. If the design requires multiple indexes, reconsider whether it represents a logical data member. If not, use a method.
- Use only one indexed property per class, and make it the default indexed property for that class. This rule is enforced by indexer support in the C# programming language.
- Do not use nondefault indexed properties. C# does not allow this.
- Name an indexed property Item. For example, see the DataGrid.Item Property. Follow this rule, unless there is a name that is more obvious to users, such as the Chars property on the String class. In C#, indexers are always named Item.
- Do not provide an indexed property and a method that are semantically equivalent to two or more overloaded methods. In the following code example, the Method property should be changed to GetMethod(string) method. Note that this not allowed in C#.

Examples:

```
// Change the MethodInfo Type.Method property to a method.
MethodInfo Type.Method[string name]
MethodInfo Type.GetMethod (string name, Boolean ignoreCase)

// The MethodInfo Type.Method property is changed to
// the MethodInfo Type.GetMethod method.
MethodInfo Type.GetMethod(string name)
MethodInfo Type.GetMethod (string name, Boolean ignoreCase)
```

8.2.2 Event Usage Guidelines

The following rules outline the usage guidelines for events:

- Choose a name for your event based on the recommended Event Naming Guidelines.
- When you refer to events in documentation, use the phrase, "an event was raised" instead of "an event was fired" or "an event was triggered."
- In languages that support the void keyword, use a return type of void for event handlers, as shown in the following C# code example:

public delegate void MouseEventHandler(object sender, MouseEventArgs e);

- Use strongly typed event data classes when an event conveys meaningful data, such as the coordinates of a mouse click.
- Event classes should extend the *System.EventArgs* Class, as shown in the following example:

  public class MouseEvent: EventArgs {}

- Use a protected (Protected in Visual Basic) virtual method to raise each event. This technique is not appropriate for sealed classes, because classes cannot be derived from them. The purpose of the method is to provide a way for a derived class to handle the event using an override. This is more natural than using delegates in situations where the developer is creating a derived class. The name of the method takes the form *OnEventName*, where *EventName* is the name of the event being raised.

  Example:

```
public class Button
{
  ButtonClickHandler onClickHandler;

  protected virtual void OnClick(ClickEventArgs e)
  {
    // Call the delegate if non-null.
    if (onClickHandler != null)
        onClickHandler(this, e);
  }
}
```

- The derived class can choose not to call the base class during the processing of OnEventName. Be prepared for this by not including any processing in the OnEventName method that is required for the base class to work correctly.
- You should assume that an event handler could contain any code. Classes should be ready for the event handler to perform almost any operation, and in all cases the object should be left in an appropriate state after the event has been raised. Consider using a try/finally block at the point in code where the event is raised. Since the developer can perform a callback function on the object to perform other actions, do not assume anything about the object state when control returns to the point at which the event was raised.

  Example:

```
public class Button
{
  ButtonClickHandler onClickHandler;

  protected void DoClick()
  {
    // Paint button in indented state.
```

```
        PaintDown();
          try
        {
          // Call event handler.
          OnClick();
        }
        finally
        {
          // Window might be deleted in event handler.
          if (windowHandle != null)
            // Paint button in normal state.
            PaintUp();
        }
      }

      protected virtual void OnClick(ClickEvent e)
      {
        if (onClickHandler != null)
          onClickHandler(this, e);
      }
    }
```

- Use or extend the *System.ComponentModel.CancelEventArgs* Class to allow the developer to control the events of an object. For example, the *TreeView* control raises a *BeforeLabelEdit* when the user is about to edit a node label. The following code example illustrates how a developer can use this event to prevent a node from being edited:

```
public class Form1: Form
{
  TreeView treeView1 = new TreeView();

  void treeView1_BeforeLabelEdit(object source,
    NodeLabelEditEventArgs e)
  {
    e.CancelEdit = true;
  }
}
```

Note: In this case, no error is generated to the user. The label is read-only.

Cancel events are not appropriate in cases where the developer would cancel the operation and return an exception. In these cases, you should raise an exception inside of the event handler in order to cancel. For example, the user might want to write validation logic in an edit control as shown:

```
public class Form1: Form
{
  EditBox edit1 = new EditBox();
```

```
    void TextChanging(object source, EventArgs e)
    {
      throw new InvalidOperationException("Invalid edit");
    }
  }
```

8.2.3 Method Usage Guidelines

The following rules outline the usage guidelines for methods:

- Choose a name for your event based on the recommended Method Naming Guidelines.
- Do not use Hungarian notation.
- By default, methods are nonvirtual. Maintain this default in situations where it is not necessary to provide virtual methods. For more information about implementing inheritance, see Base Class Usage Guidelines.

8.2.3.1 Method Overloading Guidelines

Method overloading occurs when a class contains two methods with the same name, but different signatures. This section provides some guidelines for the use of overloaded methods:

- Use method overloading to provide different methods that do semantically the same thing.
- Use method overloading instead of allowing default arguments. Default arguments do not version well and therefore are not allowed in the Common Language Specification (CLS). The following code example illustrates an overloaded String.IndexOf method:

  ```
  int String.IndexOf (String name);
  int String.IndexOf (String name, int startIndex);
  ```

- Use default values correctly. In a family of overloaded methods, the complex method should use parameter names that indicate a change from the default state assumed in the simple method. For example, in the following code, the first method assumes the search will not be case-sensitive. The second method uses the name ignoreCase rather than caseSensitive to indicate how the default behavior is being changed:

  ```
  // Method #1: ignoreCase = false.
  MethodInfo Type.GetMethod(String name);
  // Method #2: Indicates how the default behavior of method #1 is being // changed.
   MethodInfo Type.GetMethod (String name, Boolean ignoreCase);
  ```

- Use a consistent ordering and naming pattern for method parameters. It is common to provide a set of overloaded methods with an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that you specify, the more detail the developer can specify. In the following code example, the overloaded Execute method has a consistent parameter order and naming pattern variation. Each of the Execute method variations uses the same semantics for the shared set of parameters.

  Example:

  ```
  public class SampleClass
  {
  ```

```
    readonly string defaultForA = "default value for a";
    readonly int defaultForB = "42";
    readonly double defaultForC = "68.90";

    public void Execute()
    {
      Execute(defaultForA, defaultForB, defaultForC);
    }

    public void Execute (string a)
    {
      Execute(a, defaultForB, defaultForC);
    }

    public void Execute (string a, int b)
    {
      Execute (a, b, defaultForC);
    }

    public void Execute (string a, int b, double c)
    {
      Console.WriteLine(a);
      Console.WriteLine(b);
      Console.WriteLine(c);
      Console.WriteLine();
    }
  }
```

Note: The only method in the group that should be virtual is the one that has the most parameters and only when you need extensibility.

- If you must provide the ability to override a method, make only the most complete overload virtual and define the other operations in terms of it. The following example illustrates this pattern:

```
public class SampleClass
{
  private string myString;

  public MyClass(string str)
  {
    this.myString = str;
  }

  public int IndexOf(string s)
  {
    return IndexOf (s, 0);
  }
```

```
    public int IndexOf(string s, int startIndex)
    {
      return IndexOf(s, startIndex, myString.Length - startIndex );
    }

    public virtual int IndexOf(string s, int startIndex, int count)
    {
      return myString.IndexOf(s, startIndex, count);
    }
  }
```

8.2.3.2 Methods With Variable Number of Arguments

You might want to expose a method that takes a variable number of arguments. A classic example is the printf method in the C programming language. For managed class libraries, use the *params* (*ParamArray* in Visual Basic) keyword for this construct. For example, use the following code instead of several overloaded methods:

void Format(string formatString, params object [] args)

You should not use the VarArgs or ellipsis (...) calling convention exclusively because the Common Language Specification does not support it.

For extremely performance-sensitive code, you might want to provide special code paths for a small number of elements. You should only do this if you are going to special case the entire code path (not just create an array and call the more general method). In such cases, the following pattern is recommended as a balance between performance and the cost of specially cased code.

void Format(string formatString, object arg1)

void Format(string formatString, object arg1, object arg2)

void Format(string formatString, params object [] args)

8.2.4 Constructor Usage Guidelines

The following rules outline the usage guidelines for constructors:

- Provide a default private constructor if there are only static methods and properties on a class. In the following example, the private constructor prevents the class from being created:

```
public sealed class Environment
{
  // Private constructor prevents the class from being created.
  private Environment()
  {
    // Code for the constructor goes here.
  }
}
```

- Minimize the amount of work done in the constructor. Constructors should not do more than capture the constructor parameter or parameters. This delays the cost of performing further operations until the user uses a specific feature of the instance.
- Never call virtual methods inside the constructor and any instance methods that call virtual methods. Virtual method table will not point to methods from a child class when the base class constructor is executing.
- Provide a constructor for every class. If a type is not meant to be created, use a private constructor. If you do not specify a constructor, many programming language (such as C#) implicitly add a default public constructor. If the class is abstract, it adds a protected constructor.
- Be aware that if you add a nondefault constructor to a class in a later version release, the implicit default constructor will be removed which can break client code. Therefore, the best practice is to always explicitly specify the constructor even if it is a public default constructor.
- Provide a *protected* (*Protected* in Visual Basic) constructor that can be used by types in a derived class.
- You should not provide constructor without parameters for a value type struct. Note that many compilers do not allow a struct to have a constructor without parameters. If you do not supply a constructor, the runtime initializes all the fields of the struct to zero. This makes array and static field creation faster.
- Use parameters in constructors as shortcuts for setting properties. There should be no difference in semantics between using an empty constructor followed by property set accessors, and using a constructor with multiple arguments. The following three code examples are equivalent:

  Example 1:

  Class SampleClass = new Class();
  SampleClass.A = "a";
  SampleClass.B = "b";

  Example 2:
  Class SampleClass = new Class("a");
  SampleClass.B = "b";

  Example 3:
  Class SampleClass = new Class ("a", "b");

- Use a consistent ordering and naming pattern for constructor parameters. A common pattern for constructor parameters is to provide an increasing number of parameters to allow the developer to specify a desired level of information. The more parameters that you specify, the more detail the developer can specify. In the following code example, there is a consistent order and naming of the parameters for all the *SampleClass* constructors:

  ```
  public class SampleClass
  {
    private const string defaultForA = "default value for a";
    private const string defaultForB = "default value for b";
    private const string defaultForC = "default value for c";
  ```

```
    public MyClass():this(defaultForA, defaultForB, defaultForC) {}
    public MyClass (string a) : this(a, defaultForB, defaultForC) {}
    public MyClass (string a, string b) : this(a, b, defaultForC) {}
    public MyClass (string a, string b, string c)
}
```

8.2.5 Field Usage Guidelines

The following rules outline the usage guidelines for fields:

- Do not use instance fields that are *public* or *protected* (*Public* or *Protected* in Visual Basic). If you avoid exposing fields directly to the developer, classes can be versioned more easily because a field cannot be changed to a property while maintaining binary compatibility. Consider providing **get** and **set** property accessors for fields instead of making them public. The presence of executable code in **get** and **set** property accessors allows later improvements, such as creation of an object on demand, upon usage of the property, or upon a property change notification. The following code example illustrates the correct use of private instance fields with **get** and **set** property accessors:

```
public struct Point
{
  private int xValue;
  private int yValue;

  public Point(int x, int y)
  {
    this.xValue = x;
    this.yValue = y;
  }

  public int X
  {
    get
    {
      return xValue;
    }
    set
    {
      xValue = value;
    }
  }
  public int Y
  {
    get
    {
      return yValue;
    }
    set
    {
      yValue = value;
```

```
    }
  }
}
```

- Expose a field to a derived class by using a **protected** property that returns the value of the field. This is illustrated in the following code example:

```
public class Control : Component
{
  private int handle;
  protected int Handle
  {
    get
    {
      return handle;
    }
  }
}
```

- Use the *const* (*Const* in Visual Basic) keyword to declare constant fields that will not change. Language compilers save the values of **const** fields directly in calling code.
- Use public static read-only fields for predefined object instances. If there are predefined instances of an object, declare them as public static read-only fields of the object itself. Use Pascal case because the fields are public. The following code example illustrates the correct use of public static read-only fields:

```
public struct Color
{
  public static readonly Color Red = new Color(0x0000FF);
  public static readonly Color Green = new Color(0x00FF00);
  public static readonly Color Blue = new Color(0xFF0000);
  public static readonly Color Black = new Color(0x000000);
  public static readonly Color White = new Color(0xFFFFFF);

  public Color(int rgb)
  { // Insert code here.}
  public Color(byte r, byte g, byte b)
  { // Insert code here.}

  public byte RedValue
  {
    get
    {
      return …
    }
  }
  public byte GreenValue
  {
    get
```

```
    {
      return .
    }
  }
  public byte BlueValue
  {
    get
    {
      return …
    }
  }
}
```

- Spell out all words used in a field name. Use abbreviations only if developers generally understand them. Do not use uppercase letters for field names. The following is an example of correctly named fields.

```
class SampleClass
{
  string url;
  string destinationUrl;
}
```

- Do not use Hungarian notation for field names. Good names describe semantics, not type.
- Do not apply a prefix to field names or static field names that differentiate between static and instance fields. For example, applying a g_ or s_ prefix is incorrect.
- You may apply an underscore prefix "_" to instance fields (not static). Instance fields whose names start with an underscore are properly supported Visual Studio .NET and other tools. They are shown on the top by IntelliSense – the technology that suggests names during coding.

  Example:

```
class SampleClass
{
  string _url;
}
```

8.3 Type Usage Guidelines

8.3.1 Base Class Usage Guidelines

A class is the most common kind of type. A class can be abstract or sealed. An abstract class requires a derived class to provide an implementation. A sealed class does not allow a derived class. It is recommended that you use classes over other types.

Base classes are a useful way to group objects that share a common set of functionality. Base classes can provide a default set of functionality, while allowing customization though extension.

You should explicitly provide a constructor for a class. Compilers commonly add a public default constructor to classes that do not define a constructor. This can be misleading to a user of the class, if your intention is for the class not to be creatable. Therefore, it is best practice to always define at least one constructor for a class. If you do not want it to be creatable, make the constructor private.

8.3.1.1 Base Classes vs. Interfaces

An interface type is a specification of a protocol, potentially supported by many object types. Use base classes instead of interfaces whenever possible. From a versioning perspective, classes are more flexible than interfaces. With a class, you can ship Version 1.0 and then in Version 2.0 add a new method to the class. As long as the method is not abstract, any existing derived classes continue to function unchanged.

Because interfaces do not support implementation inheritance, the pattern that applies to classes does not apply to interfaces. Adding a method to an interface is equivalent to adding an abstract method to a base class; any class that implements the interface will break because the class does not implement the new method.

Interfaces are appropriate in the following situations:

- Several unrelated classes want to support the protocol.
- These classes already have established base classes (for example, some are user interface (UI) controls, and some are XML Web services).
- Aggregation is not appropriate or practical.

In all other situations, class inheritance is a better model.

8.3.1.2 Protected Methods and Constructors

Provide class customization through protected methods. The public interface of a base class should provide a rich set of functionality for the consumer of the class. However, users of the class often want to implement the fewest number of methods possible to provide that rich set of functionality. To meet this goal, provide a set of nonvirtual or final public methods that call through to a single protected method that provides implementations for the methods. This method should be marked with the Impl suffix. Using this pattern is also referred to as providing a Template method. The following code example demonstrates this process.

```
public class MyClass
{
  private int x;
  private int y;
  private int width;
  private int height;
  BoundsSpecified specified;

  public void SetBounds(int x, int y, int width, int height)
  {
     SetBoundsImpl(x, y, width, height, this.specified);
  }
```

```
  public void SetBounds(int x, int y, int width, int height,
    BoundsSpecified specified)
  {
    SetBoundsImpl(x, y, width, height, specified);
  }

  protected virtual void SetBoundsImpl(int x, int y, int width, int
    height, BoundsSpecified specified)
  {
      // Add code to perform meaningful opertions here.
      this.x = x;
      this.y = y;
      this.width = width;
      this.height = height;
      this.specified = specified;
  }
}
```

Many compilers, such as the C# compiler, insert a public or protected constructor if you do not. Therefore, for better documentation and readability of your source code, you should explicitly define a protected constructor on all abstract classes.

8.3.1.3 Sealed Class Usage Guidelines

Use sealed classes if there are only static methods and properties on a class.

8.3.2 Value Type Usage Guidelines

A value type describes a value that is represented as a sequence of bits stored on the stack. For a description of all the .NET Framework's built-in data types, see Value Types. This section provides guidelines for using the structure (struct) and enumeration (enum) value types.

8.3.2.1 Struct Usage Guidelines

It is recommended that you use a struct for types that meet any of the following criteria:

- Act like primitive types.
- Have an instance size under 16 bytes.
- Are immutable.
- Value semantics are desirable.

The following example shows a correctly defined structure:

```
public struct Int32 : IComparable, IFormattable
{
  public const int MinValue = -2147483648;
  public const int MaxValue = 2147483647;

  public static string ToString(int i)
  {
    // Insert code here.
```

```
    }

    public string ToString(string format, IFormatProvider formatProvider)
    {
      // Insert code here.
    }

    public override string ToString()
    {
      // Insert code here.
    }

    public static int Parse(string s)
    {
      // Insert code here.
      return 0;
    }

    public override int GetHashCode()
    {
      // Insert code here.
      return 0;
    }

    public override bool Equals(object obj)
    {
      // Insert code here.
      return false;
    }

    public int CompareTo(object obj)
    {
      // Insert code here.
      return 0;
    }
  }
```

- Do not provide a default constructor for a struct. Note that C# does not allow a struct to have a default constructor. The runtime inserts a constructor that initializes all the values to a zero state. This allows arrays of structs to be created without running the constructor on each instance. Do not make a struct dependent on a constructor being called for each instance. Instances of structs can be created with a zero value without running a constructor. You should also design a struct for a state where all instance data is set to zero, false, or null (as appropriate) to be valid.

8.3.2.2 Enum Usage Guidelines

The following rules outline the usage guidelines for enumerations:

- Do not use an *Enum* suffix on enum types.

- Use an enum to strongly type parameters, properties, and return types. Always define enumerated values using an enum if they are used in a parameter or property. This allows development tools to know the possible values for a property or parameter. The following example shows how to define an enum type:

  ```
  public enum FileMode
  {
    Append,
    Create,
    CreateNew,
    Open,
    OpenOrCreate,
    Truncate
  }
  ```

  The following example shows the constructor for a *FileStream* object that uses the *FileMode* enumeration:

  ```
  public FileStream(string path, FileMode mode);
  ```

- Use an enum instead of static final constants. An exception to this rule is when using string enumerations because .NET doesn't support using *String* as an enumeration base type. Use a sealed class with a private constructor and public const string fields instead.
- Do not use an enum for open sets (such as the operating system version).
- Use the *System.FlagsAttribute* Class to create custom attribute for an enum only if a bitwise OR operation is to be performed on the numeric values. Use powers of two for the enum values so that they can be easily combined. This attribute is applied in the following code example:

  ```
  [Flags()]
  public enum WatcherChangeTypes
  {
    Created = 1,
    Deleted = 2,
    Changed = 4,
    Renamed = 8,
    All = Created | Deleted | Changed | Renamed
  }
  ```

  Note: An exception to this rule is when encapsulating a Win32 API. It is common to have internal definitions that come from a Win32 header. You can leave these with the Win32 casing, which is usually all capital letters.

- Consider providing named constants for commonly used combinations of flags. Using the bitwise OR is an advanced concept and should not be required for simple tasks. This is illustrated in the following example of an enumeration:

  ```
  [Flags()]
  public enum FileAccess
  ```

```
{
  Read = 1,
  Write = 2,
  ReadWrite = Read | Write,
}
```

- Use type *Int32* as the underlying type of an enum unless either of the following is true:
  - The enum represents flags and there are currently more than 32 flags, or the enum might grow to have many flags in the future.
  - The type needs to be different from int for backward compatibility.
- Do not assume that enum arguments will be in the defined range. It is valid to cast any integer value into an enum even if the value is not defined in the enum. Perform argument validation as illustrated in the following code example:

```
public void SetColor (Color color)
{
  if (!Enum.IsDefined (typeof(Color), color)
    throw new ArgumentOutOfRangeException();
}
```

8.4 Exception Management

8.4.1 Exception Usage Guidelines

The following rules outline the guidelines for raising and handling errors:

- All code paths that result in an exception should provide a method to check for success without throwing an exception. For example, to avoid a *FileNotFoundException* you can call *File.Exists*. This might not always be possible, but the goal is that under normal execution no exceptions should be thrown.
- End Exception class names with the *Exception* suffix as in the following code example:

```
public class FileNotFoundException : Exception
{
  // Implementation code goes here.
}
```

- Use the common constructors shown in the following code example when creating exception classes. Base class *Exception* implements interface *ISerializable* which means that custom exception classes must also implement that deserialization constructors, otherwise those exceptions will not be serialized/deserialized across remote boundaries (.NET Remoting).

  Example:

```
public class XxxException : ApplicationException
{
  public XxxException() {... }
  public XxxException(string message) {... }
  public XxxException(string message, Exception inner) {... }
```

```
  public XxxException(SerializationInfo info, StreamingContext context) {...}
}
```

- In most cases, use the predefined exception types. Only define new exception types for programmatic scenarios, where you expect users of your class to catch exceptions of this new type and perform a programmatic action based on the exception type itself. This is in lieu of parsing the exception string, which would negatively impact performance and maintenance.
- For example, it makes sense to define a *FileNotFoundException* because the developer might decide to create the missing file. However, a *FileIOException* is not something that would typically be handled specifically in code.
- Do not derive all new exceptions directly from the base class *SystemException*. Inherit from *SystemException* only when creating new exceptions in *System* namespaces. Inherit from *ApplicationException* when creating new exceptions in other namespaces.
- Use a localized description string in every exception. When the user sees an error message, it will be derived from the description string of the exception that was thrown, and never from the exception class.
- Provide exception properties for programmatic access. Include extra information (other than the description string) in an exception only when there is a programmatic scenario where that additional information is useful. You should rarely need to include additional information in an exception. Remember to serialize and deserialize those custom properties in deserialization constructor and *ISerializable.GetObjectData* method.
- Do not use exceptions for normal or expected errors, or for normal flow of control.
- You should return null for extremely common error cases. For example, a *File.Open* command returns a null reference if the file is not found, but throws an exception if the file is locked.
- Design classes so that in the normal course of use an exception will never be thrown. In the following code example, a *FileStream* class exposes another way of determining if the end of the file has been reached to avoid the exception that will be thrown if the developer reads past the end of the file.

  Example:

```
class FileRead
{
  void Open()
  {
    FileStream stream = File.Open("myfile.txt", FileMode.Open);
    byte b;

    // ReadByte returns -1 at end of file.
    while ((b = stream.ReadByte()) != true)
    {
      // Do something.
    }
  }
}
```

- Throw the *InvalidOperationException* exception if a call to a property set accessor or method is not appropriate given the object's current state.

- Throw an *ArgumentException* or create an exception derived from this class if invalid parameters are passed or detected.
- Be aware that the stack trace starts at the point where an exception is thrown, not where it is created with the new operator. Consider this when deciding where to throw an exception. It affects the value of *StackTrace* property of an Exception.
- Use the exception builder methods. It is common for a class to throw the same exception from different places in its implementation. To avoid repetitive code, use helper methods that create the exception using the new operator and return it. The following code example shows how to implement a helper method:

```
class File
{
  string fileName;
  public byte[] Read(int bytes)
  {
    if (!ReadFile(handle, bytes))
        throw NewFileIOException();
  }

  FileException NewFileIOException()
  {
    string description =
      // Build localized string, include fileName.
    return new FileException(description);
  }
}
```

- Throw exceptions instead of returning an error code or HRESULT.
- Throw the most specific exception possible.
- Create meaningful message text for exceptions, targeted at the developer.
- Set all fields on the exception you use.
- Use Inner exceptions (chained exceptions). However, do not catch and re-throw exceptions unless you are adding additional information or changing the type of the exception.
- Do not create methods that throw *NullReferenceException* or *IndexOutOfRangeException*.
- Perform argument checking on protected (Family) and internal (Assembly) members. Clearly state in the documentation if the protected method does not do argument checking. Unless otherwise stated, assume that argument checking is performed. There might, however, be performance gains in not performing argument checking.
- Clean up any side effects when throwing an exception. Callers should be able to assume that there are no side effects when an exception is thrown from a function. For example, if a *Hashtable.Insert* method throws an exception, the caller can assume that the specified item was not added to the *Hashtable*.

8.4.2 Standard Exception Types

The following table lists the standard exceptions provided by the runtime and the conditions for which you should create a derived class.

Exception type	Base type	Description	Example
Exception	Object	Base class for all exceptions.	None (use a derived class of this exception).
SystemException	Exception	Base class for all runtime-generated errors.	None (use a derived class of this exception).
IndexOutOfRangeException	SystemException	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside of its valid range: arr[arr.Length+1]
NullReferenceException	SystemException	Thrown by the runtime only when a null object is referenced.	object o = null; o.ToString();
InvalidOperationException	SystemException	Thrown by methods when in an invalid state.	Calling Enumerator.GetNext() after removing an Item from the underlying collection.
ArgumentException	SystemException	Base class for all argument exceptions.	None (use a derived class of this exception).
ArgumentNullException	ArgumentException	Thrown by methods that do not allow an argument to be null.	String s = null; "Calculate".IndexOf (s);
ArgumentOutOfRangeException	ArgumentException	Thrown by methods that verify that arguments are in a given range.	String s = "string"; s.Chars[9];
ExternalException	SystemException	Base class for exceptions that occur or are targeted at environments outside of the runtime.	None (use a derived class of this exception).
COMException	ExternalException	Exception encapsulating COM Hresult information.	Used in COM interop.

SEHException	ExternalException	Exception encapsulating Win32 structured Exception Handling information.	Used in unmanaged code Interop.

8.4.3 Exception Wrapping

Errors that occur at the same layer of classes should throw an exception that is meaningful that layer. The Exception that is thrown should inherit from *ApplicationException* and implement all *ISerializable* functionality. Layer exceptions should not be part of its layer assembly but they should be defined in a common assembly used across all application layers. Such organization helps to maintain a clear multi-tier separation.

Example exception class:

```
namespace Roche.Pglc.WebReports.Common
{
  public class DataAccessLayerException : ApplicationException
  {
    public DataAccessLayerException() : base()
    {
    }

    // constructor with exception message
    public DataAccessLayerException(string message) : base(message)
    {
    }

    // constructor with message and inner exception
    public DataAccessLayerException(string message, Exception inner) : base(message, inner)
    {
    }

    // protected constructor to de-seralize state data
    protected DataAccessLayerException(SerializationInfo info, StreamingContext context) : base (info,
context)
    {
    }

    // override GetObjectData to serialize state data
    [SecurityPermission(SecurityAction.Demand, SerializationFormatter = true)]
    public override void GetObjectData(SerializationInfo info, StreamingContext context)
    {
      base.GetObjectData(info, context);
    }
  }
}
```

Example exception catching and rethrowing:

```
public class UsersDAL
{
  public DataSet LoadUsers()
  {
    try
    {
      // loading users from DBMS
    }
    catch (Exception e)
    {
      throw new DataAccessLayerException ("Could not load users", e);
    }
  }
}
```

8.4.4 Exception Logging

All exceptions must be logged which is obvious. The standard exception logging technology for .NET is „Microsoft Exception Management Application Block" which is a special library provided in a source code form. Exception Management Application Block uses a concept of Exception Publishers which are custom classes responsible for publishing exceptions. Configuration of publishers is kept in the application configuration file which is *appname.exe.config* for executables or *web.config* for web applications.

Caught exceptions should be logged before rethrowing (and wrapping), so the previous example changes into:

```
public class UsersDAL
{
  public DataSet LoadUsers()
  {
    try
    {
      // loading users from DBMS
    }
    catch (Exception e)
    {
      Microsoft.ApplicationBlocks.ExceptionManagement.ExceptionManager.Publish(e);
      throw new DataAccessLayerException ("Could not load users", e);
    }
  }
}
```

8.5 Documenting .NET Code

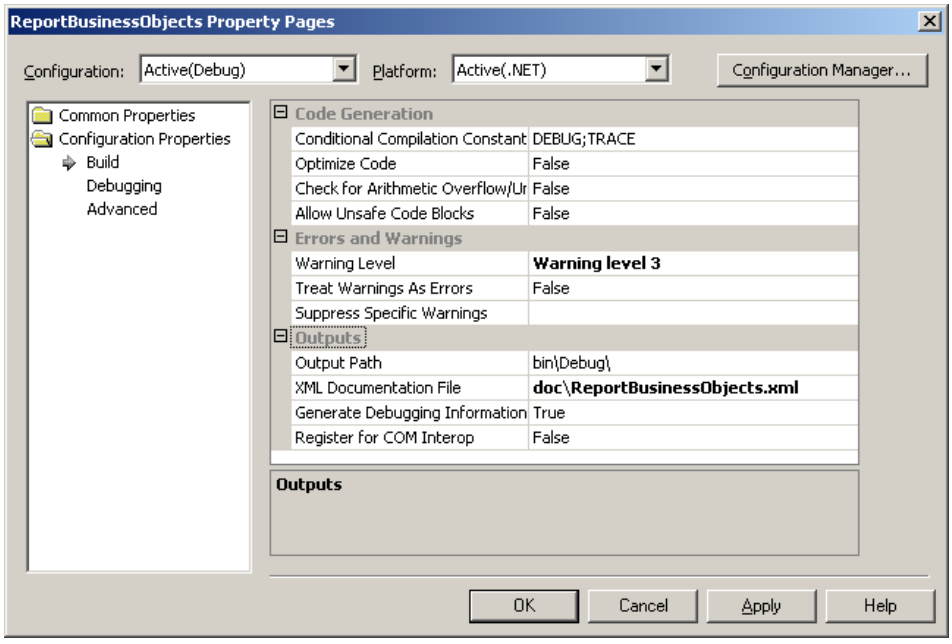
8.5.1 XML Documentation Concept

Similar to Java, all .NET languages (C#, VB.NET and others) provide an ability to use specialized comments before the definition of classes and class members. Those documentation comments must start with /// text at the beginning of every line. Their contents is defined using specialized XML tags that are understood by Visual Studio .NET.

Example comments:

```
/// <summary>
///   Loads all roles from RDE_Role table into the given dataset
///   If the <paramref name="dataSet"/> parameter is null, then
///   a new dataset is instantiated, filled with records and returned.
/// </summary>
/// <param name="dataSet">Dataset to be filled or null</param>
/// <returns>Dataset with a list of all roles</returns>
public RDE_RoleDS LoadRDE_RoleAll(RDE_RoleDS dataSet)
{
  // ...
}
```

.NET compilers can extract XML documentation into a separate XML file during compilation. Those files are used when a developer writes a code that calls a documented method. IntelliSense technology will show the description of method during typing. The description is kept in *summary* XML tag. Generation of XML documentation file by Visual Studio .NET is controlled at a project level using the given screen:



Parameters that must be set are:

- **XML Documentation File** – name of the output file with XML documentation, recommended file location is the *doc* subfolder and the name of the file must be the same as the project name.
- Warning Level – should be decreased to *Warning Level 3* which will not generate warnings during compilation when not all methods are documented. In fact all methods should be documented but auto generated files like typed datasets don't have documentation comments.

HTML pages similar to Javadoc concept may be generated directly from Visual Studio .NET, use *Tools -> Build Comment Web Pages...* or using specialized tools like a freeware *NDoc* project.

8.5.2 XML Documentation Tags

8.5.2.1 <c> Tag

Usage: <c>**text**</c>
where:

- **text** – The text you would like to indicate as code.

Remarks:
The <c> tag gives you a way to indicate that text within a description should be marked as code. Use <code> to indicate multiple lines as code.

Example:

```
/// text for class MyClass
public class MyClass
{
  /// <summary><c>MyMethod</c> is a method in the <c>MyClass</c> class.
  /// </summary>
  public static void MyMethod(int Int1)
  {
  }
  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.2 <code> Tag

Usage: <code>**content**</code>
where:

- **content** – The text you want marked as code.

Remarks:
The <code> tag gives you a way to indicate multiple lines as code. Use <c> to indicate that text within a description should be marked as code.

Example:

```
/// text for class MyClass
public class MyClass
{
  /// <summary>
  /// The GetZero method.
  /// </summary>
  /// <example> This sample shows how to call the GetZero method.
  /// <code>
  ///   class MyClass
  ///   {
  ///      public static int Main()
  ///      {
  ///         return GetZero();
  ///      }
  ///   }
  /// </code>
  /// </example>
  public static int GetZero()
  {
    return 0;
  }
  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.3 <example> Tag

Usage: <example>**description**</example>
where:

- **description** – A description of the code sample.

Remarks:
The <example> tag lets you specify an example of how to use a method or other library member. Commonly, this would involve use of the <code> tag.

Example:

```
/// text for class MyClass
public class MyClass
{
  /// <summary>
  /// The GetZero method.
  /// </summary>
  /// <example> This sample shows how to call the GetZero method.
  /// <code>
  ///   class MyClass
  ///   {
```

```
  ///      public static int Main()
  ///      {
  ///         return GetZero();
  ///      }
  ///   }
  /// </code>
  /// </example>
  public static int GetZero()
  {
     return 0;
  }
  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.4 <exception> Tag

Usage: <exception **cref="member**">**description**</exception>
where:

- **cref = "member"** – A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates member to the canonical element name in the output XML. member must appear within double quotation marks (" ").
- **description** – A description.

Remarks:
The <exception> tag lets you specify which exceptions can be thrown. This tag is applied to a method definition.

Example:

```
using System;

/// comment for class
public class CustomException : ApplicationException
{
  // class definition ...
}

/// <exception cref=" CustomException ">Thrown when... .</exception>
class TestClass
{
  public static void Main()
  {
    try
    {
    }
    catch(Exception e)
```

```
    {
      throw new CustomException(e);
    }
  }
}
```

8.5.2.5 <include> Tag

Usage: <include file='**filename**' path='**tagpath**[@**name**="**id**"]' />
where:

- **filename** – The name of the file containing the documentation. The file name can be qualified with a path. Enclose filename in single quotation marks (' ').
- **tagpath** – The path of the tags in filename that leads to the tag name. Enclose the path in single quotation marks (' ').
- **name** – The name specifier in the tag that precedes the comments; name will have an id.
- **id** – The ID for the tag that precedes the comments. Enclose the ID in double quotation marks (" ").

Remarks:
The <include> tag lets you refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file.
The <include> tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your <include> use.

Example:

This is a multifile example. The first file, which uses <include>, is listed below:

```
// xml_include_tag.cs
/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
  public static void Main()
  {
  }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
  public void Test()
  {
  }
}
```

The second file, xml_include_tag.doc, contains the following documentation comments:

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

8.5.2.6 <list> Tag

Usage:

```
<list type="bullet" | "number" | "table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>
```

where:

- ***term*** – A term to define, which will be defined in text.
- ***description*** – Either an item in a bullet or numbered list or the definition of a term.

Remarks:
The *<listheader>* block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for term in the heading.
Each item in the list is specified with an *<item>* block. When creating a definition list, you will need to specify both term and text. However, for a table, bulleted list, or numbered list, you only need to supply an entry for text.
A list or table can have as many *<item>* blocks as needed.

Example:

```
// xml_list_tag.cs

/// text for class MyClass
public class MyClass
{
```

```
   /// <remarks>Here is an example of a bulleted list:
   /// <list type="bullet">
   /// <item>
   /// <description>Item 1.</description>
   /// </item>
   /// <item>
   /// <description>Item 2.</description>
   /// </item>
   /// </list>
   /// </remarks>
   public static void Main ()
   {
   }
}
```

8.5.2.7 <para> Tag

Usage: <para>**content**</para>
where:

- **content** – The text of the paragraph.

Remarks:
The <para> tag is for use inside a tag, such as <summary>, <remarks>, or <returns>, and lets you add structure to the text.

Example:

```
/// text for class MyClass
public class MyClass
{
   /// <summary>MyMethod is a method in the MyClass class.
   /// <para>Here's how you could make a second paragraph in a description. <see cref="System.Console.WriteLine"/> for information about output statements.</para>
   /// <seealso cref="MyClass.Main"/>
   /// </summary>
   public static void MyMethod(int Int1)
   {
   }

   /// text for Main
   public static void Main ()
   {
   }
}
```

8.5.2.8 <param> Tag

Usage: <param name=”'**name**”>**description**</param>
where:

- **name** – The name of a method parameter. Enclose the name in single quotation marks (' and ').

- **description** – A description for the parameter.

Remarks:
The **<param>** tag should be used in the comment for a method declaration to describe one of the parameters for the method.
The text for the **<param>** tag will be displayed in IntelliSense, the Object Browser, and in the Code Comment Web Report.

Example:

```
/// text for class MyClass
public class MyClass
{
  /// <param name="Int1">Used to indicate status.</param>
  public static void MyMethod(int Int1)
  {
  }
  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.9 <paramref> Tag

Usage: <paramref name=”**name**” />
where:

- **name** – The name of the parameter to refer to. Enclose the name in double quotation marks (" ").

Remarks:
The <paramref> tag gives you a way to indicate that a word is a parameter. The XML file can be processed to format this parameter in some distinct way.

Example:

```
/// text for class MyClass
public class MyClass
{
  /// <remarks>MyMethod is a method in the MyClass class.
  /// The <paramref name="Int1"/> parameter takes a number.
  /// </remarks>
  public static void MyMethod(int Int1)
  {
  }
  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.10 <permission> Tag

Usage: <permission **cref=”member”**>**description**</permission>
where:

- **cref = "member"** – A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates member to the canonical element name in the output XML. member must appear within double quotation marks (" ").
- **description** – A description of the access to the member.

Remarks:
The <permission> tag lets you document the access of a member. The System.Security.PermissionSet lets you specify access to a member.

Example:

```
using System;
class TestClass
{
  /// <permission cref="System.Security.PermissionSet">
  ///  Everyone can access this method.
  /// </permission>
   public static void Test()
   {
   }
   public static void Main()
   {
   }
}
```

8.5.2.11 <remarks> Tag

Usage: <remarks>**description** </remarks>
where:

- **description** – A description of the member.

Remarks:
The <remarks> tag is used to add information about a type, supplementing the information specified with <summary>. This information is displayed in the Object Browser and in the Code Comment Web Report.

Example:

```
/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class MyClass
```

```
{
  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.12 <returns> Tag

Usage: <returns>**description**</returns>
where:

- **description** – A description of the return value.

Remarks:
The <returns> tag should be used in the comment for a method declaration to describe the return value.

Example:

```
/// text for class MyClass
public class MyClass
{
  /// <returns>Returns zero.</returns>
  public static int GetZero()
  {
    return 0;
  }

  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.13 <see> Tag

Usage: <see **cref=”member”**/>
where:

- **cref = "member"** – A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes member to the element name in the output XML. **member** must appear within double quotation marks (" ").

Remarks:
The <see> tag lets you specify a link from within text. Use *<seealso>* to indicate text that you might want to appear in a See Also section.

Example:

```
/// text for class MyClass
public class MyClass
```

```
{
  /// <summary>MyMethod is a method in the MyClass class.
  /// <para>Here's how you could make a second paragraph in a description. <see cref="System.Console.WriteLine"/> for information about output statements.</para>
  /// <seealso cref="MyClass.Main"/>
  /// </summary>
  public static void MyMethod(int Int1)
  {
  }

  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.14 <seealso> Tag

Usage: <seealso **cref=”member”**/>
where:

- **cref = "member"** – A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes member to the element name in the output XML. member must appear within double quotation marks (" ").

Remarks:
The <seealso> tag lets you specify the text that you might want to appear in a See Also section. Use *<see>* to specify a link from within text.

Example:

```
/// text for class MyClass
public class MyClass
{
  /// <summary>MyMethod is a method in the MyClass class.
  /// <para>Here's how you could make a second paragraph in a description. <see cref="System.Console.WriteLine"/> for information about output statements.</para>
  /// <seealso cref="MyClass.Main"/>
  /// </summary>
  public static void MyMethod(int Int1)
  {
  }

  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.15 <summary> Tag

Usage: <summary>**description**</summary>
where:

- **description** – A summary of the object.

Remarks:
The <summary> tag should be used to describe a type or a type member. Use <remarks> to add supplemental information to a type description.
The text for the <summary> tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser and in the Code Comment Web Report.

Example:

```
/// text for class MyClass
public class MyClass
{
  /// <summary>MyMethod is a method in the MyClass class.
  /// <para>Here's how you could make a second paragraph in a description. <see cref="System.Console.WriteLine"/> for information about output statements.</para>
  /// <seealso cref="MyClass.Main"/>
  /// </summary>
  public static void MyMethod(int Int1)
  {
  }

  /// text for Main
  public static void Main ()
  {
  }
}
```

8.5.2.16 <value> Tag

Usage: <value>**property-description**</value>
where:

- **property-description** – A description for the property.

Remarks:
The <value> tag lets you describe a property. Note that when you add a property via code wizard in the Visual Studio .NET development environment, it will add a <summary> tag for the new property. You should then manually add a <value> tag to describe the value that the property represents.

Example:

```
using System;
/// text for class Employee
public class Employee
{
  private string name;
  /// <value>Name accesses the value of the name data member</value>
```

```
  public string Name
  {
    get
    {
      return name;
    }
    set
    {
      name = value;
    }
  }
}

/// text for class MainClass
public class MainClass
{
  /// text for Main
  public static void Main()
  {
  }
}
```

8.6 CVS Directory Structure and Naming Conventions

The directory structure of the CVS repository and naming conventions that should be used for each project is described in this chapter.

For details regarding CVS please refer to "PGL, Source Code Handling and Release Management (Using CVS - Concurrent Versioning System), Global SOP" document.

The directory structure must be clear, consistent and contain separate subdirectories needed for storing in order all the files required to build, compile and unit test the entire application. The directory structure meant to help with keeping order within the different application files in the CVS repository is shown in Figure 5.

```
…---Application_Name
    |
    |___ Source
    |
    |___ Documents
    |
    |___ Tests
    |
    |___ Configuration
    |
    |___ Tools
    |
    |___ Scripts
    |
    |___ Libraries
    |
    |___ Binaries
```

Figure 5: "The possible inner structure of "Application_Name" folder for .NET and C# languages"

"Application_Name" it is the folder containing all the source, documents, tests, configuration, tools, scripts, libraries, binaries and other files belonging to or related to the SW application. The naming conventions for this folder are the following:

APP_FullName

Where:

- APP : Abbreviation for SW Application.
- FullName : Full name of the application.
- If abbreviation for SW application together with the FullName of the application exceeds 44 characters then we are leaving only the abbreviation.

e.g. PSS_Patient_Summary_System

The purpose of the sub-folders within the *"Application_Name"* folder is the following:

- *"Source"* – In this folder all the source code, graphics and other files being integral part of the Application shall be stored. In this directory should be also stored the source code developed for Unit Tests.
- *"Documents"* – In this folder shall be stored all the files documenting the module. If all the files documenting the application are already in PLv2, listed in README file within the *"Application_Name"* folder and no other documents exist that should be stored here this folder is not necessary.
- *"Tests"* – In this folder all shall be stored all the data files and other supporting documents related to Unit Test prepared for the module.
- *"Configuration"* – This is the folder where all configuration data/scripts/documents should be stored.
- *"Tools"* – This is the folder where all supporting tools used during the application development should be stored. In case of the self-developed tools, the source and binaries should be located here. In case when developer have used external tools the installation files, other binaries and source files (if available) should be stored in this directory.
- *"Scripts"* – This is the folder where all the additional supporting scripts (UNIX scripts, SQL scripts, Shell scripts) should be stored.
- *"Libraries"* – This is the folder where the binary, installation and source (if available) versions of the libraries used during the development should be stored. This folder should also be used as a reference for the application during the development in case when binary version of the library is used.
- *"Binaries"* – This is the folder where the binary and preferably installation versions of the official production releases of the application should be stored. All the production binaries for production release must be packed into one zip archive. The naming conventions for this file are the following:
  APP_X_Y_Z_A.zip
  Where:
  - APP : Abbreviation for SW Application.
  - X: Version number Major.
  - Y: Version number Minor.
  - Z: Release number.
  - A: Optional (e.g. Built number).

  e.g. PSS_3_1_5_23.zip

Folders *"Source"*, *"Tests", "Binaries"* and *"Tools"* are mandatory for each application. Other folders should be created depending on the developer needs.

Note: Every *"Application_Name"* directory should contain a README file that covers:

- short description of the Application,
- system versions, e.g. OC version, Windows version, where the SW application will work,
- build and install directions,
- direct people to all the related resources:
  - directories of source,
  - online documentation,
  - paper documentation,
  - design documentation,
- anything else that might help someone.

It is preferred that a README file will be in text format but it can also be MS Word or PDF file.

It is important to update the README file with every production release.

Every software release must be properly tagged in CVS according to guidelines given in the "PGL, Release, Tag, and Branch Management, Global Instruction" document.

9 References

9.1 Roche Documents

[PGL - 400] PGL, Source Code Handling and Release Management (Using CVS - Concurrent Versioning System), Global SOP

[PGL - 402] PGL, Release, Tag, and Branch Management, Global Instruction

9.2 External References

[1] “Code Conventions for the Java Programming Language”, SUN Microsystems Inc.

[2] "How to Write Doc Comments for Javadoc", SUN Microsystems Inc.
(URL: http://java.sun.com/products/jdk/javadoc/writingdoccomments.html)

[3] ISO Standard 3166, 1981

Project Name: *5089-10 PGL SOPs*

Document Name: **Warsaw CoE 9.1.1. Guide to Codnig Standards and Naming Conventions, local SPT**

Version: **2.0**

This document has been signed by:

Meaning: Sign for Review
Function: Author
User Name: Popko, Leszek (popkol)
Date: 27-Feb-2006 14:02 W. Europe Daylight Time

Meaning: Sign for Review
Function: Author
User Name: Czabanowski, Adrian (czabanoa)
Date: 27-Feb-2006 14:26 W. Europe Daylight Time

Meaning: Sign for Review
Function: Author
User Name: Suski, Marek (suskim)
Date: 27-Feb-2006 14:34 W. Europe Daylight Time

Meaning: Sign for Review
Function: Author
User Name: Szokalski, Daniel (szokalsd)
Date: 27-Feb-2006 14:55 W. Europe Daylight Time

Meaning: Sign for Review
Function: Author
User Name: Postupalski, Pawel (postupap)
Date: 28-Feb-2006 09:02 W. Europe Daylight Time

Meaning: Sign for Review
Function: Author
User Name: Marszalek, Marek (marszam1)
Date: 28-Feb-2006 09:26 W. Europe Daylight Time

Meaning: Sign for Review
Function: Author
User Name: Zawadzki, Zbigniew (zawadzkz)
Date: 28-Feb-2006 10:52 W. Europe Daylight Time

Meaning: Sign for Review
Function: Author
User Name: Poltorak, Marek (poltorm1)
Date: 28-Feb-2006 13:17 W. Europe Daylight Time

Project Name: *5089-10 PGL SOPs*

Document Name: **Warsaw CoE 9.1.1. Guide to Codnig Standards and Naming Conventions, local SPT**

Version: **2.0**

Meaning: Sign for Review
Function: Quality Assurance
User Name: Schumacher, Wolfgang (schumaw2)
Date: 28-Feb-2006 16:08 W. Europe Daylight Time

Meaning: Sign for Review
Function: Technical Review
User Name: Majczak, Miroslaw (majczakm)
Date: 01-Mar-2006 14:11 W. Europe Daylight Time

Meaning: Sign for Review
Function: Technical Review
User Name: Czarnas, Piotr (czarnasp)
Date: 02-Mar-2006 11:42 W. Europe Daylight Time

Meaning: Sign for Approval
Function: Warsaw CoE Site Head
User Name: Hill, John W (hillj16)
Date: 03-Mar-2006 15:41 W. Europe Daylight Time