|  |  |
| --- | --- |
| Coding Standards | |
|  | |
| hp_logo | |
|  | |
|  |  |
| Version: | A.10 |
| Effective Date: | 08-14-03 |

Document Background

Abstract

All developers should understand and use the coding conventions and standards described herein. This guide provides the standards for naming, formatting, and documenting code. Specifically, these standards target development using the C# language for the .Net platform. Careful attention has been given to adopt a standard that is semantically congruent (and strives for syntactically similar) to well-known coding standards for Java development.

DESIGN INITIATION

Initiation Date 7/22/2003

Coordinating Author Billy Shaw

Key Contributors Vu Le, R.Tang, P.Johansen, L. Fogtman, Gopalakrishna, Jagannath T

Approver Steven Lopez

Revision History

|  |  |  |
| --- | --- | --- |
| Version | Date | Description of Revisions |
| A.0 | 07-22-03 | Preliminary Draft |
| A.1 | 07-25-03 | VHL Draft |
| A.2 | 07-26-03 | JLF Edit |
| A.3 | 07-29-03 | JLF/VHL Merge |
| A.4 | 07-29-03 | JLF Edit |
| A.5 | 07-30-03 | JLF notation for enterprise scoping |
| A.6 | 08-05-03 | JLF Editing, Incorporate Comments and more information for namespace discussion |
| A.7 | 08-11-03 | JLF – incorporate suggestions from EMEA (Pete Saunders) |
| A.8 | 08-12-03 | Gopalakrishna – Validated Core Changes Suggested by core group and incorporated them |
| A.9 | 08-13-03 | JLF – incorporate suggestions from Reza/MSDN standards document (see green highlighted sections) |
| A.10 | 8-13-03 | JLF – revisions from Conference Call with standards group |
| A.11 | 8-15-03 | JLF – minor corrections and namespace finalization.  This revision marks the end of the RFC period for the PSG IT standards working group.  This revision is being sent to the HP C&I team for potentially broader adoption in the enterprise.  This revision still lacks the directory structure diagram in section 2.1 |
| A.12 | 03/05/2008 | JaganG -- Added Single Source Code Process Standards |
| A.13 | 11/18/2009 | PGS – incorporated standards from GADSC PMO |

Table of Contents

[1. Introduction 1](#_Toc48557353)

[1.1 Purpose 1](#_Toc48557354)

[1.2 Why Standards? 1](#_Toc48557355)

[1.3 The Prime Directive 2](#_Toc48557356)

[1.4 Scope 2](#_Toc48557357)

[1.5 Intended Audience 2](#_Toc48557358)

[1.6 Definitions and Acronyms 2](#_Toc48557359)

[1.7 References 3](#_Toc48557360)

[2. File Structure 4](#_Toc48557361)

[2.1 CLS Types – Interfaces and Classes 4](#_Toc48557362)

[2.2 Class Template 4](#_Toc48557363)

[2.3 Regions (.Net specific) 5](#_Toc48557364)

[2.4 File/Copyright Header 5](#_Toc48557365)

[2.5 References Block 6](#_Toc48557366)

[2.6 Sample Template File 7](#_Toc48557367)

[3. Namespace Taxonomy 8](#_Toc48557368)

[3.1 Example namespaces 8](#_Toc48557369)

[3.2 Organization/Function Component 8](#_Toc48557370)

[3.3 Region Component 8](#_Toc48557371)

[3.4 Project (or Platform) Component 9](#_Toc48557372)

[3.5 Vertical Component 9](#_Toc48557373)

[3.6 Layer Component 9](#_Toc48557374)

[3.7 Domain Component 9](#_Toc48557375)

[3.7.1 Sub-Domain Component 10](#_Toc48557376)

[4. Documentation 10](#_Toc48557377)

[4.1 Best practices in commenting 10](#_Toc48557378)

[4.2 Constants 11](#_Toc48557379)

[4.3 XML Documentation 11](#_Toc48557380)

[4.3.1 Recommended Tags for Documentation Comments 12](#_Toc48557381)

[4.4 Interface comments 12](#_Toc48557382)

[4.5 Class Level Comment 12](#_Toc48557383)

[4.6 Method Level Comment 13](#_Toc48557384)

[4.7 Property Level Comment 13](#_Toc48557385)

[5. Formatting 14](#_Toc48557386)

[5.1 Matching Braces 14](#_Toc48557387)

[5.2 Indentation 15](#_Toc48557388)

[5.2.1 Line Length 15](#_Toc48557389)

[5.2.2 Wrapping Lines 16](#_Toc48557390)

[5.3 White Spaces 16](#_Toc48557391)

[5.3.1 Blank Spaces 16](#_Toc48557392)

[5.3.2 Blank Lines 17](#_Toc48557393)

[5.4 Method Parameters 17](#_Toc48557394)

[5.5 Member and Local Variables 18](#_Toc48557395)

[5.6 Statements 19](#_Toc48557396)

[5.6.1 Simple Statements 19](#_Toc48557397)

[5.6.2 Conditional Statements 19](#_Toc48557398)

[5.6.3 Looping Statements 20](#_Toc48557399)

[5.6.4 Switch Statements 21](#_Toc48557400)

[6. Naming Conventions 21](#_Toc48557401)

[6.1 Capitalization Styles 21](#_Toc48557402)

[6.1.1 Pascal Casing 22](#_Toc48557403)

[6.1.2 Camel Casing 22](#_Toc48557404)

[6.1.3 Constant Casing 22](#_Toc48557405)

[6.2 General Naming Guidelines 22](#_Toc48557406)

[6.2.1 Best Practice 22](#_Toc48557407)

[6.2.2 Acronyms and Abbreviations 23](#_Toc48557408)

[6.2.3 Case Sensitivity 23](#_Toc48557409)

[6.3 Quick Reference for Each Identifier Type 23](#_Toc48557410)

[6.4 Naming Examples for each Identifier 25](#_Toc48557411)

# Introduction

## Purpose

This document requires or recommends certain practices for developing programs using the C# language. The objective of this standard is to have a positive effect on

* the readability and maintainability of code across the entire organization
* promote consistent conventions (as opposed to arbitrary rules upon developers)
* developer productivity by dissuading wasteful practices
* the overall quality of the software organization

## Why Standards?

Superior coding techniques and programming practices are hallmarks of a professional programmer. The bulk of programming consists of making many small choices, which collectively attempt to solve a large set of problems. A programmer's skill and expertise largely determine the wisdom of those choices.

A comprehensive coding standard encompasses all aspects of code construction. While developers should prudently implement a standard, it should be adhered to whenever practical. Completed source code should reflect a harmonized style, as if a single developer wrote the code in one session. At the inception of a software project, establish a coding standard to ensure that all developers on the project are working in concert. When the software project incorporates existing source code, or when performing maintenance on an existing software system, the coding standard should state how to deal with the existing code base.

The readability of source code has a direct impact on how well a developer comprehends a software system. Code maintainability refers to how easily that software system can be changed to add new features, modify existing features, fix bugs, or improve performance. Although readability and maintainability are the result of many factors, one particular facet of software development upon which all developers have an influence is coding technique. The easiest method to ensure a team of developers will yield quality code is to establish a coding standard, which is then enforced at routine code reviews.

Using solid coding techniques and good programming practices to create high-quality code plays an important role in software quality and performance. In addition, if you consistently apply a well-defined coding standard, apply proper coding techniques, and subsequently hold routine code reviews, a software project is more likely to yield a software system that is easy to comprehend and maintain.

## The Prime Directive

No standard is perfect and no standard is applicable to all situations: it is likely to find a situation where one or more standards do not apply. This leads to the introduction of the prime directive of standards [Ambler]:

**When you go against a standard, document it.** All standards, except for this one, can be broken. If you do so, you must document why you broke the standard, the potential implications of breaking the standard, and any conditions that may/must occur before the standard can be applied to this situation. The bottom line is that you need to understand each standard, understand when to apply them, and just as importantly when not to apply them.

## Scope

This standard pertains to the use of the C# language. With very few exceptions, it does **not** discuss the use of the .NET class libraries. Certain items that deserve attention have been identified, but have **not** been included in this document because treatment in separate documents appears more appropriate. These include items such as: Design Patterns, Unmanaged code, COM, Multi-threading, Localization (languages, Unicode), Remoting, WinForms, Security, etc. Also, the standards discussed do not address some development processes like management of Visual Studio project and solution structures (please see the Team Development Process document for guidance on this topic). Ideally, these standards can be adopted by any team using any IDE. This document makes assumptions that Visual Studio.Net will be used by most teams but does not mandate its usage.

## Intended Audience

Readers are expected to be familiar with generally accepted techniques for software engineering. For an introduction to software engineering, readers are directed to the references listed below.

## Definitions and Acronyms

|  |  |
| --- | --- |
| **Term** | **Definition** |
| .Net | Microsoft .NET is a set of Microsoft software technologies for connecting information, people, systems and devices. It enables a high level of software integration through the use of XML Web Services. |
| C# | C# (pronounced C-sharp) is the core programming language for the .Net framework developed by Microsoft in 1999. |
| Java | JavaTM is a programming language developed by Sun Microsystems in 1995 and is now standardized under the direction of the Java Community Process. |
| Web Service | Web services are self-contained business functions that operate over the Internet or Intranet using standard and open technologies. |
| SOA | A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed. SOA is the evolution of proprietary RPC technologies (OLE, CORBA, DCOM, and RMI) to a more open suite of technologies (XML, SOAP, HTTP). |
| IDE | Integrated Development Environment is a tool that enables development of applications targeted to specific technologies. The IDE usually includes debugging, editing, compiling and re-factoring capabilities. |
| Sandcastle | An open source tool that generates API documentation in HTML and Help file by by reflecting over the source assemblies and optionally integrating XML Documentation Comments (see <http://sandcastle.codeplex.com/>). |
| ~~NDoc~~ | ~~An open source tool that generates API documentation in HTML and Help file formats by processing inline XML documentation (see~~ [~~http://ndoc.sourceforge.net~~](http://ndoc.sourceforge.net) ~~)~~ |

## References

1. Scott Ambler; *Writing Robust Java Code*; [www.ambysoft.com](http://www.ambysoft.com) ; 2000
2. MSDN; *Design Guidelines for Class Library Developers* ; [www.microsoft.com](http://www.microsoft.com) ; 2002
3. Bruce Eckel; *Thinking in C++, Second Edition*, Prentice Hall PTR; 1998
4. Steve McConnell; *Code Complete*; Microsoft Press; 1993
5. Tang, et. al.; *Coding Guidelines*; [www.convergencephenomenon.com](http://www.convergencephenomenon.com); 2003
6. Vic Hartog, et. al.; *Coding Standard: C#*; Philips Medical Systems; 2003
7. Reza Wajih, *Coding Techniques Programming Practices*; MSDN compilation; 2002
8. GADSC CDC PMO; *CONSTRUCT\_Net.Technical Guidelines;* [*http://teams2.sharepoint.hp.com/teams/GADSC\_CDC\_PMO/Process%20and%20Tools/Methodology%20and%20Process.aspx*](http://teams2.sharepoint.hp.com/teams/GADSC_CDC_PMO/Process%20and%20Tools/Methodology%20and%20Process.aspx)
9. HPGM-AS; *AS\_ENG3010G C Sharp Coding Standards*; <http://gm-as.hp.com/iqs/website/webpages/html_gdpg/engineering_process.htm>

# File Structure

## CLS Types – Interfaces and Classes

A file should contain only one public class or interface. The single public class defined in each source file should have the same name as the file (excluding the .cs extension). For example, a class called Employee should be defined in a file called Employee.cs

All interfaces and their implementation should be contained in the same namespace. If the namespace is too large (more than 10 classes/interfaces) then consideration should be given to further decomposition or re-structuring.

Classes must never have dependencies on classes defined in sub-namespaces of their own namespace. Ideally, classes should avoid collaborations with classes defined in sub-namespace (avoid downward dependencies in the class lattice).

Source files will reside in a folder structure that mirrors the namespace/package schema for the primary class.

~~TODO - Need to insert picture here of directory structure as an example. I will await a decision on namespace before creating the image of namespace-to-directory.~~

## Class Template

File structure should be organized as follows:

1. Copyright Header (see Section 2.4 for details)
2. Using/Import block (see Section 2.5 for details)
3. Namespace declaration
4. Class Header
5. Constants, events, delegates and enumerated types
6. Private/Protected members and structures
   1. Static members (class scope)
   2. Instance members (object scope)
7. Constructors and destructors (class and instance)
8. Public Properties
   1. Group by interface implementation
9. Public Methods
   1. Group by interface implementation
   2. Group by semantically similar behavior
10. Private Methods
11. Event handlers
12. Inner Classes

## Regions (.Net specific)

Each of the above sections will use a collapsible region similar to the following (do not use quotes around region names):

|  |
| --- |
| // GOOD  #region constructors  // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // constructors  // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  static MyClass()  {  }  public MyClass()  {  }  #endregion constructors  // Use this List of Well-known region names  #region copyright  #region references  #region header  #region constants  #region members  #region constructors  #region public methods  #region private methods  #region events |

## 

## 

## File/Copyright Header

Use the following copyright header.

It is necessary to use SCC keyword expansion in the file header. The expansion keywords are supported by all of the dominant Source repositories (VSS, CVS, Subversion, PVCS, et al). Each file header should include the Author, Archive and Version keywords. Usage of the History tag is discretionary. Note: the History tag can be verbose given many revisions. It will be necessary to use either /\* \*/ or /// style comments with the History tag as it comprises multiple lines.

**Note:** Please consider the temporal nature of this header and update this document to use the most recent copyright information for new files.

|  |
| --- |
| // ===============================================================  //  // file name here  //  // short description of what this file contains  //  // $Author: $  // $Version: $  // $Archive: $  //  // Modification History:  // Date Programmer Description  // mm/dd/yyyy programmer name comments  //  // ===============================================================  // Copyright (C) 2003 Hewlett-Packard Company  // All rights reserved.  // ======================================================== |

## References Block

Group by semantic similarity or organization. Within these groups alphabetical ordering is suggested. (C#)Use a namespace alias only if required. Never put a using statement inside a namespace.

|  |
| --- |
| #region references  using System;  using System.Drawing;  using System.Collections;  using System.Data;  using System.Data.Common;  using System.Data.OleDb;  using HP.PsgIt.Americas.Common.Data;  using HP.PsgIt.Americas.InventDirect.Business.Cart  #endregion references |

## 

## 

## Sample Template File

Copy *NewCSharpFile.cs* to this folder:

*C:\Program Files\Microsoft Visual Studio .NET\VC#\VC#Wizards\ CSharpAddClassWiz\Templates\1033.*

Create a new C# class in Visual Studio.Net. Reference the format of the sample C# class file.



# Namespace Taxonomy

The following namespace structure is intended aligned according to the business functions within HP. This hierarchy is necessary for the proper implementation of code-line ownership policies. The following nomenclature will be used for all namespace declarations. See the tables below for the definition and constraints for each component.

**HP.<Function>.<Region>.<Project>.[Vertical].<Layer>.<Domain>.[Sub-Domain]**

## Example namespaces

|  |
| --- |
| // Samples for Common Code  HP.<Function>.<Region>.<Platform>.<Vertical>.<Layer>.<Domain>  HP.Ecommerce.Worldwide.Common  HP.Ecommerce.Emea.Common  HP.Ecommerce.Apj.Common  HP.Ecommerce.Americas.Common  // Samples for Business Domain Code (Core)  HP.Ecommerce.Worldwide.eComCat.Core.Business.Catalogs  HP.Ecommerce.Emea.EasyDirect.Core.Business.Purchasing  HP.Ecommerce.Apj.InventDirect.Core.Business.Carts  HP.Ecommerce.Americas.InventDirect.Core.Business.Customers  // Samples for Business Domain Code (Specific Verticals)  HP.Ecommerce.Worldwide.eComCat.Smb.Business.Catalogs  HP.Ecommerce.Emea.EasyDirect.Smb.Business.Purchasing  HP.Ecommerce.Apj.InventDirect.Gem.Business.CartsHP.Ecommerce.Americas.InventDirect.LA.Business.Customers |

## Organization/Function Component

Use one of the following items for the functional component.

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Ecommerce | Ecommerce functions (external applications) |
| Corporate | Corporate functions (finance, legal, payroll) |
| Marketing | Marketing functions |
| Manufacturing | Manufacturing functions |
| Logistics | Logistics and Distribution functions |
| More to be defined | With the help of the business and WS team |

## Region Component

Use one of the following items for the region component.

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Worldwide | World Wide (Core) Group |
| Apj | Asia Pacific & Japan Group |
| Emea | Europe, Middle East & Africa |
| Americas | The Americas Group |

## Project (or Platform) Component

The project component is typically the unique name of the project or platform for the system that is being developed. Examples are eComCat, Vista, EasyDirect and InventDirect. This component should be unique within the parent namespace to avoid conflicts. The project (or platform) names should be well-known. Avoid using abbreviations. Do NOT use a technology name (unless it is your product) for this component (e.g. don’t use asp or servlet). Common code utilities and libraries can be aggregated at this level in the namespace (use Common as the platform name).

## Vertical Component

Vertical components will be defined for each organization and region. Usually, the verticals will be specific pipelines or industry vertical extensions to a project or platform. Core is used for the namespace components that are common across verticals. The following examples should be used and each organization will add more verticals for their respective projects.

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Core | Core items across verticals |
| Gem (Public Sector) | Government, Education and Medical vertical |
| LA | Latin America vertical |
| Smb | Small and Medium Business vertical |

## Layer Component

The layer component defines the logical location/tier to which the class belongs.

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Common | Common utilities, components and libraries |
| UI | User Interface / Presentation Tier (e.g. Web, Win32, other) |
| Business | Business Logic Tier (e.g. Application or Problem domain) |
| Data | Data Access Tier |
| Services | SOA Tier (e.g. web services, session façade) |
| Resources | Persistence based resources (durable resources) |

## Domain Component

The domain component is a logical concept that is tied to the application or problem domain. The domain component should have a domain-specific name that intuitively represents the semantically related classes that comprise the business functionality of that portion of the domain. The domain component should be determined in the design phase by performing proper OO realization into a package diagram.

### Sub-Domain Component

The sub-domain component is an optional item to allow further de-composition of the package/namespace structure into nested items. The sub-domain components are simply the nested packages realized in the OO package diagram.

# Documentation

## Best practices in commenting

* **Comments should add to the clarity of your code.** The reason why you document your code is to make it more understandable to you, your coworkers, and to any other developer who comes after you. (Nagler, 1995)
* **Sniff Test.** If another programmer cannot determine what a method or class does within 30 seconds of looking at comments and method signature(s), then there may be a problem.
* **No Special Characters in Comments.** Comments should never include any special characters such as form-feed or backspace.
* **Keep comments simple.** Some of the best comments are simple, point-form notes. You do not have to write a book; you just have to provide enough information so that others can understand your code.
* **Write the documentation before you write the code.** The best way to document code is to write the comments before you write the code. This gives you an opportunity to think about how the code will work before you write it and will ensure that the documentation gets written. Alternatively, you should at least document your code *as* you write it.
* **Preference is given to single-line (// my comment) style.** It is preferable to use single-line versus C-style (/\* my comment \*/).
* **All Comments should pass a spell-check.** Misspelled comments indicate sloppy development.
* **Do not use End line Comments.**

|  |
| --- |
| // BAD  if (grandTotal >= 1000.00)  {  grandTotal = grandTotal \* 0.95; // somecomment here  } |

* **Document why something is being done, not just what.** Fundamentally, you can always look at a piece of code and figure out what it does. For example, you can look at the code below and figure out that a 5% discount is being given on orders of $1,000 dollars or more. Why is this being done? Is there a business rule that says that large orders get a discount? Is there a limited-time special on large orders or is it a permanent program? Was the original programmer just being generous? I do not know unless it is documented somewhere, either in the source code itself or in an external document (Ambler, 1998a).

|  |
| --- |
| // code explanation or purpose is needed here  if (grandTotal >= 1000.00)  {  grandTotal = grandTotal \* 0.95;  } |

* **Attempt to document closing braces**

|  |
| --- |
| switch (someVariable)  {  // . . . long set of case statements  } // end switch  public void SomeMethod()  {  // . . . about 25 lines of code  } // end method – SomeMethod |

## Constants

Public constants and members with non-descriptive names should have comments that include the following information:

* A description of the constant or variable
* Additional information as the author sees fit

## XML Documentation

In Visual C# you can document the code you write using XML. C# is the only programming language in Visual Studio .NET with this feature. For details on creating an XML file with documentation comments, see the following topics.

|  |  |
| --- | --- |
| **For information about** | **See** |
| The C# compiler option to use | [/doc (Process Documentation Comments)](http://msdn.microsoft.com/library/en-us/cscomp/html/vcerrDocProcessDocumentationComments.asp) |
| Tags you can use to provide commonly used functionality in documentation | [Recommended Tags for Documentation Comments](#vclrftagsfordocumentationcomments) |
| The ID strings that the compiler produces to identify the constructs in your code | [Processing the XML File](http://msdn.microsoft.com/library/en-us/csref/html/vclrfprocessingxmlfile.asp) |
| How to delimit documentation tags | [Delimiters for Documentation Tags](http://msdn.microsoft.com/library/en-us/csref/html/vclrfdelimitersfordocumentationtags.asp) |
| How to document a type | [XML Documentation Tutorial](#vcwlkxmldocumentationtutorial) |
| Links to information about XML as it relates to Visual Studio feature areas | [XML in Visual Studio](http://msdn.microsoft.com/library/en-us/vsintro7/html/vxorixmlinvisualstudio.asp) |

### Recommended Tags for Documentation Comments

C# gives the capability to document your code while writing the code. Developer would be able to generate the documentation of his or her program by including specific tags in the program.

In source code files, lines that begin with /// and that precede a user-defined type such as a class, delegate, or interface; member such as a field, event, property, or method; or a namespace declaration can be processed as comments and placed in a file.

The C# compiler will process documentation comments in your code to an XML file. ~~Processing the XML file to create documentation can be implemented using the NDoc tool (see~~ [~~http://ndoc.sourceforge.net~~](http://ndoc.sourceforge.net) ~~).~~

Tags are processed on code constructs such as types and type members.

**Note**Tags are not processed on namespaces.

The compiler will process any tag that is valid XML. The following tags provide commonly used functionality in user documentation:

|  |  |  |
| --- | --- | --- |
| [<c>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfc.asp) | [<para>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfpara.asp) | [<see>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfsee.asp)1 |
| [<code>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfcode.asp) | [<param>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfparam.asp)1 | [<seealso>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfseealso.asp)1 |
| [<example>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfexample.asp) | [<paramref>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfparamref.asp)1 | [<summary>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfsummary.asp) |
| [<exception>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfexception.asp)1 | [<permission>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfpermission.asp)1 | [<value>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfvalue.asp) |
| [<include>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfinclude.asp)1 | [<remarks>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfremarks.asp) | [<typeparam>](http://msdn.microsoft.com/en-us/library/ms173191%28VS.80%29.aspx) |
| [<list>](http://msdn.microsoft.com/library/en-us/csref/html/vclrflist.asp) | [<returns>](http://msdn.microsoft.com/library/en-us/csref/html/vclrfreturns.asp) | [<typeparamref>](http://msdn.microsoft.com/en-us/library/ms173192%28VS.80%29.aspx) |

## Interface comments

Interfaces should be well documented. Copious comments should be applied to interfaces that define boundaries between systems and logical application tiers.

## Class Level Comment

After creating the class, generate the class level documentation by typing /// and Visual Studio.NET will create a stub for comments. From this stub, remember to add the following:

* Remarks – use the <remarks> tag to add a more lengthy description of the class.
* Example – use the <example> tag to add an example of how to use the class.

After populating the stub generated by Visual Studio.NET and adding additional comments, the class level comments should be similar to the following.

|  |
| --- |
| /// <summary>short summary description for theclass</summary>  /// <remarks>more lengthy description of the classhere</remarks>  /// <example> description of the example  /// <code>  /// sample code showing how to access this class  /// </code>  /// </example> |

## Method Level Comment

After creating the method signature, generate the method level documentation by typing /// and Visual Studio.NET will create a stub for comments. From this stub, remember to add the following:

* Always dictate whether the method attempts to encapsulate exceptions for most known failure points or if the function makes no effort to trap errors.
* Always specify the conditions for an exception explicitly stated to be thrown.
* Always document the impact (state change) to the owning object.
* If applicable, describe synchronization issues if the method touches shared data

After populating the stub generated by Visual Studio.NET and adding additional comments, the method level comments should be similar to the following.

|  |
| --- |
| /// <summary>short description of the method here</summary>  /// <param name="a">short description of parameter a</param>  /// <param name="b">short description of parameter b</param>  /// <returns>short description of any returned value</returns>  /// <exception cref="reference to an exception type">  /// condition which the exception will be thrown  /// </exception>  /// <remarks>a more lengthy description of method here</remarks> |

## 

## Property Level Comment

After creating the property, generate the property level documentation by typing /// and Visual Studio.NET will create a stub for comments.

|  |
| --- |
| /// <summary>short description of property</summary>  /// <value>description of the property value here</value>  /// <exception cref="reference to an exception type">  /// condition which the exception will be thrown  /// </exception>  /// <remarks> more lengthy description of property</remarks> |

# Formatting

## Matching Braces

This topic is open to regional change. It is only necessary that the spirit of this rule is enforced: create consistent, readable code for your organization. It is possible to employ code beautifiers/formatters (lint, jindent, etc.) to standardize inside of your organization. A reasonable practice is to couple the code beautifying, automated review and check-in process into a single script or macro.

Align matching braces vertically, with no code on the same line as a brace.

|  |
| --- |
| // GOOD  try  {  if (condition)  {  ++counter;  }  }  catch (Exception e)  {  // log  }  finally  {  // exiting  }  // BAD (the brace alignment is inconsistent)  try {  if (condition)  {  ++counter;  }  }  catch (Exception e) {  // log  }  finally  {  // exiting  } |

## 

## Indentation

* Set indentations to four (4) spaces. If applicable, set the **Keep Tabs** option in Visual Studio.NET IDE(preferred method). Tools--> Options; select "Text Editor"; select "C#"; select "Tabs"; ensure "Tab size" = 4 and "Indent size" = 4; "Keep tabs" radio button should be selected. This is the default setting in Visual Studio.NET IDE.

### Line Length

* Limit lines to 80 characters in length. This rule is open for regional extension. The possible alternative for the regions is a 100 character maximum.
* Limit code examples in documentation to 70 characters in length
* Wrap lines in the following manner:
* Break a line after a comma
* Break a line after an operator

### Wrapping Lines

When an expression will not fit on a single line, break it according to these 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.

|  |
| --- |
| // GOOD  someMethod(longArgument1, longArgument2, longArgument3,  longArgument4, longArgument5, longArgument6);  someMethod(longArgument1,  longArgument2,  longArgument3,  longArgument4,  longArgument5,  longArgument6);  longName1 = longName2 \* (longName3 + longName4 - longName5)  + 4 \* longName6;  // BAD  someMethod(longArgument1, longArgument2, longArgument3, long…  longName1 = longName2 \* (longName3 + longName4  – longName5) + 4 \* longName6; |

## White Spaces

### Blank Spaces

* Use a single space after a comma (e.g. insert blank spaces after commas in argument lists).

|  |
| --- |
| // GOOD  TestMethod(a, b, c)  // BAD  TestMethod(a,b,c) |

Surround operators with a single space. Pay careful attention to for statement declarations for readability.

|  |
| --- |
| // GOOD  a = b;  for (int i = 0; i < 10; i++)  // BAD  a=b;  for(int i=0;i<10;i++) |

### 

### Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Blank lines (one or two) should always be used in the following cases:

* Between regions/sections of a source file
* Between class and interface definitions
* Between methods
* Between the local variables in a method and its first statement
* Between logical sections inside a method to improve readability

## Method Parameters

If the parameter list exceeds the screen width then place each method parameter on a separate line.

|  |
| --- |
| // GOOD  public void MethodOne(  int first,  int second)…  public void MethodTwo(  int first,  int second,  int third)…  public void MethodWithOneParameter(int first)…  public void MethodOne(int first, int second)  // BAD – these will exceed screen width  public void MethodOne(int firstLongArgument, int secondLongArg…)  public void MethodTwo(int first, int second, int third, int …) |

## Member and Local Variables

* Place only one declaration per line

|  |
| --- |
| // GOOD  int firstNumber = 1;  int secondNumber = 2;  // BAD  int firstNumber = 1, secondNumber = 2; |

Explicitly initialize each value type variable to the intended default value. Avoid redundant initialization in the constructor. Don’t let the system handle initialization for you as it can lead to unexpected behavior and results for your application.

|  |
| --- |
| // GOOD  int someVariable = 0;  bool someFlag = true;  // BAD  int someVariable;  bool someFlag; |

DDo not abbreviate (except for very well known acronyms like TCP or UI)

|  |
| --- |
| // GOOD  int firstNumber = 1;  int tcpRetryCount = 2;  long startTime = 2;  // BAD  int firstNum = 1  long startTm = 2; |

## Statements

### Simple Statements

Each line should contain at most one statement.

|  |
| --- |
| // GOOD  someNumber++;  anotherNumber--;  // BAD  someNumber++; anotherNumber--; |

### Conditional Statements

Consider the code examples below. Although they are both equivalent, at least on first inspection, the code on the left compiles and the code on the right does not. Why? Because the second if statement isn’t doing a comparison, it’s doing assignment – you can’t assign a new value to a constant value such as 0. This can be a difficult bug to find in your code (at least without a sophisticated testing tool). By placing constants on the left side of comparisons you achieve the same effect and the compiler will catch it if you accidentally use assignment instead of comparison.

|  |  |
| --- | --- |
| // compiles OK  if (something == 1) {…}  if (x = 0) {…} | // does NOT compile  if (1 == something) {…}  if (0 = x) {…} |

Do not use the ternary operator.

|  |
| --- |
| // BAD  someVariable = condition ? TRUE\_VALUE : FALSE\_VALUE;  // GOOD  if (condition)  {  someVariable = TRUE\_VALUE;  }  else  {  someVariable = FALSE\_VALUE;  } |

// BAD

someVa

If-else statements ALWAYS use braces. Place braces around a single-line expression that follows a conditional.

|  |
| --- |
| // GOOD  if (condition == true)  {  counter++;  }  else  {  counter--;  }  // BAD  if (condition == true)  counter++;  else  counter--; |

### Looping Statements

Loop statements (While, For and Do) ALWAYS use braces

|  |
| --- |
| // GOOD  for (int i = 0; i < 10; i++)  {  counter--;  }  // BAD  for (int i = 0; i < 10; i++)  counter--; |

### Switch Statements

Always note when a case falls through to the next case. Preference is to avoid fall through scenarios by using a break statement in all cases. Every switch statement should include a default case. The default case should be listed last in order.

|  |
| --- |
| // GOOD – provide a comment like below  switch (someVariable)  {  // NOTE: case 1 falls thru to case 2  case CONSTANT\_1:  case CONSTANT\_2:  anotherVariable = someMethod(CONSTANT\_2);  break;  default:  anotherVariable = someMethod(CONSTANT\_DEFAULT);  } // end switch |

# Naming Conventions

## Capitalization Styles

Consider the following example for capitalization styles.

|  |
| --- |
| // Pascal Casing  BackColor  // Camel Casing  backColor  // Constant Casing  BACK\_COLOR |

### Pascal Casing

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

### Camel Casing

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

### Constant Casing

All letters are capitalized, and the underscore is used to separate words.

## General Naming Guidelines

### Best Practice

* Use full US-English descriptors that accurately describe the code element such as employeeName, orderAmount, or CorporateCustomer. Short names should only be used as control variables for a loop. The exception is when a short variable is still meaningful. For example, use x and y when performing a square-foot calculation.
* Use terminology applicable to the domain. If your users refer to their clients as customers, use the term Customer for the class, not Client. Many developers mistakenly create generic terms for Concepts when perfectly good terms already exist in the industry/domain.
* Difficulty in selecting a proper name may indicate that you need to further analyze or define the purpose of an item.
* Do not use Hungarian notation.
* Do not use underscore characters to separate words in names (this is what mixed case is used for). The exception is scope designation of class and instance members (a prefixed underscore).
* Avoid names longer than 20 characters.
* Do not use letters that can be mistaken for digits and vice versa.

|  |
| --- |
| // BAD  // this looks like obfuscated code  bool b001 = (lo == l0) ? (I1 == 11) : (lOl != 101); |

### Acronyms and Abbreviations

Generally, avoid abbreviations if possible.

Acronyms and abbreviations used in class, interface, method, member, and local variable names will only have the first letter capitalized. Two-letter acronyms, abbreviations, or trade marked names can have all characters capitalized if used alone. Those used in conjunction with other words will follow Pascal casing rules.

|  |
| --- |
| // GOOD  HP.Ecommerce.UI  TcpClient  HpService  // BAD  ABC.Ecommerce.UI  TCPClient  HPService |

### Case Sensitivity

Do not use names that require case sensitivity. Components 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.

## Quick Reference for Each Identifier Type

Please Print and Use the following table to associate rules and styles with the appropriate identifier as a quick reference guide.

|  |  |  |
| --- | --- | --- |
| Identifier | Case | Notes |
| Class | PascalCase | * **DO** use a noun or noun phrase * **DO** use names that are domain specific. Rely on the subject expert to give proper names. * **DO** use similar names when defining a class/interface pair (IComponent and Component) * Do **NOT** use names that conflict with the namespace * Do **NOT** use the C prefix for class names (use FileStream rather than CFileStream) * Do **NOT** use the underscore in a class name |
| Interface | PascalCase | * **DO** usenouns or noun phrases, or adjectives describing behavior * **DO** usea prefix of I (ISerializable) followed by capitalizing the letter following the prefixed I. * **DO** usesimilar names when defining a class/interface pair (IComponent and Component) |
| Attribute Class | PascalCase | * **DO** usea suffix of *Attribute* |
| Exception Class | PascalCase | * **DO** usea suffix of *Exception* |
| Constant | ConstantCase | * **DO** useconstants in lieu of inline literals |
| Enumerations | PascalCase | * **DO** useenums when possible for parameters or properties * **DO** usea singular name * **DO** usea plural name for bit fields * **DO** usethe Flags custom attribute if the numeric values are meant to be bitwise ORed together * Do **NOT** use a family-name prefix * Do **NOT** use an *Enum* suffix * Do **NOT** use enums for open sets |
| Event, EventArgs, Delegates | PascalCase | * **DO** usea verb or verb phrase to name an event * **DO** use*EventHandler* as a suffix * **DO** usea 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 the post-event. * **DO** usethe EventArgs suffix for event argument classes * **DO** usetwo parameters named sender and e (object that raised the event which is always of type object, and an event class) * Do **NOT** use BeforeXxx/AfterXxx pattern. * In general, you should provide a protected method called OnXxx on types with events that can be overridden in a derived class. This methods should have the event parameter *e* only, as the sender is always the instance of the type. |
| Instance variable | camelCase | * **DO** usean underscore for instance members as a scope identifier |
| Local variable | camelCase | * **DO** usecomputation qualifiers (Avg, Sum, Min, Max, Index) to the end of a variable name where appropriate * **DO** usea protected property instead of a protected member * Do **NOT** use a family-name prefix |
| Method | PascalCase | * **DO** usea verb or verb phrase * **DO** useverb-object order instead of object-verb order * **DO** usenames that describe *What* the behavior is, not *How* the behavior is implemented. * **DO** usea name that describes the return value if a method has a return value |
| Namespace | PascalCase | * **DO** use plural namespaces where appropriate (to avoid conflicting class and namespace names) * See Namespace Taxonomy Section for more information * Use the following nomenclature   HP.<Function>.<Region>.<Platform>.<Vertical>.<Layer>.<Domain>.<Sub-Domain> |
| Property | PascalCase | * **DO** usea noun or noun phrase * Do **NOT** use the class name redundantly in the property name. Use Book.Title instead of Book.BookTitle (where Book is the class). |
| Public Instance Field | PascalCase | * **Rarely** used (use a property instead) * **DO** usea public property instead of a public member field |
| Protected Instance Field | camelCase | * **Rarely** used (use a property instead) * **DO** usea protected property instead of a protected member field |
| Parameter | camelCase | * **DO** usedescriptive names such that a parameter’s name and type clearly imply its meaning. Prefer names based on a parameter’s meaning, to names based on a parameter’s type. * Do **NOT** reserve parameters for future use. If more are needed in the future an overload can be added |
| User Interface & Control instance variables | camelCase | * **DO** use the control-type name as a suffix (e.g. dataEntryForm, firstNameTextBox) |

## Naming Examples for each Identifier

Please Print and Use the following examples for a quick reference.

|  |
| --- |
| // Class  public class Cart  public class WishList |

|  |
| --- |
| // Interface  public interface IPublisher  public interface IManagerFactory  // Methods and Paramters  public IList GetCustomers()  public Employee GetEmployee(int employeeNumber)  public void UpdateEmployee(string firstName)  // Property  public string FirstName { … }  // Constants  public const int HTTP\_PORT = 80;  public const string HTTP\_SERVER = “www.abc.com”;  // Delegate and EventArgs  public delegate void MouseEventHandler(object sender,  MouseEventArgs args)  public class MouseEventArgs : EventArgs  // Enum  public enum PrimaryColors  {  Red,  Green,  Blue  }  // Instance Members  private IList \_employees = null;  private static object \_instance = null;  // Exception classes  public class ReallyUnexpectedException : Exception  public class CartNotExistException : Exception  // Attribute class  public class MySampleAttribute |

|  |
| --- |
| **Single Source Code Standards** |
| **Layer Architecture**  With single source code in mind, the concept of thick Manager, thin UI and DAC have vanished and the design would be done the other way. UI will have access to bulk of the data transformed in a uniform way |
|  |
| UI |
| 1. All variable user controls will be shown based on flag ‘X’ (name to consistent with user control to be displayed) and the same flag X should be used to assign that controls’ value to the Value Object. 2. CSS file to be hosted in ISCS Shared project so that all SF deployments can use the same file. 3. UI Project’s Request Interceptor to be loaded from web’s config file. All page redirection in case of error or success method call needs to happen from this class. 4. Applying transformation to happen in this layer. 5. Access to API to happen from web form page and not web user control. 6. If control is not used in more than one place, it should not exist. 7. Data to control to be passed through properties or method calls where it would bind it. The neat implementation would be to set the data through properties and on control’s load event, if data is not available than the no data available would be displayed. 8. Control can access resource files as they are UI related. 9. Resource file to host UI element label values only and all data for the actual values should come from DB. 10. Common English values to be hosted in Common Resource file and deployment specific values are to be in deployment resource file. 11. Any changes to application Config file for future deployments will need to be added with correct value in previous deployment Config files |
|  |
| **Controller** |
| 1. For changing methods and new methods, please make sure the only object that be read from IscsContext is Application. All other properties like locale, country and catalog id are to be passed as parameters. 2. Controller cannot make multiple calls to Director in same method. 3. Controller to use partial class methodology to implement classes having too many methods. |
|  |
| **API – Business** |
| 1. All calls to Manager from Director to go through CreateManager implementation. 2. Integration Specific calls to have try catch blocks and all catch blocks to implement Error Logging using ILogger in Common project. 3. Director can call different manager in same method. 4. No web service references allowed within this project. 5. No localized content to be returned from ISCS API layer (except for those coming in from Integration services). 6. Seal all concrete implementation classes 7. Reduce the overload method concept in this layer 8. Manager cannot call other manager. |
|  |
| **API – Data Access** |
| 1. All access to data should happen only in this layer (like access to DB, File System, Web Services). 2. Always use Using statement to call Data Base. If the method knows how to handle the exception, than it implement try-catch within using block. 3. No calls to other DAC classes. 4. All integration calls to have try-catch block to log errors. |

# APPENDIX

**APPENDIX A - C# XML Domumentation Tag**

|  |  |  |
| --- | --- | --- |
|  | Tag | Used For |
| 1. | <c> | This gives you a way to indicate that text within a description should be marked as code |
| 2. | <para> | The <para> tag is for use inside a tag, such as <remarks> or <returns>, and lets you add structure to the text. |
| 3. | <see> | 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. |
| 4. | <code> | 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. |
| 5. | <param> | The <param> tag should be used in the comment for a method declaration to describe one of the parameters for the method. |
| 6. | <seealso> | 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. |
| 7. | <example> | 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. |
| 8. | <paramref> | 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. |
| 9. | <summary> | The <summary> tag should be used to describe a member for a type. Use <remarks> to supply information about the type itself. |
| 10. | <exception> | The <exception> tag lets you specify which exceptions a class can throw. |
| 11. | <permission> | The <permission> tag lets you document the access of a member. The System.Security.PermissionSet lets you specify access to a member. |
| 12. | <value> | 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. |
| 13. | <include> | 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. |
| 14. | <remarks> | The <remarks> tag is where you can specify overview information about a class or other type. <summary> is where you can describe the members of the type. |
| 15. | <list> | 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*. |
| 16. | <returns> | The <returns> tag should be used in the comment for a method declaration to describe the return value. |
| 17. | <typeparam> | The <typeparam> tag should be used in the comment for a generic type or method declaration to describe a type parameter. Add a tag for each type parameter of the generic type or method. |
| 18. | <typeparamref> | For more information on type parameters in generic types and methods. |