

IMPLEMENTATION PRACTICES AND STANDARDS

From TeamF::CGT

Revision: 3229

Contents

- 1 Abstract
- 2 Introduction
 - 2.1 Objective
 - 2.2 Scope
 - 2.3 Audience
 - 2.4 Terminology and Definitions
 - 2.5 References
- 3 Development Guidelines
 - 3.1 Development Methodology
 - 3.2 Code Inspection
- 4 C# Coding Standards
 - 4.1 Source File Organization
 - 4.1.1 Class and Source File
 - 4.1.2 Ordering
 - 4.1.3 XML Documentation
 - 4.1.4 Class and Interface Declaration
 - 4.2 Formatting
 - 4.2.1 Document Formatting in Visual Studio 2005
 - 4.2.2 General Rules
 - 4.2.3 Indentation
 - 4.2.4 Line Length
 - 4.2.5 Wrapping Lines
 - 4.3 Commenting
 - 4.3.1 Types of Comments
 - 4.3.2 General Rules
 - 4.3.3 Documentation Comments
 - 4.3.4 Comment Tokens
 - 4.4 Declarations
 - 4.4.1 One Declaration Per Line
 - 4.4.2 Initialization
 - 4.4.3 Placement
 - 4.5 Naming Conventions
 - 4.5.1 General Rules
 - 4.5.2 Namespaces
 - 4.5.3 Classes
 - 4.5.4 Generic Classes
 - 4.5.5 Exception Classes
 - 4.5.6 Interfaces
 - 4.5.7 Methods
 - 4.5.8 Variables
 - 4.5.9 Properties
 - 4.5.10 Objects References
 - 4.5.11 Constants
 - 4.5.12 Enum
 - 4.5.13 Data
 - 4.5.14 Abbreviations
 - 4.5.15 Capitalisation Summary
- 5 Programming Practices
 - 5.1 Visibility and Access Control
 - 5.2 Object Model
 - 5.3 Overriding
 - 5.4 Destructors and Finalizers
 - 5.5 Literals
 - 5.6 Variable Assignments
 - 5.7 Parameters
 - 5.8 Exceptions
 - 5.8.1 Exception Throwing
 - 5.8.2 Exception Handling
 - 5.8.3 Custom Exceptions

- 5.9 Returning Values
- 5.10 Avoid Excessive Nesting Using Guard Clause
- 5.11 Parentheses
- 5.12 Regions
- 5.13 Refactoring
- 6 Appendix
 - 6.1 Documentation Comments Sample Code

1 Abstract

This document requires or recommends certain practices and standards for developing the Corporate Governace Toolkit (CGT) software system using Visual C# 2.0, ASP.NET 2.0 and related technologies.

2 Introduction

2.1 Objective

The objective of this practices and standards document is:

- Avoidance of obvious or not-so-obvious errors/bugs due to casual programming styles
- Maintainability, by promoting some proven design principles
- Maintainability, by requiring or recommending a certain unity of style
- Performance, by dissuading wasteful practices

The document, however, does not aim to constrain on every aspect of coding habits. Instead, only those of higher importance are documented.

2.2 Scope

This document mainly focuses on the C# Language. It also mentions related .NET technologies that are deemed necessary or useful to the project. Although the C# language is implemented alongside the .NET Framework, this document does not address usage of .NET Framework class libraries. However, common patterns and problems related to C#'s usage of the .NET Framework are addressed in a limited fashion.

2.3 Audience

This document is supposed to be used by:

- Developers
- Testers

However, since each team member is required to be involved in at least implementation or testing, the audience of the document is the whole team.

2.4 Terminology and Definitions

Term	Definition	Examples
Access Modifier	A class-member-declaration can have any one of the five possible kinds of declared accessibility: public, protected internal, protected, internal, or private. Except for the protected internal combination, it is a compile-time error to specify more than one access modifier. Although default access modifiers vary, classes and most other members use the default of private. Notable exceptions are interfaces and enums which both default to public.	
Identifier	A developer defined token used to uniquely name a declared object or object instance.	
Magic Number	Any numeric literal used within an expression (or to initialize a variable) that does not have an obvious or well-known meaning. This usually excludes the integers 0 or 1 and any other numeric equivalent precision that evaluates as zero.	
Hungarian Notation	A variable name starts with one or more lower-case letters which are mnemonics for the type or purpose of that variable, followed by whatever the name the programmer has chosen; this last part is sometimes distinguished as the given name. The first character of the given name is capitalized to separate it from the type indicators.	lblSampleLabel txtTextField
Camel Case	A word with the first letter lowercase, and the first letter of each subsequent word-part capitalized.	loopCounter

2.5 References

1. C# Language Specification 1.2
2. C# Language Specification 2.0, March 2005 Draft
3. Design Guidelines for Class Library Developers
4. Coding Standard: C# - Philips Medical Systems - Software / SPI
5. C# Coding Standards for .NET

3 Development Guidelines

3.1 Development Methodology

The implementation team employs a pair-programming-by-request strategy, i.e., pair programming in its strict sense is neither encouraged nor discouraged. Instead, the choice of whether or not pair programming is conducted by a sub-team of developers who are dedicated to a particular build task is left to the developers themselves. One or more of the developers can pair up voluntarily or request to the manager to be paired up.

The decision of whether pair programming should be adopted needs to take the following into consideration:

- Type and difficulty of programming task
- Time availability of developers in question
- Expertise of the pair of developers in question
- Willingness of the developers in question to be paired up

Nevertheless, working in pairs is strongly encouraged, especially in the following two situations:

- Two developers work together during the stage of subsystem integration
- A developer and a tester work together to ensure testing is done in time and on the spot

Even though pair programming is not enforced nor encouraged, we recognise the benefits of having two developers working closely together towards a particular goal, especially when there is a general lack of expertise in a particular field. As a result, implementation task allocation will be based on the interest and skill survey that each developer has filled in early on. The management team will try to match up those with plenty of experience with those who are really interested in the field, so that hopefully there will be a lot of learning and pushing going on in the sub-team.

3.2 Code Inspection

Code inspection, just like coding itself, is to be done on a rotational basis. Team members currently free from any implementation or testing duties are candidates for conducting a code inspection.

Code inspection should happen on a weekly basis, or as soon as a build is finished, or when a milestone has been reached.

Code inspection should mainly focus on coding styles and legibility of code and comments, although the inspector is also welcome to point out any likely logical errors that have not been detected by unit test cases.

The inspection report should be mainly checklist based. At the end of the inspection, the inspector is expected to produce a checklist report and also needs to make sure all his findings are conveyed to the responsible developers and/or implementation team manager.

The inspection report will at least need to include the following items:

1. Compilation errors / Obvious bugs
2. Code Styles
3. Documentation / Code legibility
 1. XML documentation comments
 2. Normal comments
4. Other recommendations

For each item, the following may need to be specified:

- Inspection target
- Error type
- List of errors of that type
- Suggestions for improvement

The inspectors need to fully understand this standards document before assessing the quality of the target code, and show references to this document wherever applicable in the inspection report.

4 C# Coding Standards

4.1 Source File Organization

4.1.1 Class and Source File

Normally, one class definition goes into one source file only. Said differently, each class definition will exist within its own file. With partial classes in .NET Framework 2.0, however, one class may consist of several partial classes that reside in different files.

In either cases, two different classes should not be put into one file, and the stem of the file name must be the same name as the name used in the class declaration. Examples are:

- A class called *Sample* is defined in a file called "*Sample.cs*"
- Two parts of a partial class called *PartialSample* are defined in "*PartialSample.properties.cs*" and "*PartialSample.methods.cs*" if one contains the properties and the other contains methods, or "*PartialSample.isampleinterface1.cs*" and "*PartialSample.isampleinterface2.cs*" if one implements the methods of *ISampleInterface1* and the other implements the methods of *ISampleInterface2*. However, the use of partial classes for the purpose other than separating IDE generated code and developer code as in Visual Studio .NET 2005 is **strongly discouraged**.

4.1.2 Ordering

C# source files have the following ordering:

1. *using* statements
2. *namespace* statement
3. *class* and *interface* declarations

4.1.3 XML Documentation

XML documentation is provided for descriptions of classes, properties and methods. XML documentation should be used wherever it is applicable.

Refer to Documentation Comments for more information.

4.1.4 Class and Interface Declaration

Sequence	Part of Class/Interface Declaration	Notes
1	Class/interface documentation	<pre>/// <summary> /// The Sample class provides ... /// </summary> public class Sample</pre>
2	<i>class</i> or <i>interface</i> statement	
3	Fields	First private, then protected, then internal, and then public.
4	Properties	First private, then protected, then internal, and then public.
5	Constructors	First private, then protected, then internal, and then public. Default first, then order in increasing complexity.
6	Methods	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.

4.2 Formatting

4.2.1 Document Formatting in Visual Studio 2005

Automatic document formatting functionality is very useful when it comes to enforcing consistent code format in Visual Studio 2005. A tentative settings file for Visual Studio 2005 can be accessed from STANDARDS-vs2005settings.vsettings and can be imported to individual installations.

4.2.2 General Rules

Rule	Preferred Style	Bad Style
Indent block contents	<pre>class MyClass { int Method() { return 3; } }</pre>	<pre>class MyClass { int Method() { return 3; } }</pre>
Indent <i>case</i> labels	<pre>switch (name) { case "John": break; }</pre>	<pre>switch (name) { case "John": break; }</pre>
Indent <i>case</i> contents	<pre>switch (name) { case "John": break; }</pre>	<pre>switch (name) { case "John": break; }</pre>
Place open brace on new line for types	<pre>class MyClass { ... }</pre>	<pre>class MyClass { ... }</pre>
Place open brace on new line for methods	<pre>int Method() { ... }</pre>	<pre>int Method() { ... }</pre>
Place open brace on new line for control blocks (<i>if</i> , <i>while</i> , <i>for</i> , <i>foreach</i> , etc.)	<pre>if (a > b) { ... }</pre>	<pre>if (a > b) { ... }</pre>
Place <i>else</i> on new line	<pre>if (a > b) { ... } else { ... }</pre>	<pre>if (a > b) { ... } else { ... }</pre>
Place <i>catch</i> on new line	<pre>try { ... } catch (Exception e) { ... }</pre>	<pre>try { ... } catch (Exception e) { ... }</pre>
Place <i>finally</i> on new line	<pre>try { ... } finally { ... }</pre>	<pre>try { ... } finally { ... }</pre>

	<pre> ... } </pre>	<pre> } </pre>
Insert space before and after binary operators (+, -, etc.)	<pre>int result = 1 + 2 * 3;</pre>	<pre>int result = 1+2*3; int result = 1 + 2*3;</pre>
Methods are separated by two blank lines	<pre> int Method1() { ... } int Method2() { ... } </pre>	<pre> int Method1() { ... } int Method2() { ... } </pre>
Each line should contain only one member declaration	<pre>private int a; private int b;</pre>	<pre>private int a, b;</pre>
Each line should contain only one statement	<pre>size++; count--;</pre>	<pre>size++; count--;</pre>

4.2.3 Indentation

Indentation is constructed with spaces rather than tabs. 2 spaces are used in place of a tab.

Visual Studio editors and formatters should be configured to insert spaces with a width of 2 characters for each tab typed in.

4.2.4 Line Length

80 characters per line is recommended, although it is not a hard rule. If a line of code becomes too long, consider breaking it down into several statements.

4.2.5 Wrapping Lines

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

- Break after an operator
- Break after a comma
- Prefer higher-level breaks to lower-level breaks

Examples are:

Preferred Style	Bad Style
<pre>longMethodCall(expr1, expr2, expr3, expr4, expr5);</pre>	<pre>longMethodCall(expr1, expr2 , expr3, expr4, expr5);</pre>
<pre>var = a * b / (c - g + f) + 4 * z;</pre>	<pre>var = a * b / (c - g + f) + 4 * z;</pre>

4.3 Commenting

4.3.1 Types of Comments

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

For implementation comments, use // but **never** /* ... */.

Documentation comments are C# only, and are delimited by special XML tags that can be extracted to external files for use in system documentation. This type of comments is discussed in detail in a later section.

4.3.2 General Rules

Implementation comments are meant for commenting out code or for comments about the particular implementation. Documentation 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.

Some general rules and recommendations for code commenting are:

- Use `//` and `///` but **never** `/* ... */`.
- 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 component is built or in what directory it resides should not be included as a comment.
- Discussion of nontrivial or obscure design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code.
- The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment for a complex section of code, consider rewriting the code to make it clearer.
- When modifying code, always keep the commenting around it **up to date**. 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.
- Comments should consist of complete sentences and follow **active** language naming responsibilities (*Adds the element* instead of *The element is added*).
- Always use comments on bug fixes and work-around code to prevent recurring problems.
- Use comments on code that consists of loops and logic branches. These are key areas that will assist the reader when reading source code.
- Separate comments from comment delimiters with white space (`// comment` instead of `//comment`).
- Throughout the application, construct comments using a uniform style, with consistent punctuation and structure.
- Avoid adding comments at the end of a line of code; end-line comments make code more difficult to read. However, end-line comments may be used when annotating variable declarations. In this case, align all end-line comments at a common tab stop.
- Avoid using clutter comments, such as an entire line of asterisks (`// *****`). Instead, use white space to separate comments from code.
- Prior to committing changes to repository or prior to deployment, remove all temporary or extraneous comments to avoid confusion during future maintenance work.
- Comments should never include special characters such as form-feed and backspace.

4.3.3 Documentation Comments

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

XML documentation is **required** for classes, delegates, interfaces, events, methods, and properties. Include XML documentation for fields that are not immediately obvious. Document comments must **not** be positioned inside a method or constructor definition block, because C# associates documentation comments with the first declaration after the comment.

The documentation must be well-formed XML, and developers are not free to create their own set of tags. The available and commonly used XML documentation tags are:

Tag	Notes
<code><c></code>	The <code><c></code> tag gives you a way to indicate that text within a description should be marked as code. Use to indicate multiple lines as code.
<code><code></code>	The <code><code></code> tag gives you a way to indicate multiple lines as code. Use <code><c></code> to indicate single line as code.
<code><example></code>	The <code><example></code> tag lets you specify an example of how to use a method or other library member. Commonly, this would involve use of the <code><code></code> tag.
<code><exception></code>	The <code><exception></code> tag lets you document an exception class. Compiler verifies syntax.
<code><include></code>	The <code><include></code> 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 <code><include></code> tag uses the XML XPath syntax. Compiler verifies syntax.
<code><list></code>	The <code><listheader></code> 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 <code><item></code> 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.
<para>	The <para> tag is for use inside a tag, such as <remarks> or <returns>, and lets you add structure to the text.
<param>	The <param> tag should be used in the comment for a method declaration to describe one of the parameters for the method. Compiler verifies syntax.
	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. Compiler verifies syntax.
<permission>	The <permission> tag lets you document the access of a member. The System.Security.PermissionSet lets you specify access to a member.
<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.
<returns>	The <returns> tag should be used in the comment for a method declaration to describe the return value.
<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. Compiler verifies syntax.
<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.
<summary>	The <summary> tag should be used to describe a member for a type. Use <remarks> to supply information about the type itself. The <summary> tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.
<value>	The <value> tag lets you describe a property.

Refer to Documentation Comments Sample Code for more information.

4.3.4 Comment Tokens

There are three special comment tokens, namely TODO, HACK and UNDONE.

When you add comments with comment tokens to your code, you automatically add shortcuts to the Task List window (under "Comments"). Double-click any comment displayed in the Task List to move the insertion point directly to the line of code where the comment begins. Note: Comments in HTML, CSS and XML markup are not displayed in the Task List.

Frequent while appropriate use of comment tokens are highly recommended during development.

Example use of comment tokens is as follows:

```
// TODO: Fix this method.
// UNDONE: Need input from someone else.
// HACK: This method works but needs to be redesigned.
```

4.4 Declarations

4.4.1 One Declaration Per Line

There should only be one declaration per line.

Preferred Style	Bad Style
private int level = 2; private int size = 8;	private int level = 2, size = 8;

4.4.2 Initialization

If possible, try to initialise local variables as soon as they are declared. For example:

Preferred Style

```
string name = myObject.Name;
```

4.4.3 Placement

Generally, don't declare a variable until its first use, especially in for or foreach loops. Examples are:

Preferred Style
<pre>for (int i = 0; i < riskTable.Rows.Count; ++i) { // do something }</pre>
<pre>foreach (DataRow riskRow in riskTable.Rows) { // do something }</pre>

4.5 Naming Conventions

4.5.1 General Rules

- A name should tell "**what**" rather than "how". By avoiding names that expose the underlying implementation, which can change, a layer of abstraction that simplifies the complexity is preserved. For example, use *GetNextOrder()* instead of *GetNextArrayElement()*.
- Implementation details, in particular type specifications, should not be mentioned in the name of a descriptor. This is a common trait in procedural languages like Visual Basic where lowercase prefixes are used to encode the data type in the name of the identifier, such as *oInvoice*. This approach is not applicable to contemporary languages where the aforementioned identifier is written simply as *invoice*.
- Names of descriptors should be chosen in such a way that they can be read like a sentence within instructions.
- Minimise the use of abbreviations. If abbreviations are used, be consistent in their use. An abbreviation should have only one meaning and likewise, each abbreviated word should have only one abbreviation. For example, if using *min* to abbreviate *minimum*, do so everywhere and do not later use it to abbreviate *minute*.
- Follow Australian spelling rules.

4.5.2 Namespaces

Rule	Preferred Style	Bad Style
Namespaces must be of the format RiskWizard.Cgt.{CustomName} .	RiskWizard.Cgt.Configuration	RiskWizard.Configuration
Namespace names should be nouns, in Pascal case. Do NOT use underscores in namespace names.	RiskWizard.Cgt.DataAccess	riskwizard.cgt.configuration RiskWizard.Cgt.Data_Access

4.5.3 Classes

Rule	Preferred Style	Bad Style
Class names should be nouns, in Pascal case.	TreeControl	treeControl
Use whole words and avoid acronyms and abbreviations unless the abbreviation is much more widely used than the long form.	Url Html XmlWriter	XmlConf TreeVw

4.5.4 Generic Classes

Rule	Preferred Style	Bad Style
Generic class names should always use "T" or "K" as Type identifier, or identifier prefix followed by a number as in <i>T1</i> , <i>T2</i> .	GenericList<T> Generic3DView<T1, T2, T3>	GenericList<A> Generic3DView<X, Y, Z>

4.5.5 Exception Classes

Rule	Preferred Style	Bad Style
Exception class names should always have the suffix "Exception".	CorruptConfigurationException	CorruptConfiguration CorruptConfigurationError

4.5.6 Interfaces

Rule	Preferred Style	Bad Style
Interface names should begin with the letter "I".	IConnectionProxy	ConnectionProxy
Interface names should be nouns, in Pascal case.	ITreeView	IViewTree

4.5.7 Methods

Rule	Preferred Style	Bad Style
Method identifiers should contain active verbs or verb phrases, in Pascal case.	Connect() CreateTable()	connect() TableCreator()
Generally, do NOT include the noun name when the active verb refers directly to the containing class.	Socket.Open();	Socket.OpenSocket();
Avoid elusive names that are open to subjective interpretation.		Analyze()
Use the verb-noun form for naming methods that perform some operation on a given object.	CalculateInvoiceTotal()	
All overloaded methods with the same name should perform a similar function.		

4.5.8 Variables

Rule	Preferred Style	Bad Style
Variable names should be in camel case.	tableWidth	TableWidth
Do NOT use any special prefix characters to indicate that the variable is scoped to the class. Always use the <i>this</i> keyword when referring to members at a class's root scope from within a lower level scope.	this.name	_name m_name
Do NOT use Hungarian notation for variable names. Good names describe semantics, not type.	cancelButton	btnCancel
Prepend computation qualifiers (avg, sum, min, max, index) to the beginning of a variable name where appropriate.	avgRiskImpact	riskImpact
It is redundant to include class names in the names of class properties.	Book.Title	Book.BookTitle

Collections should be named as the plural form of the singular objects that the collection contains.	A collection of <i>Book</i> objects is named <i>Books</i> .	
Boolean variable names should contain "Is" or "is" which implies Yes/No or True/False values.	<code>isFound</code> <code>isSuccess</code>	<code>found</code> <code>success</code>
Avoid using terms such as Flag when naming status variables, which differ from Boolean variables in that they may have more than two possible values.	<code>orderFlag</code>	<code>orderStatus</code>
Even for a short-lived variable that may appear in only a few lines of code, still use a meaningful name. Use single-letter variable names such as <i>i</i> or <i>j</i> only for short-loop indexes.		
Constants should NOT be all uppercase with underscores between words. Constants follow the same naming rules as properties.	<code>NumDaysInWeek</code>	<code>NUM_DAYS_IN_WEEK</code>
Temporary variables should always be used for one purpose only. Otherwise, several variables should be declared.		

4.5.9 Properties

Rule	Preferred Style	Bad Style
Property names should be in Pascal case.	<code>public int RiskId</code>	<code>public string lastName</code>
Property names should directly reflect the underlying attribute.	<pre>private string firstName; public string FirstName { set { firstName = value; } }</pre>	<pre>private int riskImpact; public string Impact { get { return riskImpact; } }</pre>

4.5.10 Objects References

Rule	Preferred Style	Bad Style
Objects references should be in camel case when non-public and Pascal case when public (although references should never be public without any good reason).	<pre>Risk risk = new Risk(); public string Name;</pre>	<code>public string name;</code>
Generally, objects should be named after their class. An exception to this rule would be when two or more objects of the same class are needed within the same scope.	<pre>XmlWriter xmlWriter = new XmlWriter(); ... { Employee eric = new Employee("Huang", "Eric"); Employee sieming = new Employee("Huang", "Sie Ming"); } ...</pre>	<code>XmlWriter output = new XmlWriter();</code>
Avoid naming object references as abbreviations or acronyms of the class name.		<code>XmlWriter writer = new XmlWriter();</code>

4.5.11 Constants

Rule	Preferred Style	Bad Style
Constants are in Pascal case. They should NOT be all uppercase with words separated by underscores.	<code>const int NumDaysInWeek = 5;</code>	<code>const int NUM_DAYS_IN_WEEK = 5;</code>

4.5.12 Enum

Rule	Preferred Style	Bad Style
Enum types and values should be in Pascal case.	<code>enum Status { Blocked, ReadyToRun, Running };</code>	<code>enum status { blocked, readyToRun, running };</code>
Do NOT use an Enum suffix on Enum type names.	<code>enum Status</code>	<code>enum StatusEnum</code>
Generally, use a singular name for most Enum types.	<code>enum RiskType</code>	<code>enum RiskTypes</code>

4.5.13 Data

Rule	Preferred Style	Bad Style
When naming tables, express the name in the singular form.	<code>Employee</code>	<code>Employees</code>
When naming columns of tables, do not repeat the table name.	<code>LastName</code> in table <code>Employee</code>	<code>EmployeeLastName</code> in table <code>Employee</code>
Do NOT incorporate the data type in the name of a column. This will reduce the amount of work needed should it become necessary to change the data type later.	<code>Impact</code>	<code>ImpactDecimal</code>
Do NOT prefix stored procedures with <code>sp_</code> , because this prefix is reserved for identifying system-stored procedures.		

4.5.14 Abbreviations

Rule	Preferred Style	Bad Style
When using acronyms, use camel case for acronyms more than two characters long. However, acronyms that consist of only two characters should be capitalised.	<code>HtmlButton System.IO</code>	<code>HTMLButton System.io</code>
Do not use abbreviations in identifiers or parameter names.	<code>GetWindow</code>	<code>GetWin</code>
Do not use acronyms that are not generally accepted in the computing field.		

4.5.15 Capitalisation Summary

Type	Case	Notes
Project File	Pascal Case	
Source File	Pascal Case	
Other File	Pascal Case	
Namespace	Pascal Case	
Class/Struct	Pascal Case	

Interface	Pascal Case	Starts with I
Generic class	Pascal Case	Use T or K as Type identifier
Exception class	Pascal Case	Ends with Exception
Enum value	Pascal Case	
Delegate	Pascal Case	
Event	Pascal Case	
<i>public/protected</i> field	Pascal Case	
Method	Pascal Case	
Property	Pascal Case	
<i>private</i> field	Camel Case	
Parameter	Camel Case	
Inline Variable	Camel Case	

5 Programming Practices

5.1 Visibility and Access Control

Classes and most other members use the default of *private*. Interfaces and enums both default to *public*.

However, do NOT omit access modifiers. Explicitly declare all identifiers with the appropriate access modifier instead of allowing the default.

Do NOT make any instance or class variable *public* or *protected*, leave them as *private*.

Use properties instead. You may use *public static* or *const* fields as an exception to this rule, but it should better be avoided.

5.2 Object Model

Always prefer composition over inheritance.

Always prefer interfaces over abstract classes.

Try to append the design pattern name to class names where appropriate.

Make members *virtual* if they are designed and tested for extensibility.

Do NOT ever seal a class. *sealed* classes cannot be extended by others while the decision as to whether to seal a class or not is often too subjective to foresee future needs. Often, the only reason for sealing a class is library security. However, it does not apply to this project, where extensibility is one of the top priorities.

5.3 Overriding

Consider overriding *Equals()* on a *struct*.

Always override the Equality Operator (*==*) when overriding the *Equals()* method.

Always override the String Implicit Operator when overriding the *ToString()* method.

5.4 Destructors and Finalizers

Generally, do NOT use finalizers. Use destructors instead. Do NOT create *Finalize()* method. Examples are:

Preferred Style	Bad Style
<pre>~SampleClass() { ... }</pre>	<pre>void Finalizer() { ... }</pre>

Always call *Close()* or *Dispose()* on classes that offer it.

Wrap instantiation of *IDisposable* objects with a "using" statement to ensure that *Dispose()* is automatically called. For example:

Preferred Style

```
using (SqlConnection sqlConnection = new SqlConnection(this.connectionString))
{
    ...
}
```

Always implement the *IDisposable* interface and pattern on classes referencing external resources. For example:

Preferred Style

```
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        // Free managed objects
    }
    // Free unmanaged objects, i.e., your own state
    // Set large fields to null
}

// C# finalizer (optional)
~Base()
{
    Dispose(false);
}
```

5.5 Literals

Do NOT use magic numbers, i.e., place constant numerical values directly into the source code. The only exception is -1, 0 or 1, which can appear in a for loop as counter values.

5.6 Variable Assignments

Avoid assigning several variables to the same value in a single statement. For example, do NOT use:

Bad Style

```
oldRisk.Impact = newRisk.Impact = impact;
```

Do not use embedded assignments in an attempt to improve run-time performance. This is redundant. For example, do NOT use:

Bad Style

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

5.7 Parameters

Check the validity of parameter arguments. Perform argument validation for every *public* or *protected* method and property *set* accessor. Throw meaningful exceptions to the developer for invalid parameter arguments. Use specific subclasses of the *System.ArgumentException* class, or a class derived from *System.Exception*.

Avoid declaring methods with more than 7 parameters. Refactor or consider passing a *struct* or *class* instead.

5.8 Exceptions

5.8.1 Exception Throwing

Consider terminating the process by calling `System.Environment.FailFast(System.String)` (a .NET Framework version 2.0 feature) instead of throwing an exception, if your code encounters a situation where it is unsafe to continue executing.

Do not use exceptions for normal flow of control, if possible. Except for system failures and operations with potential race conditions, framework designers should design APIs so that users can write code that does not throw exceptions. For example, you can provide a way to check preconditions before calling a member so that users can write code that does not throw exceptions.

Do document all exceptions thrown by publicly callable members because of a violation of the member contract (rather than a system failure) and treat them as part of your contract. Exceptions that are a part of the contract should not change from one version to the next.

Do not have public members that return exceptions as the return value or as an out parameter. It is acceptable to use a private helper method to construct and initialize exceptions.

Consider using exception builder methods. It is common to throw the same exception from different places. To avoid code bloat, use helper methods that create exceptions and initialize their properties.

Avoid explicitly throwing exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

Consider throwing existing exceptions residing in the `System` namespaces instead of creating custom exception types.

Do create and throw custom exceptions if you have an error condition that can be programmatically handled in a different way than any other existing exceptions. Otherwise, throw one of the existing exceptions.

Do throw the most specific (the most derived) exception that is appropriate. For example, if a method receives a null argument, it should throw `System.ArgumentNullException` instead of its base type `System.ArgumentException`.

Do not throw `System.Exception` or `System.SystemException`. Do not catch `System.Exception` or `System.SystemException` in framework code, unless you intend to re-throw. Avoid catching `System.Exception` or `System.SystemException`, except in top-level exception handlers.

Do use value for the name of the implicit value parameter of property setters.

Do not allow publicly callable APIs to explicitly or implicitly throw `System.NullReferenceException`, `System.AccessViolationException`, `System.InvalidCastException`, or `System.IndexOutOfRangeException`. Do argument checking to avoid throwing these exceptions. Throwing these exceptions exposes implementation details of your method that may change over time.

Do not explicitly throw `System.StackOverflowException`. This exception should be explicitly thrown only by the common language runtime (CLR). Do not catch `System.StackOverflowException`.

Do not explicitly throw `System.OutOfMemoryException`. This exception should be thrown only by the CLR infrastructure.

5.8.2 Exception Handling

Do not swallow errors by catching non-specific exceptions, such as `System.Exception`, `System.SystemException`, and so on, in framework code.

Avoid swallowing errors by catching non-specific exceptions, such as `System.Exception`, `System.SystemException`, and so on, in application code. There are cases when swallowing errors in applications is acceptable, but such cases are rare.

Do not exclude any special exceptions when catching for the purpose of transferring exceptions.

Consider catching specific exceptions when you understand why it will be thrown in a given context.

Do not overuse `catch`. Exceptions should often be allowed to propagate up the call stack.

Do use `try-finally` and avoid using `try-catch` for cleanup code. In well-written exception code, `try-finally` is far more common than `try-catch`.

Do prefer using an empty throw when catching and re-throwing an exception. This is the best way to preserve the exception call stack.

5.8.3 Custom Exceptions

Avoid deep exception hierarchies.

Do derive exceptions from `System.Exception` or one of the other common base exceptions.

Do end exception class names with the `Exception` suffix.

Do make exceptions serializable. An exception must be serializable to work correctly across application domain and remoting boundaries.

Do provide (at least) the following common constructors on all exceptions. Make sure the names and types of the parameters are the same as those used in the following code example.

```
public class NewException : BaseException, ISerializable
{
    public NewException()
    {
        // Add implementation.
    }
    public NewException(string message)
    {
        // Add implementation.
    }
}
```

```

}
public NewException(string message, Exception inner)
{
    // Add implementation.
}

// This constructor is needed for serialization.
protected NewException(SerializationInfo info, StreamingContext context)
{
    // Add implementation.
}
}

```

Do report security-sensitive information through an override of System.Object.ToString only after demanding an appropriate permission. If the permission demand fails, return a string that does not include the security-sensitive information.

Do store useful security-sensitive information in private exception state. Ensure that only trusted code can get the information.

Consider providing exception properties for programmatic access to extra information (besides the message string) relevant to the exception.

5.9 Returning Values

Try to make the structure of your program match the intent. For example:

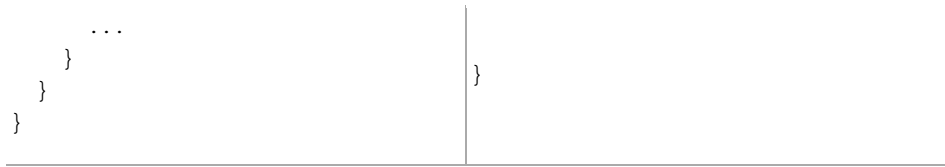
Preferred Style	Bad Style
<pre>return condition;</pre>	<pre>if (condition) { return true; } else { return false; }</pre>
<pre>return (condition ? x : y);</pre>	<pre>if (condition) { return x; } return y;</pre>

5.10 Avoid Excessive Nesting Using Guard Clause

Nesting happens when one control structure exists within another control structure, and possibly even another control structure. When reading code that resides within many nested blocks, the programmer must maintain an awareness of the pre-conditions that lead to the code being executed. Nesting becomes increasingly more ambiguous near the end of the control structures.

Using the complement of the conditional expression leads to an early resolution of control structures and a flattening of the nesting. For example:

Preferred Style	Bad Style
<pre>public SampleMethod() { for (int i = 1, i < 100, i++) { // Guard if (i <= 10) { continue; } ... if (someVariable == someNumber) {</pre>	<pre>public SampleMethod() { for (int i = 1, i < 100, i++) { if (i > 10) { ... if (someVariable == someNumber) { ... } } } }</pre>



5.11 Parentheses

Use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems or understanding problems for other programmers.

5.12 Regions

Be careful about using `#region` directive. `#region` should be used for providing useful grouping of code rather than hiding unstructured code.

If you are using `#region` to group a part of code, it is probably too big, and may be a candidate for refactoring. If you hide the code in a region it defeats the purpose of encouraging others to refactor your code later on.

5.13 Refactoring

Refactoring is a technique to restructure code in a disciplined way. Refactoring follows a set of rules. These rules are named and published in a catalog in a similar fashion to Design Patterns.

Refactoring is not only a way to repair old code, or to make existing code more flexible, but it is also a way to write new code based on a system of best practices. Not all refactorings are useful at the outset, but many are, and knowledge of the techniques is invaluable.

Refer to the online version of the refactoring catalog at <http://www.refactoring.com/catalog/index.html>.

Visual Studio 2005 natively provides a number of refactoring options, which are beyond the scope of this document.

6 Appendix

6.1 Documentation Comments Sample Code

```
// XmlSample.cs
using System;

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member
/// through the remarks tag.
/// </remarks>
public class SomeClass
{
    /// <summary>
    /// Store for the name property.
    /// </summary>
    private string name;

    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (this.name == null)
            {
                throw new Exception("Name is null");
            }
            return myName;
        }
    }
}
```

```

}

/// <summary>
/// The class constructor.
/// </summary>
public SomeClass()
{
    // TODO: Add Constructor Logic here
}

/// <summary>
/// Description for SomeMethod.
/// </summary>
/// <param name="s">Parameter description for s goes here.</param>
/// <seealso cref="String">
/// You can use the cref attribute on any tag to reference a type
/// or member and the compiler will check that the reference exists.
/// </seealso>
public void SomeMethod(string s) {}

/// <summary>
/// Some other method.
/// </summary>
/// <returns>
/// Return results are described through the returns tag.
/// </returns>
/// <seealso cref="SomeMethod(string)">
/// Notice the use of the cref attribute to reference a specific method.
/// </seealso>
public int SomeOtherMethod()
{
    return 0;
}

/// <summary>
/// The entry point for the application.
/// </summary>
/// <param name="args">A list of command line arguments.</param>
public static int Main(String[] args)
{
    // TODO: Add code to start application here
    return 0;
}
}

```

Retrieved from "http://www.eirikr.net/projects/cgt/wiki/index.php?title=Implementation_Practices_and_Standards"

This page has been accessed 217 times.

This page was last modified 2006-08-18 17:21:06.

Content is available under Attribution-NonCommercial-ShareAlike 2.5.