Encodo Systems AG – Garnmarkt 1 – 8400 Winterthur
Telephone +41 52 511 80 80 – www.encodo.com

Encodo C# Handbook

Conventions, Style & Best Practices

Abstract

This document covers many aspects of programming with C#, from naming, structural and formatting conventions to best practices for using existing and developing new code.

Authors

Marco von Ballmoos
Remo von Ballmoos
Marc Dürst

Version 1.5.2 – 19.10.2009

Table of Contents

Version History

Version	Datum	Author	Comments
0.1	03.12.2007	MvB	Document Created
1.0	28.01.2008	MvB	First Draft
1.1	06.02.2008	MvB	Updated sections on error handling and naming
1.2	07.03.2008	MvB	Change to empty methods; added conditional compilation section; updated section on comments; made some examples customer-neutral; fixed some syntax-highlighting; reorganized language elements.
1.3	31.03.2008	MvB	Added more tips for documentation; added white-space rules for regions; expanded rules for line-breaking; updated event naming section.
1.4	18.04.2008	MvB	Updated formatting for code examples; added section on using the `var` keyword; improved section on choosing names; added naming conventions for lambda expressions; added examples for formatting methods; re-organized error handling/exceptions section; updated formatting.
1.5	20.05.2008	MvB	Updated line-breaking section; added more tips for generic methods; added tips for naming delegates and delegate parameters; added rules for object and array initializers; added rules and best practices for return statements; added tips for formatting complex conditions; added section on formatting switch statements.
1.5.1	24.10.2008	MvB	Incorporated feedback from the forums at the MSDN Code Gallery.
1.5.2	19.10.2009	MvB	Expanded "8.1 – Documentation" with examples; added more tips to the "2.3 – Interfaces vs. Abstract Classes" section; added "7.21 – Restricting Access with Interfaces"; added "5.3.7 – Extension Methods" and "7.18 – Using Extension Methods".
1.5.3	Unreleased	MvB	Added reference and notes from "The Little Manual of API Design"; added section "7.15 – Using Partial Classes".

Referenced Documents

Nr./Ref.	Document	Version	Date
[1]	Microsoft Design Guidelines for Developing Class Libraries	2.0	
[2]	Microsoft Internal Coding Guidelines	2.0	
[3]	IDesign C# Coding Standards	2.32	
[4]	Coding Standard: C# by Philips Medical Systems	1.3	

Nr./Ref.	Document	Version	Date
[5]	De gustibus non est disputandum. (blog post by Anthony Steele)		
[6]	The Little Manual of API Design by Jasmin Blanchette, Trolltech, a Nokia company		19.06.2008

Open Issues

Nr./Ref.	Document	Version	Date
Naming patterns for MVC			
Formatting for lambda expressions			

Terms and Abbreviations

Term / Abbreviation	Definition / Explanation
Encodo Style	This document (Encodo C# Handbook)
IDE	Integrated Development Environment
VS	Microsoft Visual Studio 2005/2008
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
DRY	Don't Repeat Yourself

1 General

1.1 Goals

The intent of this document is *not* to codify current practice at Encodo as it stands at the time of writing. Instead, this handbook has the following aims:

- To maximize readability and maintainability by prescribing a unified style.
- To maximize efficiency with logical, easy-to-understand and justifiable rules that balance code safety with ease-of-use.
- To maximize the usefulness of code-completion tools and accommodate IDE- or framework-generated code.
- To prevent errors and bugs (especially hard-to-find ones) by minimizing complexity and applying proven design principles.
- To improve performance and reliability with a list of best practices.

Wherever possible, however, the guidelines include a specific justification for each design choice. Unjustified guidelines must be either justified or removed.

Whereas the Encodo Style draws mostly from in-house programming experience, it also includes ideas from Microsoft's C# coding style [1, 2], and benefits from both the IDesign [3] and Philips [4] coding styles as corroborative sources.

1.2 Scope

This handbook mixes recommendations for programming with C#, .NET and the Encodo libraries. It includes rules for document layout (like whitespace and element placement) as well as design guidelines for elements and best practices for their use. It also assumes that you are using *Microsoft Visual Studio 2005* or newer.

This document is a work-in-progress and covers only those issues that Encodo has encountered and codifies only that which Encodo has put into practice and with which Encodo has experience. Therefore, some elements of style and design as well as some implicit best practices are probably not yet documented. Speak up if you think there is something missing.

1.3 Fixing Problems in the Handbook

Unless otherwise noted, these guidelines are not optional, nor are they up to interpretation.

- If a guideline is not sufficiently clear, recommend a clearer formulation.
- If you don't like a guideline, try to get it changed or removed, but don't just ignore it. Your code reviewer might not be aware that you are special and not subject to the rules.

1.4 Fixing Problems in Code

If code is non-conforming, it should be fixed at the earliest opportunity.

- If the error is small and localized, you should fix it with the next check-in (noting during the code review that the change was purely stylistic and unrelated to other bug fixes).
- If the error is larger and/or involves renaming or moving files, you should check the change in separately in order to avoid confusion.
- If the problem takes too long to repair quickly, you should create an issue in the bug-tracker to address the problem at a later time.

1.5 Working with an IDE

The coding conventions in this document are to be strictly followed for both manually-written and code generated by Encodo tools. However, projects will contain code from other sources as well.

Modern IDEs generate code; this is very helpful and saves a lot of work. In order to profit from this, we will need to turn a blind eye to style infractions in generated code. Specific exceptions to rules are mentioned with the rules themselves, but here are the general guidelines:

- Use preferences and options to enforce Encodo Style as much as possible.[1]
- Names entered in visual designers must conform to the Encodo coding guidelines; auto-generated control names are not acceptable if they are non-conforming.
- Any code generated by Encodo tools must conform to the Encodo coding guidelines
- Don't bother enforcing Encodo guidelines for files that are generated by non-Encodo tools (e.g. *.Designer files).
- In files containing hand-written code as well as auto-generated code, Encodo guidelines should be enforced for member order, spacing and namespaces for all elements, but not necessarily naming (e.g. event handlers generated by the environment may contain underscores instead of being Pascal-case).
- "Format Document" is your friend and should be used to re-format auto-generated code to the Encodo Style guidelines as much as possible.
- Use the highest warning level available (level 4 in *Visual Studio*) and make sure all warnings are addressed (either by fixing the code or explicitly ignoring them) before checking in code.
- Release builds should treat warnings as errors to make sure that all warnings are removed before release.

[1] The freeware product *Code Style Enforcer* is very helpful, as is the commercial product *ReSharper*.

2 Design Guide

In general, design decisions that involve thinking about the following topics should not be made alone. You should apply these principles to come up with a design, but should always seek the advice and approval of at least one other team member before proceeding.

2.1 Abstractions

The first rule of design is "don't overdesign". Overdesign leads to a framework that offers unused functionality and has interfaces that are difficult to understand and implement. Only create abstractions where there will be more than one implementation or where there is a reasonable need to provide for other implementations in the future.

This leads directly to the second rule of design: "don't under-design". Understand your problem domain well enough before starting to code so that you accommodate reasonably foreseeable additional requirements. For example, whether or not there is a need for multiple implementations (in which case you should define interfaces) or a need for code sharing (in which case abstract interfaces are in order). You should create abstractions where they prevent repeated code (applying the DRY principle) or where they provide decoupling.

If you do create an abstraction, make sure that there are tests which run against the abstraction rather than a concrete implementation so that all future implementations can be tested. For example, database access for a particular database should include an abstraction and tests for that abstraction that can be used to verify all supported databases.

2.2 Inheritance vs. Helpers

The rule here is to only use inheritance where it makes semantic sense to do so. If two classes could share code because they perform similar tasks, but aren't really related, do not give them a common ancestor just to avoid repeating yourself. Extract the shared code into a helper class and use that class from both implementations. A helper class can be `static`, but may also be an instance.

2.3 Interfaces vs. Abstract Classes

Whether or not to use interfaces is a hotly-debated topic. On the one hand, interfaces offer a clean abstraction and "interface" to a library component and, on the other hand, they restrict future upgrades by forcing new methods or properties on existing implementations. In a framework or library, you can safely add members to classes that have descendents in application code without forcing a change in that application code. However, abstract methods—which are necessary for very low-level objects because the implementation can't be known—run into the same problems as new interface methods. Creating new, virtual methods with no implementation to avoid this problem is also not recommended, as it fails to impart the intent of the method.

Where interfaces can be useful is in restricting *write-access* to certain properties or containers. That is, an interface can be declared with only a getter, but the implementation includes both a getter and setter. This allows an application to set the property when it works with an internal implementation, but to restrict the code receiving the interface to a read-only property. See "7.21 –

Restricting Access with Interfaces" for more information.

2.4 Modifying interfaces

- In general, be extremely careful of modifying interfaces that are used by code not under your control (i.e. code that has shipped and been integrated into other codebases).
- If a change needs to be made, it must be very clearly documented in the release notes for the code and must include tips for implementing/updating the implementation for the interface.
- Another solution is to develop a parallel path that consumes a new interface inherited from the existing one.

2.5 Delegates vs. Interfaces

Both delegates and interfaces can be used to connect components in a loosely-coupled way. A delegate is more loosely-coupled than an interface because it specifies the absolute minimum amount of information needed in order to interoperate whereas an interface forces the implementing component to satisfy a set of clearly-defined functionality.

If the bridge between two components is truly that of an event sink communicating with an event listener, then you should use event handlers and delegates to communicate. However, if you start to have multiple such delegate connections between two components, you'll want to improve clarity by defining an interface to more completely describe this relationship.

2.6 Methods vs. Properties

Use methods instead of properties in the following situations:

- For transformations or conversions, like `ToXml()` or `ToSql()`.
- If the value of the property is not cached internally, but is expensive to calculate, indicate this with a method call instead of a property (properties generally give the impression that they reference information stored with the object).
- If the result is not idempotent (yields the same result no matter how often it is called), it should be a method.
- If the property returns a *copy* of an internal state rather than a direct reference; this is especially significant with array properties, where repeated access is very inefficient.
- When a getter is not desired, use a method instead of a write-only property.

For all other situations in which both a property and a method are appropriate, properties have the following advantages over methods:

- Properties don't require parentheses and result in cleaner code when called (especially when many are chained together).
- It clearly indicates that the value is a logical property of the construct instead of an operation.

2.7 Virtual Methods

- Choose carefully which methods are marked as `virtual` as they incur design, test and maintenance costs not shared by non-virtual methods.

2.8 Choosing Types

- Use the least-derived possible type for local variables and method parameters; this makes the expected API as explicit and open as possible.
- Use existing interfaces wherever possible—even when declaring local or member variables. Interfaces should be useful in most instances; otherwise they've probably been designed poorly.

```
IMessageStore messages = new MessageStore();
IExpressionContext context = new ExpressionContext(this);
```

- Use the actual instantiated class for the member type when you need to access members not available in the interface. Do not modify the interface solely in order to keep using the interface instead of the class.

2.9 Design-by-Contract

Use assertions at the beginning of a method to assert preconditions; assert post-conditions if appropriate.

- Use `Debug.Assert` or throw standard system exceptions (see section 7.23.2 – Throwing Exceptions) for pre- and post-conditions.
- You may throw the exception on the same line as the check, to mirror the formatting of the assertion.

```
if (connection == null) { throw new ArgumentNullException("connection"); }
```

- If the assertion cannot be formulated in code, add a comment describing it instead.[1]
- If class invariants are not supported, describe the restrictions in the class documentation or note the invariant in commented form at the end of the class.
- All methods and properties used to test pre-conditions must have the same visibility as the method being called.

2.10 Controlling API Size

- Be as stingy as possible when making methods public; smaller APIs are easier to understand.
- If another assembly needs a type to be public, consider whether that type could not remain internalized if the API were higher-level. Use the Object Browser to examine the public API.
- To this end, frameworks that are logically split into multiple assemblies can use the `InternalsVisibleTo` attributes to make "friend assemblies" and avoid making things public. Given three assemblies, `Quino`, `QuinoWinform` and `QuinoWeb` (of which a standard Windows application would include only the first two), the `Quino` assembly can make its internals visible to `QuinoWinform` and `QuinoWeb`. This increases their interdependence, but also makes the public API of `Quino` smaller.
- The perceived size of an API can be controlled by re-using idioms and names to make it easy-to-learn and memorize. That is, lessons learned in one part of the API should apply to other parts.
- Use assemblies to separate concepts, so that each "library" is strongly focused on one task instead of having a potpourri of methods and interfaces.

[1] The spec# project at Microsoft Research provides an integration of Design-By-Contract mechanisms into the C# language; at some point in the future, this may be worthwhile to include.

3 Structure

3.1 File Contents

- Only one namespace per file is allowed.
- Multiple classes/interfaces in one file are allowed.
- The contents of a file should be obvious from the file name.
- If there are multiple classes in a file, then the file name should usually describe the subsystem to which all these classes belong.
- If the file name does match a class in the file, the other types in that file should be supporting types for that class and may include other classes, `enums`, or `structs`. The class with the same name as the file must come first.
- If a file is intended to only ever have one element (class or interface), give it the same name as the element. If a file is already named this way, do not add more classes or interfaces (unless they are supporting types, as noted above); instead create a new file named after the subsystem and move all types there.
- An `enum` should never go in its own file; instead, place it in the file with the other types that use it.
- If a file has multiple classes and interfaces, the interfaces should go into their own file of the same name as the original file, but prefixed with "I". For example, the interfaces for Encodo's metadata framework are in `IMetaCore.cs`, whereas the classes are in `MetaCore.cs`.
- If a file starts with "I", it should contain only interfaces. The only exception to this rule is for descendents of `EventArgs` and `enums` used by events declared in the interfaces.
- Do not mix third-party or generated code and manually-written project or framework code in the same file; instead consider using partial classes.
- Tests for a file go in `<FileName>Tests.cs` (if there are a lot of tests, they should be split into several files, but always using the form `<FileName><Extra>Tests.cs`) where `Extra` identifies the group of tests found in the file.
- Generated partial classes belong in a separate file, using the same root name as the user-editable file, but extended by an identifier to indicate its purpose or origin (as in the example below). This extra part must be Pascal-cased. For example:

  `Company.cs` (user-modifiable file)
  `Company.Metadata.cs` (properties generated from metadata)

- Files should not be too large; files of 1000 lines or more are noticeably slower in the *Visual Studio* debugger. Separate logical groups of classes into multiple files using the rule above to avoid this problem (even in generated code).
- Each file should include an Encodo header, whether auto-generated or not; template files (like `*.html` or `*.xml`) should also include a header if this does not conflict with visual editing or IDE-synchronization.
- The header should contain expandable tags for the check-in date, user and revision that can be maintained by a source-control system.
- Namespace `using` statements should go at the very top of the file, just after the header and just before the namespace declaration.

3.2 Assemblies

- Assemblies should be named after their content using the following pattern: `Encodo.<Component>.Dll`.
- Fill out company, copyright and version for all projects, both projects and libraries.
- Use a separate assembly for external code whenever possible; avoid mixing third-party code into the same assembly as project or framework code.
- Only one `Main()` method per assembly is allowed; library assemblies should not have a `Main()` method.
- Do not introduce cyclic dependencies.
- Application and web assemblies should have as little code as possible. Business logic should go into a class library; view/controller logic should go in the application itself.

3.3 Namespaces

3.3.1 Usage

- Do not use the global namespace; the only exception is for ASP.NET pages that are generated into the global namespace.
- Avoid fully-qualified type names; use the `using` statement instead.
- If the IDE inserts a fully-qualified type name in your code, you should fix it. If the unadorned name conflicts with other already-included namespaces, make an alias for the class with a `using` clause.
- Avoid putting a `using` statement inside a namespace (unless you must do so to resolve a conflict).
- Avoid deep namespace-hierarchies (five or more levels) as that makes it difficult to browse and understand.

3.3.2 Naming

- Avoid making too many namespaces; instead, use catch-all namespace suffixes, like "Utilities", "Core" or "General" until it is clearer whether a class or group of classes warrant their own namespace. Refactoring is your friend here.
- Do not include the version number in a namespace name.
- Use long-lived identifiers in a namespace name.
- Namespaces should be plural, as they will contain multiple types (e.g. `Encodo.Expressions` instead of `Encodo.Expression`).
- If your framework or application encompasses more than one tier, use the same namespace identifiers for similar tasks. For example, common data-access code goes in `Encodo.Data`, but metadata-based data-access code goes in `Encodo.Quino.Data`.
- Avoid using "reserved" namespace names like `System` because these will conflict with standard .NET namespaces and require resolution using the `global::` namespace prefix.

3.3.3 Standard Prefixes

- Namespaces at Encodo start with `Encodo`
- Namespaces for Encodo products start with `Encodo.<ProductName>`
- Namespaces for customer products start with `Encodo.<CustomerName>.<ProductName>`

3.3.4 Standard Suffixes

Namespace suffixes are to be added to the end of an existing namespace under the following conditions:

Suffix	When to Use
`Designer`	Contains types that provide design-time functionality for a base namespace
`Generator`	Contains generated objects for a Quino model defined in a base namespace

3.3.5 Encodo Namespaces

Listed below are the namespaces used by Encodo at the time of writing, with a description of their contents.

Namespace	Code Allowed
`*.Properties`	Project-specific properties (e.g. properties for the Quino project are in `Encodo.Quino.Properties`)
`Encodo`	None; marker namespace only
`Encodo.Data`	Helper code for working with the `System.Data` namespace
`Encodo.Core`	Very generalized code, applicable to many problem domains; not product or project-specific
`Encodo.Expressions`	Basic expression tree builder and evaluator support (includes parsers for string format and expressions as well as operation evaluation for native operators)
`Encodo.Messages`	Code for issuing, recording and storing messages (e.g. errors, warnings, hints); not the same as Trace, which is much lower-level
`Encodo.Security`	Code describing general access control
`Encodo.Testing`	All tests for code in child namespaces of `Encodo`.
`Encodo.Utilities`	Code that is not product or project-specific, but addresses a very specific problem (like XML, tracing, logging etc.)
`Encodo.Quino`	None; marker namespace only; non-product-specific code that uses Quino metadata should be in a sub-namespaces of this one.
`Encodo.Quino.Core`	All domain-independent metadata definitions and implementations (e.g. `IMetaClass` is here, but `IViewClassAspect` is defined in `Encodo.Quino.View`)
`Encodo.Quino.Data`	Data access using Quino metadata
`Encodo.Quino.Expressions`	Expressions containing metadata references
`Encodo.Quino.Meta`	Contains definitions for core interfaces and classes in the Quino metadata library.
`Encodo.Quino.Models`	None; marker namespace only. Contains other namespaces for models used by Quino testing or internals. Generated objects should be in an Objects namespace (e.g. The

Namespace	Code Allowed
	Punchclock model resides in `Encodo.Quino.Models.Punchclock` and its objects in `Encodo.Quino.Models.Punchclock.Objects`)
`Encodo.Quino.Objects`	Concrete instances of persistable objects (e.g. `GenericObject`); expressly put into a separate namespace so that code cannot just use `GenericObject` (you have to add a namespace). This encourages developers to reference interfaces rather than use `GenericObject` directly.
`Encodo.Quino.Persistence`	Metadata-assisted storage and retrieval using the data layer.
`Encodo.Quino.Properties`	Properties for the Quino project only; do not place code here.
`Encodo.Quino.Schema`	Code related to database schema import and export.
`Encodo.Quino.Testing`	All tests for code in child namespaces of `Encodo.Quino`.
`Encodo.Quino.View`	Platform-independent visualization code.
`Encodo.Quino.Winform`	Winform-dependent code based on standard .NET controls.
`Encodo.Quino.Winform.DX`	DevExpress-dependent code
`Encodo.Quino.Web`	ASP.NET-dependent code

3.3.6 Grouping and ordering

The namespaces at the top of the file should be in the following order:[1]

`System.*`	.NET framework libraries
`Third party`	Non-Encodo third-party libraries
`Encodo.*`	Organize in order of dependency
`Encodo.Quino.*`	Organize in order of dependency

[1] *Visual Studio* 2008 and *ReSharper* offer tools for automatically sorting the using statements in a file.

4 Formatting

The formatting rules were designed for use with C#. Where possible, they should be applied to other languages (CSS, JavaScript, etc.) as well.

4.1 Indenting and Spacing

- An indent is two spaces[1]; it is never a tab.
- Use a single space after a comma (e.g. between function arguments).
- There is no space after the leading parenthesis/bracket or before the closing parenthesis/bracket.
- There is no space between a method name and the leading parenthesis, but there is a space before the leading parenthesis of a flow control statement.
- Use a single space to surround *all*[2] infix operators; there is no space between a prefix operator (e.g. "-" or "!") and its argument.
- Do not use spacing to align type members on the same column (e.g. as with the members of an enumerated type).

4.2 Braces

- Curly braces should—with a few exceptions outlined below—go on their own line.
- A line with only a closing brace should never be preceded by an empty line.
- A line with only an opening brace should never be followed by an empty line.

4.2.1 Properties

- Simple getters and setters should go on the same line as all brackets.
- Abstract properties should have `get`, `set` and all braces on the same line
- Complex getters and setters should have each bracket on its own line.

This section applies to .NET 3.5 and newer.

- Prefer automatic properties as it saves a lot of typing and vastly improves readability.

4.2.2 Methods

- Completely empty functions, like constructors, should have a space between brackets placed on the same line:

```
SomeClass(string name)
  : base(name)
{ }
```

4.2.3 Enumerations

- Use the trailing comma for the last member of an enumeration; this makes it easier to move them around, if needed.

[1] The official C# standard, however, is four spaces. We use two; deal with it.

[2] This includes the => lambda expression operator.

4.2.4 Return Statements

See 7.6 – Exit points (continue and return) for advice on how to use return statements.

- Use single-line, bracketed syntax for one-line returns with simple conditions:

```
if (Count != other.Count) { return false; }
```

- If a return statement is not the only statement in a method, it should be separated from other code by a single newline (or a line with only a bracket on it).

```
if (a == 1) { return true; }

return false;
```

- Do *not* use else with return statements (use the style shown above instead):

```
if (a == 1)
{
  return true;
}
else  // Not necessary
{
  return false;
}
```

4.2.5 Switch Statements

- Contents under switch statements should be indented.
- Braces for a `case`-label are not indented; this maintains a nice alignment with the brackets from the `switch`-statement.
- Use braces for longer code blocks under `case`-labels; leave a blank line above the `break`-statement to improve clarity.

```
switch (flavor)
{
  case Flavor.Up:
  case Flavor.Down:
  {
    if (someConditionHolds)
    {
      // Do some work
    }

    // Do some more work

    break;
  }
  default:
    break;
}
```

- Use braces to enforce tighter scoping for local variables used for only one `case`-label.

```
switch (flavor)
{
  case Flavor.Up:
  case Flavor.Down:
  {
    int quarkIndex = 0; // valid within scope of case statement
    break;
  }
  case Flavor.Charm:
  case Flavor.Strange:
    int isTopOrBottom = false;  // valid within scope of switch statement
    break;
  default:
    break;
}
```

- If brackets are used for a `case`-label, the `break`-statement should go inside those brackets so that the bracket provides some white-space from the next `case`-label.

```
switch (flavor)
{
  case Flavor.Up:
  case Flavor.Down:
  {
    int quarkIndex = 0;
    break;
  }
  case Flavor.Charm:
  case Flavor.Strange:
  {
    int isTopOrBottom = false;
    break;
  }
  default:
  {
    handled = false;
    break;
  }
}
```

4.3 Parentheses

- C# has a different operator precedence than Pascal or C, so you can write `context != null && context.Count > 0` without confusing the compiler. However, you should use the form `(context != null) && (context.Count > 0)` for legibility's sake.
- Do not use parentheses around the parameter(s) in a lambda expression.
- To make it more readable, use parentheses around the condition of a ternary expression if it uses an infix operator.

```
return (_value != null) ? Value.ToString() : "NULL";
```

- Prefix operators (e.g. "!") and method calls should not have parentheses around them.

```
return !HasValue ? Value.ToString() : "EMPTY";
```

4.4 Empty Lines

In the following list, the phrase "surrounding code" refers to a line consisting of more than just an opening or closing brace. That is, no new line is required when an element is at the beginning or end of a methods or other block-level element.

Always place an empty line in the following places:

- Between the file header and the `namespace` declaration or the first `using` statement.
- Between the last `using` statement and the `namespace` declaration.
- Between types (classes, `structs`, interfaces, delegates or `enums`).
- Between public, protected and internal members.
- Between preconditions and ensuing code.
- Between post-conditions and preceding code.
- Between a call to a `base` method and ensuing code.
- Between `return` statements and surrounding code (this does not apply to return statements at the beginning or end of methods).
- Between block constructs (e.g. `while` loops or `switch` statements) and surrounding code.
- Between documented `enum` values; undocumented may be grouped together.
- Between logical groups of code in a method; this notion is subjective and more a matter of style. You should use empty lines to improve readability, but should not overuse them.
- Between the last line of code in a block and a comment for the next block of code.
- Between statements that are broken up into multiple lines.
- Between a `#region` tag and the first line of code in that region.
- Between the last line of code in a region and the `#endregion` tag.

Do not place an empty line in the following places:

- After another empty line; the Encodo style uses only single empty lines.
- Between retrieval code and handling for that code. Instead, they should be formatted together.

```
IMetaReadableObject obj = context.Find<IMetaReadableObject>();
if (obj == null)
{
  context.Recorder.Log(Level.Fatal, String.Format("Error!"));
  return null;
}
```

- Between any line and a line that has only an opening or closing brace on it (i.e. there should be no leading or trailing newlines in a block).
- Between undocumented fields (usually private); if there are many such fields, you may use empty lines to group them by purpose.

4.5 Line Breaking

- No line should exceed 100 characters; use the line-breaking rules listed below to break up a line.
- Use line-breaking only when necessary; do not adopt it as standard practice.
- If one or more line-breaks is required, use as few as possible.

- Line-breaking should occur at natural boundaries; the most common such boundary is between parameters in a method call or definition.
- Lines after such a line-break at such a boundary should be indented.
- The separator (e.g. a comma) between elements formatted onto multiple lines goes on the same line after the element; the IDE is much more helpful when formatting that way.
- The most natural line-breaking boundary is often before and after a list of elements. For example, the following method call has line-breaks at the beginning and end of the parameter list.

```
people.DataSource = CurrentCompany.Employees.GetList(
  connection, metaClass, GetFilter(), null
);
```

- If one of the parameters is much longer, then you add line-breaking between the parameters; in that case, *all* parameters are formatted onto their own lines:

```
people.DataSource = CurrentCompany.Employees.GetList(
  connection,
  metaClass,
  GetFilter("Global.Applications.Updater.PersonList.Search"),
  null
);
```

- Note in the examples above that the parameters are indented. If the assignment or method call was longer, they would no longer fit on the same line. In that case, you should use *two* levels of indenting.

```
Application.Model.people.DataSource =
  Global.ApplicationEnvironment.CurrentCompany.Employees.GetList(
    connection,
    metaClass,
    GetFilter("Global.Applications.Updater.PersonList.Search"),
    null
  );
```

- If there is a logical grouping for parameters, you may apply line-breaking at those boundaries instead (breaking the all-on-one-line or each-on-its-own-line rule stated above). For example, the following method specifies Cartesian coordinates:

```
Geometry.PlotInBox(
  "Global.Applications.MainWindow",
  topLeft.X, topLeft.Y,
  bottomRight.X, bottomRight.Y
);
```

4.5.1 Method Calls

- The closing parenthesis of a method call goes on its own line to "close" the block (see example below).

```
result.Messages.Log(
  Level.Error,
  String.Format(
    "Class [{0}] has the same metaid as class [{1}].",
    dbCls.Identifier,
    classMap[cls.MetaId]
  )
);
```

- If the result of calling a method is assigned to a variable, the call may be on the same line as the assignment if it fits.

```
people.DataSource = CurrentCompany.Employees.GetList(
  connection,
  ViewAspectTools.GetViewableWrapper(cls),
  GetFilter().Where(String.Format("PersonId = {0}", personId)),
  null
);
```

- If the call does not fit easily—or if the function call is "too far away" from the ensuing parameters, you should move the call to its own line and indent it:

```
WindowTools.GetActiveWindow().GetActivePanel().GetActiveList().DataSource =
  CurrentCompany.Organization.MainOffice.Employees.GetList(
    connection,
    ViewAspectTools.GetViewableWrapper(cls),
    GetFilter().Where(String.Format("PersonId = {0}", personId)),
    null
  );
```

4.5.2 Method Definitions

- Stay consistent with line-breaking in related methods within a class; if one is broken up onto multiple lines, then all related methods should be broken up onto multiple lines.
- The closing brace of a method definition goes on the same line as the last parameter (unlike method calls). This avoids having a line with a solitary closing parenthesis followed by a line with a solitary opening brace.

```
public static void SetupLookupDefinition(
  RepositoryItemLookUpEdit lookupOptions,
  IMetaClass metaClass)
{
  // Implementation...
}
```

- Generic method constraints should be specified on their own line, with a single indent.

```
string GetNames<T>(IMetaCollection<T> elements, string separator, NameOption options)
  where T : IMetaBase;
```

- The generic method constraint should line up with the parameters, if they are specified on their own lines.

```
public static void SetupLookupFromData<T>(
  RepositoryItemLookUpEdit lookupOptions,
  IDataList<T> dataList)
  where T : IMetaReadable
{
  SetupLookupFromData<T>(lookupOptions, dataList, dataList.MetaClass);
}
```

4.5.3 Multi-Line Text

- Longer string-formatting statements with newlines should be formatted using the @-operator and should avoid using concatenation:

```
result.SqlText = String.Format(
  @"FROM person
      LEFT JOIN
        employee
          ON person.employeeid = employee.id
      LEFT JOIN
        company
          ON person.companyid = company.id
      LEFT JOIN
        program
          ON company.programid = program.id
      LEFT JOIN
        settings
          ON settings.programid = program.id
    WHERE
       program.id = {0} AND person.hiredate <= '{2}';
  ",
  settings.ProgramId,
  state,
  offset.ToString("yyyy-MM-dd")
);
```

- If the indenting in the string argument above is important, you may break indenting rules and place the text all the way to the left of the source in order to avoid picking up extra, unwanted spaces. However, you should consider externalizing such text to resources or text files.
- The trailing double-quote in the example above is not required, but is permitted; in this case, the code needs to include a newline at the end of the SQL statement.

4.5.4 Chained Method Calls

- Chained method calls can be formatted onto multiple lines; if one chained function call is formatted onto its own line, then they should all be.

```
string contents = header.
  Replace("{Year}", DateTime.Now.Year.ToString()).
  Replace("{User}", "ENCODO").
  Replace("{DateTime}", DateTime.Now.ToString());
```

- If a line of a chained method call opens a new logical context, then ensuing lines should be indented to indicate this. For example, the following example joins tables together, with the last three statements applied to the last joined table. The indenting helps make this clear.

```
query.
  Join(Settings.Relations.Company).
  Join(Company.Relations.Office).
  Join(Office.Relations.Employees).
    WhereEquals(Employee.Fields.Id, employee.Id)
    OrderBy(Employee.Fields.LastName, SortDirection.Ascending)
    OrderBy(Employee.Fields.FirstName, SortDirection.Ascending);
```

4.5.5 Anonymous Delegates

- All rules for standard method calls also apply to method calls with delegates.
- Anonymous delegates are always written on multiple lines for clarity.
- Do not use parentheses for anonymous delegates if there are no parameters.
- Anonymous delegates should be written with an indent, as follows:

```
IMetaCollection<IMetaProperty> persistentProps =
  model.ReferencedProperties.FindAll(
    delegate(IMetaProperty prop)
    {
      return prop.Persistent;
    }
  );
```

- Even very short delegates benefit from writing in this fashion (the alternative is much messier and not so obviously a delegate when browsing through the code):

```
public string[] Keys
{
  get
  {
    return ToStrings(
      delegate(T item)
      {
        return item.Identifier;
      }
    );
  }
}
```

- This notation is also useful for long function calls with many or long parameters. If, for example, a delegate is one of the parameters, then you should make a block out of the whole function call, like this:

```
_context = new DataContext(
  Settings.Default.ConfigFileName,
  DatabaseType.PostgreSql,
  delegate
  {
    return ModelGenerator.CreateModel();
  }
);
```

  In the example above *each* parameter is on its own line, as required.

- Here's a fancy one, with a delegate in a constructor base; note that the closing parenthesis is on the same line as the closing brace of the delegate definition.

```
public Application()
  : base(
      DatabaseType.PostgreSql,
      delegate()
      {
        return ModelGenerator.CreateModel();
      })
    { }
```

4.5.6 Lambda Expressions

This section applies to .NET 3.5 and newer.

- All rules for standard method calls also apply to method calls with lambda expressions.
- Very short lambda expressions may be written as a simple parameter on the same line as the method call:

```
ReportError(msg => MessageBox.Show(msg));
```

- Longer lambda expressions should go on their own line, with the closing parenthesis of the method call closing the block on another line. Any calls attached to the result—like `ToList()` or `Count()`—should go on the same line as the closing parenthesis.

```
people.DataSource = CurrentCompany.Employees.Where(
  item => item.LessonTimeId == null
).ToList();
```

- Longer lambda expressions should not be both wrapped and used in a `foreach`-statement; instead, use two statements as shown below.

```
var appointmentsForDates = data.Appointments.FindAll(
  appt => (appt.StartTime >= startDate) && (appt.EndTime <= endDate)
);

foreach (var appt in appointmentsForDates)
{
  // Do something with each appointment
}
```

4.5.7 Object Initializers

This section applies to .NET 3.5 and newer.

Longer initialization blocks should go on their own line; the rest can be formatted in the following ways (depending on line-length and preference):

- Shorter initialization blocks (one or two properties) can be specified on the same line:

```
var personOne = new Person { LastName = "Miller", FirstName = "John" };
```

- The new-clause is on the same line as the declaration:

```
IViewPropertySizeAspect sizeAspect = new ViewPropertySizeAspect
{
  VerticalSizeMode = SizeMode.Absolute,
  VerticalUnits = height
};

prop.Aspects.Add(sizeAspect);
```

- The new-clause is on its own line:

```
IViewPropertySizeAspect sizeAspect =
  new ViewPropertySizeAspect
  {
    VerticalSizeMode = SizeMode.Absolute,
    VerticalUnits = height
  };

prop.Aspects.Add(sizeAspect);
```

- The initializer is nested within the method-call on the same line as the declaration:

```
prop.Aspects.Add(new ViewPropertySizeAspect { VerticalUnits = height });
```

- The initializer is nested within the method-call on its own line:

```
prop.Aspects.Add(
  new ViewPropertySizeAspect
  {
    VerticalSizeMode = SizeMode.Absolute,
    VerticalUnits = height
  });
```

  If the initializer goes spans multiple lines, then the new-clause must also go on its own line.

4.5.8 Array Initializers

This section applies to .NET 3.5 and newer.

The type is usually optional (unless you're initializing an empty array), so you should leave it empty.

4.5.9 Ternary and Coalescing Operators

- Do not use line-breaking to format statements containing ternary and coalescing operators; instead, convert to an `if/else` statement.

5 Naming

The naming rules were designed for use with C#. Where possible, they should be applied to elements of other languages (CSS, JavaScript, etc.) as well.

5.1 Basic Composition

5.1.1 Valid Characters

- Identifiers should contain only alphabetic characters.
- The underscore is allowed only as a leading character for fields (or when included in a member generated by the IDE).
- Numbers are allowed only for local variables and method parameters and may then only appear as a suffix. Avoid using numbers wherever possible. (A valid use of a number in an identifier is in a sorting routine that accepts two elements; in that case, "value1" and "value2" are appropriate.)

5.1.2 General Rules

- Names are in US-English (e.g. use "color" instead of "colour").
- Names conform to English grammatical conventions (e.g. use `ImportableDatabase` instead of `DatabaseImportable`).
- Names should be as short as possible without losing meaning.
- Prefer whole words or stick to predefined short forms or abbreviations of words (as seen in section 5.4.1 – Local Variables and Parameters).
- Make sure to capitalize compound words correctly; if the word is not hyphenated, then it does not need a capital letter in the camel- or Pascal-cased form. For example, "metadata" is written as `Metadata` in Pascal-case, not `MetaData`.
- Acronyms should be Pascal-case as well (e.g. "Xml" or "Sql") unless they are only two letters long. Acronyms at the beginning of a camel-case identifier are always all lowercase.
- Identifiers differing only by case may be defined within the same scope only if they identify different language elements (e.g. a local variable and a property).

```
public void UpdateLength(int newLength, bool refreshViews)
{
  int length = Length;
  // ...
}
```

- You may not use identifiers that are keywords in C#; neither may you use the @-symbol to turn a keyword into a valid identifier.

5.1.3 Collision and Matching

- Do not name an element with the same identifier as its containing element (e.g. don't create a static class named `Expressions` within a namespace called `Expressions`).
- Since C# allows it, you should use the same identifier for a property as its type if that is the most appropriate name in that context (this is often the case with `enum` properties).

5.2 Capitalization

The following table lists the capitalization and naming rules for different language elements. Pascal-case capitalizes every individual word within an identifier, including the first one. Camel-case capitalizes all but the first word in an identifier.

Language Element	Case
Class	Pascal
Interface	Pascal w/leading "I"
Struct	Pascal
Enumerated type	Pascal
Enumerated element	Pascal
Properties	Pascal
Generic parameters	Pascal
Public or protected `readonly` or `const` field	Pascal
Private field	Camel with leading underscore[1]
Method argument	Camel
Local variable	Camel
Attributes	Pascal with "Attribute" suffix
Exceptions	Pascal with "Exception" suffix
Event handlers	Pascal with "EventHandler" suffix

5.3 The Art of Choosing a Name

5.3.1 General

- Readability is paramount, so while short names are desirable, make sure the identifier is clear and reads well (e.g. the property `bool UpdatesAutomatically` is better than `bool AutoUpdate`).
- Do not use a prefix for members in order to "group" them; this applies to enumerated type members as well.
- Do not use a generalized prefixing notation (e.g. Hungarian).
- Use the plural form to indicate lists instead of a suffix like "List" (e.g. use `appointments` instead of `appointmentList`).
- Avoid abbreviations unless that abbreviation is *very* standardized and accepted; this rule applies to abbreviations like using "max" instead of "maximum" or "num" instead of "count", but does not apply to acronyms like "XML" or "SQL".

[1] This is not CLS-compliant because protected variables in Visual Basic always start with an underscore, so the code could not be re-generated in that language. We've chosen not to care.

5.3.2 Namespaces

- Do not use a "library" prefix for types (e.g. instead of `QnoDatabase`, use a more descriptive name, like `MetaDatabase` or `RelationalDatabase`).
- Avoid very generic type names (e.g. `Element`, `Node`, `Message` or `Log`), which collide with types from the framework or other commonly-used libraries. Use a more specific name, if at all possible. [1]
- If there are multiple types encapsulating similar concepts (but with different implementations, for example), you should use a common suffix to group them. For example, all the expression node types in the Encodo expressions library end in the word `Expression`.

5.3.3 Interfaces

- Prefix interfaces with the letter "I".

5.3.4 Classes

- If a class implements a single interface, it should reflect this by incorporating the interface name into its own (e.g. `MetaList` implements `IList`).
- Static classes used as toolkits of static functions[2] should use the suffix "Tools" and should go in a file ending in "Tools.cs".

5.3.5 Properties

- Properties should be nouns or adjectives.
- Prepend "`Is`" to the name for Boolean properties only if the intent is unclear without it. The next example shows such a case:

  ```
  public bool Empty { get; }
  public bool IsEmpty { get; }
  ```

  Even though it's a property not a method, the first example might still be interpreted as a verb rather than an adjective. The second example adds the verb "Is" to avoid confusion, but both formulations are acceptable.

- A property's backing field (if present) must be an underscore followed by the name of the property in camel case.
- Use common names, like `Item` or `Value`, for accessing the central property of a type.
- Do not include type information in property names. For example, for a property of type `IMetaRelation`, use the name `Relation` instead of the name `MetaRelation`.
- Make the identifier as short as possible without losing information. For example, if a class named `IViewContext` has a property of type `IViewContextHandler`, that property should be called `Handler`.

---

[1] Name collisions can be resolved using aliases or simply by using `global::` namespace resolution, but this makes working with and reading the code both more difficult. For example, suppose we have made an interface for metadata properties in the `Encodo.Quino.Meta` namespace. The natural name for the interface is `IProperty`, but that's too common a name, so we should use something like `IMetaProperty` instead. Other metadata interfaces would share this prefix, like `IMetaClass`, `IMetaRelation` and so on.

[2] Such toolkits are also called "Convenience Methods" and often contain extension methods; see "7.18 – Using Extension Methods" for more information.

- If there are two properties that could be shortened in this way, then neither of them should be. If the class in the example above has another property of type `IEventListHandler`, then the properties should be named something like `ViewContextHandler` and `EventListhandler`, respectively.
- Avoid repeating information in a class member that is already in the class name. Suppose, there is an interface named `IMessages`; instances of this interface are typically named messages. That interface should not have a property named Messages because that would result in calls to `messages.Messages.Count`, which is redundant and not very readable. Instead, name the property something more semantically relevant, like `All`, so the call would read `messages.All.Count`.

5.3.6 Methods

- Methods names should include a verb.
- Method names should not repeat information from the enclosing type. For example, an interface named `IMessages` should not have a method named `LogMessage`; instead name the method `Log`.
- State what a method does; do not describe the parameters (let code-completion and the signature do that for you).
- Methods that return values should indicate this in their name, like `GetList()`, `GetItem()` or `CreateDefaultDatabase()`. Though there is garbage collection in C#, you should still use `Get` to indicate retrieval of a local value and `Create` to indicate a factory method, which always creates a new reference. For example, instead of writing:

```
public IDataList<GenericObject> GetList(IMetaClass cls)
{
  return ViewApplication.Application.CreateContext<GenericObject>(cls);
}
```

  You should write:

```
public IDataList<GenericObject> CreateList(IMetaClass cls)
{
  return ViewApplication.Application.CreateContext<GenericObject>(cls);
}
```

- Avoid defining everything as a noun or a manager. Prefer names that are logically relevant, like `Missile.Launch()` rather than `MissileLauncher.Execute(missile)`.
- Methods that set a single property value should begin with the verb `Set`.
- The most generalized version of a method name should be reserved for the method that the framework wishes to encourage or that is used most often. An example from [6] is reproduced below:

> *Suppose you have two event-delivery mechanisms, one for immediate (synchronous) delivery and one for delayed (asynchronous) delivery. The names `sendEventNow()` and `sendEventLater()` suggest themselves. Now, if you want to encourage your users to use synchronous delivery (e.g., because it is more lightweight), you could name the synchronous method `sendEvent()` and keep `sendEventLater()` for the asynchronous case.*

5.3.7 Extension Methods

- Extension methods for a given class or interface should appear in a class named after the class or interface being extended, plus the suffix "Tools". For example, extension methods for the class `Enum` should appear in a class named `EnumTools`.
- In the case of interfaces, the leading "I" should be dropped from the class name. For example, extension methods for the interface `IEnumerable<T>` should appear in a class named `EnumerableTools`.

5.3.8 Parameters

- Prefer whole words instead of abbreviations (use `index` instead of `idx`).
- Parameter names should be based on their intended use or purpose rather than their type (unless the type indicates the purpose adequately).
- Do not simply repeat the type for the parameter name; use a name that is as short as possible, but doesn't lose meaning. (E.g. a parameter of type `IDataContext` should be called `context` instead of `dataContext`.
- However, if the method also, at some point, makes use of an `IViewContext`, you should make the parameter name more specific, using `dataContext` instead.
- For copy constructors or equality operators, name the object to be copied or compared "`other`".

5.3.9 Local Variables

Since local variables are limited to a much smaller scope and are not documented, the rules for name-selection are somewhat more relaxed.

- Try to use a name from section 5.4.1 – Local Variables and Parameters, if possible.
- Avoid using `temp` or `i` or `idx` for loop indexes. Use the suffix `Index` together with a descriptive prefix, as in `colIndex` or `itemIndex` or `memberIndex`.
- Names need only be as specific as the scope requires.
- The more limited the scope, the more abbreviated the variable may be.

5.3.10 Return Values

- If a method creates a local variable expressly for the purpose of returning it as the result, that variable should be named `result`.

```
object result = this[Fields.Id];

if (result == null) { return null; }

return (Int32)result;
```

5.3.11 Events

- Single events should be named with a noun followed by a descriptive verb in the past tense.

```
event EventHandler MessageDispatched;
```

- For paired events—one raised before executing some code and one raised after—use the gerund form (i.e. ending in "ing") for the first event and the past tense (i.e. ending in "ed") for the second event.

```
event EventHandler MessageDispatching;
event EventHandler MessageDispatched;
```

- Event receivers are like any other methods and should be named according to their task, not the event to which they are attached. The following method updates the user interface; it does this regardless of whether it is attached as an event receiver for the `MessageDispatched` event.[1]

```
void UpdateUserInterface(object sender, EventArgs args)
{
  // Implementation
}
```

- Never start an event receiver method with "On" because Microsoft uses that convention for raising events.
- To trigger an event, use `Raise[EventName]`; this method must be `protected` and `virtual` to allow descendents to perform work before and after calling the base method.
- If you are raising events for changes made to properties, use the pattern "`Raise[Property]Changed`".

5.3.12 Enumerations

- Simple enumerations have singular names, whereas bit-sets[2] have plural names.

5.3.13 Generic Parameters

- If a class or method has a single generic parameter, use the letter `T`.
- If there are two generic parameters and they correspond to a key and a value, then use `K` and `V`.
- Generic methods on classes with a generic parameter should use `TResult`, where appropriate. The example below shows a generic class with such a method.

```
public class ListBookmarkSelection<T> : IBookmarkSelection
{
  public IList<TResult> GetObjects<TResult>()
  {
    // Convert list contents from T to TResult
  }
}
```

- Generic conversion functions should use `TInput` and `TOutput`, respectively.

[1] This applies only to event handlers written by hand; those generated by the IDE should be left untouched and will, in general, start with the name of the control, followed by an underscore, followed by the name of the event. Therefore, the click event for the control `_okButton` will be `_okButton_Click`. These may be left as generated.

[2] Enumerations marked with the `[flags]` attribute (discussed below).

```
public static IList<TOutput> ConvertList<TInput, TOutput>(IList<TInput> input)
{
  // Convert list contents from TInput to TOutput
}
```

- If there are multiple parameters, but no pattern, name the "contained" element T (if there is one) and the other parameters something specific starting with the letter `T`.

5.3.14 Delegates and Delegate Parameters

- Use a descriptive verb for delegate names, like the following examples:

```
delegate string TransformString(T item);
delegate bool ItemExists(T item);
delegate int CompareItems(T first, T second);
```

- Delegate method parameter names should use the same grammar as the type, so that it sound natural when executed:

```
public string[] ToStrings(TransformToString transform)
{
  // ...
  result[] = transform(item);
  // ...
}
```

This section applies to .NET 3.5 and newer.

- If a delegate is only used once or twice and has a relatively simple syntax, use `Func<>` for the parameter signature instead of declaring a delegate. The example above can be rewritten as:

```
public string[] ToStrings(Func<T, string> transform)
{
  // ...
  result[] = transform(item);
  // ...
}
```

  This practice makes it much easier to determine the expected signature in code-completion as well.

5.3.15 Lambda Expressions

- Do not use the highly non-expressive `x` as a parameter name[1].
- Parameters in a lambda expression should follow the same conventions as for parameters in standard methods.
- Do not make overly-complex lambda expressions; instead, define a method or use a delegate.

[1] This is the default variable name taken from other functional languages, but it's completely non-descriptive and thus doesn't conform to conventions.

5.3.16 Compiler Variables

- Compiler variables are all capital letters, with words separated by underscores.

```
#if ENCODO_DEVELOPER
      return true;
#else
      return false;
#endif
```

5.4 Common Names

5.4.1 Local Variables and Parameters

The following table presents the method parameter or local variable name you should use for common types in the .NET and Encodo libraries. For other types, try to follow the spirit of the naming convention by choosing complete words or known abbreviations.

Type	Parameter & Variables	Variables only
IMessageStore	messages	msgs
Exception	exception	
IMetaBase	metadata	
IMetaModel	model	
IMetaClass	cls, metaClass	
IMetaProperty	prop	
IMetaRelation	relation	
IMessageRecorder	recorder	
IMetaEndpoint	source, target	
IMetaBase, IMetaElement	elt, element	
IDbCommand	cmd, command	
IDbConnection	conn, connection	
I*Database	db, database	
IQuery	query	
I*Context	context[1]	
I*Handler	handler	
IMessage	msg, message	
IOperator	op, operation	
IExpression	expr, expression	
*Item	item	
String	name, message, msg, text (not n, txt)	
*Session	session	
*Application	application, app	
*EventArgs	args[2]	

[1] Though ctx or ctxt for context are common enough, they are not recommended.

[2] This differs explicitly from the Microsoft recommendation, which advocates using e.

Type	Parameter & Variables	Variables only
*List	list	
*Collection	collection, coll	
*Type	Type	
*Column	column, col	
CultureInfo	Culture	
Encoding	Encoding	
*Bookmark	Bookmark	
Appointment	Appointment	appt

5.4.2 User Interface Components

UI Elements should not include a suffix that indicates their control type unless there is another member that already uses that name or there are two controls that would use that name.[1] If a suffix must be used, use one from the table below that best matches the control's type.

UI Element	Suffix	Example
Menu or toolstrip item	Item	_saveItem
Menu (context or main)	Menu	_logMenu
Listview, listbox, etc.	List	_fileList
Trees	Tree	_folderTree
Columns	Column	_lastNameColumn
Grids	Grid	_programsGrid
Groups	Group	_optionsGroup
Data Sources	Data	_programsData
Panels	Panel	_optionsPanel
Text boxes	Text	_lastNameText
Labels	Label	_lastNameLabel
Check boxes	Checkbox	_isEmployeeCheckbox
Page Control Tabs	Tab	_optionsTab
Radio Buttons	RadioButton	_salariedRadioButton
Page Controls	Pages	_preferencesPages
Dialogs	Dialog	_saveModelDialog
Image lists	Images	_smallMenuImages
Generic controls	Control	_someControl

5.4.3 ASP Pages

- Do not add a suffix to the names of ASP pages since that name is used in the URL (at least in classic ASP.NET; naming patterns for MVC will come later).
- Do not add a prefix or suffix to pages used as AJAX-dialogs; instead, collect these pages into a sub-folder named "Dialogs".

[1] For example, this conflict can arise when you have a list control with a popup menu attached to it.

6 Language Elements

6.1 Declaration Order

- Constructors (in descending order of complexity)
- public constants
- properties
- public methods
- protected methods
- private methods
- protected fields
- private fields

6.2 Visibility

- The visibility modifier is required for all types, methods and fields; this makes the intention explicit and consistent.
- The visibility keyword is always the first modifier.
- The `const` or `readonly` keyword, if present, comes immediately after the visibility modifier.
- The `static` keyword, if present, comes after the visibility modifier and `readonly` modifier.

```
private readonly static string DefaultDatabaseName = "admin";
```

6.3 Constants

- Declare all constants other than `0`, `1`, `true`, `false` and `null`.
- Use `true` and `false` only for assignment, never for comparison.
- Avoid passing `true` or `false` for parameters; use an `enum` or constants to impart meaning instead.
- If there is a logical connection between two constants, indicate this by making the initialization of one dependent on the other.

```
public const int DefaultCacheSize = 25;
public const int DefaultGranularity = DefaultCacheSize / 5;
```

6.3.1 readonly vs. const

The difference between `const` and `readonly` is that `const` is compiled and `readonly` is initialized at runtime.

- Use const only when the value really is constant (e.g. `NumberDaysInWeek`); otherwise, use `readonly`.
- Though `readonly` for references only prevents writing of the reference, not the attached value, it is still a helpful hint for both the compiler and the reader.

6.3.2 Strings and Resources

- Do not hardcode strings that will be presented to the user; use resources instead. For products in development, this text extraction can be performed after the code has crystallized somewhat.
- Resource identifiers should be alphanumeric, but may also include a dot (".") to logically nest resources.

- Do not use constants for strings; use resource tables instead (this aids translation, if necessary).
- Configuration data should be moved into application settings as soon as possible.

6.4 Properties

- In the event that setting a property caused an exception, then the existing value should be restored.
- Use read-only properties if there is no logical reason for calling code to be able to change the value.
- Properties should be commutative; that is, it should not matter in which order you set them. Avoid enforcing an ordering by using a method to execute code that you would want to execute from the property setter. The following example is incorrect because setting the password before setting the name causes a login failure.

```
class SecuritySystem
{
  private string _userName;

  public string UserName
  {
    get { return _userName; }
    set { _userName = value; }
  }

  private int _password;

  public int Password
  {
    get { return _password; }
    set
    {
      _password = value;
      LogIn();
    }
  }

  protected void LogIn()
  {
    IPrincipal principal = Authenticate(UserName, Password);
  }

  private IPrincipal Authenticate(string UserName, int Password)
  {
    // Authenticate the user
  }
}
```

Instead, you should take the call `LogIn()` out of the setter for `Password` and make the method `public`, so the class can be used like this instead:

```
SecuritySystem system = new SecuritySystem();
system.Password = "knockknock";
system.UserName = "Encodo";
system.LogIn();
```

In this case, `Password` can be set before the `UserName` without causing any problems.

6.4.1 Indexers

- Provide an indexed property only if it really makes sense in the context of the class.
- Indexes should be 0-based.

6.5 Methods

- Methods should not have more than 200 lines of code
- Avoid returning `null` for methods that return collections or strings. Instead, return an empty collection (declare a static empty list) or an empty string (`String.Empty`).
- Default implementations of empty methods should have both brackets on the same line:

```
protected virtual void DoInitialize(IMessageRecorder recorder) { }
```

- Overrides of abstract methods or implementations of interface methods that are explicitly left empty should be marked with NOP:

```
protected override void DoBeforeSave()
{
  // NOP
}
```

- Consider using partial methods to reduce the number of explicitly declared virtual methods if you are using C# 3.0.

6.5.1 Virtual

- Prefer making `virtual` methods `protected` instead of `public`, but do not create an extra layer of method calls just to do so. If a method has logical pre-conditions or post-conditions (i.e. the pre-condition checks for more than just whether a parameter is `null`), consider making the method protected and wrapping it in a public method with the contracts in it (as below):

```
public void Update(IQuery query)
{
  Debug.Assert(query != null);
  Debug.Assert(query.Valid);
  Debug.Assert(Updatable);

  DoUpdate(query);

  Debug.Assert(UpToDate);
}

protected virtual void DoUpdate(IQuery query)
{
  // Perform update
}
```

  Always use "Do" as a prefix for such protected, helper methods.

- If a protected method is not `virtual`, make it private unless it will be used from a descendent.

6.5.2 Overloads

- Overloads are encouraged for methods that are in the same family and either serve the same purpose or have similar behavior. Do not use the types of parameters to distinguish these functions from one another. For example, the following is incorrect

```
void Update();
void UpdateUsingQuery(IQuery query);
void UpdateUsingSql(string sql);
```

Instead, use an overload, letting the method signature describe the different functions. This reduces the perceived size of the API and makes it easier to understand.

```
void Update();
void Update(IQuery query);
void Update(string sql);
```

- If an overloaded method must be marked virtual, make only one version virtual and define all of the others in terms of that one. Using the example above, this would yield:

```
public void Update()
{
  Update(QueryTools.NullQuery); // Accesses a static global “null” query
}

public virtual void Update(IQuery query)
{
  // Perform update
}

public void UpdateUsingSql(string sql)
{
  Update(new Query(sql));
}
```

- If two or more overloads share a parameter, that parameter name should be the same in all overloads.
- Similarly, standardize parameter positions as much as possible between overloads and even just similar methods.

6.5.3 Parameters

- Methods should not have more than 5 parameters (consider using a `struct` instead).
- Methods should not have more than 2 `out` or `ref` parameters (consider using a `struct` instead).
- `ref`, then `out` parameters should come last in the list of parameters.
- The implementation of an interface method should use the same parameter name as that given in the interface method declaration.
- Do not declare reserved parameters (use overloads in future library versions instead).
- If a method follows the Try* pattern—which returns a `bool` indicating success, and accepts a single out parameter—the parameter should be named “result”. The method should be prefixed with “Try”.
- Do not assign new values to parameters; use a local variable instead. Assignments to parameters are easy-to-miss in larger methods. If you use a local variable instead, a reader

knows right away to look for initializations of that variable rather than to look for changes to the parameter value.

6.5.4 Constructors

- Base constructors should be on a separate line, indented one level.
- Consider including the default `base()` call in constructors to make it clear which constructor is called (and to provide a way of quickly jumping to the implementation in the IDE).
- A constructor is considered to be valid if it doesn't crash and the object can be used without crashing or causing unwarranted exceptions (null reference, etc.). Any properties required by the constructor to make it valid should be passed in as parameters.
- All constructors should satisfy all class invariants; that is, you cannot require a user to set properties on an object in order to make it valid. A class may, however, require that some properties should be set before being able to use certain functions of a class. The example below shows such a class, which has an empty constructor and requires that certain properties are set before calling `Connect()` or `LogIn()`.

```
internal abstract class BackEnd
{
  void BackEnd()
  { }

  internal abstract string ServerName { get; set; }

  internal abstract string UserName { get; set; }

  internal abstract string Password { get; set; }

  internal abstract void Connect();

  internal abstract void LogIn();
}
```

As an aside, this is not a recommended design. The example above would work much better as follows:

```
abstract class BackEnd
{
  void BackEnd()
  { }

  abstract void Connect(IConnectionSettings settings);

  abstract void LogIn(IUser user);
}
```

- Avoid doing more than setting properties in a constructor; provide a method on the class to perform any extra work after the object has been constructed.

- Avoid calling virtual methods from a constructor because the most-derived version will be called, but before the constructor for that most-derived class has executed. The example below illustrates this problem, where the override `CaffeineAddict.GoToWork()` uses `Coffee` before it has been initialized.

```
public interface IBeverage
{
  public bool Empty { get; }
}

public abstract class Employee
{
  public Employee()
  {
    GoToWork();
  }

  protected abstract void GoToWork();

  protected void Drink(IBeverage beverage)
  {
    if (!beverage.Empty) // Crashes when initializing CaffeineAddict
    {
      // drink it
    }
  }
}

public class CaffeineAddict : Employee
{
  public CaffeineAddict(IBeverage coffee)
    : base()
  {
    _coffee = coffee;
  }

  public IBeverage Coffee
  {
    get { return _coffee; }
  }

  protected override void GoToWork()
  {
    Drink(Coffee);
  }

  private IBeverage _coffee;
}
```

- To avoid duplicating code, but also to avoid exposing an unwanted default constructor, use a protected default constructor. For example:

```
protected Query()
{
  _restrictions = new List<IRestriction>();
  _sorts = new List<ISort>();
}

public Query(IMetaClass model)
  : this()
{
  Debug.Assert(model != null);
  Model = model;
}

public Query(IDataRelation relation)
  : this()
{
  Debug.Assert(relation != null);
  Relation = relation;
}
```

- Consider using a static factory method (e.g. on a `*Tools` class) if construction of an object is very complex or would require a large number of constructor parameters.

6.6 Classes

- Never declare more than one field per line; each field should be an individually documentable entity.
- Do not use public or protected fields; use private fields exposed through properties instead.

6.6.1 Abstract Classes

Please see section 2.1 – Abstractions for a discussion of when to use abstract classes.

- Do not define `public` or `protected internal` constructors for abstract types; instead, define a `protected` or `internal` one.
- Consider providing a partial implementation of an abstract class that handles some of the abstraction in a standard way; implementers can use this class as a base and avoid having to repeat code in their own implementations. Such classes should use the "Base" suffix.

6.6.2 Static Classes

- Do not mark a class as static if it has instance members.
- Do not create too many static classes; instead, determine whether new functionality can be added to an existing static class.

6.6.3 Sealed Classes & Methods

- Do not declare protected or virtual members on sealed classes
- Avoid sealing classes unless there is a very good reason for doing so (e.g. to improve reflection performance).
- Consider sealing only selected members instead of sealing an entire class.

- Consider sealing members that you have overridden if you don't want descendents to avoid your implementation.

6.7 Interfaces

Please see section 2.1 – Abstractions for a discussion of when to use interfaces.

- Use interfaces to "fake" multiple-inheritance.
- Define interfaces if there will be more than one implementation of a hierarchy; without multiple-inheritance, this is the only way to remain flexible as to the implementation.
- Define interfaces to clearly define what comprises an API; an interface will generally be smaller and more tightly-defined that the class that implements it. A class-based hierarchy runs the risk of mixing interface methods with implementation methods.
- Consider using a C# attribute instead of a marker interface (an interface with no members). This makes for a cleaner inheritance representation and indicates the use of the marker better (e.g. NUnit tests as well as the serializing subsystem for .NET use attributes instead of marker interfaces).
- Re-use interfaces as much as possible to avoid having many very similar interfaces that cause confusion as to which one should be used where.
- Keep interfaces relatively small in order to ease implementation (5-10 members).
- Where possible, provide an abstract class or default descendent that application code can use for implementing an interface. This provides both an implementation example and some protection from future changes to the interface.
- Use interfaces where the functionality isn't the direct purpose of the object or to expose a part of the class's functionality (as with aspect-oriented programming).
- Use explicit interface implementation where appropriate to avoid expanding a class API unnecessarily.
- Each interface should be used at least once in non-testing code; otherwise, get rid of it.
- Always provide at least one, tested implementation of an interface.

6.8 Structs

Consider defining a structure instead of a class if most of the following conditions apply:

- Instances of the type are small (16 bytes or less) and commonly short-lived.
- The type is commonly embedded in other types.
- The type logically represents a single value and is similar to a primitive type, like an `int` or a `double`.
- The type is immutable.
- The type will not be boxed frequently.[1]

Use the following rules when defining a `struct`.

- Avoid methods; at most, have only one or two methods other than overrides and operator overloads.
- Provide parameterized constructors for initialization.

[1] In scenarios that require a significant amount of boxing and un-boxing, value types perform poorly as compared to reference types.

- Overload operators and equality as expected; implement `IEquatable` instead of overriding `Equals` in order to avoid the negative performance impact of boxing and un-boxing the value.
- A `struct` should be valid when uninitialized so that consumers can declare an instance without calling a constructor.
- Public fields are allowed (even encouraged) for structures used to communicate with external APIs through unmanaged code.

6.9 Enumerations

- Always use enumerations for strongly-typed sets of values
- Use enumerations instead of lists of static constants *unless* that list can be extended by descendent code; if the list is not logically open-ended, use an `enum`.
- Enumerations are like interfaces; be extremely careful of changing them when they are already included in code that is not under your control (e.g. used by a framework that is, in turn, used by external application code). If the enumeration must be changed, use the `ObsoleteAttribute` to mark members that are no longer in use.
- Do not assign a type to an `enum` unless absolutely necessary; use the default type of `Int32` whenever possible.
- Do not include sentinel values, such as `FirstValue` or `LastValue`.
- Do not assign explicit values to simple enumerations except to enforce specific values for storage in a database.
- The first value in an enumeration is the default; make sure that the most appropriate simple enumeration value is listed first.

6.9.1 Bit-sets

- Use the `[Flags]` attribute to make a bit-set instead of a simple enumeration.
- Bit-sets always have plural names, whereas simple enumerations are singular.
- Assign explicit values for bit-sets in powers of two; use hexadecimal notation.
- The first value of a bit-set should always be `None` and equal to `0x00`.
- In bit-sets, feel free to include commonly-used aliases or combinations of flags to improve readability and maintainability. One such common value is `All`, which includes all available flags and, if included, should be defined last. For example:

```
[Flags]
public enum QuerySections
{
  None = 0x00,
  Select = 0x01,
  From = 0x02,
  Where = 0x04,
  OrderBy = 0x08,
  NotOrderBy = All & ~OrderBy,
  All = Select | From | Where | OrderBy,
}
```

The values `NotOrderBy` and `All` are aliases defined in terms of the other values. Note that the elements here are not aligned because it is expected that they will be documented, in which case column-alignment won't make a difference in legibility.

- Avoid designing a bit-set when certain combinations of flags are invalid; in those cases, consider dividing the enumeration into two or more separate enumerations that are internally valid.

6.10 Nested Types

- Nested types should not replace namespaces for organization.
- Use nested types if the inner type is logically within the other type (e.g. a `TableOfContents` class may have an `Options` inner class or a `Builder` inner class).
- Use nested types if the inner type should have access to all members of the outer type.
- Do not use public nested types unless you have a good reason for doing so (e.g. in the case of the `Options` class described above).
- If a nested type needs a public constructor so that other types can create instances, then it probably shouldn't be nested.
- Delegate declarations should not be nested within the type because this reduces re-use of delegate declarations between types.
- Use a nested type to group private or protected constants.

6.11 Local Variables

- Declare a local variable as close as possible to its first use (and within the most appropriate scope).
- Local variables of the same type may be declared together, but only if they are not initialized.

```
IMetaEndpoint source, target;
```

- If a local variable is initialized, put the initialization on the same line as the declaration. If the line gets too long, use multiple lines as described in section 4.5 – Line Breaking.
- Local variables that need to be initialized cannot be declared on the same line unless they have the same initialization value.

```
int startOfWord = firstCharacter = 0;
```

6.12 Event Handlers

You should use the pattern and support classes for event-handling provided by the .NET library.

- Do not expose delegates as `public` members; instead declare events using the `event` keyword.
- Do not add a method to a delegate with `new EventHandler(…)`; instead, use delegate inference.
- Do not define custom delegates for event handling; instead use `EventHandler<T>`.
- Put all extra event data into an `EventArgs` descendent; subsequent versions can alter this descendent without changing the signature.
- Use `CancelEventArgs` as the base class if you need to be able to cancel an event.
- Neither the `sender` parameter nor the `args` parameter may be `null`; this avoids forcing event handlers to check for null.
- `EventsArgs` descendents should declare only properties, not methods or other application logic.

6.13 Operators

- Be extremely careful when overloading operators; in general, you should only do so for `structs`. If you feel that an operator overload is especially clever, it probably isn't; check with another developer before coding it.
- Avoid overriding the == operator for reference types; override the `Equals()` method instead to avoid redefining reference equality.
- If you do override `Equals()`, you should also override `GetHashCode()`.
- If you do override the == operator, consider overriding the other comparison operators (`!=`, `<`, `<=`, `>`, `>=`) as well.
- You should return false from the `Equals()` function if the objects cannot be compared. However, if they are different types, you may throw an exception.

6.14 Loops & Conditions

6.14.1 Loops

- Do not change the loop variable of a for-loop.
- Update `while` loop variables either at the beginning or the end of the loop.
- Keep loop bodies short and avoid excessive nesting.
- Consider using an inner class or other private methods if the body of a loop gets too complex.

6.14.2 If Statements

- Do not compare to `true` or `false`; instead, compare pure Boolean expressions.
- Initialize Boolean values with simple expressions rather than using an if-statement; always use parentheses to delineate the assigned expression.

```
bool needsUpdate = (Count > 0 && Objects[0].Modified);
```

- Always use brackets for flow-control blocks (`switch`, `if`, `while`, `for`, etc.)
- Do not add useless `else` blocks. An "`if`" statement may stand alone and an "`else if`" statement may be the last condition.[1]

```
if (a == b)
{
  // Do something
}
else if (a > b)
{
  // Do something else
}

// No final "else" required
```

[1] This is noted only because some style guides explicitly require that the last statement in an "`if/else if`" block is an empty "`else`" block if none is otherwise needed.

- Do not force really complicated logic into an "`if`" statement; instead, use local variables to make the intent clearer. For example, imagine we have a lesson planner and want to find all unsaved lessons that are either unscheduled or are scheduled within a given time-frame. The following condition is too long and complicated to interpret quickly:

```
if (!lesson.Stored && (((StartTime <= lesson.StartTime) && (lesson.EndTime <=
EndTime)) || ! lesson.Scheduled))
{
  // Do something with the lesson
}
```

Even trying to apply the line-breaking rules results in an unreadable mess:

```
if (!lesson.Stored &&
  (((StartTime <= lesson.StartTime) && (lesson.EndTime <= EndTime)) ||
  ! lesson.Scheduled))
{
  // Do something with the lesson
}
```

Even with this valiant effort, the intent of the `||`-operator is difficult to discern. With local variables, however, the logic is much clearer:

```
bool lessonInTimeSpan = ((StartTime <= lesson.StartTime) && (lesson.EndTime <= EndTime));
if (!lesson.Stored && (lessonInTimeSpan || ! lesson.Scheduled))
{
  // Do something with the lesson
}
```

6.14.3 Switch Statements

- Include a default statement that `asserts` or `throws` if all valid values are handled. This also applies for `enums` because the compiler does not realize that no default statement is needed.
- Use the following form when values initialized by the switch-statements are to be used elsewhere in the method.

```
IDatabase result = null;
switch (type)
{
  case DatabaseType.PostgreSql:
    result = new PostgreSqlMetaDatabase();
    break;
  case DatabaseType.SqlServer:
    result = new SqlServerMetaDatabase();
    break;
  case DatabaseType.SQLite:
    result = new SqliteMetaDatabase();
    break;
  default:
    Debug.Assert(false, String.Format("Unknown database type: {0}", type));
}

// Work with "result".

return result;
```

- In the case where the switch statement is the either the entire method or the final block in a method, use return statements directly from the case labels. In this case, the assertion is replaced with an exception or it won't compile.

```
switch (type)
{
  case DatabaseType.PostgreSql:
    return new PostgreSqlMetaDatabase();
  case DatabaseType.SqlServer:
    return new SqlServerMetaDatabase();
  case DatabaseType.SQLite:
    return new SqliteMetaDatabase();
  default:
    throw new ArgumentException("type", String.Format("Unknown type: {0}", type));
}
```

- The `default` label must always be the last label in the statement.

6.14.4 Ternary and Coalescing Operators

The ternary operator is a specialized form of an `if/then` statement with the following form:

```
return (_value != null) ? Value.ToString() : "NULL";
```

If the condition (`_value != null` in this case) is true, the operator returns the value after the question mark; otherwise, it returns the value after the colon.

The coalescing operator is a specialized form of the ternary operator, which has the following form:

```
return Target ?? Source;
```

The operator returns the expression before the two question marks if it is not `null`; otherwise, it returns the expression after the two question marks.

- Use these operators for simple expressions and results.
- Do not use these operators with long conditions and values; instead, use an `if/then` statement.
- Do not break statements with these operators in them over multiple lines.

6.15 Comments

6.15.1 Formatting & Placement

- Comments are indented at the same level as the code they document.
- Place comments above the code being commented.

6.15.2 Styles

- Use the single-line comment style—`//`—to indicate a comment.
- Use four slashes —`////`—to indicate a single line of code that has been temporarily commented.
- Use the multi-line comment style—`/* … */`—to indicate a commented-out block of code. In general, code should never be checked in with such blocks.

- Consider using a compiler variable to define a non-compiling block of code; this practice avoids misusing a comment.

```
#if FALSE
      // commented code block
#endif
```

- Use the single-line comment style with `TODO` to indicate an issue that must be addressed. Before a check-in, these issues must either be addressed or documented in the issue tracker, adding the URL of the issue to the TODO as follows:

```
// TODO http://issue-tracker.encodo.com/?id=5647: [Title of the issue in the issue tracker]
```

6.15.3 Content

- Good variable and method names go a long way to making comments unnecessary.
- Comments should be in US-English; prefer a short style that gets right to the point.
- A comment need not be a full, grammatically-correct sentence. For example, the following comment is too long

```
// Using a granularity that is more than 50% of the size causes a crash!

int Granularity = Size / 5;
```

  Instead, you should stick to the essentials so that the warning is immediately clear:

```
int Granularity = Size / 5; // More than 50% causes a crash!
```

- Comments should be spellchecked.[1]
- Comments should not explain the obvious. In the following example, the comment is superfluous.

```
public const int Granularity = Size / 5; // granularity is 20% of size
```

- Use comments to explain algorithms or tricky bits that aren't immediately obvious from a quick read.
- Use comments to indicate where a hard-won bug-fix was added; if possible, include a reference to a URL in an issue tracker.
- Use comments to indicate assumptions not already evident from assertions or thrown exceptions.
- Longer comments should always precede the line being commented.[2]
- Short comments may appear to the right of the code being commented, but only for lines ending in semicolon (i.e. marking the end of a statement). For example:

```
int Granularity = Size / 5; // More than 50% causes a crash!
```

- Comments on the same line as code should *never* be wrapped to multiple lines.

[1] CodeSpell is a good and relatively inexpensive spellchecker for *Visual Studio* 2005 and 2008; if you're already using ReSharper, the Agent Smith plugin provides superb integration with multiple dictionaries.

[2] A newline separating the comment from its code is the recommended style, as it tends to separate the comments and the code into separate, but interleaved blocks. This is, however, just a suggestion.

6.16 Grouping with #region Tags

- Use `#region` tags to distinguish groups of functions; use the auto-implement macro in *Visual Studio* to ensure that interface implementations are surrounded in `#region` tags describing which interface is being implemented by the enclosed functions.
- Use regions for generated code blocks.
- You may use regions to demarcate logical groups of members or types.
- A region should generally enclose more than one element.

6.17 Compiler Variables

- Avoid using `#define` in the code; use a compiler define in the project settings instead.
- Avoid suppressing compiler warnings.

6.17.1 The [Conditional] Attribute

Use the `Conditional` attribute instead of the `#ifdef/#endif` pair wherever possible (i.e. for methods or classes).

```
public class SomeClass
{
  [Conditional("TRACE_ON")]
  public static void Msg(string msg)
  {
    Console.WriteLine(msg);
  }
}
```

6.17.2 #if/#else/#endif

For other conditional compilation, use a static method in a static class instead of scattering conditional options throughout the code.

```
public static class EncodoCompilerOptions
{
  public static bool DeveloperBuild()
  {
#if ENCODO_DEVELOPER
    return true;
#else
    return false;
#endif
  }
}
```

This approach has the following advantages:

- The compiler checks all code paths instead of just the one satisfying the current options[1]; this avoids unknowingly retaining incompatible code in a library or application.
- Code formatting and indenting is not broken up by (possibly overlapping) compile conditions; the name `EncodoCompilerOptions` makes the connection to the compiler obvious enough.

[1] One drawback is that the editor doesn't display the "unused" code as disabled, as it does when using compiler options directly.

- The compiler option is referenced only once, avoiding situations in which some code uses one compiler option (e.g. `ENCODO_DEVELOPER`) and other code uses another, misspelled option (e.g. `ENCODE_DEVELOPER`).

7 Patterns & Best Practices

7.1 Safe Programming

- Use early-binding—which can be checked by the compiler—wherever possible; use late-binding only when necessary.
- Avoid using `new` whenever possible; use `override` instead or restructure the code to avoid it.
- Do not use `goto` unless you have gotten the approval of a senior member of your team.
- Use `unsafe` code only for integrating external libraries.
- If a reference may be `null`, test it before using it; otherwise, use `Debug.Assert` to verify that it is not `null`.
- Do not re-use local variable names, even though the scoping rules are well-defined and allow it. This prevents surprising effects when the variable in the inner scope is removed and the code continues to compile because the variable in the outer scope is still valid.
- Do not modify a variable with a prefix or suffix operator more than once in an expression. The following statement is not allowed:

```
items[propIndex++] = ++propIndex;
```

7.2 Side Effects

A side effect is defined as a change in an object as a result of reading a property or calling a method that causes the result of the property or method to be different when called again.

- Reading a property may not cause a side effect.
- Writing a property may cause a side effect.
- Methods have side effects by definition.
- Avoid writing methods that return results and cause side effects. A method should either retrieve information or it should execute an operation, but not both.

7.3 Null Handling

- If a value of `null` is allowed for a parameter of an interface type, consider making an empty implementation of that interface (named with the prefix "Null").

```
interface IWeapon
{
  bool Ready { get; }
  void Aim();
  void Fire();
}

class NullWeapon : IWeapon
{
  bool Ready
  {
    get { return false; }
  }

  void Aim()
  {
    // NOP
  }

  void Fire()
  {
    // NOP
  }
}
```

Additionally, you should make an instance of this empty implementation available via a global static. This makes it easier to write code that uses the interface, as it can assert that the parameter is non-null and callers can simply pass in the empty implementation to satisfy the pre-condition.

7.4 Casting

- Use the `as`-operator when testing and using a type.[1] If you are just testing a type, use the `is`-operator.

```
Class1 other = obj as Class1;
if (other == null) { return false; }
```

- If you are using the type of an object to route to a type-specific method, then you can use the `is`-operator and the casting operator, as shown below.

```
if (this is IMetaProperty)
{
  HandleProperty((IMetaProperty)this);
}
else if (this is IMetaMethod)
{
  HandleMethod((IMetaProperty)this);
}
else
{
  HandleDefault(this);
}
```

- If you are sure of the type, use the casting operator to assert the type because it will throw an `InvalidCastException` if it fails; this avoids having to test for `null`.

```
((IMetaClass)obj).RunShow();
```

7.5 Conversions

C# types can define `explicit` and `implicit` conversions to and from other types.

- Provide conversions only where they are logically justified; this goes double for implicit conversions. For example, the Quino library object `MetaString` enhances a string with multiple language representations; it can be freely converted to and from a string using the default language. However, you should not be able to implicitly or explicitly cast a control that displays a web page to a `String` (representing either the page content or the URL).
- Generally, you should only provide implicit conversions between types that are in the same domain (like converting between string representations).
- Do not provide implicit conversions if the conversion would cause data-loss.
- Implicit conversions cannot throw exceptions; explicit conversions can. Use `InvalidCastException` for such errors.

[1] This does not work for value types; in that case, you should test with the is-operator and use a cast.

7.6 Exit points (continue and return)

- Multiple return statements are allowed if all exit points are relatively close to one another. For example, the following code has two, clearly visible exit points.

```
if (a == 1) { return true; }

return false;
```

- Use return statements to avoid additional indentation for easy-to-catch conditions.

```
if (String.IsNullOrEmpty(key)) { return null; }
```

- Do not spread return statements out throughout a longer method; instead, create a local variable named `result` and return it from the end of the method.

```
bool result = parameters.Count == 0;

if (someConditionHolds())
{
  // Perform operations
  result = false;
}
else
{
  // Perform other operations

  if (someOtherConditionHolds())
  {
    // Perform operations
    result = false;
  }
  else
  {
    // Perform other operations
    result = true;
  }
}

return result;
```

- The only code that may follow the last `return` statement is the body of an exception handler.

```
try
{
  // Perform operations

  return true;
}
catch (Exception exception)
{
  throw new DirectoryAuthenticatorException(exception);
}
```

- Avoid the use of `continue`; rewrite the program logic instead by reversing the condition. Instead of the following logic:

```
foreach (SearchResult search in searches)
{
  if (!search.Path.Contains("CN=")) { continue; }

  // Work with valid searches
}
```

Use the following logic:

```
foreach (SearchResult search in searches)
{
  if (search.Path.Contains("CN="))
  {
    // Work with valid searches
  }
}
```

7.7 Object Lifetime

- Always use the `using` statement with objects implementing `IDisposable` to limit their lifetimes.
- Set objects that are no longer needed to `null` so that the garbage collector can collect them.
- Avoid using destructors because they incur a performance penalty in the garbage collector.
- Do not access other object references inside the destructor as those objects may already have been garbage-collected (there is no guaranteed order-of-destruction as in other languages).
- Use `try/finally` blocks (or related constructs, like `using` or `lock`) to clean up objects allocated in a method. Local variables are automatically de-referenced when exiting the method, so there's no need to set them to `null`.

7.8 Using Dispose and Finalize

`Dispose` and `Finalize` provide control over how an object is garbage-collected. `Finalize` is called when an object is reclaimed by the garbage collector; overriding it prevents resources from leaking if a consumer of your class fails to call `Dispose`.

- If you implement `Dispose`, use the `IDisposable` interface to allow explicit recovery of resources.
- Make sure that `Dispose` can be called multiple times safely; all other methods should raise an `ObjectDisposedException` if `Dispose` has already been called.
- If there is a more appropriate domain-specific name for `Dispose`, implement the `IDisposable` interface explicitly and provide a method with the new name that calls `Dispose`.
- Call the `GC.SuppressFinalize` method to prevent Finalize from being executed if `Dispose()` has already been called.
- If you implement `Dipose`, implement `Finalize` only if you actually have costly external resources.
- `Finalize` should simply include a call to `Dispose`.
- There are performance penalties for implementing `Finalize`, so proceed with caution.

- Finalize should never be public; call only the `base.Finalize()` method directly.

7.9 Using base and this

- Use `this` only when referring to other constructors.
- Do not use `this` to resolve name-clashes; instead, change one of the conflicting names.
- Use `base` only from a constructor or to call a predecessor method.
- You may only call the base method of the method being executed; do not call other base methods. In the following example, the call to `CheckProcess()` is not allowed, whereas the call to `RunProcess()` is.

```
public override void RunProcess()
{
  base.CheckProcess();  // Not allowed
  base.RunProcess();
 }
```

7.10 Using Value Types

Some value types have both Pascal- and camel-case versions. Though `string` is simply an alias of `String`, you should not mix and match them everywhere. Instead, follow the rules below.

- Use types from the System namespace when calling static functions (e.g. `String.Format` or `String.IsNullOrEmpty`).
- Use the value types when declaring variables or fields (e.g. `string` instead of `String`).

7.11 Using Strings

- Use the "`+`"-operator for concatenating up to three values; otherwise, use `String.Format`;
- Use a `StringBuilder` for more complex situations or when concatenation occurs over multiple statements.
- In comparisons, use "" instead of `String.Empty` as it is clearer and shorter. The generated code is the same.
- For function results, use `String.Empty`.
- When checking whether a string is empty, use `String.IsNullOrEmpty` instead of (`s == null`) or (`s.Length == 0`).

7.12 Using Checked

- Applications should always have range-checking during development and debugging.
- Range-checking may be disabled in release builds if there is a valid performance reason for doing so.
- Overflow- and underflow-prone operations should have explicit `checked` blocks (i.e. if there was a range-checking problem at some point, the block should be marked with a `checked` block). This way, even if range-checking is disabled, these blocks will still be checked.

7.13 Using Floating Point and Integral Types

- Be extremely careful when comparing floating-point constants; unless you are using `decimals`, the representation will have limited precision and lead to subtle bugs.
- One exception to this rule is when you are comparing constants of fixed, known value (like `0.0` or `1.0`, but not `3.14`).

- Be extremely careful when casting representations with different sizes (e.g. `Int64` to `Int32`); always `Assert` that the value fits within the new representation so as to localize the point-of-failure.

7.14 Using Generics

- Use the generic version of a class if available (e.g. use `IList<T>` instead of `IList`).
- Do not use casting in generic classes; use a generic constraint (`where`) instead.
- Use generic parameters and constraints instead of forcing a base type.
- Avoid constraints in delegates.
- Avoid constraints in generic methods; consider whether the problem can be solved another way.
- If inheriting from both a generic and non-generic interface, implement the non-generic version explicitly and implement it using the generic interface (e.g. when implementing `IEnumerable` and `IEnumberable<T>`).

7.15 Using Partial Classes

- Use partial classes to separate private or protected inner classes away from the "main" implementation.
- Use partial classes to separate larger blocks of private or protected methods away from the public interface (this helps reduce file size).

7.16 Using Event Handlers

- Be careful with events in performance-sensitive code; handlers can affect performance in non-predictable ways.
- Assume that the state of the object triggering an event has changed in unpredictable ways (i.e. that the code executed in the event handler may have changed the state of the calling object).
- Assume that code in an event handler may trigger exceptions and act accordingly in the `Raise*` triggering method.
- Be aware of which thread is triggering the event handler and which thread is handling the event. Events handled in the main (UI) thread must be synchronized (this is a limitation of Windows UI programming).
- To avoid multi-threading problems, get a reference to the handler in a local variable before checking it and calling it. For example:

```
protected virtual void RaiseMessageDispatched()
{
  EventHandler handler = MessageDispatched;
  if (handler != null) { handler(this, new EventArgs()); }
}
```

The following code is an example of a simple event handler and receiver with all of the Encodo style conventions applied.

```
public class Safe
{
  public event EventHandler Locked;

  public void Lock()
  {
    // Perform work

    RaiseLocked(new EventArgs());
  }

  protected virtual void RaiseLocked(EventArgs args)
  {
    EventHandler handler = Locked;
    if (handler != null) { handler(this, args); }
  }
}

public static class StoreManager
{
  private static void SendMailAboutSafe(object sender, EventArgs args)
  {
    // Respond to the event
  }

  public static void TestSafe()
  {
    Safe safe = new Safe();
    safe.Locked += SendMailAboutSafe;
    safe.Lock();
  }
}
```

7.17 Using System.Linq

When using Linq expressions, be careful not to sacrifice legibility or performance simply in order to use Linq instead of more common constructs. For example, the following loop sets a property for those elements in a list where a condition holds.

```
foreach (var pair in Data)
{
  if (pair.Value.Property is IMetaRelation)
  {
    pair.Value.Value = null;
  }
}
```

This seems like a perfect place to use Linq; assuming an extension method `ForEach(this IEnumerable<T>)`, we can write the loop above using the following Linq expression:

```
Data.Where(pair => pair.Value.Property is IMetaRelation).ForEach(pair =>
pair.Value.Value = null);
```

This formulation, however, is more difficult to read because the condition and the loop are now buried in a single line of code, but a more subtle performance problem has been introduced as

well. We have made sure to evaluate the restriction ("`Where`") first so that we iterate the list (with "`ForEach`") with as few elements as possible, but we still end up iterating twice instead of once. This could cause performance problems in border cases where the list is large and a large number of elements satisfy the condition.

7.18 Using Extension Methods

This section applies to .NET 3.5 and newer.

Extension methods are a way of adding available methods to a type without actually defining those methods on the type itself. Though these methods may contribute to API bloat, they are encouraged as long as they are used often (more than once, please) and provide a clear service—either making calling code much cleaner or handling a tricky task, or both.

- Use extension methods to extend system or third-party library types.
- Use extension methods to avoid cluttering interfaces with methods that are not central to the interface and can be implemented using other members of the interface.
- Use extension methods to provide a common implementation of some functionality for all implementations of an interface, so that each implementation is not required to implement functionality that can be provided centrally (i.e. poor-man's multiple-inheritance).
- Do not use extension methods when an interface implementation could possibly provide an optimized version of the functionality were the method declared on the interface instead. That is, do not restrict an implementation's efficiency because an extension instead of interface method was used.
- Do not mix extension methods with other static methods. If a class contains extension methods, it should contain only extension methods and private support methods.
- Each extended type should have its own extension methods class; do not mix extension methods for different types in one class.

7.19 Using "var"

This section applies to .NET 3.5 and newer.

The introduction of the keyword `var` for implicitly-typed variables is a boon to writing legible code. Using `var` can eliminate a lot of repeated text from code and make the intent much clearer. However, the goal is to make code *more* legible, not just to use implicit typing as much as possible.[1]

- It is not sufficient that the code compile; a human reader must also be able to (relatively) easily understand the code.
- You should always use semantically relevant names; this goes doubly so for local variables using `var`.
- Do not use `var` when the return type is a basic type, like `int` or `string`.
- Use of `var` in larger scopes requires more care; within smaller scopes, its use is almost always ok.
- Use `var` when the type or intent is already clear from the context; otherwise, specify the type to improve understandability.

[1] It is assumed that this keyword be used in an environment (e.g. *Visual Studio* 2008) that offers the code-completion and quick navigation that makes implicitly-typed variables usable.

- Removing too many types not only reduces readability, but also reduces navigability (i.e. one can no longer navigate to related types using `F12`).

7.19.1 Examples

- The classic case involves direct instantiation using the `new`-operator.

  ```
  IList<Airplane> planes = new List<Airplane>();
  ```

  This type of declaration needlessly clutters the code and should be replaced:

  ```
  var planes = new List<Airplane>();
  ```

- The rule does not state that the full type must be stated—simply that the code be understandable. The following example can use `var` because it also uses appropriate variable- and method-naming conventions.

  ```
  IDataList<Airplane> planes = connection.GetList<Airplane>();
  ```

  Naming the variable `planes` in the plural and using the method name `GetList` is sufficient to get the point across that the variable is a list of `Airplane` objects.

  ```
  var planes = connection.GetList<Airplane>();
  ```

- In the following example, the right-hand side offers no hint as to the type (other than that implied by the plural name `GetAirplanes`).

  ```
  IDataList<Airplane> planes = hanger.GetAirplanes(connection);
  ```

  However, you can use `var` here because the name of the variable `planes` is semantically relevant.

  ```
  var planes = hangar.GetAirplanes(connection);
  ```

7.20 Using out and ref parameters

One case in which you may return error codes instead of throwing exceptions is when writing a library that will be used external code that does not support passing exceptions across process or domain boundaries.

- If you must use error codes, do not return them; use `out` and `ref` parameters instead.
- Use `out` and `ref` parameters as little as possible.
- Instead of using many out and ref parameters, consider defining a `struct` instead; this improves the maintainability of the API.

7.21 Restricting Access with Interfaces

One advantage in using interfaces over abstract classes is that interfaces can restrict write-access to properties or lists. That is, the interface declares a getter, but no setter so that clients of the interface can only read the property. However, the backing implementation is free to add a setter as well, allowing the creator of the backing object to set the property externally, if desired.

The following example illustrates this principle for properties:

```
interface IMaker
{
  bool Enabled { get; }
}

class Maker : IMaker
{
  bool Enabled { get; set; }
}
```

The principle extends to making read-only sequences as well, using `IEnumerable<T>` in the interface and exposing `IList<T>` in the backing class, using explicit interface implementation, as shown below:

```
interface IProcessedCommands
{
  IEnumerable<ISwitchCommand> SwitchCommands { get; }
}

class ProcessedCommands
{
  IEnumerable<ISwitchCommand> IProcessedCommands.SwitchCommands
  {
    get { return SwitchCommands; }
  }

  IList<ISwitchCommand> SwitchCommands { get; private set; }
}
```

In this way, the client of the interface cannot call `Add()` or `Remove()` on the list of commands, but the provider has the convenience of doing so as it prepares the backing object. Using Linq, the client is free to convert the result to a list using `ToList()`, but will create its own copy, leaving the `SwitchCommands` on the interface unaffected.

7.22 Error Handling

7.22.1 Strategies

- Exceptions are the primary means of signaling errors (see 7.22.3 – Exceptions).
- Consider carefully whether an error is truly exceptional or whether the component should handle the error and set a property to indicate a status instead. This is especially true of code that performs an asynchronous task (e.g. a communications component), where the initiation point is separated from the completion point.
- Do not design methods that change error-handling strategy depending on a parameter; use the Try*-pattern instead.

- If a method is expected to encounter one or more errors, don't use exceptions; use an `IMessageRecorder` instead.
- Reserve the result for semantically relevant data. If there is no semantically relevant result, use `void`. The following function is incorrect because `-1` is not a semantically relevant result for the method name.

```
public int GetNumberOfPeopleInFile(string fileName)
{
  if (CanLoadFile(fileName))
  {
    // Return actual number of people.
  }

  return -1;
}
```

Instead, you should use something like the following code, throwing exceptions for situations that the API is not meant to handle.

```
public int GetNumberOfPeopleInFile(string fileName)
{
  File people = LoadFile(fileName); // throws exception

  // Return actual number of people.
}
```

7.22.2 Error Messages

- The standards for error messages are the same as for any other text that might be shown to a user; use grammatically correct English (though translations may also be provided).
- Avoid question marks and exclamation points in messages (even assertion messages). Use the error level or exception type to indicate severity or let the handler of the exception determine what level of urgency to assign.
- Messages should always end in a period.
- Lower-level, developer messages should be logged to sources that are available only to those with permission to view lower-level details.
- Applications should avoid showing sensitive information to end-users. This applies especially to web applications, which must never show exception traces in production code. The exact message returned by an exception can vary depending on the permission level of the executing code.
- Be as specific as possible when throwing exceptions.
- The message should include as much information as possible, though it is highly recommended to provide both a longer, low-level message (for logging and debugging) and a shorter, high-level message for presenting to the user.
- Error messages should describe how the user or developer can avoid the exception.

7.22.3 The Try* Pattern

The Try* pattern is used by the .NET framework. Generally, Try*-methods accept an `out` parameter on which to attempt an operation, returning true if it succeeds.

- If you provide a method using the `Try*` pattern (), you should also provide a non-try-based, exception-throwing variant as well.

```
public IExpression Parse(string text, IMessageRecorder recorder)
{
  // parse the string to an expression
}

public bool TryParse(string text, IMessageRecorder recorder, out IExpression
result)
{
  try
  {
    result = Parse(text, recorder);
    return true;
  }
  catch (ExpressionException)
  {
    result = null;
    return false;
  }
}
```

In the example above, you'll note the parse process also accepts an `IMessageRecorder`, which is used to record hints, warnings and errors. The function throws an `ExpressionException` if any unrecoverable errors were detected. The contents of the exception message should include the recorded messages as well.

7.23 Exceptions

7.23.1 Defining Exceptions

- Do not simply create an exception type for every different error.
- Create a new type only if you want to expose additional properties on the exception; otherwise, use a standard type.
- Use a custom exception to hold any information that more completely describes the error (e.g. error codes or structures).

```
throw new DatabaseException(errorInfo);
```

- Custom exceptions should always inherit from `Exception` (do not use `ApplicationException` as its use has been deprecated).
- Custom exceptions should be `public` so that other assemblies can catch them and extract information.
- Avoid constructing complex exception hierarchies; use your own exception base-classes if you actually will have code that needs to catch all exceptions of a particular sub-class.

- An exception should provide the three standard constructors (as well as the serialization constructor if you are supporting serialization) and should use the given parameter names:

```
public class ConfigurationException : Exception
{
  public ConfigurationException()
  { }

  public ConfigurationException(string message)
    : base(message)
  { }

  public ConfigurationException(string message, Exception inner)
    : base(message, innerException)
  { }

  public ConfigurationException(SerializationInfo info, StreamingContext context)
    : base(message, innerException)
  { }
}
```

- Do not cause exceptions during the construction of another exception (this sometimes happens when formatting custom messages) as this will subsume the original exception and cause confusion.
- If an exception must be able to work across application domain and remoting boundaries, then it must be serializable.

7.23.2 Throwing Exceptions

- If a member cannot satisfy its post-condition (or, absent a post-condition, fulfill the promise implied in its name or specified in its documentation), it should throw an exception.
- Use standard exceptions where possible.
- Do not throw `Exception` or `SystemException`.[1]
- Never throw `Exception`. Instead, use one of the standard .NET exceptions when possible. These include `InvalidOperationException`, `NotSupportedException`, `ArgumentException`, `ArgumentNullException` and `ArgumentOutOfRangeException`.
- When using an `ArgumentException` or descendent thereof, make sure that the `ParamName` property is non-empty.
- Your code should not explicitly or implicitly throw `NullReferenceException`, `System.AccessViolationException`, `System.InvalidCastException`, or `System.IndexOutOfRangeException` as these indicate implementation details and possible attack points in your code. These exceptions are to be avoided with pre-conditions and/or argument-checking and should never be documented or accepted as part of the contract of a method.
- Do not throw `StackOverflowException` or `OutOfMemoryException`; these exceptions should only be thrown by the runtime.
- Do not explicitly throw exceptions from `finally` blocks (implicit exceptions are fine).

[1] *Visual Studio* generates code that throws an `Exception` when to indicate that it has not yet been implemented; these are only temporary and do not need to be changed.

7.23.3 Catching Exceptions

- Be as specific as possible as to which exceptions are caught (even if this means you have to repeat some lines of handling code). This allows unexpected exceptions (e.g. `NullReferenceException`) to show up during testing instead of being swallowed and logged with other, expected errors.
- Catch and re-throw an exception in order to reset the internal state of an object to satisfy a post-condition.
- In the case of a misbehaving third-party component, catch specific exceptions as much as possible, but also catch all exceptions to prevent third-party problems from crashing the application.

```
try
{
  // Use third-party code
}
catch (DatabaseException)
{
  // Handle problems with database
}
catch (ArgumentException)
{
  // Handle problems with arguments
}
catch (Exception)
{
  // Handle misbehaving third-party code
}
```

- Do not catch exceptions and reroute them to an event; this practice separates the point-of-failure from the logging/collection point, increasing the likelihood that an exception is ignored and making debugging very difficult.

7.23.4 Wrapping Exceptions

- Only catch an exception if you plan to wrap it in another exception, if you plan to handle it by logging or setting an internal state, or
- Use an empty `throw` statement to re-throw the original exception in order to preserve the stack-trace.
- Wrapped exceptions should *always* include the original exception in order to preserve the stack-trace.
- Lower-level exceptions from an implementation-specific subsection should be caught and wrapped before being allowed to bubble up to implementation-independent code (e.g. when handling database exceptions).

7.23.5 Suppressing Exceptions

- Only suppress expected exceptions; do not write a catch-all exception handler unless you are re-throwing the exception.
- You may only suppress an exception if you either set an internal state indicating the exception on the catching object or if you log it (e.g. to an `IMessageRecorder` or a `TraceSource`).

- If it is unsafe to continue executing after an error, consider calling `System.Environment.FailFast` instead of throwing an exception.

7.23.6 Specific Exception Types

- Catch and suppress `System.Exception` only from the most external layer of code in an application or module. In other words, use catch-all exception handling when the exception can be presented to the user or must be passed across an API-boundary (e.g. out of a DLL loaded by legacy code).
- Do not catch `ArgumentExceptions` unless causing the error is unavoidable (i.e. you should check preconditions before calling a method).
- Do not catch a `StackOverflowException` as the state of the running application that has encountered a stack-overflow is not defined.
- Catching an `OutOfMemoryException` should be done rarely or not at all.

7.24 Generated code

- Avoid modifying generated code unless there is an extremely good reason for doing so.
- Use a custom build step to perform the modification to make sure that the required change is not lost if the environment re-generates the file.
- Do not add application logic to `AssemblyInfo.cs`; add only attributes.

7.25 Setting Timeouts

- Use `TimeSpans` to encapsulate timeout values.
- Prefer passing the timeout value as a parameter to the method that will use it rather than offering it as a property on the class. This more closely associates the timeout value with the operation that uses it.
- The method can decide which values of `TimeSpan` are valid (including `TimeSpan.Zero` and `TimeSpan.MaxValue`).

7.26 Configuration & File System

- An assembly or application should never make any assumptions about the location from which it's running; nor should it make assumptions about folder structure on a hard drive. Use the members of `System.Environment.SpecialFolder` instead.
- Do not use the registry to store application information; save user settings to a user-accessible file instead.

7.27 Logging and Tracing

- Use `IMessageRecorder` and `IMessageStore` wherever possible to collect multiple errors from a process instead of throwing a single exception. Use these classes when output should go to a user or might be collected by a UI; otherwise, use tracing.
- Use `TraceSources` to send output to logs; do not log directly to the console or screen

7.28 Performance

- If possible, set the list capacity in the constructor.

8 Processes

8.1 Documentation

8.1.1 General

- Documentation should be written in English and must be grammatically correct (i.e. do not just use lists of keywords or short phrases; prefer full sentences or clauses).
- All documentation ends in a period.
- Use XML documentation to document public and protected elements.
- Method-level documentation is the absolute minimum (as it will be displayed by code-completion).
- Private and internal code should only be documented when needed; do not document small methods or methods whose purpose is evident from their implementation.
- Class documentation should include references to the important members for quick navigation and introduction to the class.
- External tutorials and samples are strongly encouraged.
- Frameworks should provide a more in-depth documentation of the API that includes high-level architecture, diagrams, examples and tutorials.
- Stay consistent when documenting similar members (e.g. properties); it's ok to repeat yourself or to use the same exact structure for all members (see below for examples) as long as the documentation is useful for that member.
- Do not add "using namespace" declarations just to resolve documentation references; instead, use as much of the namespace path as necessary in the documentation reference itself.[1] It's better to reserve `using` for actual code, as the reference from the documentation is not resolved by the C# compiler when generating an executable, whereas the `using` is.
- Do not ever add an assembly reference just to resolve a documentation reference. As above, for the *code* to pull in another assembly is fine, but the documentation should not introduce such a dependency.

8.1.2 XML Tags

- Make sure the `<summary>` does not simply restate what is already obvious from the element; however, it should also be relatively short so that it is useful in code-completion.
- Use a `<remarks>` section to indicate usage and to link to related members. It's sometimes good to include references to other types or methods in descriptive sentences in addition to listing them in the `<seealso>` section.
- Use `<c>` tags for the keywords `null`, `false` and `true`.
- Use `<see>` tags to refer to properties, methods and classes.
- Use `<paramref>` and `<typeparamref>` tags to refer to method parameters.
- Although you might want to use `<c>` tags for subsequent references to code elements if you want to keep the documentation readable in the code, it is highly recommended to use the `<paramref>` and `<typeparamref>` for all references because those can be refactored when the name changes.
- Use `<seealso>` tags to link documentation for overloads or related methods.

[1] *ReSharper* will offer to use namespaces in order to resolve documentation; you should ignore it.

- Use the `<inheritdoc/>` tag for method overrides or interface implementations to avoid repeating documentation.[1] Add a `<remarks>` section to document specifics of the implementation (though this is actually quite rare, in practice).

```
/// <inheritdoc/>
bool Exists { get; }
```

- Use the `<include>` tag to include larger blocks of documentation (e.g. large `<remarks>` or `<example>` sections).
- Avoid applying possessives to code elements; instead of writing '`<paramref name="prop">'s value`', write '`the value of <paramref name="prop">`'.

8.1.3 Tool Support

- The automatically-generated documentation from tools like *Ghostdoc* isn't too bad, but you should enhance it so that it offers more than just a reformulation of what was obvious from the signature.
- Avoid manually wrapping documentation; instead use a tool like the *Agent Smith* plugin for *ReSharper*.

8.1.4 Classes

- If the class is an abstract implementation of an interface, then use this formulation:

```
/// <summary>
/// A base implementation of the <see cref="IMaker"/> interface.
/// </summary>
public abstract class MakerBase : IMaker { }
```

- If the class is the standard (or only) implementation of an interface, then use this formulation:

```
/// <summary>
/// The standard implementation of the <see cref="IMaker"/> interface.
/// </summary>
public class Maker : IMaker { }
```

- If the class is one of several implementations, then use this formulation:

```
/// <summary>
/// An implementation of the <see cref="IMaker"/> interface that works with a
/// Windows service.
/// </summary>
public class WindowsServerBasedMaker : IMaker { }
```

8.1.5 Methods

- Parameters should be documented in the order that they appear in the method definition.
- Use the word `"given"` to refer to parameters.
- Summary documentation should not simply restate what is obvious from the method name and signature unless there is really nothing else to say.

[1] *Ghostdoc* will make a copy of documentation from overridden or implemented interface methods. Do not use this documentation; used `<inheritdoc/>` instead.

- The documentation should indicate which values are acceptable inputs for a parameter (e.g. whether or not it can be `null` or empty (for strings) or the range of acceptable values (for numbers). The example below demonstrates all of these principles:

```
/// <summary>
/// Fills the <see cref="Body"> with random text using the given
/// <paramref name="generator"> and <paramref name="seedValues">.
/// </summary>
/// <param name="generator"> The generator to use to create the random text;
/// cannot be <c>null</c>.</param>
/// <param name="seedValues">The values with which to seed the random generator;
/// cannot be <c>null</c>.</param>
void FillWithRandomText(IRandomGenerator generator, string seedValues);
```

- For methods that return a Boolean value, use the following form:

```
/// <summary>
/// Returns <c>true</c> if the value of <paramref name="prop"/> value has
/// changed since it was loaded from or stored to the database; otherwise
/// <c>false</c>.
/// </summary>
/// <param name="obj">The object to test; cannot be <c>null</c>.</param>
/// <param name="prop">The property to test; cannot be <c>null</c>.</param>
/// <returns>Returns <c>true</c> if the value has been modified; otherwise
/// <c>false</c>.</returns>
bool ValueModified(object obj, IMetaProperty prop);
```

- It is highly recommended that exceptions thrown by a method be documented.
- Exceptions should begin with "If…" as shown in the example below:

```
/// <summary>
/// Gets the <paramref name="value"/> as formatted according to its type.
/// </summary>
/// <param name="value">The value to format; can be <c>null</c>.</param>
/// <param name="hints">Hints indicating context and formatting
/// requirements.</param>
/// <returns>The <paramref name="value"/> as formatted according to its
/// type.</returns>
/// <exception cref="FormatException">If <paramref name="value"/> cannot be
/// formatted.</exception>
string FormatValue(object value, CommandTextFormatHints hints);
```

8.1.6 Constructors

- Though you can use inherited documentation for constructors, this is not recommended; instead, you should be as specific as possible on the parameter documentation. The default constructor documentation created by *Ghostdoc* is generally quite good.
- Where possible, the documentation for a parameter should consist only of indicating to which property the parameter is assigned and the acceptable inputs (as with other methods). Let the linked property documentation describe the effect of the parameter; this accounts for many constructor parameters.
- Parameters assigned to read-only properties should use the form "Initializes the value of …" and parameters assigned to read/write properties should use the form: "Sets the initial value of …"

The example below shows a class with constructor and properties documented according to the rules given:

```
class SortOrderAspect
{
  /// <summary>
  /// Initializes a new instance of the <see cref="SortOrderAspect"/> class.
  /// </summary>
  /// <param name="sortOrderProperty">Initializes the value of <see
  /// cref="SortOrderProperty"/>.</param>
  /// <param name="sortOrderProperty">Sets the initial value of <see
  /// cref="Enabled"/>.</param>
  public SortOrderAspect(IMetaProperty sortOrderProperty, bool enabled)
  { }

  /// <summary>
  /// Gets the property used to create a manual sorting for objects using the
  /// <see cref="IMetaClass"/> to which this aspect is attached.
  /// </summary>
  /// <value>The property used to create a manual sorting for objects using the
  /// <see cref="IMetaClass"/> to which this aspect is attached.</value>
  IMetaProperty SortOrderProperty { get; private set; }

  /// <summary>
  /// Gets or sets a value indicating whether this <see cref="SortOrderAspect"/> is
  /// enabled.
  /// </summary>
  /// <value><c>true</c> if enabled; otherwise, <c>false</c>.</value>
  IMetaProperty Enabled { get; set; }
}
```

8.1.7 Properties

- If a property has a non-public setter, do not include the "or sets" part to avoid confusion for public users of the property.
- Include both `<summary>` and `<value>` tags (they generate to different places in the documentation, despite their apparent redundancy). Yes, this means you will have repeated text (see the examples below).
- In general, copy the text from the "summary" to the "value", adjusting grammar so that it makes sense.
- The documentation for read-only properties should begin with "Gets" and the `value` elements should begin with the word "The".

  ```
  /// <summary>
  /// Gets the environment within which the database runs.
  /// </summary>
  /// <value>The environment within which the database runs.</value>
  IDatabaseEnvironment Environment { get; private set; }
  ```

- The documentation for read/write properties should begin with "Gets or sets", as follows:

  ```
  /// <summary>
  /// Gets or sets the database type.
  /// </summary>
  /// <value>The type of the database.</value>
  DatabaseType DatabaseType { get; }
  ```

- Boolean properties should have the following form, formatting the value element as follows:

```
/// <summary>
/// Gets a value indicating whether the database in <see cref="Settings"/> exists.
/// </summary>
/// <value><c>true</c> if the database in <see cref="Settings"/> exists;
/// otherwise, <c>false</c>.</value>
bool Exists { get; }
```

- For properties with generic names, take care to specify exactly what the property does, rather than writing vague documentation like "gets or sets a value indicating whether this object is enabled". Tell the user what "enabled" means in the context of the property being documented:

```
/// <summary>
/// Gets or sets a value indicating whether automatic updating of the sort-order
/// is enabled.
/// </summary>
/// <value><c>true</c> if sort-orders are automatically updated; otherwise,
/// <c>false</c>.</value>
bool Enabled { get; }
```

8.1.8 Full Example

The example below includes many of the best practices outlined in the previous sections. It includes `<seealso>`, `<exception>` and several `<paramref>` tags as well as clearly stating what it does with those parameters and their acceptable values as well as including extra detail in the `<remarks>` section instead of the `<summary>`.

```
/// <summary>
/// Copies the entire contents of the given <paramref name="input"/> stream to the
/// given <paramref name="output"/> stream.
/// <param name="input">The stream from which to copy data; cannot be
/// <c>null</c>.</param>
/// <param name="output">The stream to which to copy data; cannot be
/// <c>null</c>.</param>
/// <remarks>
/// Uses a 32KB buffer; use the <see cref="CopyTo(Stream,Stream,int)"/> overload to
/// use a different buffer size.
/// </remarks>
/// <seealso cref="CopyTo(Stream,Stream,int)"/>
/// <exception cref="ArgumentNullException">If either the <paramref name="input"/> or
/// <paramref name="output"/> is <c>null</c>.</exception>
/// <exception cref="ArgumentException">If the <paramref name="input"/> cannot be read
/// or is not at the head of the stream and cannot perform a seek or if the
/// <paramref name="output"/> cannot be written.</exception>
public static void CopyTo(this Stream input, Stream output)
```

8.2 Testing

- All code paths should be tested in a high-level manner, using integration tests; there is no utility in doing method testing.[1]
- Use the `nUnit` testing framework to create tests.[1]

[1] Classic unit-testing involves testing each and every property and method individually, which is largely a waste of time and often fails to test the interaction between components, which is actually more important.

8.3 Releases

- Include range-checking if it doesn't hamper performance.
- Include debugging information.
- Include code optimization.

[1] Both *ReSharper* and *TestMatrix* offer excellent *Visual Studio* integration for *nUnit* testing. *Visual Studio's* own *msTest* is currently being considered as a replacement for *nUnit*.