``

C# Coding Conventions

and

Best Practice Guidelines

DRAFT

08/26/2016

By

Barry Feigenbaum, Ph. D.

``

TOC

``

``

``

``

``

Introduction

TODO: use consistent font on code examples.

Note: This is a working document and will be extended often; add new items at the end only. Need to add examples in several items.

Coding Conventions are rules/guidelines/recommendations intended to achieve increased consistency of coding style. Adherence to coding conventions is critical for code quality - it improves readability, learnability and maintainability. Lack of conventions or adherence to a mix of conflicting convention only causes confusion to any reader and needs to be avoided. Lack of or inconsistent use of conventions is a serious form of technical debt.

From http://en.wikipedia.org/wiki/Computer_programming:

In computer programming, readability refers to the ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It affects the aspects of quality above, including portability, usability and most importantly maintainability.

Readability is important because programmers spend the majority of their time reading, trying to understand and modifying existing source code, rather than writing new source code. Unreadable code often leads to bugs, inefficiencies, and duplicated code. A study[14] found that a few simple readability transformations made code shorter and drastically reduced the time to understand it.

Following a consistent programming style often helps readability. However, readability is more than just programming style. Many factors, having little or nothing to do with the ability of the computer to efficiently compile and execute the code, contribute to readability.[15] Some of these factors include:

- *Different indentation styles (whitespace)*
- *Comments*
- *Decomposition*
- *Naming conventions for objects (such as variables, classes, procedures, etc.).*

``

Often there are multiple ways to approach a problem and thus multiple forms of code to implement the associated function. A *Best Practice* is the form of code accepted as the "best" (per context) approach to solving a coding problem. This document describes several best practices to be considered.

From http://en.wikipedia.org/wiki/Best_practice:

*A **best practice** is a method or technique that has consistently shown results superior to those achieved with other means, … can evolve to become better as improvements are discovered. Best practices are used to maintain quality ….*

All code in new source files should follow these conventions and practices. Code in existing source files should retrofit to these conventions/practices as time permits; any such retrofits should be done a whole source file (at least) at a time.

Assessment of conformance to these conventions/practices shouldl be done during code reviews. Absolute compliance is not (typically) required, but a high level of compliance is expected.

Retrofitting conventions (and especially best practices) can be both tedious and time consuming. The more code that exists in a code base to retrofit, the more work there is to do. Therefore, if you plan to conform to any such conventions in a code base, do so as early in the development of the code base as possible.

This is a pay-me-now or pay-me-more-later situation. If you wait too long the technical debt piles up and the recovery effort can become so large that the work will never get done; eventually the code base can become so unmaintainable that it will need to be abandoned, very often at a particularly bad time from a competitive business point of view.

Note: most of the items in this document were identified by reading existing code bases and observing (often multiple) violations of these conventions in code snippets. Some examples of correct code, based on code from thses code bases, violate multiple (not all fixed in the each example) conventions; they do not void the need to conform to all those other conventions.

Items

Items are in no particular order.

Context Item Importance Indicator legend (used on item headers):

1. **MD** Must Do – A direct source of actual or potential bugs if violated or greatly improves readability/ maintainability; has code class indicator

``

2. **I** Important – Significantly improves readability/maintainability
3. **NTH** Nice to Have – helps improve readability or is generally beneficial
4. **X** Not important or just to keep in mind
5. **TBD** To be Determined – importance is yet to be determined

Code Classification Legend (currently applied only to MD items):

- Ren Rename variable/type/method/namespace
- Mov Move code to new location
- Com Comment add/change/remove
- Cod Localized (say within a method/block) code change
- Fac Refactor code (generally extensive (cross method/type) change)
- Dec Declaration change

The above importance levels apply most strongly to pre-existing code. All conventions are considered to be at least important to new code (ex. new source file) and should always be applied to any new code. On pre-existing code, development tasks should be created and scheduled to apply (at least) all the MD and some I items; subsequent code reviews should not pass until all MD items (in a single source file) are applied. Any time a source file is opened for an extensive (not localized to a small part (say a single method) of the source file) change/enhancement is also an opportunity to apply any of these items.

I - Only One Public Class per File

It is best if there is only one public type (class, interface, enum, etc.) per source file. The file name should match the type name (and the file's path should match any namespace). This makes it easier to find the source of a type. As they are only one-liners, delegate and event types are often an exception.

Public top-level (vs. nested) helper classes, such as enums, used only in the classes in the same source file, can be collocated in the same file but its best to externalize them (or make them non-public) so other source files can also use them.

This may explode the sheer number of source files, but it is worth it for management reasons. This also helps keep source files small; see I - Limit Source File Extents.

``

I - Namespace Names use Initial Capital Words

Namespaces group types and values into distinct spaces such that the same name can be used to identify different things in different spaces.   They are always public.

Namespace names should consist of a sequence of names separated by periods (“.”). Each such name shall be an initial capital letter followed by lowercase letters.

- Avoid use of numeric digits or graphic characters (Ex. “Stream1”).   Do NOT use underscores (“_”) (Ex. “XXX_YYY_ZZZ”); replace the underscore with a period (Ex. “XxxYyyZzz”).
- Each name should be a single word; typically, either a noun or adjective or (rarely) a verb.
- Avoid compound names (Ex. “XXXAccessLib”); split them with “.” (Ex. “Xxx.Access.Library”). It is OK to violate this convention where the word sequence represents one thing. For example an name like “`VirtualStandbyAccessLib`” can be mapped to a namespace name with fewer periods like “`VirtualStandby.AccessLibrary`”.
- Avoid the use of plural names (ex. “Contracts”).
- Avoid abbreviations (ex. “Lib” for “Library”).

I - Public Constants Use All Uppercase

All public constant (`const`) declarations should use all uppercase names separated by underscores (“_”),.

For example: `THIS_IS_A_PUBLIC_CONSTANT`.

Non-public constants can use normal variable name conventions or the above style.

I - Public Names should Start with Uppercase and use Camel-case

Any public identifier, such as a type, method or property name, should start in uppercase. The rest of the name should use camel-case (each word is initial caps with a lowercase body).   Do not use embedded underscores (“_”) in non-constant names.

For example: `ThisIsACamelCaseClassName`.

Avoid names like: `RestartXXX_InitializationWorkaround`.

I - Protected/Private Names should Start with Lowercase and use Camel-case

``

Any protected or private identifier, such as a type, method or property name, should start in lowercase. The rest of the name should use camel-case (each word is initial caps with a lowercase body). Do not embedded use underscores ("_") in non-constant names. Generally, types and properties should be public.

For example: thisIsACamelCaseMethodName.

I - Use Initial Lowercase Camel-case Names for Local Variable and Parameter Names

Always use initial lowercase names for local variables. Do not use underscores ("_") in local names.

For example: thisIsAGoodLocalName.

Avoid names like: restartXXX_InitializationWorkaround_Attempted.

Avoid using underscores (especially as first or last characters) in non-field names:

```
public VirtualMachine(string _host, string _path, HyperVisorType _type)
{
  this.host = _host;
  this.path = _path;
  this.type = _type;
}
```

Instead do this:

```
public VirtualMachine(string host, string path, HyperVisorType type)
{
  _host = host;
  _path = path;
  _type = type;
}
```

NTH - Use Short Names in For Clauses

Too much use of long names can reduce readability, especially in narrow contexts. Favor shorter names as "for" loop variables (both for and foreach). Use acronyms for long type names. For collections, use the name of the element type as a basis. For integer indexes, use the highly conventional letters I, j, k, l, m, n (as needed for nesting); reserve these identifiers for use only as loop indexes. In general, use a "var" declaration (vs. an explicit type) for the index.

``

For example : `foreach (var oc in listOfOrderedCollection) { … }.`

And: `for (int i = 0; i < xxx.length; i++) { … }`

More examples:

Avoid this verbose form:

```
foreach (Configuration configuration in Configurations) …
```

Instead Do:

```
foreach (var c in Configurations)  …
```

I - Use Shorter Names for Locals and Longer Names for Fields, Methods, Properties and Types

Too much use of long names can reduce readability. Local names (which should have very limited range of reference (method or block)) should be fairly short (often 10 or fewer characters). They are often acronyms. Global (used beyond one method) names, such as type, method and field/property names, should have longer names (generally 10 or more characters). Global names need to be very descriptive of their usage so they are often multi-word phrases.

For example, in a local context, instead of this:

```
List<Configuration> tempConfigurations;
```

Do this:

```
List<Configuration> aecl;
```

Or when initialized (and the type can be inferred), be even briefer and do this:

```
var aecl = …;
```

X - Avoid using just Type Names as Variable Names

``

Names like `stringBuilder` /`aStringBuilder` or `file` /`aFile` are to be avoided as they provide little added value to the name. It is OK to use type names if they help but a role description is also needed.

For example:
`File inputFile = …, outputFile = …` are appropriate.

If the use range is very limited, then abbreviated names like this are OK:

```
StringBuilder sb = …;
```

NTH - Use “S”/”A” suffix for Collection Types

When declaring variables of collection (including array) types, add a short plural type indicator. Avoid longer qualifiers such as “List” or “Array” that are repeating the declaration.

For example: `List<File> files = …` (instead of `List<File> fileList = …`).

Use `File[] files …` then `File f = …`:

```
File[] files = getAllFiles(…);

foreach(File file in files) { … }
```

Or for narrow contexts, an even briefer use:

```
File[] fa = getAllFiles(…);

foreach(File f in fa) { … }
```

Or use:

```
List<File> files = getAllFiles(…);

foreach(File file in files) { … }
```

``

MD – ren - Acronyms in Names should be Initial Caps

Any name with an embedded acronym should use mixed case. Otherwise they are hard to read and confusing. Never use underscore ("_") to separate acronyms.

For example, to use WWW and URL in a name, use this form: `WwwUrlOperation`.

Never abut acronyms if they are both in all uppercase (ex. `WWWHTTPServer` should be `WwwHttpServer`).

MD – mov - Place Field Declarations near the Beginning of the Containing Type/Namespace

Place all field declarations in a consistent position (near the either the top/bottom of the type that contains them). Place all methods (except access methods/properties for fields) in a consistent position (either before or after all the fields).

An exception is for fields used as properties (with explicit or automatic get/set methods). They should be located adjacent to the get methods.

MD – mov - Place Constructors and Object Method Overrides after all Fields/Properties and before any other Methods

Place constructors after all fields/properties and before any other methods that follow fields.

Place any overrides of inherited methods (ex. `ToString()`) before any other methods.

MD – ren - Always Initialize Local variables

Uninitialized local variables can have arbitrary values, so always supply an initial value on the declaration, even if it may be reassigned to the same value later.

An exception is when the value is set immediately after the declaration inside an `if/else`, `for` or `switch` (and possibly simple try/catch) statements.

``

Note: any read of an uninitialized variable should get a compiler error.

I - Avoid Initializing Fields to Default Values in Constructors

Setting a field to its default (or initialized) value in a constructor is redundant. Only set the field value if the value is not the default value.

Do not do this:

```
public class MyClass
{
  private int count;  // implied set to 0
  :
  public MyClass()
  {
      count = 0;  // avoid this
  }
}
```

NTH - Prefer Initializing Fields in their Declarations (vs. Constructor)

If you can create the desired initial field value in a simple expression (or via a simple static method), then it is preferred to set the initial value of a field in its declaration (vs. using a constructor). This creates a reference pattern to be used to determine the initial value. Using a constructor may cause the field to be set multiple times (to its default, and then to the constructor's value). Also it is easy to miss initializations if multiple constructors exist.

Avoid setting a field's value in both the declaration and a constructor, even if the value is the same.

It is unnecessary to explicitly initialize fields to their default types. It is preferred to omit the value altogether (to imply the default value) rather than explicitly set the default value.

For example, instead of:

``

```
int counter = 0;  // avoid, 0 is redundant
```

Just do this:

```
int counter;
```

MD – ren - Make Field Names easy to Distinguish

In the absence of IDE tools (ex. *Visual Studio*) that do this (say by color) it is best if all fields look different from local variables. The recommendation is to prefix instance fields with an underscore (“_”) and prefix and suffix static fields with underscores. Avoid using any “typed” qualifiers (such as “m_” for non-static member). Avoid using “__” (vs “_”) to distinguish static from non-static fields; they look too much alike.

Example instance field: `private int _count;`

Example static field: `private static int _count_;`

An alternate is to prefix all instance field references with “`this.`” and static field references with “`<typename>.`”. If this option is used, this must be used in all cases in the same source file (this option is hard to enforce and thus not recommended). This option is often used in constructors to distinguish field references from parameter names when the same name is used; this is acceptable.

Example instance field: `int x = this.count;`

Example static field: `int x = MyType.count;`

MD – ren - Distinguish Methods Names from Type/Value Names

As both public type names and method (and field/property) names are often both in initial capitals, some easy to recognize means to distinguish them is needed. Therefore, all type and value names (classes, interfaces, enums, properties, etc.) are to be nouns while all method names are to be verbs.

``

MD – ren - Start Interface Names with “I”

In order to make it easy to distinguish class and interface names, start all interface names with an “I” character. For example: `IFileStream`. All implementations should end with the “core” interface name (after the “I”, or a subset of it when it has many words).

For example:

`MyFileStream` (or briefer `MyStream`).

If there is a preferred default implementation, use just the core name or an augmented form (such as an Abstract” or “Base” for abstract classes or “Basic” or “Default” prefix for concrete classes).

For example:

`AbstractFileStream`.

NTH - Maximize use of Dependency Injection

Each unit (typically a public class) needs to be able to be tested independently and optimally reused in unpredictable circumstances. To make this easier, avoid creating (often in a constructor) instances of dependent objects (an object your code needs to function) in the code that depends (consumes) on the object. Instead provide setters (and optionally overloaded constructors) that take the dependent object[1] as a parameter. Then set the dependent class in the same code that creates the consuming class. This moves the responsibility for acquiring dependencies from the consumer to its constructing method (potentially recursively up to the “main” method).

*** add example ***

I - Avoid Heavyweight Types/Objects

Construction of objects should, in general, always succeed. Any object should be able to be reliably (no exceptions thrown) created in a default state via a default constructor. Such an object is *lightweight*. If the object requires complex (often time consuming or error prone) action just to be created it is

[1] Or parameters used to create the object (ex. the name of file instead of a file object)

``

heavyweight. Always provide a Default constructor to do lightweight initialization. Lightweight objects allow circular dependencies.

Lightweight objects may not be fully operational (i.e., may require additional configuration before use) after construction. For example, consider a File object; it can be created but cannot be used to read data until an “open” operation is performed. Often the configuration method is called “init” or “Open” or “Configure”. It is best that if an operational method is invoked on a non-configured instance, that some exception be thrown at that time (ex. do a “read” on an un-open file). Most lightweight objects that have an initialization method also provide a “Close” (or “Dispose”) function to use when processing is complete.

Alternatives are using one or more setter methods and/or having additional constructors that take configuration parameters. If the parameters of such initialization are other objects, this is a form of dependency injection.

For example:

```
File f = new File(“myfile”);

try (f.Open()) {

  string[] lines = f.readAllLines();

  // process the lines

} finally {

  f.Close();

}
```

X - Take Care Calling Virtual Methods in Constructors

In the .NET runtime there is a (dubious) design in that it processes constructor calls in subclass to superclass order yet it does instance field initialization in superclass to subclass order. If a superclass constructor calls a virtual member function this can lead to the situation where, when the virtual method is running, the fields in its class (or any below the class invoking the method) are not yet initialized to any provided initialization value (default values will be set). So if you do call virtual methods, they must be written to deal with this situation (ex. a reference field may be (temporarily) null even when initialized to a non-null value).

``

MD – com - All Public Methods should have a Comment

All public (and recommended protected) methods need (at least) a remark (“// …”) describing its function. Use of documentation comments (“/** … */” or “/// …”) are preferred. If the method implements an interface method the comment may be omitted if the interface declaration provides a comment for the method.

MD – com - All Public Types must have a Comment

All public (and suggested protected) type declarations need commentary. Use of documentation comments (“/** … */” or “/// …”) is required. The commentary must describe the purpose of the type. The detailed function of the type can be left to any method commentary.

I - Limit Source File Extents

A single source file should have a reasonable size. Source files having more than (say) 1000 lines of code are probably too long. Split the source (use partial classes or better yet, break the contained classes into smaller forms in separate source files) to maintain this limit.

Each line should have a reasonable length. Entire lines should stay visible without scrolling (on modern displays) so a maximum length of between 100 and 120 characters often appropriate. Split/wrap lines to maintain this limit. Indent any secondary segments or split lines several spaces right of the indentation of the first segment. If multiple secondary segments exists it is best if they all align at the same column.

In general, use spaces instead of tabs to indent. Spaces are the same width on all systems, tabs are not.

MD – com - Use Spacing Consistently

Erratic spacing, especially line spacing, causes reduced code readability. Use spacing consistently.

Use (only) a single blank line between sections (methods, blocks, types, #region, related declarations, etc.). Avoid using blank lines before block closures (i.e., before “}”).

``

For example, the following:

```
switch (typeName)

{
     case "Hyper-V":
         virtualStandby = …;
         break;
      case "VMWare ESX(i)":
         virtualStandby = …;

         break;

      case "Oracle VirtualBox":

          virtualStandby = …;
          break;

      case "VMWare Workstation":

          virtualStandby = …;

          break;
      default:
          virtualStandby = …;          break;
}
```

Is better with this spacing:

```
switch (typeName)

{
     case "Hyper-V":
         virtualStandby = …;
         break;

      case "VMWare ESX(i)":
         virtualStandby = …;
         break;

      case "Oracle VirtualBox":
          virtualStandby = …;
          break;

      case "VMWare Workstation":
          virtualStandby = …;
          break;
```

``

```
      default:
          virtualStandby = …;
          break;
}
```

Do the same in expressions. In particular, always separate binary operators from their operands by a space. Never separate a unary operator from its operand. Spaces are optional, but recommended, around parens; the main exception is parens used on method calls, if and for statements.

For example, instead of this:

```
int a=(x+y)*2+--z;
```

Do this:

```
int a = ( x + y ) * 2 + --z;
```

This is OK:

```
int a = myfunc((x + y) * 2 + --z);
```

When indenting code (ex., split statements, nested blocks, etc.), use a consistent level of indentation. Typically use two or four spaces for blocks and three or five spaces for split lines. This is required within a method and should be applied across a source file.

MD – fmt - Always use Blocks after Grouping Statements

On `if/else`, `while`, `do`, `using`, `try/catch/finally`, for and `foreach`, always use a block statement, even if the block contains only zero or one statement. An exception is a `using` statement that is the body of another `using` statement; only the most nested `using` statement requires a block.

For example, instead of:

```
if(x > 10) y = x; else y = 0;
```

``

Or:

```
if(x > 10)
   y = x;
else
  y = 0;
```

Do this:

```
if(x > 10)
{
   y = x;
}
else
{
  y = 0;
}
```

It is recommended tp always place the braces (“{“ and “}”) on their own lines aligned with the parent statement but the style

```
if(x > 10) {
   y = x;
}
```

Is also acceptable if done consistently (in same source file).

The above allows easy addition of lines in any block.

It also helps prevent the following class of error:

```
if(x > 10)
   if(z > 2)
```

```
``
     y = x;

else

  y = 0;
```

Which the compiler interprets as:

```
if(x > 10)

   if(z > 2)

     y = x;

   else

     y = 0;
```

I - Use Ternary Expressions on Short Relational Expressions

Use Ternary Expressions (vs. `if/else`) on short expressions (short being defined as fits easily on one line). Do this especially on initial values for declarations.

(Too) Verbose form:

```
int x;

if(x >= 0) {

  x = 10;

}

 else {

  x = -10;

}
```

Preferred form:

```
int x = a >= 0 ? 10 : -10;
```

``

NTH - Avoid Testing Booleans with Equals

Use just the Boolean variable as a test.

Avoid :	`if(aBoolean == true) …`

Preferred:	`if(aBoolean) …`

Avoid:	`if(aBoolean == false) … or if(aBoolean != true) …`

Preferred:	`if(!aBoolean) …`

NTH - Return Boolean Results Directly

Return expressions that result in Boolean values directly.

For example, instead of:

```
if(x > 10)

{

  return true;

}

else

{

  return false;

}
```

Use the briefer:

```
return x > 10 ? true : false;
```

Best just use:

```
return x > 10;
```

``

I – Use Methods Over Operators

Whenever a method exists that subsumes the role of an operator, use the method instead. These methods generally exist to add value beyond the operator's basic behavior.

For example, instead of:

```
if(name == "barry") ...
```

Instead do:

```
if(Equals(name, "barry")) ...
```

Note: Equals() above is really the static Object.Equals() method. Object.Equals() behaves well (consistently) when either operand is null.

Some methods exist that replace certain combinations of operators. Use them instead of the multiple operators.

For example, instead of:

```
if(name != null && name.Length() > 0) ...
```

Instead do:

```
if(!string.ifNullOrEmpty(name)) ...
```

Often these methods are static methods of various system types. Some common examples are:

```
Object.Equals(object o1, object o2)

Object.ReferenceEquals(object o1, object o2)

Overloaded String.Equals(string o1, string o2)

Overloaded String.Compare(string s1, string s2)
```

``

```
Multiple String.IsNullOr???(string s1, string s2)

Overloaded String.Concat(string s1, string s2)

Overloaded String.Join(string s1, string s2)
```

MD – cod - Test Literals First

When doing comparisons between variables/expressions and literals, always use the literal on the left side of the comparison.

For example, instead of:

```
if(name.Equals(“barry”)) ...
```

Instead do:

```
if(“barry”.Equals(name)) ...
```

A safer, and thus recommended, alternate, where operand order is immaterial, is to use the static `Object.Equals()`:

```
if(Object.Equals(“barry”, name)) ...
```

Note: `Object.Equals` can be expressed as just `Equals`, so this is sufficient:

```
if(Equals(“barry”, name)) ...
```

This can significantly reduce the number of null reference exceptions.

MD – fac - A Type Should have a Limited Role

A type (i.e., Interface or Class) should provide limited and cohesive function. Classes or interfaces with many methods typically indicate a non-cohesive condition. Often over time function is added to a type

``

until it becomes a "spaghetti" type. When this happens refactoring is needed. These types should be broken down in to less functional types each with a single purpose. A good rule of thumb is no more than (say) twenty public methods in a class and (say) six or fewer in an interface.

Remember that interfaces are additive (can be multiply inherited) so err on the side of small interfaces. Often single method interfaces are best.

I - Keep Classes Small

Classes that have a lot of methods, especially large methods, are difficult to understand and maintain. Refactor these classes into smaller, more digestible, forms. Often this can be done by breaking the single class into a hierarchy of classes. Move the general methods into the one or more super-classes and the detailed methods into one or more subclasses. Often an existing method needs to be split to extract the general and specific portions of it. A good rule of thumb is keeping a class under (say) 500 lines of code, including field/property declarations. Certainly classes over 1000 lines should be split up.

Often large size is due to use of repeated cut & paste, so that can easily be reduced by method refactoring. Another common cause is implementing large or multiple interfaces. Reducing the number and/or size of the implemented interfaces can allow the implementation class to be broken into smaller classes.

I - Keep Method's Purpose and Size Reasonable

A method's source code should be contained in at most one "screen" (<< page if printed, window[2] if displayed) that is visible without scrolling. It is better to err on the side of smallness (ideally < 20 lines; best if (much) less than 50 lines, certainly less than 500 lines). If necessary, create smaller methods by splitting out the insides of a larger method into private (or perhaps protected) methods.

Often bodies of loops or `if/else` clauses can be made into such helper methods very easily (some IDEs have an extract function that automates this). If this is done, methods of less than fifty lines will be typical.

Large methods tend to be doing more than one task. Each task can be split out into a distinct method. This greatly increases the chance of reusability of methods (vs. cut & paste duplication).

[2] Of reasonable size, say at least 25-30 lines.

``

Note: making methods smaller helps in interpreting exception stack traces. Often they indicate the method in error but not the line number. If methods are small, it helps make the location of the exception more obvious.

NTH - Make All (non-static) Protected Methods Virtual

A protected method exists to allow a subclass to use it. Any subclass should also be able to override it, so it needs to be `virtual`.

NTH - Make Most (non-static) Public Methods Virtual

Unless the class is sealed, public methods exist to allow a client- or a sub-class to use it. Any subclass should be able to override it, so it needs to be `virtual`. In general, be in the habit of adding the `virtual` keyword to all public instance method declarations. I.e. err on the side of being too `virtual`.

NTH- Do not Create Protected Methods/Fields in Sealed Classes

A `sealed` class cannot be extended, so there is no purpose for a new protected method or field to be declared. Use private instead.

MD – cod - Provide only One Primary Constructor

A "primary" constructor is one the makes sure all fields are initialized (by default or explicit initialization) and calls a primary super constructor, if any. There should only be one such constructor (often the default constructor or a constructor with a parameter for all fields). All other constructors should (directly or indirectly) call this constructor. This ensures full initialization of objects even as new fields are added to the class.

``

I – Provide a Copy Constructor for Every Class

A *Copy Constructor* is a constructor that has a single parameter of the class type. Its purpose is to make the newly created object be "equal" (but not identical) to the argument object. Often this is achieved by copying all fields from the parameter instance to the new instance. The copy can be shallow or deep (depending on the structure of the class). In a deep copy, all objects are duplicated (often recursively); in a shallow copy, only the top-level objects are duplicated and the lower levels contain just references. For simple objects (like the example below) they are the same.

For example:

```
class MyClass

{

  int x, y, x;

  string name;

  :

  // provide a copy constructor

  public MyClass(MyClass other)

  {

    this.x = other.x;

    this.y = other.y;

    this.z = other.z;

    this.name = other.name;

  }

  :

}
```

Some classes, especially ones that can support both shallow and deep copies, offer explicit copy methods:

```
``
public MyClass shallowCopy()

{

}

public MyClass deepCopy()

{

}

public MyClass copy()

{

  return deepCopy();   (or shallowCopy() as appropriate)

}
```

This makes it easy to clone instances. To make a clone:

```
MyClass source = = …;

Myclass clone = new MyClass(source);

Assert(Equals(source, clone));
```

Some classes implement the *ICloneable* interface. Having a copy constructor makes this easy.

```
public Object Clone()

{

  return new MyClass(this);  (or copy() if implemented)

}
```

Note: it is recommended all data-oriented classes implement ICloneable.

I - ToString Should Include the Type Name

``

Except for basic types, like strings, numbers and enums, the `ToString()` method (required for all public classes) should use consistent formatting. Always include the type name. ToString should always be declared virtual.

For example:

```
public virtual override string ToString()
{
   return GetType().Name + ‘[‘  +  <any fields> + ‘]’;
}
```

All (included) fields should be of the form: <name>”:”<value> (or <name>”=”<value>) and separated by “, “.

An alternate (and perhaps preferred) implementation is:

```
public virtual override string ToString()
{
   StringBuilder sb = new StringBuilder();
   sb.Append(GetType().Name);
   sb.Append(‘[‘]);
   bodyToString(sb);
   sb.Append(‘)‘]);
   return sb.ToString();
}
```

Where in top most class:

```
protected virtual void bodyToString(StringBuilder sb)
{
   sb.Append(<any field>);
   :
```

```
``
}
```

Where in subclasses:

```
protected override string bodyToString(StringBuilder sb)
{
   base.bodyToString(sb);
   sb.Append(<any field>);
   :
 }
```

NTH - ToString/Equals/GetHashCode/CompareTo Should Support Inheritance

To support inheritance, `ToString()` should be written like this (in the top most class):

```
public virtual override string ToString()
{
   return GetType().Name + ‘[‘  +  bodyToString() + ‘]’;
}
```

Where in top most class:

```
protected virtual string bodyToString()
{
   return <any fields>  or “”;
}
```

Where in subclasses:

```
protected override string bodyToString()
{
   return base.bodyToString() + “, “ + <any added fields>;
}
```

``

The <any … fields> string should be created by use of `String.Format()`.

The `Equals`, `GetHashCode` and `CompareTo` methods should follow a similar pattern.

NTH - ToString Should be Concise

The `ToString()` function is primarily used for diagnostics (tracing, etc.). It should output only critical (not necessarily all) field information, generally in single line format. If the type needs to format itself for easier human consumption, provide another method (recommend the name `HumanToString`) for this purpose. `HumanToString` can use multiple lines, indentions, etc., as fit to make the output more readable.

NTH - All Public Classes Should provide an Explicit ToString

If the class is public (available for general use) and adds fields, it must provide an explicit `ToString` method to support printing it in diagnostics. All classes that add fields (original or sub-classed) should provide this method; see I - ToString Should Include the Type Name.

NTH - All Public Classes Should provide Explicit Equals, GetHashCode and CompareTo

If the class is public (available for general use) and adds fields, it should provide explicit `Equals` and `GetHashCode` methods for comparisons. The methods should be based on field values (only rarely should the default identity comparison be used). All classes that add fields (original or sub-classed) should provide these methods; see I - ToString Should Include the Type Name.

If the type is comparable (instances can be sorted) then implement the `ICompareable` interface. From MSDN:

*The role of **IComparable** is to provide a method of comparing two objects of a particular type. This is necessary if you want to provide any ordering capability for your object. Think of **IComparable** as providing a default sort order for your objects. For example, if you have an array of objects of your type, and you call the **Sort** method on that array, **IComparable** provides the comparison of objects during the sort.*

For example, in a `Car` class (that implements `ICompareable`):

``

```
public int CompareTo(object obj)
{
   Car other = (Car)obj as Car;
   if(other == null)
   {
       throw new ArgumentException(“other not a Car”);
   }
   int rc = string.Compare(this.make, other.make);
   if(rc == 0)
   {
      rc = string.Compare(this.model, other.model);
   }
   if(rc == 0)
   {
      rc = this.year - other.year;
   }
   return rc;

}
```

In general, using the generic form of `ICompareable<T>` is preferred. For example, in a `Car` class (that implements `ICompareable<Car>`):

```
public int CompareTo(Car other)
{
   // Note: no type tests needed
   // same “rc” code as above

}
```

Note: `Equals` and `GetHashCode` must be consistent. Any equal instances MUST have the same hash code value. In general, any comparison only uses fields included in `Equals`.

It is highly recommended that the hash code not change over the like of an instance (especially if the instance can ever be used as a dictionary/hash key). This is easily achieved by my making the instance immutable.

MD – cod - Avoid Changing Equals/GetHashCode upon Mutation

`Equals` and `GetHashCode` are used to place objects into hashes (ex. Dictionary). For objects in hashes, if an object’s hash code changes, the hash must be re-indexed. Often this is a forgotten action

``

leaving the hash broken. It is better if Equals and GetHashCode are agnostic to any mutation of the object.

Therefore, in general, it **is best if only immutable types[3] be used as hash keys** therefore avoid mutable types as hash keys.

X - Never Lower the Visibility of a Type/Field/Method

Unless you have code access to all client code, never reduce the visibility (ex. public ➔ protected ➔ private) of any type, field, property or method. Err on the side of creating such items with narrow visibility (i.e., use private); visibility can always be increased (ex. private ➔ protected ➔ public). Avoid all use of default visibility; explicitly state the visibility on all (non-local) declarations.

X - Never Remove non-private Members

Unless you have code access to all client code, never remove a non-private type, field, property, constructor or method defined in a class. If the item is obsolete, mark it with an [Obsolete] attribute.

A corollary: never rename a non-private member. To rename a property or method, simply create a new alias property/method and delegate to the existing form. Use [Obsolete] on the existing form.

X - Do not Wrap Return Values in Parens

While popular in the C language (makes return look like an operator/function), it is not conventional in C#. Instead of this:

```
return(theResult);   or return (theResult);
```

Simply do this:

[3] Like numbers and strings.

```
``
return theResult;
```

MD – cod - Have only One Return Point per Method

Avoid using multiple `return` statements. In general, always exit at the end of the method. If necessary, use `if/else` to achieve this. Omit any `return` on void methods. For non-`void` methods declare an initialized return variable (suggested name `result`) at the start of the method and then return it at the end. Set the value in the body (often conditionally).

This greatly reduces the fragility of code to added paths over time.

Note: this rule does not cover exceptions (`throw` statements); you may throw as many exceptions as needed from any location in a method. If any cleanup is required, `use try/finally` to enforce such cleanup.

For example:

```
public int conditionalDoubler(int x)

{

  int result = -1;

  if(_aField > 0)

  {

    result = x * 2;

  }

  return result;

}
```

Here is a more realistic example:

```
/// Get a Logger

public static ILog getLogger()

{

    if (isInitialized && (logger != null))
``
    {

        return logger;

    }

    return null;

}
```

It should be like this:

```
/// Get a Logger

public static ILog getLogger()

{

    ILog result = null;  (or other appropriate default)


    if (isInitialized && (logger != null))

    {

        result = logger;

    }

    return result;

}
```

In particular, avoid code like this:

```
if (attributes.Length > 0)
{
     return attributes[0].Description;
}
else
{
     return value.ToString();
}
```

Instead do this:

``

```
return attributes.Length > 0 ? attributes[0].Description :
value.ToString();
```

Or this (especially if the case text is long):

```
return attributes.Length > 0
   ? attributes[0].Description
   : value.ToString();
```

Never use `return` in the body of a switch case. Avoid the following:

```
var status = …;
:
switch (status)
{
    case "0":
        return VirtualMachineStatus.Unknown;
    case "1":
        return VirtualMachineStatus.Other;
    case "2":
        return VirtualMachineStatus.Enabled;
    :
    case "10":
        return VirtualMachineStatus.Starting;
    default:
        return VirtualMachineStatus.Unknown;
}
```

Do this instead:

```
var status = …;
:
VirtualMachineStatus result;
switch (status)

{
    case "0":
        result = VirtualMachineStatus.Unknown;
    case "1":
        result = VirtualMachineStatus.Other;
    case "2":
        result = VirtualMachineStatus.Enabled;
    :
    case "10":
        result = VirtualMachineStatus.Starting;
    default:
        result = VirtualMachineStatus.Unknown;
}
```

```
``
return result;
```

Never return from inside a loop, such as:

```
Object vm = …;
:
foreach (var settingData in SystemSettingData)
{
   String data = settingData["DataRoot"];

   if (data.Equals(“target”))
   {
       return vm;
   }
}
```

Instead do this:

```
Object vm = …;
:
Object result = null;
foreach (var settingData in SystemSettingData)
{
   String data = settingData["DataRoot"];

   if (data.Equals(“target”))
   {
      result = vm;
      break;

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

X - Do not Mark the End of a Method with Comments

Avoid code like this:

```
/// Delete the current event source and event log

public void deleteEventSourceAndLog()
```

```
``
{

    :

} // deleteEventSourceAndLog()
```

Simply omit the end comment/remark.

If the I - Keep Method’s Purpose and Size Reasonable rule is used this is unnecessary. Most IDEs provide this service dynamically. Also it is tedious to do this and easily goes bad if the method is renamed (especially via IDE rename function).

I - Avoid Use of Continue in Loops

Use `if/else` instead of `continue` (exception is right after the start of the loop) in loops; especially from nested locations. Continue is fragile to change of logic flow in the body.

For example, instead of this:

```
for(int i = start; i < end; i++)

{

  if(i < 0)

  {

    continue;

  }

   : rest of loop

}
```

Do this:

```
for(int i = start; i < end; i++)

{

  if(i >= 0)
```

``

```
   : rest of loop
  }
}
```

I - Avoid Multi-level Breaks

Use break to escape only an immediate loop or switch. Avoid use of labels and the `break` (or `goto`) to label statement.

For example, avoid of this:

```
for(int i = start; i < end; i++)
{
  for(int j = start; j < end; j++)
  {
    for(int k = start; k < end; k++)
   {
      if(i < 0 || j < 0 || k < 0)
     {
       goto exit;
     }
     // do something with I, j and k
  }
}
exit:
```

I - Place Field Declarations and Accessor Methods Together

``

Place the field declaration and access methods together. This makes them easier to remove (or rename). Use a similar approach with field and property definitions. In general, prefer using properties (over fields and explicit get/set methods); take full advantage of anonymous fields when using properties.

Example:

```
private bool initialized;

public bool isInitialized()

{

    return initialized;

}

public void setInitialized(bool f)

{

    initialized = f;

}
```

MD – dec - Make All (non-constant) Fields Private

In general, all non-constant (`const`) fields (static and instance) should be declared `private` and then access methods/properties should be provided if they need to be used outside the declaring type.

The above rule is rather strong in one case; one can relax it to make fields `protected` so that they can be directly accessed by subclasses. If you do this remember it is difficult to make the field `private` in the future, so take care in making this decision.

I - Avoid "Hungarian" Notation for Names

While popular in some C code conventions do not try to "encode" (abbreviated type) the type of a variable into its name (if you must do this use the actual type name as a suffix). Avoid prefixes like "f" or "b" for flags, "i" for integers, "c" for counters, "str" (or "sz" or "lpsz") for strings, etc. Instead use more

``

descriptive names (that imply the type). This practice started due to the relative lack of type-safety in C. C# does not have this issue so it does not need this fix.

For example, instead of this:

```
int cUsers = 0;

Date dBirth;
```

Do this:

```
int userCount = 0;

Date birthDate;
```

MD – com - Make Sure Comments are Accurate/Understandable/Meaningful

Comments help any code reader understand the code. If the comments are obsolete or wrong, they become more harmful that not have any comments at all. So if you use comments, make sure they stay accurate over time.

Use comments that add information, not just repeat the obvious reading of the code. Be thoughtful and clear in your commentary.

Instead of this:

```
int count = -1;  // set count to -1
```

Do this:

```
int count = -1;  // count starts out as an invalid value
```

I - Use Enums/Classes/Namespaces over Name Mangling

To collect a set of identifiers into a group use either enums (if integer values are sufficient) or classes, or (rarely) a namespace, (for non-integer values) to make the set of names unique.

For example, instead of global definitions like this (ex. that use the suffix _CN/_PN/etc. to create a name space):

```
``
public const String ManagedElement_CN = "CIM_ManagedElement";
public const String ComputerSystem_CN = "CIM_ComputerSystem";
public const String HostedService_CN = "CIM_HostedService";
:
public const String PreviousInstance_PN = "PreviousInstance";
```

Instead do this:

```
public class ClassNames
{
    public const String ManagedElement = "ManagedElement";
    public const String ComputerSystem = "ComputerSystem";
    public const String HostedService = "HostedService";
    :
}

public interface PropertyNames
{
    :
    public const String PreviousInstance = "PreviousInstance";
}
```

MD – fac - Factor Repeated Method Calls into Local Variables

If a method/block contains repeated calls to the same method (that are expected to return the same value) or property, factor out (i.e., DRY principle) the call to a local variable, and then replace the call with the local variable.

For example, rather than this:

```
if(user.Address.City != null)

{

   Console.writeln(String.Format
      (“{0} lives at {1}.”,
      user.Name, user.Address.City);

}
```

Do this:

```
var city = user.Address.City;
```

``

```
if(city != null)

{

   Console.writeln(String.Format
      (“{0} lives at {1}.”,
      user.Name, city);

}
```

This makes it easier to do improvements like:

```
var city = user != null ?
  (user.Address != null ? user.Address.City : null) : null;
```

:

X - Use “+” Operator/Format Method to Build Composed Strings

Most developers think use of “+” to compose strings has poor performance and should be avoided. This is typically not the case. The C# compiler will automatically convert repeated sequences of string addition (+) to use a sequences of StringBuilder.Append() calls. If there are just few terms (ex. ”x” + “y”) then String.Concat() may be used instead.

Creating a StringBuilder for one Append() is wasteful.

```
public override string ToString()

{

    StringBuilder sb = new StringBuilder();

    sb.Append("OldVersion : " + oldVersion.ToString() +  ",
NewVersion : " + newVersion.ToString());

    return sb.ToString();

}
```

It is OK for multiple Append()s, but often redundant to what the compiler is doing:

```
public override string ToString()
```

``

```
{
    StringBuilder sb = new StringBuilder();

    sb.Append("OldVersion : ");

    sb.Append(oldVersion.ToString());

    sb.Append(", NewVersion : ");

    sb.Append(newVersion.ToString());

    return sb.ToString();

}
```

There is an `Append(Object)` that under the covers calls `ToString()` on its argument, so any use of `ToString()` on the parameter expression is redundant. Also `StringBuilder.Append()` returns the receiver, thus the above can be shortened (made more fluid) to:

```
public override string ToString()

{

    StringBuilder sb = new StringBuilder();

    sb.Append("OldVersion : ").Append(oldVersion).
       Append(", NewVersion : ").Append(newVersion);

    return sb.ToString();

}
```

Or even:

```
public override string ToString()

{

   return new StringBuilder().Append("OldVersion : ").
        Append(oldVersion). Append(", NewVersion : ").
        Append(newVersion).ToString();

}
```

``

Which is exactly what the following is converted to (the compiler uses StringBuilder or Concat() under the covers) :

```
public override string ToString()

{

    return "OldVersion : " + oldVersion +
           ", NewVersion : " + newVersion;

}
```

Often it is better to use a “format string” rather than a series of “+” operators. Consider this equivalent approach:

```
public override string ToString()

{

    return String.Format("OldVersion : {0}, NewVersion : {1}",
                  oldVersion, newVersion)
}
```

Here is an example that combines the use of “+” and Append() in an interesting way:

```
public override string ToString()
{
    StringBuilder sb = new StringBuilder();

    sb.AppendLine("ID : " + ID);
    sb.AppendLine("isLocal : " + isLocal);
    sb.AppendLine("HyperVisor");
    sb.AppendLine("Name: " + hypervisor.Name);
    sb.AppendLine("Type: " + hypervisor.Type);
    sb.AppendLine("connectivity: " + hypervisor.connectivity);

    sb.AppendLine();
    sb.AppendLine("AgentVMInfo:");
    sb.AppendLine("AgentName :" + agentVMInfo.AgentName);
    sb.AppendLine("VMLocation :" + agentVMInfo.VMLocation);
    sb.AppendLine("VMName :" + agentVMInfo.VMName);
    sb.AppendLine("VMStatus :" + agentVMInfo.VMStatus);

    sb.AppendLine();
    sb.AppendLine("Export:");
```

``

```
    sb.AppendLine("exportPercentage : " + Export.ExportPercentage);
    sb.AppendLine("status   : " + Export.status);
    sb.AppendLine("lastExportTime : " + Export.LastExportTime);


    return sb.ToString();
}
```

Note: the above uses spaces before the colon: "status : ". This is the recommended form: "status: ". Also recommended is consistent use of case on the field names.

The above can be improved with the use of a helper method:

```
public override string ToString()
{
    StringBuilder sb = new StringBuilder();

    ForToString(sb, "ID : " , ID);
    ForToString(sb, "isLocal : " , isLocal);
    ForToString(sb, "HyperVisor");
    :

    return sb.ToString();
}
```

It is highly recommended to use the format string approach whenever possible as substitutions ("{…}") can have additional control over the formatting applied that "+" does not offer. Also the single format string is much easier to translate into other languages (if that is a requirement) than a string with "+" parts.

Also note that: `String x = "hello"; x += "world";`

Creates a new string x' that is different from the original x as all string instances are immutable.

If generating strings in loops, then explicit use of StringBuilder and Append is much preferred.

```
StringBuilder sb = …;

for(int i = 0; i < 1000000; i++) {
  sb.Append("a string");
}

string result = sb.ToString();
```

``

Is much faster than:

```
String result = "";

for(int i = 0; i < 1000000; i++) {
  result += "a string";
}
```

The first loop creates a few objects, the second over 1,000,000 objects. BTW a smart compiler could rewrite the second form into one using StringBuilder.

MD – fac - Always Use Finally when Using Locks

On code like this:

```
public Xxx get()

{

   Xxx retVal;

   ReadWriteLock.AcquireReaderLock(readWriteLockTimeout);

   retVal = new someArbitaryMethodThatMayThrowException (…);

   ReadWriteLock.ReleaseReaderLock();

   return retVal;

}
```

If there is any chance the code between the lock and unlock can throw an exception, the code MUST be wrapped in `try/finally`. Here it is uncertain if *`someArbitaryMethodThatMayThrowException`* throws an exception or not. One can read the code of `someArbitaryMethodThatMayThrowException` to find out but that is brittle to future changes to `someArbitaryMethodThatMayThrowException` so best to always use `try/finally`.

So the code should be:

```
public Xxx get()

{

   Xxx retVal;
```

``

```
   ReadWriteLock.AcquireReaderLock(readWriteLockTimeout);

   try {

     retVal = new someArbitaryMethodThatMayThrowException (…);

   } finally {

     ReadWriteLock.ReleaseReaderLock();

   }

   return retVal;

}
```

It is better if some approach like this is used:

```
delegate void LockedTarget(void);

  :

public static void doLocked(ReaderWriterLock l, LockedTarget t) {

  l.AcquireReaderLock();

  try {

     t();
  }

  finally

  {
     l.ReleaseReaderLock();
  }
}
```

Use as follows:

```
public Xxx get()

{

      Xxx result = null;
``
        doLocked(rwLock, () => {
            result = new someArbitaryMethodThatMayThrowException (…); }
        );

        return result;

}
```

The above is briefer, safer and more obvious. The locking code is factored out into one place (i.e., upholds DRY principle).

```
public Xxx get()

{

        Xxx result = null;

        doLocked("rwLock", () => {
            result = new someArbitaryMethodThatMayThrowException (…); }
        );

        return result;

}
```

The above is even more abstract as it hides the actual locking mechanism (by providing instead a key to lookup/select the (unknown) implementation) used and allows for it to change overtime without impacting all the usage sites. Thus this approach should be used even if no exception is possible.

NTH - Use Thread-Safe Collections over Explicit Locking

Use a *Thread-Safe Collection* (TSC) instead of adding locks around standard collections if the lock is only for a single operation (ex. add/remove). Continue to use locking (on both standard and safe collections) if there is a sequence of operations (ex., search for an item, and if not found, then add it).

As stated in other items, it is best to wrap such locking in utility methods.

Use these types

Queue ➔ ConcurrentQueue

Dictionary ➔ ConcurrentDictionary

``

Stack ➔ ConcurrentStack

MD – fac - Don’t Repeat Yourself (DRY Principle)

Many examples, of varying scale (single clause of expression to whole methods), of duplication can exist. Such duplication is often the result of Copy & Paste sequences. Avoid or remove any such duplication; often by creating helper methods.

On common example like this:

```
if (ThreadState == ThreadState.Stopped)
{
    pollingThread = new Thread(ts);
    pollingThread.Start();
}
else
{
    pollingThread.Start();
}
```

Which should be reduced to:

```
if (ThreadState == ThreadState.Stopped)
{
    pollingThread = new Thread(ts);
}
pollingThread.Start();
```

``

Or consider code like this:

```
if (!Handle.isFeatureEnabled())

{

  FeatureDetails.set(

      New FeatureDetails(
        FeatureInstallationStatus.NotInstalled,
        Feature_NotInstalled, "")

      );

}

else

{

  FeatureDetails.set(

      new FeatureDetails(
        FeatureInstallationStatus.Installed,
        Feature_Installed, "")

      );

}
```

Note the highly repetitive nature of the if and else bodies. This is a common result of copy/paste programming combined with minimal thinking about the result.

This should be re-expressed something like this:

```
bool enabled = Handle.isFeatureEnabled();

FeatureDetails.set(

      new FeatureDetails(
          enabled
            ? FeatureInstallationStatus.Installed
            : FeatureInstallationStatus.NotInstalled,
          enabled
            ? Feature_Installed
``
            : Feature_NotInstalled,
          ""));
```

Where the decision logic is pushed lower.

One could go further:

```
bool enabled = Handle.isFeatureEnabled();

FeatureDetails.set(

    new FeatureDetails(getStatus(enabled)),
                                    getMessage(enabled), ""));

:

virtual protected FeatureInstallationStatus
             getStatus(bool installed)
{

   return installed
         ? FeatureInstallationStatus.Installed
         : FeatureInstallationStatus.NotInstalled;

}

virtual protected FeatureInstallationMessage

             getStatus(bool installed)
{

   return installed
         ? Feature_Installed
         : Feature_NotInstalled;

}
```

This approach is more readable and allows better reuse and a potential to override to change.

Consider this code, which result in four log lines, none of which strongly identify the source of the information.

```
public void LogToString()

{

   logMessage(LEVEL_FINE, "Status " + this.status);

   logMessage(LEVEL_FINE, "AltStatus " + this.AltStatus);
``
   logMessage(LEVEL_FINE, "Out_Msg " + this.Out_Msg);

   logMessage(LEVEL_FINE, "Alt_Msg " + this.Alt_Msg);
}
```

The above is likely the result of use of a copy & paste action. It should be rewritten:

```
{public} static void logHelper(string message)

{

    logMessage(LEVEL_FINE, message);

}

public void LogToString()

{

    logHelper("Status " + this.status);

    logHelper("AltStatus " + this.AltStatus);

    logHelper("Out_Msg " + this.Out_Msg);

    logHelper("Alt_Msg " + this.Alt_Msg);

}
```

`logHelper` is now a reusable method. Think beyond just minimal reuse. Consider creating enhanced/more flexible versions, such as:

```
public static void log(string format, params object[] args)

{

    logMessage(LEVEL_FINE, string.Format(format, args));

}
```

Now it is easier to (say) change the log granularity in one place and see the result everywhere.

```
public static void log(string format, params object[] args)

{

    logMessage(LEVEL_FINEST, string.Format(format, args));

}
```

``

Rewriting like this may be better:

```
public void LogToString()

{

    logMessage(LEVEL_FINE,
       string.Format("UpdateResult: Status={0}, AltStatus={1}, “ +
                     "Out_Msg={2},Alt_Msg={3}",
                  this.status,  this.AltStatus,
                  this.Out_Msg, this.Alt_Msg));

}
```

Which has less code and fewer log lines.

If any message is very long, then multiple lines are ok. If the string needs new lines:

```
logMessage(LEVEL_FINE,
     string.Format("UpdateResult: Status={0}, +
                  “AltStatus={1}\n  Out_Msg={2}\n  Alt_Msg={3}",

                  this.status, this.AltStatus,
                  this.Out_Msg, this.Alt_Msg));
```

Note: Several other items in this document also fall into the DRY category.

MD – fac - Extract Reusable Utility Methods into Utility Classes

Whenever you identify repeated code, factor it out. If it is a simple (say 3 or less terms) expression use a local variable; else wrap it in a method. If it is one or more statements, move it to a method. Parameterize (vs. use hardcoded values) the method to make it more general.

If the method is reasonably general in nature (i.e., useful beyond the context written, many will be this way), place the (typically static) method in a shared utility class. Often the class will already exist (such as a `Util` class in the `Library` namespace) but create it first if necessary. This extraction of code/methods needs to become basic practice.

Review all utility classes (should only be a few) frequently to keep aware of new methods as they are added.

``

Here is an example of one such utility method:

```
/// Get the Description attribute value for an enum value.
public static string GetEnumDescription(Enum value)
{
   var vx = value.ToString();
   System.Reflection.FieldInfo fi = value.GetType().GetField(vx);
   DescriptionAttribute[] attributes =
      (DescriptionAttribute[])fi.GetCustomAttributes(
                  typeof(DescriptionAttribute), false);
   return attributes != null && attributes.Length > 0
      ? attributes[0].Description
      : vx;
}
```

Often these extractions create the need for short lived local variables. Since the range of their use is often very short they can have minimal names (like “vx” for value).

Utility classes should consist of only `public static` methods (especially have no non-static fields). The classes should be created as a hierarchy of classes with more specific utility methods going into subclasses while more general methods going into super-classes.

Such utility class names generally end in “Utilities” (“Utils” is OK).

For example, for a class holding string utilities, use:

```
public class StringUtilities : BasicUtilities { … }
```

Favor access via inheritance over direct (“<classname>.”) reference.

MD – fac - Encapsulate Common Code Fragments in Methods

Code often is composed of repeated use of sequences of simple code (often in the form of *idioms*). Any common such sequence should be factored out, often to a method. This allows for more mature implementations to evolve over time and for common maintenance of the idiom. Even code as small as testing against `null` can benefit from this.

Often the purpose of the code can be better revealed via the method’s name than human parsing of the code snippet.

For example, instead of doing it this way:

``

```
if(name != null) …  (easy typo of if(name == null) possible)
```

Do it this way:

```
if(!IsNull(name)) …  (typo less likely here)
```

Or:

```
if(NotNull(name)) …    (typo not possible here)
```

Or (avoiding negative names):

```
if(Exists(name)) …    (typo not possible here)
```

Or, even better, when name is a collection (i.e., a string is a collection of characters):

```
if(name != null && name.Length > 0) …
```

Should be:

```
if(!IsEmpty(name)) …
```

Where `IsNull()`, `Exists()` and `IsEmpty()` are examples of utility methods from a common utility library. `Exists` Is the opposite of `IsNull`. `IsEmpty` test for both null and zero size.

Of course, these utility methods need to exist (or be created just in time).

MD – cod - Avoid use of Boolean Flags in Control Flow Logic

Avoid use of flags (Boolean values) for control flow when simpler means are possible. For example, to skip the first element in an (index-able) collection:

```
bool seenFirst = false;
```

``

```
for (int i = 0; i < someCollection.Length; ++i)
{
   String prop = someCollection[i];
   if (seenFirst)
  {
          // do something with prop
   }
   else
  {
    seenFirst = true;
  }
}
```

Should be this briefer (and clearer):

```
for (int i = 0; i < someCollection.Length; ++i)
{
   if (i > 0)
  {
     String prop = someCollection[i];
    // do something with prop
  }
}
```

Or even better:

```
for (int i = 1; i < someCollection.Length; ++i)
{
      String prop = someCollection[i];
      // do something with prop
``
}
```

MD – fac - Encapsulate Access to All (potentially) Shared Objects

Safe access to objects that can be used concurrently is very difficult to implement correctly. Therefore all such access needed to be implemented in a shareable (library) form. This allows the implementation to done by experts and any flaws be fixed in one place. Any open (in normal functional code) use of synchronization features (such as using locks, try/finally recovery, etc.) needs to be so encapsulated in utility services used from the open code.

MD – cod - Consider Concurrent Threads when Creating Shared/Global Instances

When creating shared instances (such as singleton objects) in a (potentially) multi-threaded environment special care is need to avoid race conditions. Often the creation requires locking.

For example, creating a singleton:

```
public sealed class SomeSingleton
{
    private static volatile SomeSingleton instance;
    private static object syncObject = new Object();

    private SomeSingleton () { }

    /// Access the only instance.
    public static SomeSingleton Instance
    {
        get
        {
           // optional optimization (locking can be slow)
           if (instance == null)
           {
              lock (syncObject)
              {
                  if (instance == null)
                     instance = new SomeSingleton();
                  }
               }
           }
           return instance;
``
         }
    }
}
```

I - Prefer the C# (vs .NET) Names for System Types

Many system types have alias names in C#. For instance, the type *System.String* has alias `string` and *System.Int32* has alias `int`. The use of the C# alias (over the System name) is highly preferred. Whatever form is used, use it consistently within one source file.

I - Avoid Conditional Source

Use of the preprocessor `#if/#else/#endif` statements should be avoided; especially if the test variable can be set in the same source (e.g., where commenting out the source is an alternative).

Rarely is any code space savings worth using conditional code. Often traditional `if/else` statements can be used instead. If such conditional source is required, the reason must be clearly stated on commentary around the conditional source.

For conditional (void) methods, it is often better to use the `[Conditional(<variable>)]` attribute instead.

For example, instead of:

```
#if DEBUG
void ConditionalMethod()
{
   :
}
#endif
```

Do this:

```
[Conditional("DEBUG")]

void ConditionalMethod ()
{
   :
}
```

``

It is highly recommended that any conditional source not currently contained as a method be, if possible, wrapped in a method so the above attribute form can be used.

Once contained in a method, then inheritance can be employed. The current class adds a "no op" method. A subclass is created that includes the conditional code in an override of the method. Then either the base or subclass are created (preferably via a factory object) and used based on the state of the condition.

NTH - Maximize the use of Factory Objects

Avoid doing direct `new` operations on types (except simple types like strings, arrays and collections); especially for configurable types. Instead create "factory" methods to create them. This allows the configuration to be easily changed in one common place (the factory). It also hides a lot of the details of the object's creation from the client code. Common examples are creating a factory to create all `Exception` objects or to acquire a logger object. Often also done for Singleton objects.

As much as possible move the factory methods in to sharable utility classes as (often) static methods.

*** add example ***

I - Minimize Impacts of Cross-Cutting Concerns

Many times methods have to deal with *Cross-Cutting Concerns* (CCC). CCCs are function in the method that has nothing to do with the method's purpose but is needed to meet system requirements. A very common form of CCC code is flow tracing. Large amounts (often 50% or more) of the method code can be composed of CCC code. Not only does this code strongly interfere with the readability of the code, it is often among the most brittle of code requiring very large effort to maintain it across a code base. Also, it is best if CCC code can be easily and systematically disabled; this is near impossible if such code is intermixed with purposeful code.

Often CCC code is very repetitive (occurs in most methods) and can be factored out into (often highly reusable) utility methods that often take lambdas containing the purposeful code as arguments. Doing this needs to become routine practice.

For example, instead of this:

```
``
String funcName = "Access() constructor";
try
{
    logFuncEnter(LEVEL_FINEST, funcName);

    readWriteLock = new ReaderWriterLock();
    configurations = new ConfigurationCollection();
    infos = new InfoCollection();

    Refresh();

    logFuncExit(LEVEL_FINEST, funcName);
}
catch (Exception ex)
{
    logMessage(LEVEL_FINE, "Error", ex);
    logFuncExit(LEVEL_FINEST, funcName);
}
```

Do this:

```
traceFlow("Access() constructor" => {
    try {
      readWriteLock = new ReaderWriterLock();
      configurations = new ConfigurationCollection();
      infos = new StatusInfoCollection();

      Refresh();
    }
    catch (Exception ex)
    {
      logException("Error", ex);
    }
});
```

Note the `traceFlow(...)` library method takes a delegate/lambda and wraps it in a `try/catch/finally` that makes the logging calls.

If the only action in the exception is to log the data and re-throw it[4], this entire sequence can be reduced to:

```
traceFlow( () => {
    readWriteLock = new ReaderWriterLock();
    configurations = new ConfigurationCollection();
    infos = new InfoCollection();
```

[4] Vs. consume it. The choice to consume vs. re-throw could be a (defaulted) parameter to traceFlow().

``

```
    Refresh();
});
```

It is a simple extension to allow the trace exit message to be modified to all trace results to be supplied. As another example of how using a function like `traceFlow` and `traceResult` can clean up code consider:

```
private void RebootHandler(object sender, EventArgs e)
{
  string fName = "RebootHandler";
  Logger.Debug(fName, "Entering ..");
  customMessageBox cm = new customMessageBox();
  cm.Setup(stringResources.Reboot, stringResources.rebootRequiredExitPopup, 1, 1);
  cm.ShowDialog(_mainForm);

  if (cm.DialogResult == DialogResult.OK)
  {
    cm.Dispose();
    RebootAction(null, null);
    Logger.Debug(fName, "Exiting : ... cm.DialogResult == DialogResult.OK");
  }
  else
  {
    Logger.Debug(fName, "Exiting : ... cm.DialogResult != DialogResult.OK");
    return;
  }
  Logger.Debug(fName, "Exiting : ... ");
}
```

Changes to the simpler:

```
private void RebootHandler(object sender, EventArgs e)
{
  traceFlow(() -> {
    customMessageBox cm = new customMessageBox();
    cm.Setup(stringResources.Reboot, stringResources.rebootRequiredExitPopup, 1, 1);
    cm.ShowDialog(_mainForm);

    if (cm.DialogResult == DialogResult.OK)
    {
      traceResult("cm.DialogResult == DialogResult.OK");
      cm.Dispose();
      RebootAction(null, null);
    }
    else
    {
      traceResult("cm.DialogResult != DialogResult.OK");
    }
``
  }
}
```

Note: the default `traceResult` message is “Exit <containingMethodName>” (similar to above fName value); setting the value adds a suffix to the message. `traceFlow` generates a message similar to “Enter <containingMethodName>..”.

MD – com - Always add TODO: to Notes on Future Work

Always use the marker “TODO:” to mark remarks about pending work. This allows for consistent search for these types of notes. Use only standard remarks (“//”), not documentation comments (“/* … */” or “///”) .

For example:

```
// TODO:  Add locking around this sequence

:
```

I - Place Any Type Name Qualifier on the Left End

Always have the base type name on the right end of a type name. Add qualifiers on the left.

For example, for base type `VirtualStandby`, instead of this:
`VirtualStandbyAbstract`

Do this:

`AbstractVirtualStandby`

NTH - Always Prefer Polymorphic Selection over Switch Selection

Avoid using switch to select among cases that can be expanded; instead use polymorphism. Failure to do this leaves the code open to (often undetected until runtime) breakage as new cases are added. This, of course, may require the distinct behavior to be implemented as virtual methods on a class hierarchy.

``

Switch should only be used on when the cases are fixed (example against an enum type).

For example, for situations like this:

```
switch (DisplayExportTypeName)
{
    case "Hyper-V":
        virtualStandby = new HyperV_VirtualStandby(…);
        break;

    case "VMWare ESX(i)":
        virtualStandby = new ESX_VirtualStandby(…);
         break;

     case "Oracle VirtualBox":
         virtualStandby = new OracleVirtualBox_VirtualStandby(…);
         break;

     case "VMWare Workstation":
         virtualStandby = new VMWareWorkstation_VirtualStandby(…);
         break;

    default:
         virtualStandby = new Unknown_VirtualStandby(…);
         break;
}
```

It is better to create a hierarchy of factory classes implementing (say) `IVirtualStandbyFactory`. Then each factory returns an appropriate `IVirtualStandby` instance. Often the factories are held in a map indexed by the switch value (in this case: `DisplayExportTypeName`).

For example, the switch is replaced with:

```
var f = virtualStandbyFactories.

                       TryGetValue(DisplayExportTypeName, null);

virtualStandby = f != null ? f.makeInstance()
                           : new Unknown_VirtualStandby(…);
```

Here is another related example:

```
VirtualStandby virtualStandby =
   GetVirtualStandbyFromID(idToStart.ToString());

//create a VM based on the virtualStandby
```

```
``
if (virtualStandby.hypervisor.Type == HyperVisorType.HyperV)
{
        HyperV_VirtualMachine vm = new HyperV_VirtualMachine(…);
}
else
{
        logMessage("Invalid hypervisor type detected");
 }
```

As soon as another hypervisor type is added, this code is broken (fails badly at runtime).

I - Avoid using Abbreviations (vs. Acronyms) in Names

Avoid using abbreviations (unless they are the common way to express the name (i.e., “Ok” for Okay, “approx.” for approximate, etc.)) in names; use the full form. Acronyms are allowed.

When an exception is made, use mixed- or all lowercase only; not all uppercase. For example, use “Id” or “id”, short for identifier, instead of “ID”.

For example, instead of:

```
VirtualStandby vstndby = …;

String fileNameStr = …;
```

Do this:

```
VirtualStandby virtualStandby …;

String fileNameString = …;
```

MD – cod - Avoid Redundant Condition State

On searches use simple/non-redundant ways to determine found/non found state.

For example, instead of this:

```
VirtualStandby retVal = null;
```

``

```
List<VirtualStandby> retList = VStandbyMgmt_Handle.GetVirtualStandbys();

bool IDFound = false;
foreach (VirtualStandby vstndby in retList)
{
   if (vstndby.ID == idToFind)
   {
     IDFound = true;
     retVal = vstndby;
     break;
  }

 }

 if (IDFound)
 {
     return retVal;
 }
 else
 {
     throw new Exception(
            String.Format("VirtualStandby not found with ID {0}", idToFind));
 }
```

IDFound is redundant with retVal.

Do this:

```
VirtualStandby result = null;

var vsl = virtualStandbyManagementHandle.GetVirtualStandbys();
foreach (VirtualStandby vs in vsl)
{
   if (vs.id == idToFind)
   {
     result = vs;
     break;
  }

 }
 if (result == null)
 {
     throw new Exception(String.Format("VirtualStandby not found: {0}", idToFind));
 }

 return result;
```

I - Avoid Redundant Code

Redundant code causes reader confusion (Why is it there? What is its purpose?). So avoid writing it.

``

For example, instead of:

```
if (IDFound)
{
     return retVal;
}
else
{
     throw new Exception(String.Format(
             "VirtualStandby not found with ID {0}", idToFind));
}
```

Here the `else` clause is not needed as `return` cannot continue.

Do this:

```
if (!idFound)
{
     throw new Exception(String.Format(
             "VirtualStandby not found with ID {0}", idToFind));
}
return result;
```

NTH - Use Hashes over Lists

If you have a collection of keyed (or keyable) items, place them in a hash so you can avoid linear searches.

For example, instead of this:

```
VirtualStandby result = null;

List<VirtualStandby> vsl = VStandbyMgmt_Handle.GetVirtualStandbys();
foreach (VirtualStandby vs in vsl)
{
   if (vs.ID == idToFind)
   {
     result = vs;
     break;
  }

 }
 if (result == null)
 {
     throw new Exception(String.Format("VirtualStandby not found: {0}", idToFind));
 }

 return result
```

``

Do this:

```
var vsd = virtualStandbyManagementHandle.GetVirtualStandbys();
var result = vsd.TryGetValue(idToFind, null);
if (result == null)
 {
     throw new Exception(String.Format("VirtualStandby not found: {0}", idToFind));
 }

 return result
```

MD – ren - Always use Symbolic Names (vs. literals)

Nearly every value used in code has some meaning beyond its literal representation.   This is often captured in a variable’s names. Literals lack this feature, so create a constant name for all literals to add this feature.   This is especially true for numeric limit values and string values. This allows the actual value to change safely across multiple references (often in multiple source files).

Exceptions are basic values such as: true, false, null. ‘ ‘[5], “”, -1, 0 or 1 (int and float).

So instead of:

```
if(x > 100) …
```

Or:

```
if(“running”.Equals(status)) …
```

Do this:

```
public const int MAX_USER_COUNT = 100;

:

if(x > MAX_USER_COUNT) …
```

Or:

```
public const string  RUNNING_STATUS = “running”;

:
```

[5] I.e. space

``

```
if(RUNNING_STATUS.Equals(status)) …
```

The declaration of the constant is often far removed from its use. If the constant applies to only a single variable, define the constants at the same place the variable is declared with the same visibility.

Here is another example of what not to do. In each case the strings should be defined as constants and the constants used here (and anyplace `getServiceStatus` is called). So instead of this:

```
public static string getServiceStatus(string service)
{
    if (isServicePresent(service) == false)
        return ("Not Found");

    ServiceController srvc = new ServiceController(service);

    switch (srvc.Status)
    {
        case ServiceControllerStatus.Running:
            return "Running";
        case ServiceControllerStatus.Stopped:
            return "Stopped";
        case ServiceControllerStatus.Paused:
            return "Paused";
        case ServiceControllerStatus.StopPending:
            return "Stopping";
        case ServiceControllerStatus.StartPending:
            return "Starting";
        default:
            return "Status Changing";
    }
}
```

And:

```
string serviceStatus = getServiceStatus(serviceName);

if (String.Equals(serviceStatus, "Running"))
:
```

Do this instead:

```
public const string StatusRunning = "Running";

 :

     case ServiceControllerStatus.Running:
         return StatusRunning;
:
```

``

```
            string serviceStatus = getServiceStatus(serviceName);

            if (String.Equals(serviceStatus, StatusRunning))
           :
```

Doing this avoids many errors related to changing the string's text value or case mismatches in string comparisons.

MD – fac - Never use hard-coded values on System Interfaces

Always use any defined definition (often enums) of testable values; especially when doing system functions.

X - Do not Extend Enums Values

Enums are a related set of symbolic names (of `int` type). Often they are used in a fixed sequences of tests (i.e., `if/else if/else` or `switch`) that tend to break if (new) untested values exist. Therefore, once released to the wild, avoid extending the values in a public Enum set; instead create a new Enum with the added values and mark the old Enum as `[Obsolete]`.

NTH - Prefer Var Type over Explicit Type For Local Variables

The C# "var" declaration infers the declared type from the initial value. Since all local values are expected to be initialized (see MD – ren - Always Initialize Local variables), the `var` type can be applied.

For example:

```
List<SomeTypeName> stnl = new List<SomeTypeName>();
```

Or:

```
List<SomeTypeName> stnl = SomeMethodReturningListOfSomeTypeName();
```

``

Is better as:

```
var stnl = new List<SomeTypeName>();
```

Or:

```
var stnl = SomeMethodReturningListOfSomeTypeName();
```

Another example:

```
MachineLocation ml= exportConfiguration.Location;
```

Is better as:

```
var ml = exportConfiguration.Location;
```

MD – fac - Use Exceptions to Report Failures

Many methods return limited types, even if there is potential for more general values. A common case are methods that return execution status; especially `bool` (success/failed) methods. It is much better if such methods return multi-values types, such as int, string or enum, for status if possible.

If the result is really success/failed, then the method should return `void` on success and throw an exception on failure. In general, failure should be reported not through status code return values but through exceptions. Only variations of success should be reported via return codes or return values.

Custom Exception subclasses can be created to pass unique failure state information.

Code sequences like the following are to be avoided:

```
VirtualMachine vm = new VirtualMachine(…);
if (vm.StopVirtualMachine() == false)
{
   throw new Exception(…);
}
```

With:

``

```
public override bool StopVirtualMachine() { … }
```

Instead the `StopVirtualMachine` method should be void and throw an exception.

```
public override void StopVirtualMachine() { … }
```

Then do:

```
var vm = new VirtualMachine(…);
vm.StopVirtualMachine();
```

If the caller wants to define the exception then do this:

```
var vm = new VirtualMachine(…);
try
{
  vm.StopVirtualMachine();
}
catch (Exception e)
{
   throw new Exception(…);
}
```

NTH - Prefer Use of Using Over Fully Qualified Names

Many Types (especially nested types) can have long names (into the 100s of characters) when the full namespace path is included. Therefore it is best to use the `using` statement to bring the class into the current namespace such that it can be used without the namespace prefix. The fully qualified name is only required in the (rare) case that the same name is used in multiple namespaces and both forms are needed in the current namespace.

For example, instead of using:

```
System.IO.File f = …;
```

Do:

```
using System.IO;

:

File f = …;
```

``

MD – cod - Avoid Null Reference Exceptions

Any method that returns a reference type can conceivably return a null value. Similarly any reference variable can possibly be null. Unless the variable definition or method documentation explicitly states a null is not a possible result (ex. many methods that return arrays/collections often will always return at least an empty array/collection) then the value/results should be explicitly checked for null before being used. This can be done by either doing if tests around the code or wrapping the code in try/catch. In general, the if test approach is the preferred method to use.

For example, instead of this:

```
Dictionary inParams = vm.GetMethodParameters("xxx");
inParams["yyy"] = action;
```

Do this (preferred):

```
Dictionary inParams = vm.GetMethodParameters("xxx");
if (inParams != null)
{
    inParams["yyy"] = action;
}
else      // optional else if some recovery action possible
{
  // any recovery here
}
```

Or this (especially if other exceptions can occur):

```
Dictionary inParams = vm.GetMethodParameters("xxx");
try
{
    inParams["yyy"] = action;
}

catch (Exception)
{
  // any recovery here
}
```

This can be tedious to do but it makes the code much more robust in the face of unexpected behavior of downstream code. Also it is easy to add recovery code later if/when there is an alternative behavior available.

``

I - Avoid Undefined use of Acronyms

Do not assume the reader knows what an acronym stands for. Use of acronyms is OK, but they need to be defined (unless universally well known, such as WWW, or USA) upon first usage; just like in normal documentary writing. In general, the full name should be used in commentary until the acronym is defined, especially type header comments.

For example, the commonly used acronym(say) BMR (for Bare Metal Recovery) needs to be defined (at least once) in each source file, such as:

```
/**
  This class defines basic functions for Bare Metal Recovery (BMR).
  These BMR utilities …
*/
public class BmrUtilities { … }
```

MD – fac - Log Significant Events

Often significant events occur during program execution. Common examples are unrecoverable events/exceptions, significant changes of state (ex. database updates) or significant decision points. These events should be logged to allow subsequent examination of the program behavior to diagnose bugs or other activities. Depending on the event nature, several forms of logging are available, each with increasing persistence, content capacity and visibility. All logging code should be extracted to utility method/classes; see I - Minimize Impacts of Cross-Cutting Concerns .

Logging code can be wrapped in conditional code to allow it to be suppressed in production; see I - Avoid Conditional Source. In general, this is not required as the (performance) cost of the logging code is minimal.

Types of logging:

1. Log to Console – simplest and reliable; limited content; content is non-persistent
2. Log to File – simple but potential for failure; effectively unconstrained; content is persistent

``

3. Log to Host Event Log – highly visible but constrained; content is persistent

For example:

```
try
{
    someOperationThatMayFailWithoutRecovery();
}

catch (Exception ex)
{
    StandardLogger.getLogger().logException(ex);
}
```

MD – cod - Validate All Method Parameters

Most code only functions correctly when provided with correct inputs. All code is implemented inside methods, most with parameters. Except for enums, these parameters may have values outside the expected range (ex. an `int` that only allows the values 0, 1 or 2; a reference that cannot be `null`, etc.). The method must be defensive against this by validating all such parameters before use. Permitted parameter values should be detailed in any documentation comments supplied with the method.

Common ways of validating parameters:

1. If with Exception – all validity tests wrapped in `if` statements with `throw` statement as its body; simple and effective; cannot easily remove
2. Asserts – all tests in the form of Assert methods (implicit throw); more consistent; can remove easily at compile time; often the briefest form
3. Design By Contract – all tests follow `Contract` class form ; most flexible but most tedious to implement

In most cases, Asserts are the recommended means.

For Example:

```
public void someGreatFunction(MyData mydata, int size)

{

  if (md == null)

  {
```

``

```
  throw new ArgumentNullException(“myData is required”);
 }
 if (size <= 0)
 {
  throw new ArgumentException(“size is invalid: ” + size);
 }
 :
}
```

Often the creation of these common exceptions can be factored out into utility routines.

Or for Example:

```
public void someGreatFunction(MyData myData, int size)
{
 Assert(md != null, “myData is required”);
 Assert(size > 0, “size is invalid: {0}”, size);
 :
}
```

Note: if performance of assertions is a concern, one can pass lambda wrappers as either argument to enable the Assert method to disable the execution of the expression. For example:

```
Assert(() => size > 0, () => string.Format(“size is invalid: {0}”,
size));
```

MD – cod - Dispose All Resources

If a type implements the `IDisposable` interface all creation and use of it should be within a `using` statement. This ensures the `Dispose` method is called on the object.

``

From the MSDN documentation:

> *As a rule, when you use an **IDisposable** object, you should declare and instantiate it in a **using** statement. The **using** statement calls the Dispose method on the object in the correct way, and it also causes the object itself to go out of scope as soon as Dispose is called. Within the **using** block, the object is read-only and cannot be modified or reassigned. The **using** statement ensures that Dispose is called even if an exception occurs while you are calling methods on the object.*

For example:

```
using (var fos = new FileOutputStream("myfile.txt"))
{
    fos.writeString(aLongString);
}
```

The above guarantees the `FileOutputStream` is flushed and closed. Nested `using` statements are allowed.

Always implement `IDisposable` if the class provides a clean-up method like "close" or "terminate".

I - Maximize Use of Standard Exceptions

Use pre-existing exceptions if they are appropriate. Only create new exceptions if no (.NET) exception applies or you need to add parameter values. Extend a standard exception if possible.

Common Exceptions from the `System` name space:

- ArgumentException
- ArgumentNullExceptions
- ArgumentOutOfRangeException
- InvalidOperationException
- NotSupportedException
- NotImplementedException
- ObjectDisposedException

Search other standard namespaces as well to see if an existing exception meets your needs.

``

MD – cod - Make Fields/Properties Accessed Across Threads Volatile or Use Lock

Any field/property that can potentially be accessed across threads needs to be declared `volatile`. This ensures that any change made in a thread is seen immediately in other threads (by suppressing any compiler generated/CLR caching of the value in generated code). See MD – fac - Always Use Finally when Using Locks for an example. An alternate is to always access the variable with in a `lock` block.

I - Always Mark Read Only Fields

Often fields are set during initialization or in constructors and never expected to be changed after that (i.e., they act like late bound constants or constants set to runtime values). To indicate this intention, use the `readonly` modifier on the field's declaration. Err on the side or using `readonly`; one can always remove it later.

MD – cod - Always Catch Exceptions in Threads

Always provide a `try/catch` around the body method of a thread. Failure to so this can cause the whole program to terminate unexpectedly if an exception occurs in the thread body. Exceptions thrown in the thread body are NOT reflected to the code that starts the thread; to get such reflection use `Tasks` instead of `Threads`.

For example:

```
var t = new Thread(Body);

:

static void Body()

{

  try

  {

     // thread body here

  }
```

``

```
  catch (Exception e)

  {

    // handle exception - often log the event

  }
}
```

MD – cod - Always have a Try/Catch in the “Main” Method

Do not depend on any compiler supplied exception handling; explicitly provide your own. This is needed in the `main` method or any top-level service handler on a server.

For example:

```
static void Main(string[] args)

{

  try

  {

     // main body here

  }

  catch (Exception e)

  {

    // handle exception - often log/print the event

  }
}
```

MD – cod - Always Serialize Access to Non-Thread-safe Code

Most services (i.e., methods) are not *thread-safe* (supports concurrent reentry on different threads). Never allow such methods to be accessed from concurrent threads. When in a context where multiple threads may exist, assume, in the absence of documentation that states otherwise, that all methods are

``

not thread-safe and wrap method (or sequences of method) calls in a `lock` block. See MD – fac - Always Use Finally when Using Locks for an example.   A common example of this issue is accessing an I/O stream; concurrent read/write access to the same stream will produce unpredictable (and generally bad) results.

Often the lock wrapping can occur once in top-level methods (vs. every low-level method). Remember that when using delegates or lambdas, the method can be called from a different context and may not be under the protection any locking at the original call site; each delegate/lambda may need to acquire the lock itself.

Note: Locking does not prevent recursion (reentry on the same thread).

If you take the care to provide thread-safe methods, make sure you indicate that in the method's documentation.

It is best to indicate in interface documentation if the implementations are expected to be thread-safe.

I - Consider Character Encodings when processing Text Steams

Characters are represented in computers by integers. Any particular character may be represented by different values in different contexts (i.e., a character set). Often the character requires more than one byte to hold the value (ex. UTF-16 or UTF-32)   Streams are sequence of bytes, so characters need to be mapped to one or more bytes to be on the stream. Many such mappings (encodings) are possible. Reading a stream with a different encoding than it was written in will often result in either incorrect or unknown characters being seen.   Either find a means to permanently associate the encoding used with the stream or document the assumed encoding used by your code (often UTF-8) when accessing the stream. For example, on a method that reads a supplied text file, provide both the file (or file name) and the file's encoding as parameters.

TBD - Enable Human-Readable Text for Translation/Internationalization

Many programs are used internationally. If this is a possibility, all text (messages, forms, etc.) intended for end users (not programmers/support personal) needs to be enabled for translation.   In general, this means the text itself should not be embedded in the source program; instead it needs to be externalized to separate files/resources. Once in resources, multiple (locale dependent) versions of the resource text can be created. Used in place of the text in the source is a unique token (often a string; perhaps the

``

English message itself) used to retrieve the localized text. In general, utility methods should be create/used to retrieve the proper locale-based text for the token at runtime. The .NET runtime provides function to assist here.

The text messages should be single strings (vs. strings composed using '+'). Often they have substitution placeholders embedded in the string. Any values (ex., currency, date, time, etc.) embedded in messages must be formatted according to the target locale.

Note: doing enablement after coding (i.e., retrofitting) is very difficult. If international use is even just a possibility, do enablement right from the start.

I – Create Human-Readable Information in Clients (vs. Servers)

In general, it is the responsibility of client code (rather than server code and assuming they are separated) to create/format/present human readable information (often text or images). A server should only provide information to guide the client in creating this information. For example, a server that responds with a failure should provide an error code and any error parameters, but not the error message text itself. Any text returned by the server should be presented literally (ex. an exception trace-back).

For example, the server returns:

ErrorCode: 1001; Param: "C:\...\someFile.xxx"

Which the client translates into (say):

"Required configuration file(C:\...\someFile.xxx) not found"

Note: this means all translation resources should be associated with the client, not the server.

NTH - Use Helper Methods in Catch Clauses

Recommend the use of a helper method (suggested name `triageException`) as the body of `catch` clause that has any logic. Move the recovery logic into the helper method. This reduces redundancy and allows easy enhancement of the recovery logic. In general the helper should be a `virtual` method to enable overriding.

``

For example:

```
try
{
    // body
}
catch (Exception ex)
{
   triageException(ex, “any arguments added here”);
}
```

:

```
// process special exception types

protected virtual void triageException(Exception e,

                           params object[] args)

{

  if(e != null)

  {

     if(e.GetType() !=[6]  AssertException)
     {

       throw;  // resume normal handling

     }

     // process asserts specially

  }

}
```

As an example, the above shows special handling of particular exception types.

NTH - Use Unsigned Types for Unsigned Values

[6] Belter to use a test that matches subclasses as well.

``

When a value cannot be negative (often counts, array indexes, etc.), use an unsigned type. This avoids a whole class of out-of-bounds errors (< 0 values).

For example:

```
int[] xxx = …

for (uint i = 0; i < xxx.length; i++) { … }
```

NTH - Use Limited Range Values When Applicable

Often the allowed range of a variable is (far) less that the variable's type allows. Consider an array index variable. If of (typical) base `int` type its natural range is -2^{31}:2^{31}-1 but it legitimate range is 0:array.length-1. Use a range limited type over a base type. If necessary (not already in a reusable library form) define such a type.

Note: any such type needs to be easily convertible between base int/float types.

I - Document Changes to Code with Comments

Any change to a source file (beyond initial development/unit test) should be documented, typically with a comment at the start of the source file. If the change was motivated by a documented defect or design change, it should reference the issue tracking number. This provides a source of reference for code reviewers and other readers of the change. It is OK to copy into the source material from the reference for easier access.

Naked (no associated commentary) changes are very hard to follow (or justify why done) just from looking at the changed code.

For example:

```
// Bugzilla 824:  Renamed all XXX to YYY
```

Complex changes should be documented at each change point (line/block), especially if a code review might see changes from multiple issues at once. Add a tag in the descriptive comment and then tag each change location with a brief comment/remark.

For example:

```
``
// Bugzilla 824: (@842) Renamed all XXX to YYY

:

if(zzz > yyy)      // @842 was: zzz >= xxx
```

I - Avoid Large Number of Return Values for a Method

A method may have a main return value (`return` <value>) and zero or more `ref` (or `out`) values, which are effectively extra return values. Avoid having more than two `ref` values; instead create a "return" structure that has the needed values as fields, and return an instance of that structure. The caller can then assign (if needed) the values to any caller's variables.

MD – cod - Avoid Direct Field Usage

Instead of defining `private` fields and using hand written get/set access methods (see I - Place Field Declarations and Accessor Methods Together), use properties instead. Properties can wrap (provide access methods) existing fields or, better yet, create hidden fields. Whenever a property can be used instead a field within a class, do so, even if it is in the owning class. When the access method is trivial (just get/set a field) use the compiler generated forms.

For example (preferred compiler generated anonymous field):

```
public string Name { get; set; }
```

For example (avoid human generated field if just simple get/set like below):

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

Usage example, even in the same class, instead of:

```
if("barry".equals(_name)) …
```

Do this:

``

```
if(“barry”.equals(Name)) …
```

I - Avoid use of Internal Visibility

The `internal` visibility makes a type public within an assembly and invisible to other assemblies. It can be thought of either as an enhanced from of `private` visibility or a reduced form of `public` visibility. This ambiguity is an issue for understanding so therefore avoid using `internal` visibility; use either `private` or `public` visibility. If `internal` visibility is required, comment the use (declaration) to explain why.

NTH - Use Implicit (over Explicit) Processing of Collection Elements

Tradition programming uses `for/foreach` (AKA explicit) looping to process each element of an array or collection. It is better to use implicit (LINQ syntax or LINQ functions) looping.

For example, instead of this:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var numsPlusOne = new int[numbers.Length];
for(int i = 0; i < numbers.Length; i++)
{
   numsPlusOne[i] = n + 1;
}
```

Do this:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

var numsPlusOne = from n in numbers select n + 1;
```

MD – cod - Never use Hardcoded Array/Collection Indexes

``

Never use hardcoded index (unless the collection definition is adjacent to its use); indexes can change at any time.

For example, instead of this:

```
if (ErrorFound)
{
  result = SMISClientAccess.Controller_StateStatus_Map[2]; //error
}
else if (WarningFound)
{
  result = SMISClientAccess.Controller_StateStatus_Map[3]; //warning
}
else
{
  result = SMISClientAccess.Controller_StateStatus_Map[1]; // OK
}
```

Do this:

```
if (ErrorFound)
{
  result = SMISClientAccess.Controller_StateStatus_Map[ERROR_INDEX];
}
else if (WarningFound)
{
  result = SMISClientAccess.Controller_StateStatus_Map[WARNING_INDEX];
}
else
{
  result = SMISClientAccess.Controller_StateStatus_Map[OK_INDEX];
}
```

Define the constants (ex. `OK_INDEX`) adjacent to the collection.

I - Use Interfaces Over Classes for Parameters and Return Values

If a parameter or return type implements an interface (or interfaces), always use that (or one of several) interface type for any method parameters or return types (vs. the concrete type) when you think you need a particular concrete type. This allows the caller to (for parameters) provide any conforming implementation, not just the particular one you thought of. It also (for return values) allows the method to change the actual return type in the future as long as the new type conforms to the specified interface.

``

For example, give the definition:

```
class List<T> : IList<T>, ICollection<T>, IList, ICollection,
     IReadOnlyList<T>, IReadOnlyCollection<T>, IEnumerable<T>,
     IEnumerable
```

Instead of declaring a parameter/return type as a `List`, use one the interfaces it implements, such as `IList`. Use the most abstract (close to `Object`) type you can.

For example:

```
IList<string> getNames() { … }
```

Or:

```
void processItems(IList<string> items) { … }
```

Then use the method like this:

```
List<string> names = x. getNames();

var names = x.getNames();
```

Or:

```
List<string> items = new List<string>();

:

processItems(items);
```

Note: if you need to implement a subset (> 1) of the multiple interfaces, create a new interface that implements the ones you need and pass that. For example:

```
interface XList<T> : IList<T>, ICollection<T>, IList, ICollection
```

``

Here any `List<T>` instance also implicitly implements `XList<T>`.

NTH - Avoid Creating Return Objects in Subroutines

Avoid passing `out` parameters for reference types; use value parameters instead – especially for collections or objects where you just want to set fields/properties and/or call methods. The caller should always allocate (`new`) objects and pass them into the callee. The callee then modifies the values (ex. adds items to a supplied list). Using `out` values (where the callee creates the object) is more complex/error prone than just passing regular values into the callee.

Note: it is OK (and common) for the callee to create the values returned (i.e., `return value`) from a method (vs. via out parameters).

Given:

```
void CreateStrings(out List<string> strings)

{

  strings = new List<string>();

  :

  strings.Add("…");

}
```

For example, instead of this:

```
List<string> strings;

CreateString(out strings);
```

Given:

```
void CreateStrings(List<string> strings)

{

  strings.Add("…");
``
}
```

Do this:

```
List<string> strings = new List<string>();

CreateStrings(strings);
```

NTH – Order Comparison Expressions Consistently

Place any variables on the left of any comparison and any literal on the right. This is a more expected reading order.

For example, instead of this:

```
if (null != ex.Data && 0 < ex.Data.Count) …
```

Do this:

```
if (ex.Data != null && ex.Data.Count > 0) …
```

NTH – Create Convenience Overloads on Methods

When creating any method (especially public ones) add overloads that support likely other frequent parameter types. For example support both GUID/string or File/pathString overloads. Delegate most overloads to a base method. Err on the side of offering too many (vs. too few) overloads. This allows method clients to select the parameter forms most convenient to their situation.

For example:

```
public VirtualStandby GetVRequiredVirtualStandbyFromID(string id)
{
   :
}

public VirtualStandby GetVRequiredVirtualStandbyFromID(Guid id)
{
    return GetVRequiredVirtualStandbyFromID(id.ToString());
}
```

``

MD – fac - Never Allow Duplicate Fields

Having more than one field (with different names, but same type) that hold a particular value (such as an injected dependency object) leads to runtime errors (often null value exception when one field is set and not the other) and reader confusion. Eliminate any such duplicates. If a duplicate has a valid purpose (ex. subtle difference in use), make sure that purpose is documented in commentary at the declaration point.

Often this issue occurs when fields are declared in multiple locations in a type definition; placing all field definitions together in a type definition makes this easier to detect.

Note: often non-obvious duplicate fields can exist with different types if one type inherits from the other type; be careful to detect this case.

I - Use Range Comparisons over Equality

When comparing for a value at the extreme of a range, do not use simple equality, but instead use greater/less-than or equal. Although equals may work, this approach is safer. This is especially needed for floating types.

For example, instead of this:

```
pt.type = StorageAccessLib.JobType.Recover;
if (100 == pt.progresspercent)
{
    pt.status = StorageAccessLib.TaskStatus.Recovered;
}
else if (0 == pt.progresspercent)
{
    pt.status = StorageAccessLib.TaskStatus.RecoveryPending;
}
else
{
    pt.status = StorageAccessLib.TaskStatus.RecoveryInprogress;
}
```

Do this:

``

```
pt.type = StorageAccessLib.JobType.Recover;
if (pt.progresspercent >= 100)
{
    pt.status = StorageAccessLib.TaskStatus.Recovered;
}
else if (pt.progresspercent <= 0)
{
    pt.status = StorageAccessLib.TaskStatus.RecoveryPending;
}
else
{
    pt.status = StorageAccessLib.TaskStatus.RecoveryInprogress;
}
```

I – Avoid Constructed Messages

For any message intended for user (vs. developer) consumption (i.e., translatable messages), avoid constructing the message by appending parts or using the “+” operator. Often during translation the clauses of the message need to be reordered and such construction interferes with that.

For example, instead of:

```
Console.writeline("For "  + count + " items the cost is " + cost + ".");
```

Do this instead:

```
Console.writeline(string.Format("For {0} items the cost is {1}.", count, cost));
```

This allows for reordering such as this (in English here, but often another language):

```
Console.writeline(string.Format("The cost is {1} for {0} items.", count, cost));
```

Any form of positional or named substitution (not just that offered by string.Format()) is allowed.

``

I – Never use Default Formatting of Values in Messages

For any message intended for user (vs. developer) consumption (i.e., translatable messages), never use default (often USA English culture) formatting of values in messages; instead use user locale specific culture dependent formatting.   This is especially required for numbers, currency values, dates, times and similar values.

For example, instead of:

```
Console.writeline(string.Format("For {0} items the cost is {1}.", count, cost));
```

Do this instead:

```
Console.writeline(string.Format(activeCulture, "For {0:D} items the cost is {1:C}.",
count, cost));
```

Where "activeCulture" reflects the culture of the current client, ":D" requests decimal (or integer) formatting and ":C" requests currency formatting. The special "InvariantCulture" can be used for developer oriented messages.

Many methods are possible (beyond the examples above) to achieve this.

I – Do not use the "get" Prefix on Methods that Compute a Value

If a method computes (vs. just retrieves) a value, do not name the method starting with "get"; instead use other prefixes such as "find", "calc" or "make". The "get" prefix implies a simple retrieval (generally repeatable without change and thus cacheable) of a value. In general, if the method takes any parameters (except one that is an index/key for getting from a collection), it should not be called "get".

For example, instead of:

```
public string getDomainName(string domain)
{
   return domain != null ? "ntlmdomain:" + domain : null;
}
```

``

Do this:

```
public string makeDomainName(string domain = null)
{
   return domain != null ? "ntlmdomain:" + domain : null;
}
```

I - Avoid Obvious Duplicate Code

Avoid repeating obviously identical code in multiple places. Do this especially in if/else or switch/case statements where the code is easily combined; in other places extract the code into a sharable function and call that function at each original code occurrence.

For example, instead of this:

```
if (ConfigInfo.Instance.machineNameChanged == true)
{
  RebootRequiredEvent(…);
  throw new ProcessingThreadException();
}
else if (ConfigInfo.Instance.SymRebootRequired == true)
{
  RebootRequiredEvent(…);
  throw new ProcessingThreadException();
}
else
{
   …
}
```

Do this instead:

```
if (ConfigInfo.Instance.machineNameChanged == true ||
    ConfigInfo.Instance.SymRebootRequired == true)
{
  RebootRequiredEvent(…);
  throw new ProcessingThreadException();
}
else
{
   …
}
```

Often duplicate code exists because methods are (nearly) duplicated due to different parameters. When possible combine the methods by use of defaulted parameters.

``

For example, instead of this:

```
public static void ShutDown(int seconds)
{
    ShutdownWithDelay(seconds);
}

public static void ShutDown()
{
    ShutdownWithDelay(0);
}
```

Do this:

```
public static void ShutDown(int seconds=0)
{
    ShutdownWithDelay(seconds);
}
```

MD – fac – Refactor Repeated Near Duplicate Code

If there is near duplicate code in multiple places, refactor it into a common method with parameters. If the method is of general use make it public, and perhaps move it to a common utility class.

For example, replace this:

```
public static bool DeleteReg_Core_serviceHost()
{
    bool retVal = false;
    string funcName = getContextMethodName();
    Logger.Debug(funcName, "Entering");

    string subKey = @"SOFTWARE\AppRecovery\Core\serviceHost\";

    try
    {
        ConfigUtils.RegistryDeleteSubKeyTree(subKey);

        retVal = true;
    }
    catch (Exception ex)
    {
        Logger.Error(funcName, "Exception caught ... " + ex.ToString());
``
        retVal = false;
    }

    Logger.Debug(funcName, "Exiting with retval is " + retVal);
    return retVal;
}


public static bool DeleteReg_Core_webServer()
{
    bool retVal = false;
    string funcName = getContextMethodName();
    Logger.Debug(funcName, "Entering");


    string subKey = @"SOFTWARE\AppRecovery\Core\webServer";


    try
    {
        ConfigUtils.RegistryDeleteSubKeyTree(subKey);

        retVal = true;
    }
    catch (Exception ex)
    {
        Logger.Error(funcName, "Exception caught ... " + ex.ToString());
        retVal = false;
    }

    Logger.Debug(funcName, "Exiting with retval is " + retVal);
    return retVal;
}
```

With this:

```
private static bool deleteRegCoreKey(string subkey)
{
    bool retVal = false;
    string funcName = getContextMethodName();
    Logger.Debug(funcName, "Entering");

    try
    {
        ConfigUtils.RegistryDeleteSubKeyTree(subKey);

        retVal = true;
    }
    catch (Exception ex)
    {
        Logger.Error(funcName, "Exception caught ... " + ex.ToString());
    }

    Logger.Debug(funcName, "Exiting with retval is " + retVal);
``
    return retVal;
}

public static bool DeleteReg_Core_serviceHost()
{
    string subKey = @"SOFTWARE\AppRecovery\Core\serviceHost\";
    return deleteRegCoreKey(subkey);
}


public static bool DeleteReg_Core_webServer()
{
    string subKey = @"SOFTWARE\AppRecovery\Core\webServer";
    return deleteRegCoreKey(subkey);
}
```

Here is another example:

```
bool modelFound = false;
try
{
  var accessHandle = new PS_Access();
  modelFound = accessHandle.IsPrecheckRequired();
}
catch (Exception ex)
{
  modelFound = false;
}
```

The assignment in the catch class is completely redundant with the assignment before the try and thus should be removed as follows:

```
:
catch (Exception)
{
}
```

Avoid creating local variable for just a single use (especially if that use is immediately after the variable's declaration). For example:

```
  var accessHandle = new PS_Access();
  modelFound = accessHandle.IsPrecheckRequired();
```

Can be more concisely expressed as just:

```
  modelFound = new PS_Access().IsPrecheckRequired();
```

I – Use System Services to Access Configuration Information

``

Do not create your own mechanism to access configuration information; instead use a system supplied (ex. Window’s `ConfigurationManager`) mechanism. Especially avoid schemes where application code parses complex formats like XML files.

MD – fac – Define Configuration Information External to Code

Do not hard-code configuration information; instead place it in some form of configuration file or database. This allows the configuration to be changed without rebuilding the code.

For example, avoid the first form below; use the second form instead:

```
readonly static String InstallPath = @"C:\Program Files\ \...\Provider\";
readonly static String LogFilePath = Util.readConfig("id", "logfilepath");
```

NTH – Avoid Assignments that are not Used

Do not make assignments to variables that are not referenced later; especially if the variable is created for that purpose only. Often this is done with method calls where the result is ignored. Since an expression can be used only for its side effects, indicate that by not assigning the expression value (ex. method result) to any variable.

MD- fac – Name Functions Accurately

A function name should act like the main line of any descriptive commentary. It is best if the name completely documents the method. Make sure the name accurately reflects the purpose of the method; rename the method if it does not. Prefer longer, very descriptive, names over shorter, less obvious, names. If the name gets very complex then the method has too many responsibilities, so split the function into multiples. Name brevity, while convenient during coding, hurts understanding later.

For example: use a name like `GetRelativePathElements` instead of `GetElements`.

Avoid method names that look like property access methods (i.e `GetXxx`, `SetXxx`) unless they implement those roles.

``

If the name includes/implies the return type, use the actual return type in the name.

For example:

Use a generic name when an interface is returned:

```
static IPolicy GetPolicy()  {  // don’t use GetMyPolicy here
 :
}
```

Use a specific name when a class is returned:

```
static AAPolicy GetAAPolicy() {  // don’t use GetPolicy here
 :
}
```

Do not include any “I” prefix of any type name as part of the method name. Do this:

```
static IPolicy GetPolicy()  { …}
```

Instead of this:

```
static IPolicy GetIPolicy()  { ... }
```

I – fac – Use Properties over Fields and Access Methods

Use C# Properties over explicit creation of fields and manually written get/set methods. Rarely should any field be explicitly defined to back a property; instead use automatic field generation unless that cannot be done (and add a comment that explains why). If the property requires complex logic, define the logic with the “get” or “set” attributes of the property.

Note: if a field is required, use the property name (with initial lowercase) prefixed with “_”.

If the access method requires attributes, then explicit get/set methods (vs. properties) may be required.

For example, instead of this:

```
IPolicy staticPolicy = …;

public IPolicy GetPolicy()
{
   return staticPolicy;
``
}
public void SetPolicy(IPolicy staticPolicy)
{
   this.staticPolicy = staticPolicy;
}
```

Instead do this:

```
IPolicy _staticPolicy =…;

public IPolicy Policy { get { return _staticPolicy; } set { _staticPolicy = value; } }
```

Or better (auto-generate the field), do just this:

```
public IPolicy Policy { get; set; }
```

Auto-generation makes use of the property required, even when in the same class, which is best practice.

MD – fac – Do not use Assert to Test for Expected Failures

Assert is used to detect unexpected (conceptually impossible) conditions that code is not defined to expect/handle; do not use Asserts to test for/handle predictable error conditions. In particular, it should be possible to disable all asserts and still have the code be functional (assuming the condition does not happen).

For example, instead of this:

```
int rc = srv.VerifyStorageEnclosure(controller);
Assert.IsTrue(4096 == rc, "Unexpected return code returned: " + rc);
```

Instead do this:

```
int rc = srv.VerifyStorageEnclosure(controller);
if(rc != 4096) {
  throw new SomeException("Unexpected return code returned: " + rc);
}
```

NTH – Do not Overly Complicate Logic

``

Often how a method is first written (typically via stream-of-consciousness thinking) is sub-optimal. After first writing, review the code to see if simple rework (code reorder, reverse tests, etc.) can significantly simplify the code.

For example, instead of this:

```
public static string evaluateCapacityString(double capacityinBytes)
 {
    string capacityString = "";
    if (capacityinBytes <= BytesPerKiloByte)
    {
        capacityString = (capacityinBytes / BytesPerKiloByte).ToString("F1") + "KB";
    }
    else if (capacityinBytes > BytesPerMegaByte && capacityinBytes <= BytesPerGigaByte)
    {
        capacityString = (capacityinBytes / BytesPerMegaByte).ToString("F1") + "MB";
    }
    else if (capacityinBytes > BytesPerGigaByte && capacityinBytes <= BytesPerTeraByte)
    {
        capacityString = (capacityinBytes / BytesPerGigaByte).ToString("F1") + "GB";
    }
    else if (capacityinBytes > BytesPerTeraByte)
    {
         capacityString = (capacityinBytes / BytesPerTeraByte).ToString("F1") + " TB";
    }

    return capacityString;
}
```

``

Do this:

```
public static string evaluateCapacityString(double capacityinBytes)
{
    string capacityString = "";

    if (capacityinBytes >= BytesPerTeraByte)
    {
        capacityString = (capacityinBytes / BytesPerTeraByte).ToString("F1") + " TB";
    }
    else if (capacityinBytes >= BytesPerGigaByte)
    {
        capacityString = (capacityinBytes / BytesPerGigaByte).ToString("F1") + "GB";
    }
    else if (capacityinBytes >= BytesPerMegaByte)
    {
        capacityString = (capacityinBytes / BytesPerMegaByte).ToString("F1") + "MB";
    }
    else if (capacityinBytes >= BytesPerKiloByte)
    {
        capacityString = (capacityinBytes / BytesPerKiloByte).ToString("F1") + "KB";
    }
    else
    {
        capacityString = (capacityinBytes).ToString("F1");
    }

    return capacityString;
``
}
```

Here the test order is reversed and thus simplifying the tests.

Another example, instead of this:

```
   if (!(obj == null)) …
```

Do this:

```
   if (obj != null) …
```

I – Use Enums to Create Named Symbols

Do not use distinct variables for (especially numeric) related values; use enums instead. Enums are a much better way to create a namespace for such values.

For example, instead of this:

```
const int RequestStateChangeOK = 0;
const int RequestStateChangeDmtfReservedStart = 7;
const int RequestStateChangeDmtfReservedEnd = 4095;
const int RequestStateChangeTransitionStarted = 4096;
:
const int RequestStateChangeOutofMemory = 32778;
```

Do this:

```
enum RequestStateChangeResult
{
  OK = 0,
  DmtfReservedStart = 7,
  DmtfReservedEnd = 4095,
  TransitionStarted = 4096,
  :
  OutOfMemory = 32778
}
```

``

I – Fix All Warnings

Compiler warnings generally indicate either suspicious/error prone code or code that has a likely flaw (even though it's legal C#).

All warnings should be corrected (i.e., treated as errors). If for some reason they cannot be fixed, add a comment (or perhaps attribute) at the indicated source line(s) to explain why.

I – All Inter-Process Requests Must Indicate Failure

Often requests between processes (or machines) can fail (abort without valid results). If so, such failure needs to be reported to the caller. It is incorrect to return some "inaccurate" value (ex. an empty list on a request to get a list of items or a "false" result on a Boolean operation) as apparent success. This is particularly true when the return type cannot itself imply an error occurred (ex. Boolean results).

These approaches are recommended (in increasing order of preference):

1) Always return a type that has a particular value that indicates an error (ex. for a list type, a null (vs. empty) list)
2) Always return an error code (0 for success, other for failure) as the request result and any data values by some other means
3) Throw a remote-able exception (ex. `WebFaultException`) on failure

Be consistent as to the method used for all requests to a given service.

PS: this rule can be applied to local requests as well, but it is less critical in that case.

I – Test all Return Values for Errors Values

Often methods can fail and indicate that with specific return values (such as a negative value for `int` methods or a `null` value for reference methods) instead of using exceptions to report the failure[7]. If

[7] Throwing an Exception is preferred unless there failure is a very predictable result.

``

you use such a method, you must test the result explicitly for the error result; you many never[8] assume the request always succeeds. You may test by explicit testing (ex. `if` statements) or by wrapping the code in a `try/catch`. It is an error to assume that any caller has such a `try/catch` and thus any checking can be omitted.

For example, given the method:

```
public List<String> getNames()
 {

  List<String> result = null;
  :
  if(<some expression>)
  {
    result = new List<String>();
    :
  }
  return result;
}
```

While convenient, do not use it like this:

```
String name1 = getNames()[0];
```

This code errs in not testing for null or that an element exists; better is:

[8] Unless you did pre-checks elsewhere to ensure that a failure (ex. index out of range) is not possible.

```
``
List<String> names = getNames();

if(names != null && names.Count > 0)

{

  String name1 = names[0];

}
```

MD – Avoid Critical Sections

In any code expected to be used (directly or indirectly) in a multi-threaded environment (such as a request processor in a server) one must be careful to correctly manage *critical sections*[9]. The best approach is to avoid critical sections wherever possible. The easiest way to do that is to use only local variables in methods. Local variables are allocated on a stack (unique per thread) and are unique per method call.

Note: use of local variables that are references to objects is not intrinsically safe.

Note: use of static values is nearly always an issue as they are shared across instances.

Note: use of instance values is an issue if multiple threads can run against a single instance; this is typical of web servers.

If all critical section cannot be avoided then controlled access to them needs to be employed. There are two main ways to do this.

1. Use of the `lock` statement (or library routines that are the equivalent, such as `System.Threading.Interlocked`)

   Each access (read and write) to a value needs to be enclosed in a lock statement using the same lock value. Locks must be used if the value needs to be shared across threads (for instance when accessing static or instance variables).

2. Use of `ThreadLocal` variables

   Each shared value needs to be created as a *ThreadLocal* object; this guarantees each thread gets a separate instance.

[9] A section of code in which values (variables) are examined and changes across threads such that there are potential race conditions between the threads.

``

It is rare when any non-local variable can be safely used if one of the actions has not been taken. Failure to take the above precautions is likely to result in unpredictable (and often hard to reproduce and fix) failures.

These rules apply both to main entry points (ex. web request processors) and also any helper routines used by them.

NTH -Fail soft on Group Assignments

Often when making multiple assignments, especially to fields of an object, avoid using a single `try/catch` around all assignments. Instead wrap each assignment in a separate `try/catch`. This allows a softer failure (i.e., more complete assignment of values). Do this especially if any fields are actually properties (which might run code); since it's hard to know if the reference is to a field or property, assume it's to a property to be safe.

For example, avoid this:

```
                   try
                    {
                        lastExportTime = virtualStandby.LastExportTime;
                        exportPercent = virtualStandby.ExportPercentage;
                        location = virtualStandby.VMLocation;
                        vmName = virtualStandby.VirtualMachineName;
                        agentName = virtualStandby.AgentName;
                        hvName = virtualStandby.HyperVisorName;
                        hvPort = virtualStandby.HyperVisorPort;
                        hvType = virtualStandby.hyperVisorType;
                        JobStatus jobStatus = virtualStandby.exportStatus;
                        exportStatus = ConvertJobStatusToExportStatus(jobStatus != null ?
jobStatus : JobStatus.Unknown);
                    }
                    catch (Exception ex)
                    {
                        logMessage(LEVEL_FINE, "Exception converting values", ex);
                    }
```

Instead place a try/catch around each assignment. To make this easier, use a library helper method. So instead of this:

```
                        hvPort = virtualStandby.HyperVisorPort;
```

``

do this:

```
                        safeAssign(() => { hvPort = virtualStandby.HyperVisorPort; });
```

for each assignment, with the `try/catch` inside of `safeAssign()` instead of around all the assignments.