TABLE OF CONTENTS

1 OVERVIEW

This document may be read as a guide to writing robust and reliable programs. It focuses on programs written in C#, but many of the rules and principles are useful even if you write in another programming language.

2 FILE ORGANIZATION

2.1 C# Source Files

- Keep your classes and source files short. Try not to exceed 2000 lines of code.
- Put every class in a separate file and name the file like the class name.
- Lines should not exceed 100 characters and user can see all words of one line and they don't need to drag the scroll bar.

2.2 C# Projects

- The project should be set to "AnyCPU".
- Set the warnings level to maximum and turn warnings into errors.

2.3 Directory Layout

Create a directory for every namespace. (For MyExtension.Revit.TestTier use MyExtension/Revit/TestTier as the path, do not use the namespace name with dots.) This makes it easier to map namespaces to the directory layout.

2.4 Methods

- Avoid methods with more than 5 arguments. Use structures or options classes to pass more information.
- Avoid methods with more than 500 lines.
- If you have more than 2-3 subroutine nests, it is better to split code out into a separate method.

3 FORMATTING

3.1 Indentation

In Visual studio 2010, for a common formatting set Tab size to 4, Indent size to 4, and turn on Insert spaces. Chose "smart" auto indent and do not indent open and close braces.

3.2 Align declarations vertically

A logical block of lines should be formatted as a table:

```
string variableName = "Mr. Ed";
int myValue         = 5;
```

```
Test aTest          = Test.TestYou;
```

Use spaces for the table like formatting and not tabs because the table formatting may look strange in special tab intent levels.

3.3 Wrapping Lines

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

- Break after a comma.
- Break after an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line

Example of breaking up method calls:

```
longMethodCall(expr1, expr2,
               expr3, expr4, expr5);
```

Examples of breaking an arithmetic expression:

```
PREFER:

var = a * b / (c - g + f) +
      4 * z;

BAD STYLE – AVOID:

var = a * b / (c - g +
      f) + 4 * z;
```

The first is preferred, since the break occurs outside the parenthesized expression (higher level rule). Note that you indent with tabs to the indentation level and then with spaces to the breaking position in our example this would be:

```
> var = a * b / (c - g + f) +
> ......4 * z;
```

Where '>' are tab chars and '.' are spaces. (the spaces after the tab char are the indent with of the tab). A good coding practice is to make the tab and space chars visible in the editor which is used.

4 COMMENTS

4.1 Functional Comments

You should comment the important part of your code. The comment should explain what the following code block is doing. Also note solved problems.

- Provide comments to explain tricky or non-obvious code.
- Provide comments to explain workarounds to issues.
- Bug description
- Notes to co-developers
- Performance/feature improvement possibilities
- Use TODO in comments to indicate tasks yet to be completed.
- Except for the requirements listed above, avoid comments that simply restate the code. Most code which uses explanatory variable names should be self explanatory.

4.2 Block Comments

To create a block comment use this style:

```
//Line 1
//Line 2
//Line 3
```

Avoid C-style comments `/* */` as it is hard to see code proceeded by comments in the same line. It also helps to use the single line comments as most IDEs can automatically add and remove them.

4.3 Single Line Comments

Single line comments must be indented to the indent level when they are used for code documentation. Commented out code should be commented out in the first line to enhance the visibility of commented out code.

4.4 Documentation Comments

XML-style documentation comments must be provided for all class, interface, enum, method, and property declarations. For private members, a summary must be provided. For non-private members, a summary must be provided along with appropriate parameter, returns, and exception documentation.

All public and protected items need to have documentation comments; all others are not mandatory but are recommended.
These comments are formally single line C# comments containing XML tags. They follow this pattern for single line comments:

```
/// <summary>
/// This class represents ..
/// </summary>
```

Multi line XML comments follow this pattern:

```
/// <summary>
/// This class represents ..
/// More Comments to the class ..
/// </summary>
```

All lines must be preceded by three slashes to be accepted as XML comment lines.
Tags fall into two categories:

- Documentation items
- Formatting/Referencing

The first category contains tags like `<summary>`, `<param>` or `<exception>`. These represent items that represent the elements of a program's API which must be documented for the program to be useful to other programmers. These tags usually have attributes such as name or cref as demonstrated in the multi line example above. These attributes are checked by the compiler, so they should be valid.

The latter category governs the layout of the documentation, using tags such as `<code>`, `<list>` or `<para>`.

`<c>`	Marks a part of a comment to be formatted as code
`<code>`	As above, but multi line
`<example>`	For embedding examples in comments, usually uses <c>
`<exception>*`	Documents an Exception Class
`<include>*`	Includes documentation from other files
`<list>`	A list of <term>s defined by <description>s
`<para>`	Structures text blocks, e.g. in a <remark>
`<param>*`	Describes a method parameter
`<paramref>*`	Indicates that a word is used as reference to a parameter
`<permission>*`	Gives the access permissions to a member
`<remarks>`	For overview of what a given class or other type does
`<returns>`	Description of the return value
`<see>*`	Refers to a member or field available
`<seealso>*`	As above, but displays a 'See also' section
`<summary>`	A summary of the object
`<value>`	Describes a property

4.4.1 General guidelines

A typical xmldoc comment looks like this:

```
/// <summary>Adds two numbers.</summary>
```

Or like any of these:

```
/// <summary>
/// Adds two numbers.
/// </summary>
```

```
/// <summary>
///     Adds two numbers.
/// </summary>
```

```
/// <summary>
///     Adds two
///     numbers.
/// </summary>
```

Guidelines for comments:

- Any number of spaces or tabs before "///".
- At least one space after "///".
- Doc comments are always implemented as doc comment elements. Element types are described below.
- Comment text begins with a capital letter and ends with a period. Even if it's only one word.
- Don't omit articles. Don't say, "Gets midpoint of segment." Instead say, "Gets the midpoint of a segment".
- Note that multiple whitespace characters (including spaces, tabs, and line ends) are collapsed to one space when comments are harvested.

Between tags, you can add whitespace to format text for readability in source files, but this formatting is abandoned in the harvesting process.
Typical xmldoc comment block:

```
/// <summary>Adds two numbers.</summary>
/// <remarks>The numbers can be positive or negative.</remarks>
/// <param name="dSum">Output the sum.</param>
/// <param name="d1">Input a number.</param>
/// <param name="d2">Input a number.</param>

public ResultEnum
Adder(
   double dSum,
   double d1,
   double d2);
```

Guidelines for comment blocks:

- Every component that is exposed for public use requires a doc comment block.
- A doc comment block consists of one or more xmldoc elements.
  A <summary> element is always required and always comes first.
- Insert at least one blank line before a comment block.
- Place a comment block immediately before the declaration that it refers to.
- No blank lines within a comment block.
- No blank lines between a comment block and its declaration.
- But blank comment-lines are OK. For example:

```
/// <param name="dN1">Input a number.</param>
///
/// <param name="dN2">Input a number.</param>
```

- Note that multiple whitespace characters (including spaces, tabs, and line ends) are collapsed to one space when comments are harvested.
  See the bullet item on this subject in "Doc Comment Format" above for more information.
- Don't repeat information in doc comments that the reader can get from inspecting the declaration that follows.
- Persistent Formatting in Doc Comments

Although multiple whitespace characters are collapsed to one space when comments are harvested, which means that most doc comment formatting is abandoned in the harvesting process, there are ways to implement persistent formatting with Doc-O-Matic as the harvesting tool.
The following sections also describe xmldoc elements supported by Doc-O-Matic, the comment types that they contain, and notes on comment style.

4.4.2 Summary and Remarks Elements

A <summary> element is required for each declaration. It contains a brief description only, typically a sentence fragment without a subject, such as "Adds two numbers."

Important! Keep <summary> descriptions brief!

If you have more to say, insert one or more <remarks> elements following the <summary> element.

Begin the text for a <summary> element with a present tense verb, like *Initializes*, *Sets*, *Gets*, *Defines*, *Determines*, and so on. Don't use *Returns*, because it is reserved for the <returns> element.

Example:

```
/// <summary>Adds two numbers.</summary>
/// <remarks>The numbers can be positive or negative.</remarks>
```

4.4.3 Param Element

A <param> element is required for each parameter of a function.
For output parameters, begin <param> text with the word *Output*. For input parameters, begin with *Input*. Follow *Input* or *Output* with a description of valid argument(s) and what they represent. With the output parameter of a "get" function, indicate what the default output value is (what the data member in question is initialized to).
Every <param> element must have a *name* attribute. The value is the name of the parameter in question.
Important! The *name* value must match the corresponding parameter name in the function declaration that follows! Be especially careful when you modify parameter names to keep *name* attributes in sync.

Example:

```
/// <param name="d1">Input a number.</param>
/// <param name="dSum">Output the sum.</param>
```

4.4.4 Exception Element

Where an error is signaled by throwing an exception, insert an <exception> element:

```
/// <exception cref="ExceptionTypeName">Thrown when...[explain]....</exception>ntation
Comments
```

4.4.5 Function Comments

```
/// <summary>Adds two numbers.</summary>
/// <remarks>The numbers can be positive or negative.</remarks>
/// <param name="dSum">Output the sum.</param>
/// <param name="d1">Input a number.</param>
/// <param name="d2">Input a number.</param>

public ResultEnum
Adder(
   double dSum,
   double d1,
   double d2);
```

The following comment block includes an <exception> element:

```
/// <summary>Invokes the Define Property Alteration dialog, which
/// specifies the type of property alteration.</summary>
/// <exception cref="MapException">kErrUsrBreak in the case of
/// cancel.</exception>
/// <param name="alteration">A type of property alteration, such as
/// TextAlteration, AnnotAlteration, or HatchAlteration.</param>
```

4.4.6 Enum Comments

```
/// <summary>Classification error codes.</summary>
/// <remarks>More info, if any, which in this case there isn't.</remarks>
```

```
enum EErrCode
{
   eOk=0x1,       // Everything is OK.
   eFailed,       // The action failed.
   ...,           // ....
   EClassNotFromCurrentSchema
                  // The instance of the class AcMapObjClassDefinition
                    // comes from a different feature definition file than
                  // the current one. It is no longer usable unless the
                  // original feature definition file is reattached.
};
```

4.4.7 Struct Comments

The same as Enum comments

.

4.4.8 Class, Namespace, or Interface Comments

Class example:

```
/// <summary>A set of attached drawings.</summary>
/// <remarks>Every Autodesk Map drawing has a drawing set containing
/// zero or more attached drawings.</remarks>
class DrawingSet {
...
};
```

Do the same with namespaces and interfaces.

5 DECLARATIONS

5.1 Number of Declarations per Line

One declaration per line is recommended since it encourages commenting. In other words,

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

Do not put more than one variable or variables of different types on the same line when declaring them. Example:

```
int a, b; //What is 'a'? What does 'b' stand for?
```

5.2 Initialization

Declare variables at the point in which they are needed. Do not group all declarations at the top of a method. Try to initialize local variables as soon as they are declared. For example:

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

or

```
int val = time.Hours;
```

5.3 Const and read-only

- Use the const directive only on natural constants such as the number of days of week.
- Use the readonly directive for single initialization members.

5.4 Using

Use the using statement for declarations where it's important call Dispose() on the variable when it is no longer needed.

```
using (OpenFileDialog openFileDialog = new OpenFileDialog())
{
    ...
}
```

5.5 Braces

Put the opening brace on a new line after the declaration statement or language construct which leads into it. Put the closing brace on a new line matching the opening brace's indentation.

5.6 Class and Interface Declarations

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

- No space between a method name and the parenthesis "(" starting its parameter list.
- The opening brace "{" appears in the next line after the declaration statement.
- The closing brace "}" starts a line by itself indented to match its corresponding opening brace.

For example:

```
            /// <summary>
            /// My increment Class.
            /// </summary>
            public class Increment : MyClass, IMyInterface
            {
                  private int myInteger;

                  /// <summary>
                  /// My increment Constructor.
                  /// </summary>
                  ///    <param    name="myInteger">The    int    value    to
increment.</param>
                  public Increment(int myInteger)
                  {
                     this.myInteger = myInteger;
                  }

                  /// <summary>
                  /// My Public increment Method.
                  /// </summary>
                  public void IncrementMyInteger()
```

```
        {
            myInteger++;
        }

        /// <summary>
        /// My Private empty Method.
        /// </summary>
        private void emptyMethod()
        {
            // Do nothing
        }
    }
```

5.7 Enums

- Avoid providing explicit values for enum members.

Incorrect

```
public enum Color
{Red = 1, Green = 2, Blue = 3}
```

Correct

```
public enum Color
{Red, Green, Blue}
```

6 STATEMENTS

6.1 Simple Statements

Each line should contain only one statement.

6.2 If, if-else, if else-if else Statements

If, if-else and if else-if else statements should look like this:

```
if (condition)
{
    DoSomething();
}

if (condition)
{
    DoSomething();
}
else
{
    DoSomethingOther();
}
```

```
        if (condition)
        {
           DoSomething();
        }
        else if (condition)
        {
           DoSomethingOther();
        }
        else
        {
           DoSomethingOtherAgain();
}
```

If there is only one statement you can omit brackets.

```
        if (condition)
              DoOnething();
```

6.3 For / Foreach Statements

A for statement should have following form:

```
        for (int i = 0; i < 5; ++i)
        {
           // Do something.
}
```

or single lined (consider using a while statement instead):

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

A foreach should look like:

```
        foreach (int i in IntList)
        {
           // Do something
}
```

You can omit brackets if there is only one statement in the loop.

6.4 While/do-while Statements

A while statement should be written as follows:

```
        while (condition)
        {
            // Do something
}
```

A do-while statement should have the following form:

```
        do
        {
            // Do something
        }
        while (condition);
```

6.5 Switch Statements

A switch statement should be of following form:

```
        switch (condition)
        {
            case A:
                // Do something
                break;

            case B:
                // Do something
                break;

            default:
                // Do something
                break;
}
```

6.6 Try-catch Statements

A try-catch statement should follow this form:

```
        try
        {
            // Do something
        }
        catch (Topobase.Exception.TBException ex)
        {
            // Handle Exception.
```

```
        }

        try
        {
            // Do something
        }
        catch (Topobase.Exception.TBException ex)
        {
            // Handle Exception.
        }
        finally
        {
            // Clean up any resources.
    }
```

7 WHITE SPACE

7.1 Blank Lines

Blank lines improve readability. They set off blocks of code which are in themselves logically related.
Two blank lines should always be used between:

- Logical sections of a source file
- Class and interface definitions (try one class/interface per file to prevent this case).

One blank line should always be used between:

- Methods
- Properties
- Local variables in a method and its first statement
- Logical sections inside a method to improve readability.

Note that blank lines must be indented as they would contain a statement this makes insertion in these lines much easier.

7.2 Region

For better structure use "`#region Comment`" & "`#endregion`" in your code and also define for every region a short description, otherwise it doesn't make sense. To achieve this, please have a look at the following ordered list of grouping items:

- Constants
- Member Variables
- Constructor / Destructor
- Properties
- Methods

There is only one recommendation, do not declare the Constant and Member Variables in the bottom of the Code (These regions has to be defined at the top). The indicated order is not mandatory, but you should group at least the code which belongs together. For complex classes you have to specify a more detailed structure, e.g. create separate Region for Public and Private Properties.

7.3 Inter-term spacing

There should be a single space after a comma or a semicolon, for example:

```
TestMethod(a, b, c);
```

Single spaces surround operators (except unary operators like increment or logical not), example:

```
            a = b;
      for (int i = 0; i < 10; ++i)
```

8 NAMING CONVENTIONS

8.1 Capitalization Styles

8.1.1 Pascal Casing

This convention capitalizes the first character of each word (as in **T**est**C**ounter).

8.1.2 Camel Casing

This convention capitalizes the first character of each word except the first one, e.g. test**C**ounter.

8.1.3 Upper case

Only use all upper case for identifiers if it consists of an abbreviation which is one or two characters long. Identifiers of three or more characters should use Pascal Casing instead. For Example:

```
            /// <summary>
            /// My Math Class.
            /// </summary>
            public class Math
            {
               private const double PI = 3.14159;
               private const double E = 2.71828;
               private const double eulerConstantNumber = 0.57721;
      }
```

8.1.4 Naming Guidelines

Generally the use of underscore characters inside names and naming according to the guidelines for Hungarian notation are considered bad practice.
Hungarian notation is a defined set of pre and postfixes which are applied to names to reflect the type of the variable. This style of naming was widely used in early Windows programming, but now is obsolete or at least should be considered deprecated. Using Hungarian notation is not allowed if you follow this guide.
And remember: a good variable name describes the semantic not the type.

An exception to this rule is GUI code – See REX Design Guidelines documents . All fields and variable names that contain GUI elements like button should be post fixed with their type name without abbreviations. For example:

```
System.Windows.Forms.Button cancelButton;


System.Windows.Forms.TextBox nameTextBox;
```

Do not use a prefix for member variables (_, m_, s_, etc.). If you want to distinguish between local and member variables you should use “this.”.

8.1.5 Class Naming Guidelines

- Class names must be nouns or noun phrases.
- Use Pascal Casing

- Do not use any class prefix

8.1.6 Interface Naming Guidelines

- Name interfaces with nouns or noun phrases or adjectives describing behaviour.
- Use Pascal Casing
- Use I as prefix for the name, it is followed by a capital letter (first char of the interface name)

8.1.7 Enum Naming Guidelines

- Use Pascal Casing for enum value names and enum type names
- Don't prefix (or suffix) a enum type or enum values
- Use singular names for enums

8.1.8 Read-Only and Const Field Names

- Name static fields with nouns, noun phrases or abbreviations for nouns
- Use Pascal Casing

8.1.9 Parameter/non const field Names

- Do use descriptive names, which should be enough to determine the variable meaning and its type. But
- Prefer a name that's based on the parameter's meaning.
- Use Camel Casing

8.1.10 Variable Names

- Counting variables are preferably called i, j, k, l, m, n when used in 'trivial' counting loops.
- Use Camel Casing

8.1.11 Method Names

- Name methods with verbs or verb phrases.
- Use Pascal Casing

8.1.12 Property Names

- Name properties using nouns or noun phrases
- Use Pascal Casing
- Consider naming a property with the same name as it's type

8.1.13 Event Names

- Name event handlers with the EventHandler suffix.
- Use two parameters named sender and e
- Use Pascal Casing
- Name event argument classes with the EventArgs suffix.
- Name event names that have a concept of pre and post using the present and past tense.
- Consider naming events using a verb.

8.1.14 Capitalization summary

Identifier	Case	Example
Class / Struct	Pascal	AppDomain

Enum type	Pascal	ErrorLevel
Enum values	Pascal	FatalError
Event	Pascal	ValueChange
Exception class	Pascal	MyException **Note** Always ends with the suffix Exception.
Read-only Static field	Pascal	RedValue
Interface	Pascal	IDisposable **Note** Always begins with the prefix `I`.
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor

8.1.15 Namespace

Please consider the following REX Namespace rules:

- The Default Namespace must start with REX. Classes must have namespace starting with REX.
- The Assembly name should start by an upper case and has to reflect the location name.
- Use Pascal case for namespace.
- Use plural namespace names if it is semantically appropriate, e.g. REX.Collections rather than REX.Collection.
- Do not use the same name for a namespace and a class.

Hint:
Prefixing namespace with a well-established brand, like REX, avoids the possibility of two published namespaces having the same name.

9 PROGRAMMING PRACTICES

9.1 Visibility

Do not make any instance or class variable public, make them private.
Use properties instead. You may use public static fields (or const) as an exception to this rule, but it should not be the rule.
Make items only public when you expect to let use them. As consequent you need to check the input data (Arguments).

9.2 No 'magic' Numbers

Don't use magic numbers, i.e. place constant numerical values directly into the source code. Declare a const variable which contains the number:

```
/// <summary>
/// My Math Class.
```

```
/// </summary>
public class Math
{
   private const double PI = 3.14159;
}
```

9.3 Use defined classes and values in .NET Framework

Do not define your own values for constants like PI, the number of days in the week, or other capabilities already well defined in the .NET Framework.

Do not add code to solve problems already well served by the .NET Framework (for example, Date and time operations should use the .NET classes and utilities, not manually written code.

9.4 Class per file

There should be only one public/internal class within a file and the filename should match the class name.

9.5 Exception Handling

Handling exceptions the right way will lead to an application which provides helpful warnings and errors to the user and also useful log files to the developer.
Implementation is not 'exception rules' compliant. Therefore well-founded exceptions are allowed.

10 COMPILER WARNINGS

Basically the Microsoft compiler produces only legitimate warnings. Therefore all new projects have to be written that no warning occur in the compilation process. Existing projects must not be made warning-free since this would take weeks to clean every project.

11 LOCALIZATION

User interface will be translated on several languages. Take it into consideration all the time.

11.1 User interface

The REX Design Guidelines document provides general rules about User interface.
Our products are translated into many different languages. The look & feel among all language products should be the same.
The size of label should be made maximum size possible. A common rule is to use a text in English and add 30 %. If this condition seems impossible to realize, a dialog redesign is necessary.

11.2 Strings

All user interface texts and strings used should be set on resource files.
Use unique string and don't reuse strings from a dedicated control to another one.
Insure functionality of sorted ComboBoxes and ListBoxes according localization.
Avoid as much as possible string concatenation and font hardcoded.

11.3 Bitmap

Use culture neutral bitmaps.
Insert only symbol on bitmap. Texts couldn't be translated on images.