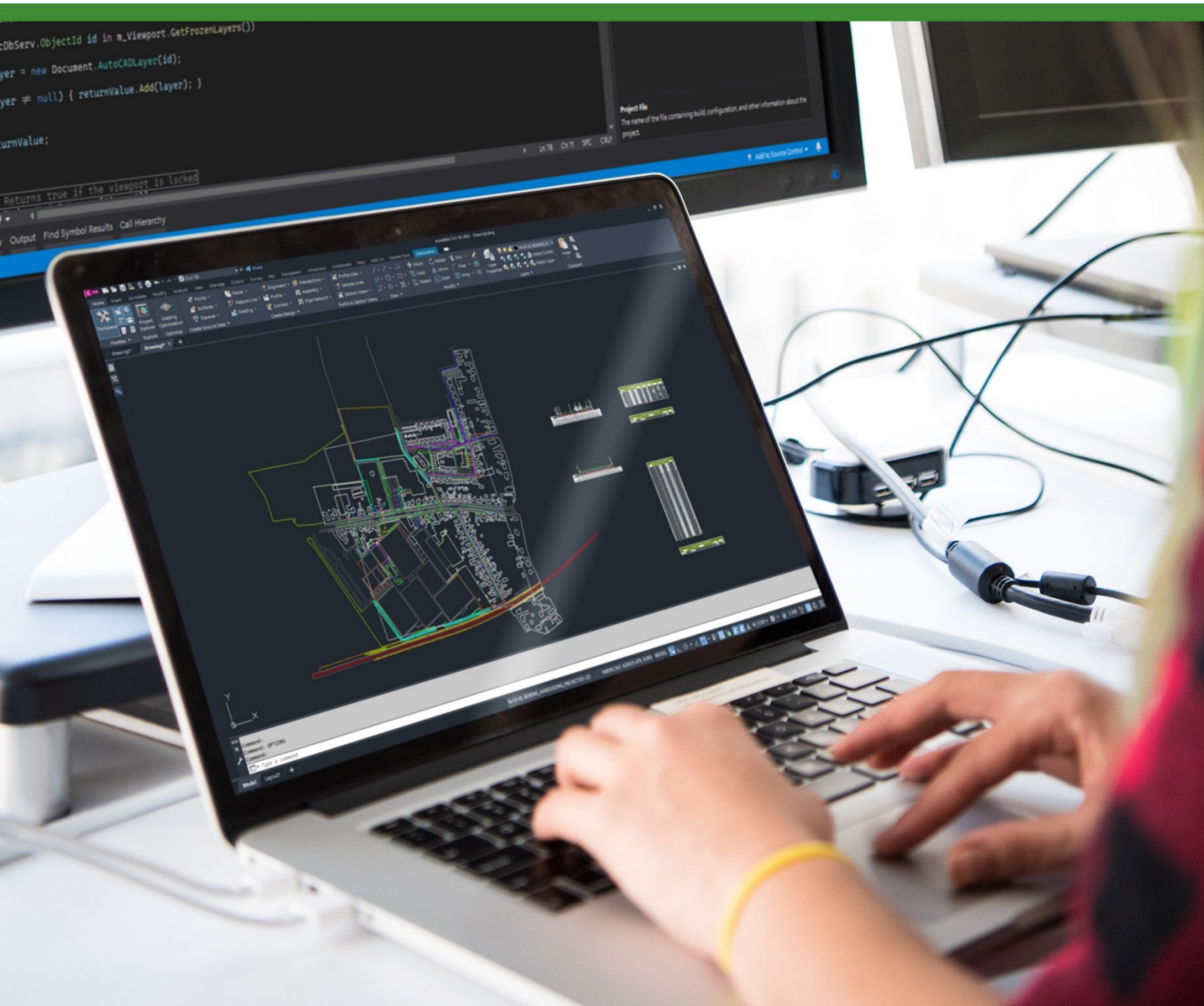
Anton Huizinga

Coding Conventions and Guidelines



Coding conventions and guidelines for writing source code

Anton Huizinga

Version 2.0 - last modified 26/05/2022

www.huiz.net/en/
www.linkedin.com/in/antonhuizinga/

FOREWORD

Using conventions and guidelines is not only indispensable for collaborating programmers, it is also essential for independent programmers to start working in a structured way.

Code that you have written and only read again months or years later can sometimes make you scratch your head because of ambiguities. Why did you write it that way then? Were you asleep perhaps, and what did you really expect from that spaghetti code?

That's why it's wise to start programming consistently, sticking to the same structures to provide other programmers and of course yourself with readable and clear code.

This book contains my agreement set. What yours will look like I don't know. You may use this set, you may modify this set, or you may write a completely new set for yourself. You may delete half of the guidelines or double them.

As long as you're going to use it.... ☺

Anton Huizinga

"Real programmers don't comment their code.
If it was hard to write, it should be hard to read."

TABLE OF CONTENTS

1 IN GENERAL

This document describes an elaboration of conventions and guidelines for writing source code.

No single set of conventions or guidelines will be adequate for all programmers in the world. The purpose of a standard is to encourage efficiency for single or collaborating programmers. It will result in fewer bugs and more maintainable code. The purpose of this standard is to make you aware of how to write better code. It hopefully triggers you to think about a set of conventions and guidelines, whether you use this set or write a whole new one.

At first, it will be difficult and tedious to start applying one/your standard, but after a while, through habituation, it will actually become enjoyable to program consistently.

PRINCIPLES

The principles of applying a standard are:

- ✓ **Understandable**: meaning that the code is readable by, for example, using full names instead of abbreviations, and applying documentation where it clarifies.
- ✓ **Correct**: meaning that the code does not cause unexpected errors by, for example, checking all values beforehand.
- ✓ **Consistent**: showing that you apply the same structures every time, this demonstrates craftsmanship and proves that you have an eye for detail.
- ✓ **Modern**: abandoning old coding techniques that are often cumbersome and unclear.
- ✓ **Legal**: no hacks or code that makes permanent changes to the computer. Any changes caused must be reversible. This does not mean deleting or creating files with the user's permission.
- ✓ **Certain**: meaning that version numbers are correct, the linked libraries are clear and there can be no confusion about which functionality and/or known bugs are or are not present in which version.

TERMINOLOGY

This booklet periodically states what you should or should not do, what you should or should not do, and why you should do something. This is represented as follows:

Do: do the following ...

Don't: don't do it this way ...

Why: because it is ...

2 GENERIC STANDARDS

This chapter applies to general issues surrounding programming and not with the specific language structure of .NET languages.

CLEAR AND CONSISTENT

Be clear and consistent.

Do: This set of conventions strives to ensure that written code is clearly readable, easily understood, and easily maintained.

Do: Consistently apply this set of conventions to all the code you write.

It may be stating the obvious, but you write (or adopt) a set of standards to improve your source code. So apply it, and take the time to apply it. Every time you take a moment to do something a different way because it seems time-saving, you will be working to make your code worse and the code becomes less maintainable.

Compare it to running a red light. Once you ignore the rule not to run a red light, it immediately results in a piece of unsafe road behavior. Both for you and for a fellow road user, even if it doesn't bother you much at the time. You can get into an accident yourself. You can kill someone because they think they can cross the road safely. Or someone has to brake very hard for you and causes a chain collision. Or other people also become nonchalant and start ignoring agreed-upon traffic rules just like you. Until society has become an anarchy.

Think about that again if you don't follow your own coding agreements.

FORMATTING AND STYLE

Don't: Using tab characters to indent code. If you take a look at the files in another editor, alternating between tabs and spaces can make for an unreadable mush.

Visual Studio can be configured to automatically convert tabs to spaces. Maintain a short spacing of, say, 2 to 4 spaces. That's more than enough to indent cleanly.

Break lines of code into readable blocks of maximum length. Usually text lines up to about 80 characters are readable on a screen. By breaking down the lines you prevent horizontal scrolling.

Use a fixed-width font, such as '`Consolas`' or '`Courier New`'. Microsoft has also made the font '`Cascadia Code`' available, especially for IDEs. The font uses ligatures (compound characters) within Visual Studio.

LIBRARIES

Don't: Linking to libraries that are not being used. This causes unnecessary ballast when compiling and loading your application. Furthermore, it can cause confusion for other programmers, where exactly is a linked library called in your code?

VARIABLES

Do: Declare variables as late in the code as possible, only when you need them or at the beginning

of a block of code in which you need them.

Limit the use of variables that you use throughout the application. If necessary, group them in a settings class.

INITIALIZATION OF VARIABLES

Do: Initialize (provide with a value) variables as soon as you declare them. Don't assume that .NET will come up with a correct initial value though.

Good:

```
string text1 = "This is text.";
string text2 = string.Empty;
```

Wrong:

```
string text;
double number;
```

Don't: Do not declare multiple variables with the same value at the same time. It can be done but it will also be able to cause errors.

Good:

```
string text1 = "This is text.";
string text2 = "This is text.";
```

Wrong:

```
string text1 = text2 = "This is text.";
```

Why: For variables that directly contain a value, the above will work, but it is worse read. Variables that do not contain a value but point to a memory location that contains the value will fail. All variables point to the same memory location. If you then change the second variable, the first one suddenly refers to it as well. Even though it is not a coding error in .NET, never do it.

PARAMETERS

Do: Sort parameters in a logical way. If you expect parameters "Left" and "Right" in this order, use them as such and use a similar name.

Good:

```
public void GetSpace(double left, double right, string name, int doors)
```

Wrong:

```
public void GetSpace(double Right, int Doors, string Name, double LeftSide)
```

If you use Out Parameters, always put them at the end. First In Parameters, then Out Parameters.

Don't: As much as possible, avoid using parameters with a default value. Just think carefully about the parameters you have to fill in. With default values, you quickly overlook what is happening.

Good:

```
public void FormatDisk(string drive)
```

Wrong (less good):

```
public void FormatDisk(string drive = "C")
```

STATEMENTS

Don't: Do not place more than one Statement on a line. It makes reading more difficult and debugging can no longer be done on that one Statement.

Good:

```
string text1 = "This is text.";
string text2 = string.Empty;
```

Wrong:

```
string text1 = "This is text."; string text2 = string.Empty;
```

ENUMS

Do: Make use of Enums if you want to use fixed values in parameters. This makes it clearer which values you can apply and you get help from the IDE.

Don't: Don't use constants or variables when an Enum is more appropriate.

Good:

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

Wrong:

```
const int Red = 0;
const int Green = 1;
const int Blue = 2;
```

Don't: Don't use the Enum.IsDefined() function because it creates quite a bit of extra load in the background.

Do: If possible, set a **0** value with a clear name. Also use this name consistently, for example "None".

Example:

```
enum Direction
{
  None = 0,
  Left,
  Right
}
```

This obviously does not fit Enums that should not return none-values. For example, with colors, it doesn't make sense to return the None option:

```
enum Color
{
  None = 0,
```

```
  Red,
  Green,
  Blue
}
```

Do: For Enums that must support Bitwise, set the Flags Attribute. Set default values with a power-to-the-two in each case.

Don't: Do not set a default value of **0** for Enums that must support Bitwise.

Good:

```
[Flags]
enum AccessRights
{
  Read = 1,
  Write = 2,
  Delete = 4,
  Copy = 8
}
```

Wrong:

```
[Flags]
enum AccessRights
{
  None = 0,
  Read = 1,
  Write = 2,
  Delete = 4,
  Copy = 8
}
```

Why: For Bitwise Enums, in the wrong example, "None" will always be valid.

WHITESPACE

Do: It is recommended to apply white space. There should be a white line at the top of each block that is separate from the previous statement. This reads much easier. Limit it to one white line, except above each function. Separate each function with at least 2 and preferably 3 white lines so you can scroll through the code faster and see where a new function begins.

CURLY BRACES

Accolades are used to join blocks of code together to form matching code. For *if* constructions, .NET does not require you to use braces if only one statement is executed, but teach yourself to do so.

Do: Always apply braces to a condition.

Good:

```
if (a == 5)
{
  b = a * a;
```

```
}
```

Wrong:

```
if (a == 5)
  b = a * a;
```

Why: With curly braces it reads much better, now you know what is executed when the condition is **true**. Moreover, in the above it is still reasonable to estimate what is executed when the condition is **true** but with a using construction it becomes more complicated.

Wrong:

```
using (Database db = Application.DocumentManager.MdiActiveDocument.Database)
  using (Transaction tr = db.TransactionManager.StartTransaction())
  {
    BlockReference brP = null;
    Point3d ptLocation = new Point3d();
    double x = 0.0;
    double y = 0.0;
    ...
```

In the above example, what is being executed within the first using? The code will work fine but is not clear to read. So always use braces.

COMMENTS

Do: Write comments that summarize what a piece of code will do. Write this as full text with initial capitalization and punctuation.

Don't: Don't repeat the entire code in the comments.

Do: Inline comments are placed either above a statement or after a statement in the same line. Moreover, you only do this where it clarifies things. For example with variables that you declare at the top, to explain why you are going to use them in the code. Or with a difficult construction to explain at which step you are now.

Don't: Don't write a piece of commentary after every line. You really don't need to write more comments than code. If you want to say more then you can also summarize that above that piece of code or above the function itself.

Do: Add a File Header to each file that includes the name of the file, a brief description of the functionality, the author, and possibly the license. Don't make it too complex either, full version control of the file is often over the top.

Do: Add comments above the declaration of a Class and above each function. Do this with an XML Documentation Comment. IntelliSense will use this to provide you with the correct information when you call a Class or a function.

Example:

```
/// <summary>
/// This is an example class and it does not do anything.
/// </summary>
public class Example {
```

In Visual Studio, you can start by typing three slashes and then the IDE itself will generate such a comment block. Once you have added this, the IDE will provide you with help.

Do: Add TODO comments where you are still working on or expect to extend something. Before you finalize the application, you can go through all the TODO comments and update them with the appropriate code. It helps you not to forget certain things and sends you directly to the right place.

TODO comments are written as:

```
// TODO: This class must be extended some day.
```

This will result in a list in the Task List. Double clicking on such a notification will take you straight to the right place.

3 .NET CODING STANDARDS

The content of this chapter focuses on conventions and guidelines related to the .NET structure.

FILES AND STRUCTURE

Do: Name the files after their contents. For example, a Class named MainForm belongs in a file named 'Mainform.cs', or possibly with a predecessor such as a Namespace: 'MyApp.MainForm.cs'.

Don't: Avoid putting multiple Classes in one file. You are less likely to find it if a Class named Color is somewhere below a Class named LineTypes in a file named 'MyApp.LineTypes.cs'. That Class belongs in its own file called 'MyApp.Color.cs'.

ASSEMBLY PROPERTIES

Always fill in the Properties of the final Assembly with clear descriptions such as company name, application name, description and version.

VERSIONS

Do: Think about how you will apply version numbering. Will you use Major and Minor? Or Builds as well? Minor changes in a Build or in a Minor?

Always update the version when you have released a build. This prevents different variants of the same version from being on the market. Moreover, it is more difficult to install if the computer already has the same version.

NAMING CONVENSIONS

Do: Always use meaningful names for everything. Also, always use the same style consistently. All functions that fetch something have them start with, for example, 'Get...' and all functions that set something then have them start with 'Set...'.

Don't: There is no shortage of space on a computer. Therefore, never use abbreviations unless it is a commonly used abbreviation. 'Proc' is a commonly used word for 'Procedure', but if it can be confused with 'Process' then it is better to write it in full.

A wonderful comment on the misuse of abbreviations: "If you say 'plz' because it is shorter than 'please', then I will say 'No' because it is shorter than 'Yes'."

Do: Always use English to denote something, unless it is a typical product for a market in a particular country.

Don't: Handle naming conventions as Visual Studio expects them, so don't have functions start with a number, preferably don't use numeric values at all in a naming convention. It is also better not to use special characters, even if the IDE would accept them.

Do: Always sort the parts of a name logically, functions that start with Get or Set always first. The rest from important to less-important.

Do: Use *PascalCase* or *camelCase* for naming objects.

Object:	*Naming:*
Class of Structure	public class **NameOfClass**
Namespace	namespace **MyCompany.Functions**
Flag Enum	public enum **MyValues**
Enum	public enum **MyValue**
Method	public void **PrintToPaper()**
Public Field	public string **NameCompany**
Private Field	private string **m_NameCompany**
Variable	string **nameCompany**

In general, you use *PascalCase*, which means that all individual words start with a capital letter. Only variables within a function are written with *camelCase*, where the first letter is written small. Private Fields used in a Class are also written with a lowercase letter. It is recommended to use an "m_" as a prefix for this so that there is a difference from variables within a function.

Do: Provide the name of an Interface with a capital I as a prefix. Provide Generic Type Parameters with an uppercase T as a prefix.

Do: parameters are written with lowercase or *camelCase*.

```
public void GetCalculatedValue(double length, double width)
```

Don't: Don't use old-fashioned prefix naming to give variables a letter according to their type.

Wrong:

```
double dAngle = 0.0;
int iCounter = 1;
string sText = string.Empty;
```

This can be useful when naming Controls on a Form. This way you can quickly find all "Btn..." Controls. Commonly used are:

Object:	*Naming:*
Button	Btn...
CheckBox	Chk...
ComboBox	Cmb...
DataGrid	Dg...
Label	Lbl...
ListBox	Lb...
Panel	Pnl...
RadioButton	Rb...
TextBox	Txt...

CONSTANTS

Don't: Do not use constants as variables within code blocks. If you use constants, place them at the top of the Class.

You can, however, use read-only Fields. This allows you to create an object from a Class and then provide it read-only within your application.

STRINGS

Do: If you want to merge a lot of text, use a StringBuilder object. This works better than merging texts with the '+' operator. You can use the latter if you need to merge short texts.

Use the Format method to apply values to a string, rather than pasting individual texts together.

Good:

```
string message = string.Format("There are {0} items found.", items.ToString());
```

Wrong:

```
string message = "There are " + items.ToString() + " items found.";
```

Do: When comparing strings, always use a value for the comparison type.

Good:

```
if (message1.Equals(message2, StringComparison.OrdinalIgnoreCase))
```

Wrong:

```
if (message1.Equals(message2))
```

ARRAYS, LISTS AND COLLECTIONS

Do: Use Collections or Lists as often as possible. These types are very flexible and pleasant to work with, unlike Arrays.

Do: If you create your own Collection type, implement IEnumerable so you can also apply LINQ to your object.

Don't: Do not return *null* in your functions that normally return a List, Array or Collection. Rather, return an empty object.

STRUCTURES

Don't: Don't use Structs if a Class works better. You can use Structs for simple objects with a fleeting existence, without more extensive functionality. Rather, create a Class.

An AutoCAD Point2d or Point3d are Structures and contain two or three fields for the X, Y, Z values. These are logical Structures. If you create your own Point type with multiple Z values and all kinds of functions to do with it, use a Class.

CLASSES: FIELDS/PROPERTIES

Do: Make use of Properties instead of Public Fields. Properties can also be used to show Private Fields. These Properties can optionally be read-only while the Private Field is not.

Do: Make Properties read-only if users do not need to modify them. This prevents a lot of trouble.

Don't: Don't use set-only Properties. You can't do much with them. Instead, you can write a function that puts the value in a Private Field.

Do: Give all Properties a value in the Constructor.

CLASSES: CONSTRUCTORS

Do: Set only the Properties and add any parameter values to the Properties. Do nothing more than that in a Constructor.

Do: If you have a Constructor with parameters, also create a default Constructor in which you give all Properties a default value. In the Overload Constructor you then only need to process the parameters.

CLASSES: METHODS

Do: If you use Out Parameters, put them at the end.

Think carefully about how you set up your function. Do you want to do a warning in the function? Do you want to return an empty object on an error? Or do you return an Exception and handle it in the code that calls the function?

CLASSES: EVENTS

Remember that Events can be error prone. Always use *Try {} Catch {}* functionality and try to avoid causing new Events in Events. You can solve this, for example, by having a static bool that keeps track of whether you are executing a function. In the Event catching code you check if you execute a function or not.

CLASSES: MEMBER OVERLOADING

Overloading is better than functions with default parameters.

Don't: Do not change parameter naming in Overloaded functions. Be consequent and consistent.

CLASSES: STATIC CLASSES

Don't: Make as little use of Static Classes as possible. In Object Oriented Programming, it is not desirable. If you do use it, it is only for auxiliary functions.

NAMESPACES

Some books suggest setting up Namespaces as: CompanyName.ApplicationName.Functionality. This is not necessary, you may also make up your own system. But be clear.

EXCEPTIONS

Do: Use the most logical Exception. For example, an ArgumentNullException if a value is **null**.

Don't: Don't use Exceptions to exit a function just because it's easy. If an empty parameter is sent (i.e., not null), give a message, return false, or do something else, but don't fire an Exception.

Don't: Do not firing Exceptions from a Finally block.

RESOURCE CLEANUP: GARBAGE COLLECTOR

Don't: Don't force the cleanup of objects. There is no need to and it can interfere with the operation

of your application if the wrong objects are cleaned up. The computer is perfectly capable of determining for itself when an object needs to be cleaned up. And computers today have ample memory.

RESOURCE CLEANUP: TRY-CATCH-FINALLY

Use a Finally block to close, clean up or empty your objects. Do not do this in a Catch block. This one is not always executed, an Finally always is.

RESOURCE CLEANUP: DISPOSE

Clean up Disposable objects by calling the Dispose method. What's even better, use these objects in a using, then they are automatically cleaned up even if an Exception occurs.

Example:

```
using (Transaction tr = db.TransactionManager.StartTransaction())
{
  ...
}
```

The Transaction is now automatically cleaned up. At compile time, this structure is converted to a *Try {} Catch {} Finally {}* structure converted with a Dispose call in the Finally block.

INTEROP

Do: Send a Win32Exception if needed after a P/Invoke error.

4 AUTOCAD API

This section describes the conventions that apply specifically to the AutoCAD API.

ALIASES

Do: Always use aliases for usings to Autodesk Namespaces. Always use the same name for these.

```
using acAppServ = Autodesk.AutoCAD.ApplicationServices;
using acDbServ = Autodesk.AutoCAD.DatabaseServices;
```

Then in the code you can refer to:

```
acDbServ.Polyline poly = new acDbServ.Polyline();
```

Autodesk Classes can sometimes conflict with System Classes. This way you won't have this problem anymore. Also, you can now write code more easily for AutoCAD look-alikes like BricsCAD.

DOUBLES

Doubles are inherently error-prone. If you do a math calculation in your function, it may produce an incorrect answer:

86.5 - 86.4 = 0.0999999999999943

After about 12 decimal places, doubles start to deviate. So comparing coordinates will always result in an error. Therefore always use a rounded number in an equation. If necessary use a function to round numbers for comparison.

You can also compare the result if it is smaller than a small number. If you want to know if the difference is 0 (and therefore equal), you can also check if the difference is smaller than 0.0001 which may not be mathematically correct but is the desired result. The chance that you want to know something even more accurate is very small, moreover you'd be better off calculating with decimals instead of doubles.

DRAWING LOCK

Always lock the drawing when you call a function from a Palette. Otherwise, this produces an eLockViolation Exception.

```
using (acAppServ.DocumentLock acLckDoc = acAppServ.Application.DocumentManager.Mdi-
ActiveDocument.LockDocument())
{
  ...
}
```

TRANSACTIONS

Always commit a transaction, even if you only read an object. Commit() closes the transaction cleanly and quickly.

Always use a *using* to a Transaction. It is always cleaned up with a Dispose without having to write all kinds of constructs for it.

DATABASE

There are several Databases that can be referenced: ActiveDatabase and WorkingDatabase. These do not have to be the same. Normally you always work in the ActiveDatabase (MdiActive-Document) and usually this is also the WorkingDatabase.

Never use a *using* around a Database. The active drawing will not be cleaned up with a Dispose() even if it is called in a *using*, but to avoid problems with a Database suddenly disappearing from memory it is best not to do so.

Good:

```
Database db = Application.DocumentManager.MdiActiveDocument.Database;
using (Transaction tr = db.TransactionManager.StartTransaction())
{
  Point3d ptLocation = new Point3d();
  ...
```

Wrong:

```
using (Database db = Application.DocumentManager.MdiActiveDocument.Database)
{
  using (Transaction tr = db.TransactionManager.StartTransaction())
  {
    Point3d ptLocation = new Point3d();
    ...
```

5 AFTERWORD

The most important thing about guidelines is that you have a structure to program consistently and consistently, which improves clarity and organization and keeps things from being open to multiple interpretations.

It shouldn't limit you. Rather, it should help guide you through.

Hopefully, these examples will give you an idea of how you want to set up your own coding conventions.

Let me know what you do with it, and if you were able to apply it successfully or not!

And if you like my book ' Start programming in .NET for AutoCAD', I would really appreciate it if you leave a positive review at Amazon.

Thanks in advance!