

Coding Conventions

Version 1.0

BorgWorks Business Framework

Table of Contents

1. About this Document	1
2. General Principles.....	2
3. Formatting.....	2
4. C# Conventions	5
5. Naming Types and Members.....	5
6. Design Guidelines	5

1. About this Document

This document specifies how C# code for the BorgWorks framework should be written and formatted.
Target Audience

Everyone who commits code in the BorgWorks Subversion repository should read and know the coding conventions. Please try to conform to the conventions as much as possible. We know that some of the conventions are to a large degree arbitrary, but consistent coding within one project does make programmers much more comfortable reading and changing other people's code.

Current Level of Compliance

You may note that some existing code does not follow the guidelines. This is because some existing code was contributed by members of the BorgWorks foundation. We will try to increase compliance with conventions as we work on those files. Please try not to be irritated and follow the guidelines for new code, rather than the existing code that does not.

Further Reading

- ▶ Microsoft has its own (rather elaborate) design guidelines. Here is the link:
<http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp>
- ▶ Not written for C#, but still recommended:
Allen Vermeulen et.al., *The Elements of Java Style*, Cambridge University Press, 2000
- ▶ Since good programming practices always come from a good understanding, here is my favourite book for people who know programming but are not familiar with the special features of .NET:
Jeffrey Richter, *Applied Microsoft .NET Framework Programming*, Microsoft Press, 2002

2. General Principles

- ▶ **Look around.** Follow the style of existing code.
- ▶ **Do it right the first time.** Even if code is experimental to a degree, if you commit it, it should comply.

3. Files and Directories

- ▶ **Use one file for each class and interface.** The filename must be `ClassName.cs`. Enumerations and delegates that are clearly associated with one single class can use the same file as the class. Other groups of enumerations and delegates that somehow form a logical group can share a single file.
- ▶ **Use one directory for each namespace level.** The directory name must be the namespace name.

Example: Imagine an Assembly `BorgWorks.Module1`

Namespace	Directory (local to project root)
<code>BorgWorks.Module1</code>	<code>.\</code>
<code>BorgWorks.Module1.GuiStuff</code>	<code>GuiStuff\</code>
<code>BorgWorks.Module1.GuiStuff.Controls</code>	<code>GuiStuff\Controls\</code>

4. Formatting

- ▶ **Use Tabs.** If you use tabs, every user can use his or her favourite indentation width.
- ▶ **Limit each line to 120 characters.** Horizontal scrolling is not fun. Break lines that are too long.
- ▶ **Continue broken lines with two tabs.** If you have to continue in the next line, indent with two tabs. Try to group logical blocks in lines. Good places for line breaks are
 - ◆ Before an argument list
 - ◆ After a comma

Avoid:

```
throw new InvalidSomethingExpression (string.Format (message,
                                                    arg),
                                     e);
```

This is correct:

```
throw new InvalidSomethingExpression (
→ | → | string.Format (message, arg),
→ | → | e);
```

- ▶ **Curly braces share a line with the statement at their outer side.** This does not include class and interface definitions; they start with an opening brace in an extra line.

Example:

```
Class MyClass
{
  public void StartWorking (int hours) {
    if (hours < 0) {
      throw new ArgumentOutOfRangeException ("hours", hours);
    }
  }
}
```

- ▶ **Use curly braces even for single-line statements.**

Avoid this:

```
i f (condi ti on) doSome thi ng();

i f (condi ti on)
  doSome thi ng();

i f (condi ti on)
  {
    doSome thi ng();
  }
```

This is correct:

```
i f (condi ti on) {
  doSome thi ng();
}
```

- ▶ **Use a single line for trivial get and set accessors.** If all an accessor does is read or set a private variable, this saves some space and makes the code more readable if many properties are used.

```
public string Col or {
  get { return _col or; }
  set { _col or = val ue; }
}
```

- ▶ **Whitespace is free.** Whitespace does not cost anything but makes code so much more readable.

Use spaces to separate expressions.

Avoid this:

```
int i=3*4;
```

This is correct:

```
int i = 3 * 4;
```

Also, use blank lines to separate classes, methods, and blocks of statements that somehow belong together.

- ▶ **Use parenthesis for nested statements.** Yes, it is easy to see that $a + b * c$ is the same as $a + (b * c)$, so this is clearly a borderline case. If it gets any more complex, do not assume that everybody knows operator precedence rules by heart.
- ▶ **Parenthesis attach to their inside.** We use $f(x)$, not $f(x)$. Method calls with empty parenthesis are often written as $f()$, this is OK too.
- ▶ **return statements don't need parenthesis.** Do not write `return (expression);`
- ▶ **Do not use indentation to align columns of code.** This puts too much focus on code formatting. It also makes changes harder (and therefore less likely to happen). Refactoring tools destroy this kind of formatting anyway.

Avoid:

```
string f (int pos) {
  const string errorMessage = "There is an error at position {0}.";
  int numSomething = 0;
  StringBuilder sb = new StringBuilder (100);

  someFunctionCall (pos); // this is indented too much ...
  sb.AppendFormat (errorMessage, pos); // because of this comment here
  return sb.ToString(); // which does not fit anyway
  // ... so is this one
}
```

This is correct:

```
string f (int pos) {
    const string errorMessage = "There is an error at position {0}.";
    int numSomething = 0;
    StringBuilder sb = new StringBuilder (100);

    someFunctionCall (pos); // this is indented just as it happend
    // no problem with this comment here which does not fit anyway
    sb.AppendFormat (errorMessage, pos);
    return sb.ToString(); // ... so is this one
}
```

This does not look as nice as the first part, but is much more maintainable. With syntax highlighting, it's still comprehensible. Source code is just not a good place for ASCII art.

- ▶ **Do not use #regions.** Regions make it impossible to use certain features of Visual Studio, such as *Collapse to definitions (Ctrl-M, Q)*.
- ▶ **Indent documentation comments.** Documentation comments sometimes nest pretty deep, so indentation within these comments helps.

```
/// <summary>
/// This is the summary of what it does.
/// </summary>
public class MyClass
```

- ▶ **For documentation comments that are very large or used more than once, use include files.** Using the `<include>` element, refer to an external file that has all the XML comments for this class. Leave a short `<summary>` description with the code though.

Formatting Example

```
namespace MyNamespace {
    /// <summary>
    /// This class provides some methods.
    /// </summary>
    /// <include file=' /doc/MyClass.xml ' path=' MyClass/Class/' />
    class MyClass: IMyInterface
    {
        /// <summary>
        /// Returns an integer depending on input.
        /// </summary>
        /// <param name="intParam">
        /// Specifies the input of this method.
        /// </param>
        public int MyMethod (int intParam) {
            int firstVariable = intParam;
            string secondVariable = intParam.ToString();

            try {
                TrySomethingDangerous();
            } catch (Exception) {
            }

            if (intParam < 0) {
                return (intParam * 3) + 4;
            } else {
                return -intParam;
            }
        }
    }
}
```

5. C# Conventions

- ▶ Use `using` statements to import namespaces. Avoid fully qualified class names.
- ▶ Use C# alias names for declarations with basic types. Use `int` instead of `Int32` etc.

Avoid:

```
Int32 myInteger = Int32.Parse (myString);
return String.Format (...);
```

This is correct:

```
int myInteger = int.Parse (myString);
return string.Format (...);
```

- ▶ **But: Use CLR names for member names with basic types.** Some of these names are visible outside your source code. Use `Int32` instead of `int` etc.

Avoid:

```
public static MyClass FromInt (... ) {
```

This is correct:

```
public static MyClass FromInt32 (... ) {
```

- ▶ For attributes, do not write the `... Attribute` suffix.
- ▶ Declare private members with the `private` keyword.

6. Naming Types and Members

- ▶ Avoid abbreviations. Use names that make sense to the reader.
- ▶ Use singular nouns for types. Exception: For enumerations with the `[Flags]` attribute, use the plural form.
- ▶ Use combinations of verb and object for methods.

Case and Prefixes

Scope and Visibility	Kind	Coding Convention
public, protected or internal	all	Pascal Case
All	type (including enumerations and delegates), method or property	Pascal Case
private	Field	<u>_</u> camel Case
private	Constant	<u>c_</u> camel Case
private static	Field	<u>s_</u> camel Case
	local variable	camel Case

Special Constructs

Kind	Coding Convention	Example
Interface	I prefix	I MyInterface
Attribute	Attribute suffix	SpecialMarker Attribute
Exception	Exception suffix	MyEvent Exception
Event arguments	EventArgs suffix	Notification EventArgs
Event handler delegate	EventHandler suffix	Notification EventHandler

Kind	Coding Convention	Example
GUI control fields	suffix that indicates type	Cancel Button Fi rstName TextBox Gender DropDown

7. Design Guidelines

- ▶ **Keep classes and interfaces small.** Although sometimes large classes are inevitable, they are often a sign of overloading. Try to come
- ▶ **Do not use nested public types.** Rather, use appropriately named top-level types in the same namespace. Remember that enumerations and delegates can be in the same file as the class they belong to.
- ▶ **All fields are private or readonly.** Exception: Classes and structs without any behaviour may use non-private fields.
- ▶ **Non-private constructors always create valid objects.**
- ▶ **Do not call virtual methods from constructors.** This might result in the execution of overridden versions of those methods that depend on the constructors of the classes that implement them.
- ▶ **Nest constructor calls instead of writing redundant initialization code.**

Avoid:

```
public class Account
{
    public Account (string name) {
        _name = name;
        _balance = 0;
    }

    public Account (string name, float balance) {
        _name = name;
        _balance = balance;
    }
}
```

This is correct:

```
public class Account
{
    public Account (string name)
        : this (name, 0) {
    }

    public Account (string name, float balance) {
        _name = name;
        _balance = balance;
    }
}
```

- ▶ **Remember to free resources.** Use the `using (...)` statement for resources that support the `IDisposable` interface. Remember that freeing resources is essential in many cases, so make sure that it occurs in every situation.
- ▶ **Remember to make exception classes serializable.** Mark Exceptions with the `[Serializable]` attribute. If your exception class contains user data, implement the `ISerializable` interface and the special constructor this interface requires.