C# Programming Style Guide

 CJ v0.3 5/2/2011 v0.4 6/24/2011 v0.5 11/ 2013 v0.6 1/15/2014 v1 5/30/2016

This is about C# programming on the host; not about the application to be protected.
Not all phrases were invented by us.
If you want to use our style guide for your own company, feel free to do so.

Table of Contents

Prime directives

*This is a **guide**. We expect our engineers to be top professionals and don't want any rules which make their life harder. Exceptions to these rules are occasionally justified.*

When the code and the comments disagree, probably both are wrong.

This is a living document; as we learn more, we will consider what we learned and improve the rules.

Different projects might have different guidelines, but it is considered granted that whatever some projects rules say is motivated by making programs more readable, while not looking too unusual.

Terminology

The words Camel-casing and Pascal-casing are used in most C# style guides, so do we. (Nevermind that Pascal does NOT define what is now called Pascal casing.)

*Unusual White Hawk Software standards are marked with **; just to underline that such deviation may be on purpose and not accidental.*

Consistency

When adding or modifying code, adhere to the style of the original. When adding less then a page

to an existing file, the addition should use consistent style with the previous content of the file.

When adding code, the code added must be as well (or better) documented then the existing code.

New project members read and copy the code style used by the more senior programmers.

If you are only an occasional bug-fixer for a module, additionally comment your changes with your id, date or bug number, at least whenever the fix is not obvious. A primary maintainer of that particular module will remove comments which only serve as change-markers, but leave comments serving to understand the code..

Corner cases

Use simple C#. Use of unusual cases or corner cases needs to be documented.
It is sometimes person-dependent what is a regular case and what is a corner case. Sadly, that is unavoidable; err towards more documentation.

Try to avoid using language features with known differences between Mono and VisualStudio.

*There used to be an argument that programmers **must** know their tool. That statement sounds very good, but it is not completely true anymore. Languages do change; new languages are invented. A programmer might use multiple languages and should be knowledgeable about computer science and software architecture. Detail-knowledge of any one particular language may be of much lesser importance now.*

White Space

Be generous with white space between functionalities.

Try to use little white space within features, so more text fits on a screen.
The number of empty lines before a function must not be less then the number of empty lines within a function.

The following rules are arbitrary and serve only to make code more consistent. When adhered to systematically, they give an extra clue for the reader.

Never use space between a function name and the left parenthesis for arguments.
Use a space between keywords and a left parenthesis.
(This makes it easier for humans to parse programs and grasp the difference.)
Use a space after comma; no space before comma.
Use a space after semicolon; no space before semicolon.

Use spaces only; avoid tabs.
This is a coin-flip rule, but the flipping of that coin in our company has already been done. It implies that elaborate rules for tabbing can be reserved for other purposes.

Comments

Do not comment the obvious.
Do comment every type, namespace or function, unless obvious.
Keep comments short (but make plenty of them). If a comment is too long to read, it may be useless!
Make code as self-documenting as reasonably possible.

Destructurs and constructors benefit from comments. *(The "syntax" clues are too weak for programmers not grown up using C++)*

Extension-methods must be commented. *(We consider the syntax alone not obvious enough)***

In comments use "this" rather than "the" when refering to instances of the current class. Avoid "this" for refering to non-instances..

Comment the file name., as files may be copied to strange places.
Comment the file-purpose.

See the chapter on "Brackets" for more.

Identifiers, naming

Careful choice of names is important. *(Not only in programing-languages.)*

Choose identifiers which make sense to their users:

> *For an interface, the client programmer is the user, not the the class implementor nor the protection engineer.*

It is good to use terminology of the customers domain. However only programmers read the program: for this reason customer standards do **NOT** override programming standards.**

Avoid overly long identifiers. *But accept names to be a tiny little bit clumsy for the benefit of somebody not knowing all conventions.*

Frequently two word identifiers make most sense. (One word names frequently suffer from disagreement of which things is more basic.)

Typically use verbs for functions.

"Hungarian" notation

Hungarian notation is generally considered a bad idea, but rarely understood.
(Real "hungarian" notation in Microsoft uses prefix hints to mimick the **type** used for a variable.)
We DO NOT use the actual TYPE. However a prefix or postfix describing a logical category can be very useful.**
(Types can change easily as an implementation matter; logical categories rarely change.)

Yes, it may read mildly clumsy in English, but it can be very useful in programming.

For interfaces we follow the usual C# convention to start names with an extra "I" character.

Capitalization

Use camel-caps for values.
Use Pascal-caps for types.

Variables, function names** and constants** start with a small case letter (*they are values*).
Capital case first letters usually denote types (or classes).

Constants and enumeration values** start with a small case letter. (This may be unusual, but it makes little sense to make a big difference between constants and variables. Separating values and types is much more useful).

Namespaces are not consistent. If they look like a static class use Pascal casing. If they look like a C++ namespace use camel casing. **

Do not use case alone to differentiate names, EXCEPT when one is a type and the other is a value/variable of this type. ** (Delegates are types!)

Surprize experience: *We were worried about uglyness and inconsistency when using White Hawk-defined names and elsewhere-defined names in close proximity. However, as it turned out, this 'inconsistency' has given an implicit clue 'who' has defined a function, and adds valuable information to the program text.*

Brackets "{", "}"

Consistency is most important. We prefer the opening bracket on the same line with the "if","while",... and the closing bracket on a new line, aligned with the "if", "while"...

This removes noise from the left side of the code (where normal readers look first). It also saves a line. *The reason to be greedy with lines is to fit more code on a monitor.*

In contrast, functions and classes usually have the opening bracket on a new line. *This gives another optical hint for reading the code; readers may be used to this from old C code.*

For functions we like adding a comment of the function name at the closing bracket..**
This gives valuable information when a function beginning is off-screen and invisible.
This is painful for refactoring, but the above advantage is more important then this disadvantage.

For classes and namespaces we like to comment the word "class" resp. "namespace" in addition to the name.**

Do not ommit brackets even when a statement is simple enough that the brackets are optional. The

brackets may help a human reader to parse the code, and prevent future errors should the simple statement be modified later.

Comma separated list

Avoid comma separated list for declaring multiple fields of the same type. (*These are hard to read as soon as any specifiers or unusual names occur.*)

Usually declare each member instead.
Exception: list of value type with no explicite initializations.

Properties, function or fields.

Use the mechanism best suited for each declaration. The forbiding of public class variables as mentioned in many other standards is irrelevant.** Changing a field to using accessors is an easy refactoring.

"Using" directives

Do not use using directives to open too many name spaces.
Use the aliasing mechanism to create short prefixes.**
Do not confuse using directives with using statements.

Inheritance

Use inheritance for "IS A" relationship.
Use containment for "has a" relationship.
In doubt whether to use inheritance, use containment.

Avoid hiding inherited members.**

Avoid multiple inheritance, but allow mixins.

The C# language makes a big deal of not supporting multiple-inheritance, however it is a white lie: Only classes are restricted to single-inheritance; interfaces support multiple-inheritance.

Subclassing

Use the Liskov substitution principle. If you don't know what this is, look it up. It is important.

Overloading

In doubt better avoid.
(Among others reasons, text-replace is easier when identifiers are different.)

(Refactoring IDEs are great, but forcing other users to always use such an IDE is not great.)
When used sparingly, and only where obvious, overloading can occasionally be useful however.

Using "this" references

Unless naming makes it obvious that a name is local to a class, prefix it. (The actual code "this." can serve as a good prefix.) Another good prefix frequently used is just "its" ..
When readability is enhanced by class-specific prefixes, such prefixes can make sense.
This may be neglected in very short functions. It is acceptable to use different (multiple) prefixes if that makes code more readable.

Interface-method implementation

Interface method implementation (unless obvious) must be documented as such, or use an "explicit implementation".**

Avoid magic numbers

Unless it is obvious, declare a constant.
(3.1415 may be obvious, but its precision may matter despite not being obvious.)

Special cast-like code.

Unboxing must be commented, as this is not a pointer-cast which looks soo similar.**

Explicit nullable conversions must be commented. (Likewise, this is not a pointer cast which looks soo similar.)**

Extension methods

Need extra documentation, as the syntax is somewhat weak. ** (See comments)

Exceptions

Program as if the throwing of an exception would be extremely slow; use exceptions for exceptional cases only.

Thread safety

As we mostly write programming-tools, thread-safety is not assumed **in this environment,** and need not be documented. Library routines are different, and presence or absence of thread-safety should be documented.

When you have to implement thread safety, consider the method of locking to be class-specific and

document it. Avoid implied locks a la Java (because this is a design which leads to gratuidious locking errors).

LINQ

We prefer standard SQL. **
This is of course an application-specific preference only. We think that the visual difference between C# and SQL aids the comprehension; the semantic benefits LINQ are rarely useful.

Garbage collection

It is ok but rarely required to help the garbage collector to break references (See next paragraph). Keep finalization methods to where absolute necessary, as this causes an extra step during garbage collection.
The C# "dispose pattern" is overly complex; C++ destructors do not belong into C#. We recommend using your own dispose mechanism which should be easier to use. Consider making your mechanism idempotent.

Attributes

Consider attributes to be heavy weight; please think only about readability and safety. Unless the attribute has a real benefit, don't bother using it; attributes make code harder to read. **

(I have seen attribute-use as good means for internal debugging however.)

Properties

Use ONLY if use is required, or has a significant benefit. **
The set of available properties is somewhat confusing for application programmers.

Lambda

Use lambdas where they make code more readable; avoid lambdas where they make code less readable. *More versus less readable is subjective! the target reader for this decision case is a competent programer not using lambdas frequently.*

Templates

Templates most often are good. However just to avoid an occasional typecast is too weak a reason to use a template.

XML Documentation Tags

XML Documentation-tags can make comments unreadable and should be avoided. **
Code must in first order be readable from the real source code; funny documentation thingies for representations other then source-code are unnecessary, and mostly counter-productive.
Similarly, the "///" becomes useless.**

While programming, programmers read the source code, not some documentation stored somewhere else. Non-programmers cannot do anything reasonable with program source code anyway.

Deficiencies

Comment known defects and deficiencies in the program.
Use a marker like "TODO" in the comment. That allows a quick search of what still needs to be done.

Defensive programming

Use assertions freely.
Experience shows that the amount of complexity a programmer can handle is smaller then what that programmer expects.

Performance

Think in Big-O notation. O(n*log(n)) is to be expected for our specific application. O(n2) is absolutely prohibitive, at least for the most frequently occurring features in our application.

Small gains should be ignored if readability is affected.

Early optimization sometimes can be evil. However, early pessimization is even worse.

Performance of the protector can be important, but the performance of the protected application is much more important. (In protected code, some performance costs due to loss in proximity is almost unavoidable and must be accepted.)

Character set

For identifiers in the program and for file-names use Ascii characters.
For comments there is no restriction.

Reuse code versus reprogramming code

Do not hesitate to use other peoples code. Always make a local backup of reused source code, so it does not suddenly disappear. Always respect copyrights and license conditions.

Do not hesitate to rewrite existing external code. A good decision-process is:

-If reuse saves time, use that external code.

-If for you, understanding other peoples code is more work then rewriting the code, AND for other White Hawk engineers reading your code is less work then reading the external code as well, do what is necessary.**

Application-specific code versus library routine

Should you write a generic library class or an application specific class?
The decision is not as easy as it looks. If you write an application-specific class you can always rewrite the class in a generic way later if you still feels so. It might be an easier judgement later. If you write a class in a reusable way, don't do it unless you have the time to do it right. However, if a library routine is properly designed and debugged, its use for shortening the application is beneficial.

UnExample: Parsing command-line arguments. There exists a myriad of pre-existing classes to parse command-line arguments. To do it right, powerful and flexible is every library routine's goal. For the NestX86 application however, we replaced the use of a generic library by simple, straight forward code in the application specific command class. This is simpler to debug and simpler to maintain.

Special project-specific considerations

The protection program will be tamper-proofed itself before customers get the binary. While not specifying how we tamper proofe the code, lets say that certain considerations must be taken to ease the tamper-proofing, including special functions to hide data and string literals. While it would be nice to use "eat own dog food", we protect machine code and not yet C# code. See http://www.whitehawksoftware.com/eat-your-own-dog-food/

In particular, we have some tools which modify the source code. Write your code simple enough that this pre-processing step stays easy to do, and give up on some code shorteness otherwise possible.

Machine-generated code

Some of our code is tool-generated and not hand-written. Any tool generating code must produce code which itself should mostly adhere to this coding standard.
Exception: The number of lines for machine generated files, classes, or functions is NOT limited.

Generated code must be labelled as such, to prevent manuall editing which will be deleted when such code is re-generated.

Refactoring

Is highly encouraged, however please coordinate your refactoring with the team. Avoid refactoring when team-members have lots of code checked out. You refactoring is great. You creating extra work for your team-members to update their checked-out files is NOT so great.

Write your code in a way which makes future refactoring "easy". For example, try to avoid reusing identifiers.

Agile

Agile is good. However "agile" must not be used to avoid thinking nor designing.

Most agile methods are good, but at our company you will NOT get extra credit for completeness and using all methods necessary to match a given methodology, nor for following "pure" definition of agility. Please use only what is useful and don't bother about stuff which does not make sense to you.

Agile is never a goal but a method.

Object orientedness

Object oriented is good. However object orientedness is a method, not a goal nor a religion.

Programmer Ego

You don't own the code. But in our company, as primary maintainer you have a right to be heard. When you create something really neat, there should be some space for your name.