Starchy Games Coding Style Guidelines

Adapted from Coding Guidelines by Dennis Doomen: <https://csharpcodingguidelines.com/>

Last revised 9/5/19 by Graeme Holliday

**Introduction & Guiding Principles**

This document attempts to provide coding standards for all versions of C# that are both valuable and pragmatic. Notice that not all guidelines have a clear rationale. Sometimes, a guideline exists simply to provide uniformity.

When you have an issue that this document does not cover, refer to these basic guiding principles:

* The Principle of Least Surprise (or Astonishment): you should choose a solution that everyone can understand, and that keeps them on the right track.
* Keep It Simple Stupid (a.k.a. KISS): the simplest solution is more than sufficient.
* You Ain't Gonna Need It (a.k.a. YAGNI): create a solution for the problem at hand, not for the ones you think may happen later. Can you predict the future?
* Don't Repeat Yourself (a.k.a. DRY): avoid duplication within a component, a source control repository or a bounded context. Use the Rule of Three heuristic: When a segment of code appears three or more times, you should be looking to make that code modular.
* The Four Principles of Object-oriented Programming: Encapsulation, abstraction, inheritance and polymorphism.

**Class Design Guidelines**

**A class or interface should have a single purpose**

A class or interface should have a single purpose within the system it functions in. In general, a class either represents a

primitive type like an email or ISBN number, an abstraction of some business concept, a plain data structure, or is responsible

for orchestrating the interaction between other classes. It is never a combination of those.

**It should be possible to treat a derived object as if it were a base class object**

In other words, you should be able to use a reference to an object of a derived class wherever a reference to its base class

object is used without knowing the specific derived class. A very notorious example of a violation of this rule is throwing a

NotImplementedException when overriding some of the base-class methods. A less subtle example is not honoring the behavior

expected by the base class.

**Don't refer to derived classes from the base class**

Having dependencies from a base class to its sub-classes goes against proper object-oriented design and might prevent other

developers from adding new derived classes.

**Avoid bidirectional dependencies**

This means that two classes know about each other's public members or rely on each other's internal behavior. Refactoring or

replacing one of those classes requires changes on both parties and may involve a lot of unexpected work. The most obvious

way of breaking that dependency is to introduce an interface for one of the classes and using Dependency Injection.

**Classes should have state and behavior**

Use the principles of object-orientation explained above and move program logic close to the data it applies to.

**Classes should protect the consistency of their internal state**

Validate incoming arguments from public members. For example:

public void SetAge(int years) {

AssertValueIsInRange(years, 0, 200);

this.age = years;

}

**Member Design Guidelines**

**A property, method or local function should do only one thing**

A method body should have a single responsibility.

**Properties, arguments and return values representing strings or collections should never be** null

Returning null can be unexpected by the caller. Always return an empty collection or an empty string instead of a null

reference. This prevents cluttering your code base with additional checks for null.

**Define parameters as specific as possible**

If your method or local function needs a specific piece of data, define parameters as specific as that and don't take a container

object instead.

**Miscellaneous Design Guidelines**

**Throw exceptions rather than returning a status value**

A code base that uses return values to report success or failure tends to have nested if-statements sprinkled all over the code.

Quite often, a caller forgets to check the return value anyway. Structured exception handling has been introduced to allow you

to throw exceptions and catch or replace them at a higher layer. In most systems it is quite common to throw exceptions

whenever an unexpected situation occurs.

**Provide a rich and meaningful exception message text**

The message should explain the cause of the exception, and clearly describe what needs to be done to avoid the exception.

**Throw the most specific exception that is appropriate**

For example, if a method receives a null argument, it should throw ArgumentNullException instead of its base type

ArgumentException.

**Maintainability Guidelines**

**Make all members** private **by default**

To make a more conscious decision on which members to make available to other classes, first restrict the scope as much as

possible. Then carefully decide what to expose as a public member or type.

**Name a source file to the type it contains**

Use Pascal casing to name the file and don't use underscores. Don't include (the number of) generic type parameters in the file

name.

**Use** using **statements instead of fully qualified type names**

Limit usage of fully qualified type names to prevent name clashing. For example, don't do this:

var list = new System.Collections.Generic.List<string>();

Instead, do this:

using System.Collections.Generic;

var list = new List<string>();

If you do need to prevent name clashing, use a using directive to assign an alias:

using Label = System.Web.UI.WebControls.Label;

**Only use** var **when the type is obvious**

Only use var if the type is obvious from the same statement and using it would improve readability, like here:

var repository = new RepositoryFactory.Get();

var list = new ReadOnlyCollection();

**Don't make explicit comparisons to** true **or** false

It is usually bad style to compare a bool-type expression to true or false. For example:

while(condition == false) // wrong; bad style

while(condition) // OK

**Don't change a loop variable inside a** for **loop**

Updating the loop variable within the loop body is generally considered confusing, even more so if the loop variable is modified

in more than one place.

**Don’t add a block after the keywords** if**,** else**,** do**,** while**,** for**, and** foreach **unless needed.**

For example:

if(isActive)

Foo();

Make sure to indent the next line to avoid possible confusion.

**Use the ternary operator instead of an** if **/** else **statement for assignment.**

Express your intentions directly. For example:

int minimum = value < 0 ? 0 : value;

**Don't use parameters as temporary variables**

Never use a parameter as a convenient variable for storing temporary state. Even though the type of your temporary variable

may be the same, the name usually does not reflect the purpose of the temporary variable.

**Naming Guidelines**

**Use US English**

All identifiers (such as types, type members, parameters and variables) should be named using words from the American

English language. Choose easily readable, preferably grammatically correct names. Favor readability over brevity. The property name canScrollHorizontally is better than scrollableX (an obscure reference to the X-axis).

Avoid using names that conflict with keywords of widely used programming languages.

**Use proper casing for language elements**

Use camelcase for all variables and parameters with the first letter lowercase: anExample

Use camelcase for all classes, methods, and types with the first letter uppercase: AnExample

Use all uppercase for constants: ANEXAMPLE

**Properly name properties**

Name properties with nouns, noun phrases, or occasionally adjective phrases.

Name boolean properties with an affirmative phrase. E.g. canSeek instead of cannotSeek.

Consider prefixing boolean properties with is, has, can, allows, or supports.

**Documentation Guidelines**

**All methods must be documented using only multi-line comments preceding the method declaration**

All method comments must follow this form:

/\*

MethodName

Parameters: purpose

Returns: values

Throws: exceptions

Purpose: description

\*/

void DoNothing() {}

Where the comment itself is indented and the opening and closing are on separate lines.

The parameter line should be omitted if there are none.

The return line should be omitted if the function is void.

The throws line should be omitted if the function doesn’t explicitly throw an exception.

**All classes and interfaces must be documented using a multi-line comment directly preceding the class or interface declaration**

All classes and interfaces must follow this form:

using System;

/\*

ClassOrInterfaceName

Author: name

Purpose: description

\*/

class ClassName {}

**All other documentation must be single-line comments immediately preceding the block or line of code it concerns**

All other comments must follow this form:

//This block does something complex

for() {

...

}

**Layout Guidelines**

**Use a common layout**

Use an indentation of tabs, not spaces

Remove space between keywords like if and the expression and don't add spaces after ( and before ). For example:

if(condition == null).

Add a space around operators like +, -, ==, etc.

Add a space after ) but before an opening curly brace {

Always put closing curly braces on a new line nd keep opening curly braces on the line where the block starts.

Indent all code within a block using a tab.

**Place members in a well-defined order**

Maintaining a common order allows other team members to find their way in your code more easily. In general, a source file

should be readable from top to bottom, as if reading a book, to prevent readers from having to browse up and down through

the code file. Use this order:

1. Private fields and constants
2. Public fields and constants
3. Constructors and the finalizer
4. Private methods
5. Public methods