Hue Learn

Wise Work

DNAi World

C# Coding Guidelines

Best Practices to follow

by Hue Learn, Wise Work, DNAi World

Contents

1. Introduction

This document describes rules and recommendations for developing applications and class libraries using the C Sharp Language. The goal is to define guidelines to enforce consistent style and formatting and help developers avoid common pitfalls and mistakes. Coding conventions are essential for maintaining code readability, consistency, and collaboration within a development team.

2. Naming Conventions

Legend:

- "c" = camel Case
- "P" = Pascal Case
- "_" = Prefix with _Underscore
- "x" = Not Applicable

Identifier	Public	Protected	Internal	Private	Example
Project File	P	x	x	x	MyProject.csproj
Source File	P	x	x	x	UserService.cs
Other Files	P	x	x	x	Helpers.cs
Name space	P	x	x	x	MyProject.Services
Class or Structure	P	P	P	P	UserManager
Interface	P	P	P	P	IUserRepository
Generic Class	P	P	P	P	List<T>, Dictionary<TKey, TValue>
Method	P	P	P	P	SaveUser(), GetData()
Property	P	P	P	P	FirstName
Field	P	P	P	_c	_userId, _connectionString
Constant	P	P	P	_c	const int MaxRetries = 5;
Static Field	P	P	P	_c	private static string _configPath;
Enum	P	P	P	P	enum LogLevel { Info, Warning, Error }
Delegate	P	P	P	P	delegate void LogHandler(string message);
Event	P	P	P	P	public event EventHandler DataLoaded;
Inline Variable	x	x	x	c	int counter = 0;
Parameter	P	x	x	c	void SaveUser(string userName)

3. Coding Style

1. Do use Pascal Casing for class names and method names.

Right Coding Style.

```
Public class ClientActivity
{
  Public void ClearStatistics()
  {
  }
}
```

2. Do use Camel Casing for local variables and method arguments.

Right Coding Style.

```
int itemCount = logEvent.Items.Count
```

3. Do not use Hungarian notation or any other type identification in identifier.

Wrong Coding Style.

```
int iCounter;
string strName;
```

Right Coding Style.

```
int counter;
string name;
```

4. Do not use Screaming Caps for constants or read only variables.

Wrong Coding Style.

```
public static const string SHIPPINGTYPE = "DropShip"
```

Right Coding Style.

```
public static const string ShippingType = "DropShip";
```

5. Avoid using Abbreviations variables

Wrong Coding Style.

```
UserGroup usrFrp;
Assignment empAssig;
```

Right Coding Style.

```
UserGroup userGroup;
Assignment employeeAssignment;
```

6. Do not use Underscores in identifiers.

Exception: you can prefix private static variables with an underscore.

Wrong Coding Style.

```
public DateTime client_Appointment;
public Timespan time_Left;
```

Right Coding Style.

```

public DateTime clientAppointment;
public Timespan timeLeft;
```

Right Coding Style.

```
private DateTime _registrationDate;
```

7. Do use predefined type names instead of system type names like Int16, Single, UInt64, etc

Right Coding Style.

```
string firstName;
int lastIndex;
bool isSaved;
```

Wrong Coding Style.

```
string firstName;
int32 lastIndex;
bool isSaved;
```

8. Do use noun or noun phrases to name a class.

Right Coding Style.

```
public class Employee
public class BussinessLocation
public class DocumentCollection
```

9. Do prefix interfaces with the letter I. Interface names are noun (phrases) or adjectives.

Right Coding Style.

```
public interface IShape  {}
public interface IShapeCollection { }
public interface IGroupable { }
```

10. Do name the source files according to their main classes.

Exception: file names with partial classes.

Right Coding Style.

```
//Located in Task.generated.cs
public partial class Task
{

}
```

11. Do vertically align curly brackets.

Right Coding Style.

```
class program
{
    static void Main(string[] args)
    {

    }
}
```

12. Do declare all member variables at the top of a class, with static variables at the very top.

Right Coding Style.

```
public class Account
{
  public static string BankName;
  public static decimal Reserves;
  public static Number {get;set;}
  public DateTime DateOpened {get;set;}
}
```

13. Do use singular names for enums.

Right Coding Style.

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

14. Do not explicitly specify a type of an enum or values of enums.

Wrong Coding Style.

```
public enum Direction
{
  North = 1,
  East = 2,
  South = 3,
  West = 4
}
```

Right Coding Style.

```
public enum Direction
{
  North,
  East,
  South,
  West
}
```

15. Do not suffix enum names with Enum

Wrong Coding Style.

```
public enum CoinEnum
{
Penny,
Nickel,
}
```

16. Do not use names that begin with a numeric character.
17. Do not use C Sharp reserved words as names.
18. Try to prefix Boolean variables and properties with “Can”, “Is” or “Has

3.1. Formatting

1. Never declare more than 1 namespace per file.
2. Avoid putting multiple classes in a single file.
3. Always place curly braces ({ and }) on a new line.
4. Always use curly braces ({ and }) in conditional statements.
5. Always use a Tab & Indention size of 4.
6. Declare each variable independently – not in the same statement.
7. Place namespace “using” statements together at the top of file. Group .NET namespaces above custom namespaces.
8. Group internal class implementation by type in the following order:
   - Member variables.
   - Constructors & Finalizers.
   - Nested Enums, Structs, and Classes.
   - Properties
   - Methods
9. Sequence declarations within type groups based upon access modifier and visibility.
   - Public
   - Protected
   - Internal
   - Private
10. Segregate interface Implementation by using region statements.
11. Append folder-name to namespace for source files within sub-folders.
12. Recursively indent all code blocks contained within braces.
13. Use white space (CR/LF, Tabs, etc) liberally to separate and organize code.

4. Variables & Types

1. Try to initialize variables where you declare them.
2. Always choose the simplest data type, list, or object required.
3. Always use the built-in C Sharp data type aliases, not the .NET common type system (CTS).

Right Coding Style.

```
short NOT System.Int16
int NOT System.Int32
long NOT System.Int64
string NOT System.String
```

1. Only declare member variables as private. Use properties to provide access to them with public, protected, or internal access modifiers.
2. Try to use int for any non-fractional numeric values that will fit the int datatype - even variables for non negative numbers.
3. Only use long for variables potentially containing values too large for an int.
4. Try to use double for fractional numbers to ensure decimal precision in calculations.
5. Only use float for fractional numbers that will not fit double or decimal.
6. Avoid using float unless you fully understand the implications upon any calculations.
7. Try to use decimal when fractional numbers must be rounded to a fixed precision for calculations. Typically this will involve money.
8. Avoid using sbyte, short, uint, and ulong unless it is for interop (P/Invoke) with native libraries.

5. Language guidelines

5.1. Catch exceptions

Only catch exceptions that can be properly handled; **avoid catching generic exceptions.**

1. Use LINQ queries and methods for collection manipulation to improve code readability.
2. Use the language keywords for data types instead of the runtime types. For example:

- use string instead of System.String
- int instead of System.Int32

5.2. String data

1. To append strings in loops, especially when you're working with large amounts of text, use a System.Text.StringBuilder object.
2. Use string interpolation to concatenate short strings, as shown in the following code.

Right Coding Style.

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

5.3. Delegates

1. Use Func<> and Action<> instead of defining delegate types.

2. Create an instance of the delegate type and call it.

Right Coding Style.

```
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

1. try-catch and using statements in exception handling
2. Use a try-catch statement for most exception handling.
3. Simplify your code by using the C Sharp using statement. If you have a try-finally statement in which the only code in the finally block is a call to the Dispose method, use a using statement instead.

5.4. && and || operators

1. Use && instead of & and || instead of | when you perform comparisons new operator
2. Use object initializers to simplify object creation, as shown in the following example.

Right Coding Style.

```
var thirdExample = new ExampleClass { Name= "Desktop", ID = 37414, Location =
"Redmond", Age = 2.3 };
```

5.5. LINQ queries

1. Use meaningful names for query variables.
2. Align query clauses under the from clause, as shown in the previous examples.
3. Use where clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

Right Coding Style.

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

5.6. Using Directive

Place the using directives outside the namespace declaration

Right Coding Style.

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
}
```

6. Style guidelines

Use the following format for code samples:

- Use tabs for indentation.
- Align code consistently to improve readability.
- Limit lines to 65 characters to enhance code readability on docs, especially on mobile screens.
- Break long statements into multiple lines to improve clarity.
- Use the "Allman" style for braces: open and closing brace its own new line. Braces line up with current indentation level.
- Line breaks should occur before binary operators, if necessary

7. Comment style

- Use single-line comments ("//") for brief explanations.
- Avoid multi-line comments ("/* */") for longer explanations. Comments aren't localized. Instead, longer explanations are in the companion article.
- For describing methods, classes, fields, and all public members use XML comments.
- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter ("//") and the comment text.

Right Coding Style.

```
// The following declaration creates a query.
// It does not run the query.
```

8. Layout conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read.

- Use the default Code Editor settings
- Smart indenting,
- Four-character indents,
- Tabs saved as spaces
- Write only one statement per line.
- Write only one declaration per line.

- If continuation lines aren't indented automatically, indent them one tab stop (four spaces).
- Add at least one blank line between method definitions and property definitions.
- Use parentheses to make clauses in an expression apparent, as shown in the following code.

9. Securing resource access

When designing and writing your code, you need to protect and limit the access that code has to resources, especially when using or invoking code of unknown origin. So, keep in mind the following techniques to ensure your code is secure:

- Do not use Code Access Security (CAS).
- Do not use partial trusted code.
- Do not use the AllowPartiallyTrustedCaller attribute (APTCA).
- Do not use .NET Remoting.
- Do not use Distributed Component Object Model (DCOM).
- Do not use binary formatters.

10. Formatters

10.1. EditorConfig

A powerful tool built into **Visual Studio that helps standardize code formatting across teams**. You can define rules for indentation, spacing, and more in a **.editorconfig** file.

10.2. dotnet-format

: A .NET tool that applies consistent code formatting based on your project's **.editorconfig** settings. **It can be run from the command line.**

11. Linter

11.1. StyleCop Analyzers

A code analysis tool that **checks your C sharp code for style and consistency** based on StyleCop rules. It can be **integrated into Visual Studio and Visual Studio Code.**

11.2. SonarLint

A powerful linting tool that provides **in-depth static code analysis to catch potential bugs,** code smells, and security vulnerabilities in C Sharp. It supports **integration with Visual Studio.**

12. Type Checker

12.1. Roslyn Analyzer

A powerful tool built into the .NET ecosystem, specifically targeting C Sharp and Visual Basic code. It is part of the .NET Compiler Platform and provides a way to perform static code analysis, enforce coding standards, detect potential issues, and improve overall code quality. Unlike traditional linters, **Roslyn Analyzers are integrated directly into the com-**

pilation process, giving them deep access to code syntax, semantics, and project metadata.

Key Features of Roslyn Analyzer:

- Static Code Analysis:

Roslyn Analyzers analyze your code at compile-time to catch potential issues before the code is run. They provide warnings, errors, and suggestions in the IDE (e.g., Visual Studio, Visual Studio Code).

- Custom Rules:

You can create your own custom analyzers to enforce specific coding guidelines or company standards, beyond what is provided by default. For example, you can enforce naming conventions, code patterns, or deprecations.

- Code Fixes:

Analyzers can suggest automatic code fixes. For example, if an analyzer detects that a naming convention isn't followed, it can suggest a fix and allow you to apply it with a single click in the IDE.

- Real-time Feedback:

As you write code, Roslyn Analyzers provide real-time feedback in the form of squiggly lines, error messages, or warnings, helping you catch issues early.

- Integration with Build Pipelines:

Roslyn Analyzers **can be integrated into build pipelines (like CI/CD) to enforce coding rules across teams. They can fail a build if critical violations are detected.**

- Deep Language Understanding:

Since Roslyn is the compiler platform for .NET, the analyzers have full access to the code's syntax tree, semantic model, and project metadata. This allows them to perform sophisticated analysis and provide more meaningful diagnostics compared to general-purpose linters.

13. Conclusion

Adhering to these coding guidelines is vital for producing clean, maintainable, and efficient code. By following conventions such as naming standards, proper code structure, and adhering to best practices, teams can ensure that the code is both readable and consistent across projects and create software that is easier to maintain and scale over time.