Best Practices for Writing Code

Robert Brenckman

Charter Oak State College

ITE 215: Software Develop Method & Languages

COSC_ITE215012526FACR

John Rusnak

11/1/2025

Abstract

Maintainers of programming languages typically provide a ‘best practices or a style guide for developers to reference. These guides are designed as reference documentation so that developers across the world have a policy that they can adhere to, allowing most other developers to more easily read through the code. For a language like C#, this style guide is maintained by Microsoft on the Learn.Microsoft.com website. While researching C++, I found a similar style guide maintained on a GitHub repo (Stroustrup & Sutter, 2025) that was endorsed by Microsoft.

The primary goal of these style guides is to encourage readability and documentation of the code being developed. The quality and organization of these style guides, as well as the topics covered, are often similar, but some guides are more declarative than others. For example, on the topic of indentation, both the Python and C# style guides recommend the use of 4 spaces and avoidance of tabs. In contrast, the C++ style guide has this to say: “We are less concerned with low-level issues, such as naming conventions and indentation style. However, no topic that can help a programmer is out of bounds (Stroustrup & Sutter, 2025).” This C++ style guide has no other mention of indentation or file structure that I could locate.

The primary languages the remainder of the document will focus on is C# and Python.

Summary of Findings:

Naming Guidelines:

C# has many pages dedicated to their naming guidelines. Typically, PascalCasing is used for most symbols within the language. This includes namespaces, types, methods, and most other identifiers. Parameters are recommended to be camelCasing though, to easily pick out what is an and identifier and what is a parameter. Using underscores is discouraged and will generate warnings/messages in Visual Studio.

One major note on the capitalization conventions is that while C# is case-sensitive, other languages that run are the CLR are not required to be. Therefore, a developer should take care to avoid similarly-named symbols. For example, "Apple" and "apple" are distinct in C#, but may be interpreted as the same for Visual Basic and PowerShell functions that interact with the compile C# dll.

Python is case-sensitive, similar to C#. But as for the rest of naming conventions, the Python official documentation has this to say: "The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent" (Rossum et al., 2001). The only real 'rule' to naming conventions in python is that you should never introduce your own__double_leading_and_trailing_underscore__ names, as these are "magic" objects or attributes reserved for use by the interpreter. Even that won't break things though, unless it conflicts with the interpreter itself.

Python recommendations are using PascalCase ("CapWords" as they call it) for classes, which agrees with C# naming conventions. But module names in python are recommended to be short and all lower-case, using an underscore is discouraged, but is not in violation. Confusingly, python also recommends that variables, parameters, and generally everything that isn't a class or a constant use all_lower_snake_case.

Exceptions & Error Handling in C#

While studying code developed by Microsoft, I often see methods that throw exceptions in addition to the calling methods. One example of this is ArgumentNullException.ThrowIfNull(), which was made publicly available in one of the more recent .Net versions. The design guidelines have this to say regarding this practice:

"It is common to throw the same exception from different places. To avoid code bloat, use helper methods that create exceptions and initialize their properties. Also, members that throw exceptions are not getting inlined. Moving the throw statement inside the builder might allow the member to be inlined." (Microsoft, 2023)

For example, this recommendation changes the code from

```
Void DoSomething()
{
        If (x) throw new Exception ();
}
```

To

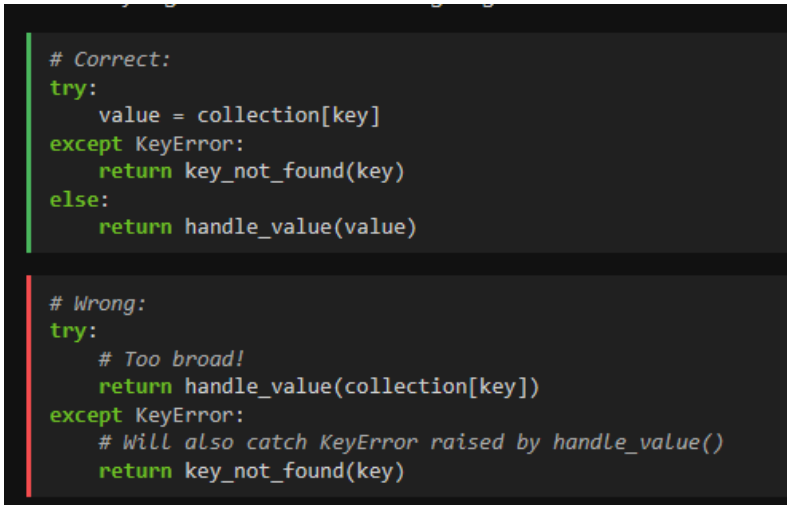
```
static void Throw() => throw new Exception();
Void DoSomething()
{
        If (x) Throw()
}
```

Additionally, it is recommended to use guard statements to avoid potentially throwing exceptions, especially on a hot path. A branch check is significantly less expensive that an exception getting thrown up the stack. This is called the "Tester-Doer Pattern".

Exceptions & Error Handling in Python

The Python guide has a well written section on exception handling, with many of the recommendations lining up with recommendations for C#. The major points both guides agree on is to catch exceptions from most specific to most generic, but also ensure that the exceptions are meaningful. Both languages recommend that when you are catching exceptions, try to avoid the base level (most generic) exceptions.

Python offers a unique way to handle exceptions compared to C# though, with the try-except-else pattern, which does not exist in C#. It is generally recommended to catch as high up as possible in C#, or where it makes sense to do so, in order to stop execution of a failing path. In Python the try-except-else pattern allows stopping execution with handling as close as possible to the error. The example provided calls two methods, both of which can generate a KeyError. When using this pattern, it can detect which method the error came from without having to inspect the stack trace, and the program can react accordingly. This same pattern in C# would involve nested try-catch statements, or declaring variables shares amongst the methods outside of the try-catch blocks.



```python
# Correct:
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

```python
# Wrong:
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

Sealed Classes & Attributes (C#)

In C#, sealing classes indicates that they cannot be inherited from. C# allows for virtual members, which derived classes can then override and change the functionality. This flexibility causes a minor performance impact, with each derivative increasing the hit. As an example, if you have A > B > C, B and C may both override a method and may even call the base method within the override. Even on type C, the CLR must check for any other overrides. Sealing class C will ensure that no additional classes can be dertived, making the performance impact max out at type C. Similarly, if class B is unsealed, but the method is sealed, then the one 1 additional lookup is performed. This change does prevent C from overriding the functionality though.

Generally, it is not recommended to seal classes, to allow other consumers the ability to customize them. In the case of virtual members, the recommendation is to mark them as protected, which should cause less lookups to occur as they are not publicly exposed. An exception to the rule of sealed classes is Attributes act as metadata about the tagged type or member. Attributes are scanned by the runtime, source generators, and by Reflection calls, so sealing these is recommended to improve the lookup of the attribute.

Attributes also have several other guidelines, as their use within the language is unique. Namely, any constructor arguments for the attribute should be publicly accessibly get-only after construction. Optional parameters should never be included in the constructors, and should instead be public Get/Set properties. Furthermore, each attribute should only have a single publicly exposed constructor with arguments, meaning that they either have just the parameterless constructor, one with arguments, or both, but never a third.

Encapsulation

Encapsulation is the idea that methods should be as small and as self-contained as possible. The goal is to reduce bugs that can occur in long-running methods by isolating the different aspects of each operation. Encapsulation also provides the ability to reuse code across the code base. When optimizations or changes are required, all callers will then receive those changes, instead of having to paste those fixes into every caller.

Abhishek Malaviya describes the following benefits when this practice is applied:

- Improved readability
- Easier Maintenance
- Enhanced Reusability
- Better Testability
- Reduced Complexity
- Improved Collaboration.

(Malaviya, 2023)

Sample Code:

Old Code

This code sample was adapted from several CodeProject.com articles for safe removal of a USB device. The goal was to modernize the code by using the 'CSWin32' source generator for the PInvoke calls. The other issue with the original code was that it was not properly ejecting the USB devices during my testing – the volume was removed but the device was still powered. A full copy of this file is uploaded with the assignment.

```
    /// <summary>
    /// Attempts to eject a drive specified by the drive letter
    /// </summary>
    /// <param name="driveLetter">a-Z</param>
    /// <returns><see langword="true"/> if successfully ejected, otherwise <see langword="false"/></returns>
    /// <exception cref="ArgumentException">Thrown if the <paramref name="driveLetter"/> is invalid or the drive is not removable.</exception>"
    /// https://www.codeproject.com/articles/How-to-Prepare-a-USB-Drive-for-Safe-Removal
    /// https://www.codeproject.com/articles/How-to-Prepare-a-USB-Drive-for-Safe-Removal-2
    public static unsafe bool Eject(char driveLetter)
    {
        if ((driveLetter >= 'A' && driveLetter <= 'Z' || driveLetter >= 'a' && driveLetter <= 'z') is false)
            throw new ArgumentException($"Invalid Drive Letter - Expected A-Z. Received '{driveLetter}'", nameof(driveLetter));

        driveLetter = Char.ToUpperInvariant(driveLetter);

        if (PInvoke.GetDriveType($"{driveLetter}:\\") != PInvoke.DRIVE_REMOVABLE)
        {
            throw new ArgumentException($"Drive '{driveLetter}:\\' is not a Removable Drive.", nameof(driveLetter));
        }

        bool success = false;

        // Get the storage device number
        using var handle = PInvoke.CreateFile(
                lpFileName: $@"\\.\{driveLetter}:",
                dwDesiredAccess: (uint)(GENERIC_ACCESS_RIGHTS.GENERIC_READ | GENERIC_ACCESS_RIGHTS.GENERIC_WRITE),
                dwShareMode: FILE_SHARE_MODE.FILE_SHARE_READ | FILE_SHARE_MODE.FILE_SHARE_WRITE,
                lpSecurityAttributes: null,
                dwCreationDisposition: FILE_CREATION_DISPOSITION.OPEN_EXISTING,
                dwFlagsAndAttributes: FILE_FLAGS_AND_ATTRIBUTES.FILE_ATTRIBUTE_DEVICE,
                hTemplateFile: null);
        var sdn = new STORAGE_DEVICE_NUMBER();
#if CSWIN32_0_3_236
        Span<byte> outBuffer = new(&sdn, Marshal.SizeOf<PREVENT_MEDIA_REMOVAL>());
        var success = PInvoke.DeviceIoControl(handle, PInvoke.IOCTL_STORAGE_GET_DEVICE_NUMBER, null, outBuffer);
#elif CSWIN32_0_3_235
        Span<byte> outBuffer = new(&sdn, Marshal.SizeOf<PREVENT_MEDIA_REMOVAL>());
        uint bytesReturned = 0;
        success = PInvoke.DeviceIoControl(handle, PInvoke.IOCTL_STORAGE_GET_DEVICE_NUMBER, null, 0u, &sdn, (uint)Marshal.SizeOf<STORAGE_DEVICE_NUMBER>()
            && bytesReturned > 0;
#endif

        if (success)
        {
            success = false;
            uint deviceInstanceID = 0;
            Guid diskGuid = PInvoke.GUID_DEVINTERFACE_DISK; // this is only type this class cares about

            // Get device interface info set handle for all devices attached to system
            var hDevInfo = PInvoke.SetupDiGetClassDevs(
                ClassGuid: &diskGuid,
                Enumerator: null, hwndParent: HWND.Null,
                Flags: SETUP_DI_GET_CLASS_DEVS_FLAGS.DIGCF_PRESENT | SETUP_DI_GET_CLASS_DEVS_FLAGS.DIGCF_DEVICEINTERFACE);

            if (hDevInfo.IsNull)
            {
                PInvoke.SetupDiDestroyDeviceInfoList(hDevInfo);
                return false;
            }

            try
            {
                // Prepare interface data
                SP_DEVINFO_DATA devInfoData = new SP_DEVINFO_DATA();
                devInfoData.cbSize = (uint)Marshal.SizeOf<SP_DEVINFO_DATA>();

                SP_DEVICE_INTERFACE_DATA devInterfaceData = new SP_DEVICE_INTERFACE_DATA();
                devInterfaceData.cbSize = (uint)Marshal.SizeOf<SP_DEVICE_INTERFACE_DATA>();

                uint index = 0;

                while (true)
                {
                    // Get device interface data
                    if (!PInvoke.SetupDiEnumDeviceInterfaces(hDevInfo, null, &diskGuid, index++, &devInterfaceData))
                    {
                        // returns false if no more interfaces
                        int err = Marshal.GetLastWin32Error();
                        if (err == (int)WIN32_ERROR.ERROR_NO_MORE_ITEMS)
                            return false;

                        Marshal.ThrowExceptionForHR(err);
                        return false;
                    }

                    // Obtain required size for SP_DEVICE_INTERFACE_DETAIL_DATA_W
                    // expected to return false with ERROR_INSUFFICIENT_BUFFER
                    uint requiredSize = 0;
                    //PInvoke.SetupDiGetDeviceInterfaceDetail(hDevInfo, &devInterfaceData, null, 0, &requiredSize, &devInfoData);
                    PInvoke.SetupDiGetDeviceInterfaceDetail(hDevInfo, &devInterfaceData, null, 0, &requiredSize, null);


                    // Call against to get the device interface details
                    SP_DEVICE_INTERFACE_DETAIL_DATA_W* detailData = (SP_DEVICE_INTERFACE_DETAIL_DATA_W*)Marshal.AllocHGlobal((int)requiredSize);
                    detailData->cbSize = (uint)Marshal.SizeOf<SP_DEVICE_INTERFACE_DETAIL_DATA_W>();
                    try
                    {
                        if (!PInvoke.SetupDiGetDeviceInterfaceDetail(hDevInfo, &devInterfaceData, detailData, requiredSize, &requiredSize, &devInfoData)
                        {
                            Marshal.ThrowExceptionForHR(Marshal.GetLastWin32Error());
                            continue;
                        }

                        if (detailData->cbSize == 0u)
                            continue;
```

Improved Code

This code is significantly easier to understand the flow of the function.

Best practices applied:

- Encapsulation – Break smaller blocks into self-contained methods.
    - For example, **DriveLetterToVolumePath(char)** converts the character into a valid volume path for the PInvoke api.
    - Should an exception occur, it will also be easier to debug as the call stack will be more direct as to where the problem lies.
- Throw Helpers – Validation is performed at various points in the class, so helpers were created to ensure validation is performed the same way every time.
    - The Throw functions themselves were made into static methods to allow inlining the validation checks, per the design guidelines.
- Naming Schemes (Pascal and camelCase) were used where appropriate.
- Documentation – Functions were appropriately documented with comments for consumers.

```
/// <summary>
/// Attempts to eject a drive specified by the drive letter
/// </summary>
/// <param name="driveLetter">a-Z</param>
/// <returns><see langword="true"/> if successfully ejected, otherwise <see langword="false"/></returns>
/// <exception cref="ArgumentException">Thrown if the <paramref name="driveLetter"/> is invalid or the drive is not removable.</exception>"
/// https://www.codeproject.com/articles/How-to-Prepare-a-USB-Drive-for-Safe-Removal
/// https://www.codeproject.com/articles/How-to-Prepare-a-USB-Drive-for-Safe-Removal-2
4 references
public static unsafe bool Eject(char driveLetter)
{
    ThrowIfInvalidDriveChar(driveLetter);
    driveLetter = Char.ToUpperInvariant(driveLetter);
    ThrowIfDriveIsNotRemovable(driveLetter);

    if (TryGetStorageDeviceNumber(DriveLetterToVolumePath(driveLetter), out var deviceNumber))
    {
        if (TryGetDeviceInstanceID(deviceNumber, out uint deviceInstanceID))
        {
            if (DismountVolume(driveLetter))
            {
                // The first parent must be ejected - otherwise it attempt to dismount the volume only and subsequent attempts fail
                var cr = PInvoke.CM_Get_Parent(out var parentDevId, deviceInstanceID, 0u);
                if (cr == CONFIGRET.CR_SUCCESS)
                {
BUG
                    Span<char> buffer = new char[PInvoke.MAX_PATH];
                    PNP_VETO_TYPE failureReason = default;
                    cr = PInvoke.CM_Request_Device_Eject((uint)parentDevId, &failureReason, buffer, 0u);

                    cr = PInvoke.CM_Request_Device_Eject((uint)parentDevId, null, null, 0u);

                }
                if (cr == CONFIGRET.CR_SUCCESS)
                    return true;

                var errorCode = (int)PInvoke.CM_MapCrToWin32Err(cr, default);
                Marshal.ThrowExceptionForHR(errorCode);
                Marshal.ThrowExceptionForHR(Marshal.GetLastWin32Error());
            }
        }
    }
    return false;
}
```

References

Armanisoft. (2012, April 30). *How to Prepare a USB Drive for Safe Removal*. CodeProject. https://www.codeproject.com/articles/How-to-Prepare-a-USB-Drive-for-Safe-Removal-2

Google. (n.d.). *Google C++ Style Guide*. google.github.io. https://google.github.io/styleguide/cppguide.htm

Malaviya, A. (2023, 09 08). *.NET C# clean code and best practice Part-2*. Medium.com. https://mabhishekit.medium.com/net-c-clean-code-and-best-practice-part-2-c7f3e133ca69

Microsoft. (2005,2009). *Framework design guidelines*. Learn.Microsoft.com. https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/

Microsoft. (2023, 10 03). *Exception Throwing*. Microsoft.com. https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/exception-throwing

Microsoft. (2025, 01 18). *Coding Conventions*. Microsoft.com. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Rossum, G. v., Warsaw, B., & Coghlan, A. (2001, 08 05). *PEP 8 – Style Guide for Python Code*. Python.org. https://peps.python.org/pep-0008/

StackOverflow. (2011). *Eject USB device via C#*. Stack Overflow. https://stackoverflow.com/questions/7704599/eject-usb-device-via-c-sharp

Stroustrup, B., & Sutter, H. (2025, 07 08). *C++ Core Guidelines*. C++ Core Guidelines. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-struct

Uwe_Sieber. (2006, April 19). *How to Prepare a USB Drive for Safe Removal*. CodeProject.com. https://www.codeproject.com/articles/How-to-Prepare-a-USB-Drive-for-Safe-Removal