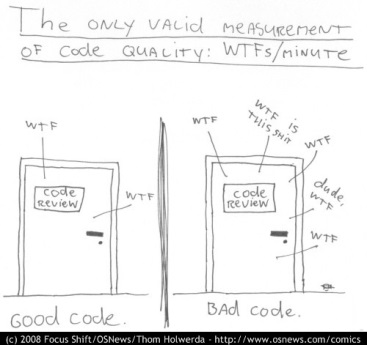
**Coding Guidelines**



**Table of contents**

[1. Introduction 9](#_Toc376426903)

[1.1 Why Have Code Standards 9](#_Toc376426904)

[1.2 Basic Principles 9](#_Toc376426905)

[1.3 Not All Coding Standards are Created Equal! 12](#_Toc376426906)

[2. Guidelines 13](#_Toc376426907)

[2.1 Class Design Guidelines 13](#_Toc376426908)

[K1100 A class or interface should have a single purpose  13](#_Toc376426909)

[K3101 Only create a constructor that returns a useful object  15](#_Toc376426910)

[K2102 An interface should be small and focused  15](#_Toc376426911)

[K3103 Use an interface rather than a base class to support multiple implementations  17](#_Toc376426912)

[K3104 Use an interface to decouple classes from each other  19](#_Toc376426913)

[K2105 Avoid static classes  21](#_Toc376426914)

[K1106 Don’t hide inherited members with the new keyword  21](#_Toc376426915)

[K1107 It should be possible to treat a derived object as if it were a base class object  22](#_Toc376426916)

[K1108 Don’t refer to derived classes from the base class  24](#_Toc376426917)

[K2109 Avoid exposing the other objects an object depends on  24](#_Toc376426918)

[K1110 Avoid bidirectional dependencies  24](#_Toc376426919)

[K1111 Classes should have state and behavior  24](#_Toc376426920)

[2.2 Member Design Guidelines 26](#_Toc376426921)

[K1200 Allow properties to be set in any order  26](#_Toc376426922)

[K1201 Use a method instead of a property  26](#_Toc376426923)

[K1202 Don’t use mutual exclusive properties  26](#_Toc376426924)

[K1203 A method or property should do only one thing  26](#_Toc376426925)

[K2204 Don’t expose stateful objects through static members  27](#_Toc376426926)

[K2205 Return an IEnumerable<T> or ICollection<T> instead of a concrete collection class  27](#_Toc376426927)

[K1206 Properties, methods and arguments representing strings or collections should never be null  27](#_Toc376426928)

[K3207 Consider using domain-specific value types rather than primitives  27](#_Toc376426929)

[2.3 Miscellaneous Design Guidelines 28](#_Toc376426930)

[K2300 Throw exceptions rather than returning some kind of status value  28](#_Toc376426931)

[K2301 Provide a rich and meaningful exception message text  28](#_Toc376426932)

[K2302 Throw the most specific exception that is appropriate  28](#_Toc376426933)

[K1303 Don’t swallow errors by catching generic exceptions  28](#_Toc376426934)

[K2304 Properly handle exceptions in asynchronous code  28](#_Toc376426935)

[K2305 Always check an event handler delegate for null  29](#_Toc376426936)

[K2306 Use a protected virtual method to handle each event  29](#_Toc376426937)

[K2305 Consider providing property-changed events  29](#_Toc376426938)

[K1306 Don’t pass null as the sender argument when raising an event  30](#_Toc376426939)

[K2307 Use generic constraints if applicable  30](#_Toc376426940)

[K1308 Evaluate the result of a LINQ expression before returning it  31](#_Toc376426941)

[2.4 Maintainability Guidelines 32](#_Toc376426942)

[K1400 Methods should not exceed 7 statements  32](#_Toc376426943)

[K1401 Make all members private and types internal by default  32](#_Toc376426944)

[K2402 Avoid conditions with double negatives  32](#_Toc376426945)

[K3403 Name assemblies after their contained namespace  32](#_Toc376426946)

[K1404 Name a source file to the type it contains  33](#_Toc376426947)

[K1405 Limit the contents of a source code file to one type  33](#_Toc376426948)

[K3406 Name a source file to the logical function of the partial type  33](#_Toc376426949)

[K1407 Use using statements instead of fully qualified type names  33](#_Toc376426950)

[K1408 Don’t use "magic” numbers  34](#_Toc376426951)

[K1409 Only use var when the type is extremely obvious  34](#_Toc376426952)

[K2410 Declare and initialize variables as late as possible  35](#_Toc376426953)

[K1411 Assign each variable in a separate statement  35](#_Toc376426954)

[K1412 Always use Object and Collection Initializers instead of separate statements  35](#_Toc376426955)

[K1413 Don’t make explicit comparisons to true or false  36](#_Toc376426956)

[K2414 Don’t change a loop variable inside a for or foreach loop  36](#_Toc376426957)

[K2415 Avoid nested loops  37](#_Toc376426958)

[K1416 Always add a block after control structures such if, else, while, for, foreach and case  37](#_Toc376426959)

[K1417 Always add a default block after the last case in a switch statement  37](#_Toc376426960)

[K2418 Finish every if-else-if statement with an else-part  38](#_Toc376426961)

[K2419 Be reluctant with multiple return statements  39](#_Toc376426962)

[K2420 Don’t use if-else statements instead of a simple (conditional) assignment  39](#_Toc376426963)

[K1421 Encapsulate complex expressions in a method or property  40](#_Toc376426964)

[K2422 Call the most overloaded method from other overloads  40](#_Toc376426965)

[K1423 Only use optional arguments to replace overloads  41](#_Toc376426966)

[K1424 Avoid using named arguments  42](#_Toc376426967)

[K1425 Don’t allow methods and constructors with more than three parameters  42](#_Toc376426968)

[K1426 Don’t use ref or out parameters  43](#_Toc376426969)

[K2427 Avoid methods that take a bool flag  43](#_Toc376426970)

[K3428 Don’t use parameters as temporary variables  43](#_Toc376426971)

[K1429 Always check the result of an as operation  43](#_Toc376426972)

[K1430 Don’t comment out code  44](#_Toc376426973)

[2.5 Naming Guidelines 45](#_Toc376426974)

[K1500 Use US-English  45](#_Toc376426975)

[K1501 Use proper casing for language elements  45](#_Toc376426976)

[K3502 Don’t include numbers in variables, parameters and type members  46](#_Toc376426977)

[K1503 Don’t prefix fields  46](#_Toc376426978)

[K2504 Don’t use abbreviations  46](#_Toc376426979)

[K2505 Name a member, parameter or variable according its meaning and not its type  46](#_Toc376426980)

[K2506 Name types using nouns, noun phrases or adjective phrases  46](#_Toc376426981)

[K2507 Name generic type parameters with descriptive names  47](#_Toc376426982)

[K1508 Don’t repeat the name of a class or enumeration in its members  47](#_Toc376426983)

[K3509 Name members similarly to members of related .NET Framework classes  47](#_Toc376426984)

[K1510 Avoid short names or names that can be mistaken with other names  48](#_Toc376426985)

[K2511 Properly name properties  48](#_Toc376426986)

[K2512 Name methods using verb-object pair  48](#_Toc376426987)

[K3513 Name namespaces using names, layers, verbs and features  48](#_Toc376426988)

[K2514 Use a verb or verb phrase to name an event  49](#_Toc376426989)

[K3515 Use -ing and -ed to express pre-events and post-events  49](#_Toc376426990)

[K3516 Prefix an event handler with On  49](#_Toc376426991)

[K3517 Use an underscore for irrelevant lambda parameters  50](#_Toc376426992)

[K2518 Group extension methods in a class suffixed with Extensions  50](#_Toc376426993)

[K2519 Postfix asynchronous methods with Async or TaskAsync  50](#_Toc376426994)

[2.6 Performance Guidelines 51](#_Toc376426995)

[K3600 Consider using Any() to determine whether an IEnumerable<T> is empty  51](#_Toc376426996)

[K3601 Only use async for low-intensive long-running activities  51](#_Toc376426997)

[K3602 Prefer Task.Run for CPU intensive activities  51](#_Toc376426998)

[K2603 Beware of mixing up await/async with Task.Wait  52](#_Toc376426999)

[K2604 Beware of async/await deadlocks in single-threaded environments  52](#_Toc376427000)

[2.7 Framework Guidelines 53](#_Toc376427001)

[K1700 Use C# type aliases instead of the types from the System namespace  53](#_Toc376427002)

[K2701 Properly name properties, variables or fields referring to localized resources  53](#_Toc376427003)

[K3702 Don’t hardcode strings that change based on the deployment  53](#_Toc376427004)

[K1703 Build with the highest warning level and treat warnings as errors 54](#_Toc376427005)

[K3704 Properly fill the attributes of the AssemblyInfo.cs file  55](#_Toc376427006)

[K3705 Avoid LINQ for simple expressions  55](#_Toc376427007)

[K2706 Use Lambda expressions instead of delegates  55](#_Toc376427008)

[K1707 Only use the dynamic keyword when talking to a dynamic object  56](#_Toc376427009)

[K1708 Favor async/await over the Task  56](#_Toc376427010)

[K1709 Use string.Empty instead of ""  56](#_Toc376427011)

[K1710 Do not use the this reference  57](#_Toc376427012)

[K1711 Do not use the base keyword to access base class members  57](#_Toc376427013)

[2.8 Documentation Guidelines 58](#_Toc376427014)

[K2800 Avoid inline comments  58](#_Toc376427015)

[K1801 Only write comments to explain complex algorithms or decisions  58](#_Toc376427016)

[K3802 Don’t use comments for tracking work to be done later  58](#_Toc376427017)

[K1803 All comments must pass spell checking  59](#_Toc376427018)

[2.9 Layout Guidelines 60](#_Toc376427019)

[K1900 Use a common layout  60](#_Toc376427020)

[K1901 Wrapping lines rules  61](#_Toc376427021)

[K1902 Blank Spaces rules  64](#_Toc376427022)

[K1903 Blank Lines rules  65](#_Toc376427023)

[K1904 Order and group namespaces according the company  65](#_Toc376427024)

[K1905 Place members in a well-defined order  66](#_Toc376427025)

[K1906 #regions are forbidden  66](#_Toc376427026)



1. Introduction

This document attempts to provide guidelines (or coding standards if you like) for coding in C# 3.0, 4.0 or 5.0 that are both useful and pragmatic. All developers need to carefully read this document at least once before start coding. This will give them a sense of the kind of guidelines the document contains. This document has been compiled using many resources from community, blog posts, books and many years of developing in C#.

* 1. Why Have Code Standards

Although some might see coding guidelines as undesired overhead or something that limits creativity, this approach has already proven its value for many years. Why? Well, because not every developer

* is aware that code is generally read 10 times more than it is changed;
* is aware that 80% of the lifetime cost of a piece of software goes to maintenance;
* is aware that hardly any software is maintained for its whole life by the original author;
* is aware of the potential pitfalls of certain constructions in C#;
* is introduced into certain conventions when using the .NET Framework such as IDisposable or the deferred execution nature of LINQ;
* is aware of the impact of using (or neglecting to use) particular solutions on aspects like security, performance, multi-language support, etc.;
* knows that not every developer is as capable in understanding an elegant, but abstract, solution as the original developer;

Code standards improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly. If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

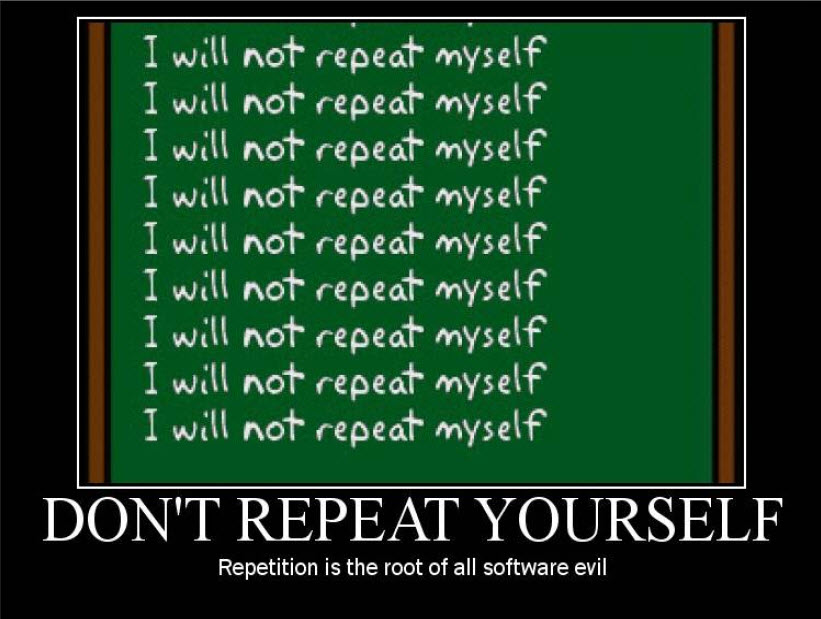
* 1. Basic Principles

There are many unexpected things I run into during my work as a consultant, each deserving at least one guideline. Unfortunately, I still need to keep this document within a reasonable size. But unlike to what some junior developers believe, that doesn’t mean that when something is not mentioned in this guidelines it must be okay.

In general, if I have a discussion with a developer about a smell that this document does not provide absolution for, I’ll refer back to a set of basic principles that apply to all situations, regardless of context. These include:

* The Principle of Least Surprise (or Astonishment), which means that you should choose a solution that does include any things people might not understand, or put on the wrong track.
* Keep It Simple Stupid (a.k.a. KISS), a funny way of saying that the simplest solution is more than sufficient.
* You Ain’t Gonne Need It (a.k.a. YAGNI), which tells you to create a solution for the current problem rather than the ones you think will happen later on (since when can you predict the future?)
* Don’t Repeat Yourself (a.k.a. DRY), which encourages you to prevent duplication in your code base without forgetting the Rule of Three heuristic.

Regardless of the elegancy of somebody’s solution, if it’s too complex for the ordinary developer, exposes unusual behavior, or tries to solve many possible future issues, it is very likely the wrong solution and needs redesign.



* 1. Not All Coding Standards are Created Equal!

To help understand the severity of each guideline I’ve assigned a level of importance to each one of them as a way to advise developers about the importance of each guideline:

Guidelines that you should never skip and should be applicable to all situations

Strongly recommended guidelines

Recommended guidelines that may not be applicable in all situations

1. Guidelines

For the sake of clarity and future references, each guideline is assigned with an identification number with 4 digit which starts with K and a digit for the classification and another for the severity.

* 1. Class Design Guidelines

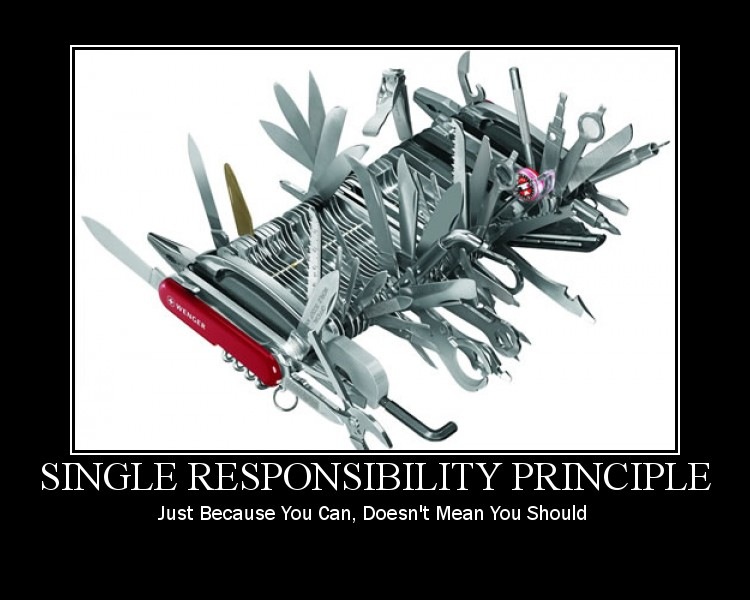
K1100 A class or interface should have a single purpose 

A class or interface should have a single purpose within the system it participates in. In general, a class is either representing a primitive type like an email or ISBN number, an abstraction of some business concept, a plain data structure or responsible for orchestrating the interaction between other classes. It is never a combination of those. This rule is widely known as the [Single Responsibility Principle](http://www.objectmentor.com/resources/articles/srp.pdf), one of the [SOLID](http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)) principles.

**Tip** A class with the word And in it is an obvious violation of this rule.

**Tip** Use [Design Patterns](http://en.wikipedia.org/wiki/Design_pattern_(computer_science)) to communicate the intent of a class. If you can’t assign a single design pattern to a class, chances are that it is doing more than one thing.

**Note** If you create a class representing a primitive type you can greatly simplify it usage by making it immutable.

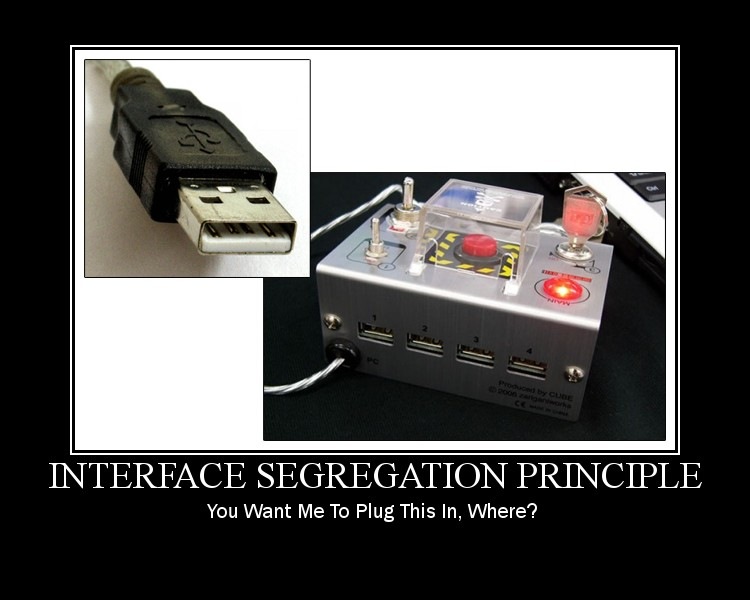


K3101 Only create a constructor that returns a useful object 

There should be no need to set additional properties before the object can be used for whatever purpose it was designed. However, if your constructor needs more than three parameters (which violates AV1561), your class might have too much responsibility (and violate K1100) and needs to be redesigned.

K2102 An interface should be small and focused 

Interfaces should have a name that clearly explains the purpose or role of that interface within the system. Do not combine many vaguely related members on the same interface just because they were all on the same class. Separate the members based on the responsibility of those members so that callers only need to call or implement the interface related to a particular task. This rule is more commonly known as the [Interface Segregation Principle](http://www.objectmentor.com/resources/articles/isp.pdf).



K3103 Use an interface rather than a base class to support multiple implementations 

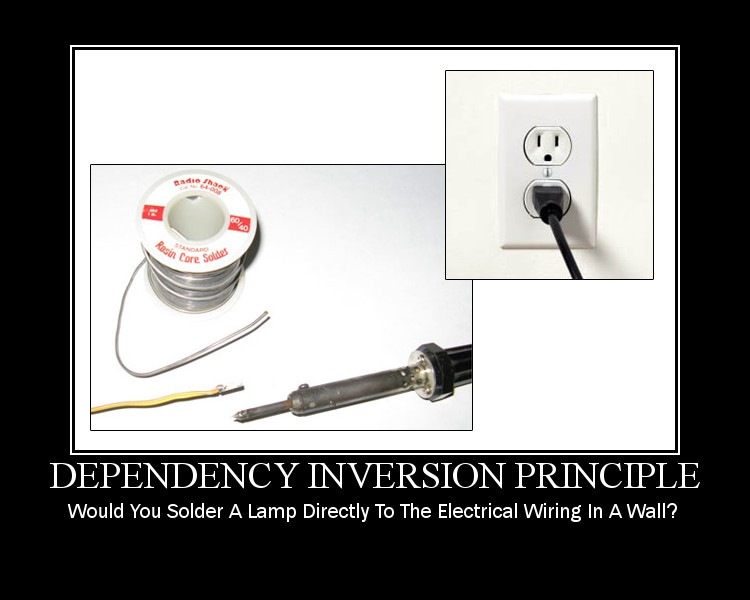
If you want to expose an extension point from your class, expose it as an interface rather than a base class. You don’t want to force users of that extension point to derive their implementations from a base-class that might have undesired behavior. However, for their convenience you may implement an (abstract) default implementation that can serve as a starting point.



K3104 Use an interface to decouple classes from each other 

Interfaces are a very effective mechanism for decoupling classes from each other.

* They can prevent bidirectional associations;
* They simplify the replacement of one implementation with another;
* They allow replacing an expensive external service or resource with a temporary stub for use in a non-production environment.
* They allow replacing the actual implementation with a dummy implementation or a fake object in a unit test;
* Using a dependency injection framework you can centralize the choice which class is going to be used whenever a specific interface is requested.



K2105 Avoid static classes 

With the exception of extension method containers static classes very often lead to badly designed code. They are also very difficult, if not impossible, to test in isolation unless you’re willing to use some very hacky tools. As a general rule, if you find yourself creating a lot of static classes then it is a smell showing that your design is not object oriented and rather structural.

**Note** If you really need that static class, mark it as static so that the compiler can prevent instance members and instantiating your class. This relieves you of creating an explicit private constructor.

K1106 Don’t hide inherited members with the new keyword 

Not only does the new keyword break [Polymorphism](http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming), one of the most essential object-orientation principles, it also makes subclasses more difficult to understand. Consider the following two classes:

public class Book

{

    public virtual void Print()

    {

        Console.WriteLine("Printing Book");

    }

}

public class PocketBook : Book

{

    public new void Print()

    {

        Console.WriteLine("Printing PocketBook");

    }

}

This will cause behavior that you would not normally expect from class hierarchies:

PocketBook pocketBook = new PocketBook();

pocketBook.Print(); // Will output "Printing PocketBook "

((Book) pocketBook).Print(); // Will output "Printing Book"

It should not make a difference whether you call Print through a reference to the base class or through the derived class.

K1107 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.

**Note** This rule is also known as the **Liskov Substitution Principle**, one of the **SOLID** principles.



K1108 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.

K2109 Avoid exposing the other objects an object depends on 

If you find yourself writing code like this then you might be violating the [Law of Demeter](http://en.wikipedia.org/wiki/Law_of_Demeter).

someObject.SomeProperty.GetChild().Foo()

An object should not expose any other classes it depends on because callers may misuse that exposed property or method to access the object behind it. By doing so, you allow calling code to become coupled to the class you are using, and thereby limiting the chance you can easily replace it in a future stage.

**Note** Using a class that is designed using the [Fluent Interface](http://en.wikipedia.org/wiki/Fluent_interface) pattern does seem to violate this rule, but it is simply returning itself so that method chaining is allowed.

**Exception** Inversion of Control or Dependency Injection frameworks often require you to expose a dependency as a public property. As long as this property is not used for anything else than dependency injection I would not consider it a violation but even this is not a good practice as Constructor Injection should always be favored to Property Injection.

K1110 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 two classes requires changes on both parties and may involve a lot of unexpected work. The most obvious way of breaking that dependency is introducing an interface for one of the classes and using Dependency Injection.

**Exception** Domain models such as defined in [Domain Driven Design](http://en.wikipedia.org/wiki/Domain-driven_design) tend to occasionally involve bidirectional associations that model real-life associations. In those cases, I would make sure they are really necessary, but if they are, keep them in.

K1111 Classes should have state and behavior 

In general, if you find a lot of data-only classes in your code base, you probably also have a few (static) classes with a lot of behavior (see [K2105](#K2105)). Use the principles of object-orientation explained in this section and move the logic as close to the data it applies to.

**Exception** The only exception to this rule are classes that are used to transfer data over a communication channel, also called [Data Transfer Objects](http://martinfowler.com/eaaCatalog/dataTransferObject.html), or a class that wraps several parameters of a method.

**Note** Particularly **i**n designing domain model classes these state-only objects are called [Anemic Domain Model](http://martinfowler.com/bliki/AnemicDomainModel.html) which must be avoided. The basic symptom of an Anemic Domain Model is that when you look at the behavior in the domain model, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters even though many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have. Instead there are a set of service objects which capture all the domain logic. These services live on top of the domain model and use the domain model for data. The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together. The anemic domain model is really just a procedural style design, exactly the opposite of what object-oriented design is all about.

* 1. Member Design Guidelines

K1200 Allow properties to be set in any order 

Properties should be stateless with respect to other properties, i.e. there should not be a difference between first setting property DataSource and then DataMember or vice versa.

K1201 Use a method instead of a property 

* If the work is more expensive than setting a field value.
* If it represents a conversion such as the Object.ToString() method.
* If it returns a different result each time it is called, even if the arguments didn’t change. For example, the NewGuid() method returns a different value each time it is called.
* If the operation causes a side effect such as changing some internal state not directly related the property (which violates the [Command Query Separation](http://martinfowler.com/bliki/CommandQuerySeparation.html)).

**Note** In general converting every parameterless method to a read only property is not a good strategy especially if it matches one of the above scenarios.

**Exception** Populating an internal cache or implementing [lazy-loading](http://www.martinfowler.com/eaaCatalog/lazyLoad.html) is a good exception.

K1202 Don’t use mutual exclusive properties 

Having properties that cannot be used at the same time typically signals a type that is representing two conflicting concepts. Even though those concepts may share some of the behavior and state, they obviously have different rules that do not cooperate.

This violation is often seen in domain models and introduces all kinds of conditional logic related to those conflicting rules, causing a ripple effect that significantly worsens the maintenance burden.

K1203 A method or property should do only one thing 

Similarly to rule [K1100](#K1100) a method should have a single responsibility.

K2204 Don’t expose stateful objects through static members 

A stateful object is an object that contains many properties and lots of behavior behind that. If you expose such an object through a static property or method of some other object, it will be very difficult to refactor or unit test a class that relies on such a stateful object. In general, introducing a construction like that is a great example of violating many of the guidelines of this chapter.

A classic example of this is the HttpContext.Current property, part of ASP.NET. Many see the HttpContext class as a source for a lot of ugly code. In fact, the testing guideline [Isolate the Ugly Stuff](http://msdn.microsoft.com/en-us/magazine/dd263069.aspx#id0070015) often refers to this class.

K2205 Return an IEnumerable<T> or ICollection<T> instead of a concrete collection class 

In general, you don’t want callers to be able to change an internal collection, so don’t return arrays, lists or other collection classes directly. Instead, return an IEnumerable<T>, or, if the caller must be able to determine the count, an ICollection<T>.

**Note** If you’re using .NET 4.5, you can also use IReadOnlyCollection<T>, IReadOnlyList<T> or IReadOnlyDictionary<TKey, TValue>.

K1206 Properties, methods and arguments 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 also prevents cluttering your code base with additional checks for null, or even worse, string.IsNotNullOrEmpty().

K3207 Consider using domain-specific value types rather than primitives 

Instead of using strings, integers and decimals for representing domain specific types such as an ISBN number, an email address or amount of money, consider created dedicated value objects that wrap both the data and the validation rules that apply to it. By doing this, you prevent ending up having multiple implementations of the same business rules, which both improves maintainability and prevents bugs.

* 1. Miscellaneous Design Guidelines

K2300 Throw exceptions rather than returning some kind of status value 

A code base that uses return values for reporting the success or failure tends to have nested if-statements sprinkled all over the code. Quite often, a caller forgets to check the return value anyhow. Structured exception handling has been introduced to allow you to throw exceptions and catch or replace exceptions at a higher layer. In most systems it is quite common to throw exceptions whenever an unexpected situation occurs.

K2301 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.

K2302 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.

K1303 Don’t swallow errors by catching generic exceptions 

Avoid swallowing errors by catching non-specific exceptions, such as Exception, SystemException, and so on, in application code. Only top-level code, such as a last-chance exception handler, should catch a non-specific exception for logging purposes and a graceful shutdown of the application.

K2304 Properly handle exceptions in asynchronous code 

When throwing or handling exceptions in code that uses async/await or a Task remember the following two rules:

* Exceptions that occur within an async/await block and inside a Task's action are propagated to the awaiter.
* Exceptions that occur in the code preceding the asynchronous block are propagated to the caller.

K2305 Always check an event handler delegate for null 

An event that has no subscribers is null, so before invoking, always make sure that the delegate list represented by the event variable is not null. Furthermore, to prevent conflicting changes from concurrent threads, use a temporary variable to prevent concurrent changes to the delegate.

event EventHandler<NotifyEventArgs> Notify;

void RaiseNotifyEvent(NotifyEventArgs args)

{

    EventHandler<NotifyEventArgs> handlers = Notify;

if (handlers != null)

    {

        handlers(this, args);

    }

}

**Tip** You can prevent the delegate list from being empty altogether. Simply assign an empty delegate like this:

event EventHandler<NotifyEventArgs> Notify = delegate {};

K2306 Use a protected virtual method to handle each event 

Complying with this guideline allows derived classes to handle a base class event by overriding the protected method. The name of the protected virtual method should be the same as the event name prefixed with On. For example, the protected virtual method for an event named TimeChanged is named OnTimeChanged.

**Note** Derived classes that override the protected virtual method are not required to call the base class implementation. The base class must continue to work correctly even if its implementation is not called.

K2305 Consider providing property-changed events 

Consider providing events that are raised when certain properties are changed. Such an event should be named PropertyChanged, where Property should be replaced with the name of the property with which this event is associated.

**Note** If your class has many properties that require corresponding events, consider implementing the INotifyPropertyChanged interface instead. It is often used in the [Presentation Model](http://martinfowler.com/eaaDev/PresentationModel.html) and [Model-View-ViewModel](http://msdn.microsoft.com/en-us/magazine/dd419663.aspx) patterns.

K1306 Don’t pass null as the sender argument when raising an event 

Often, an event handler is used to handle similar events from multiple senders. The sender argument is then used to get to the source of the event. Always pass a reference to the source (typically this) when raising the event. Furthermore don’t pass null as the event data parameter when raising an event. If there is no event data, pass EventArgs.Empty instead of null.

**Exception** On static events, the sender argument should be null.

K2307 Use generic constraints if applicable 

Instead of casting to and from the object type in generic types or methods, use where constraints or the as operator to specify the exact characteristics of the generic parameter. For example:

class SomeClass { }

    // Don't

    class MyClass<T>

    {

        internal void SomeMethod(T t)

        {

            object temp = t;

            SomeClass obj = (SomeClass) temp;

        }

    }

    // Do

    class MyClass<T> where T : SomeClass

    {

        internal void SomeMethod(T t)

        {

            SomeClass obj = t;

        }

    }

K1308 Evaluate the result of a LINQ expression before returning it 

Consider the following code snippet

public IEnumerable<GoldMember> GetGoldMemberCustomers()

            {

                const decimal GoldMemberThresholdInEuro = 1000000;

                GoldMember q = from customer in db.Customers

                        where customer.Balance > GoldMemberThresholdInEuro

                        select new GoldMember(customer.Name, customer.Balance);

                return q;

            }

Since LINQ queries use deferred execution, returning q will actually return the expression tree representing the above query. Each time the caller evaluates this result using a foreach or something similar, the entire query is re-executed resulting in new instances of GoldMember every time. Consequently, you cannot use the == operator to compare multiple GoldMember instances. Instead, always explicitly evaluate the result of a LINQ query using ToList(), ToArray() or similar methods.

* 1. Maintainability Guidelines

K1400 Methods should not exceed 7 statements 

A method that requires more than 7 statements is simply doing too much or has too many responsibilities. It also requires the human mind to analyze the exact statements to understand what the code is doing. Break it down in multiple small and focused methods with self-explaining names, but make sure the high-level algorithm is still clear.

K1401 Make all members private and types internal 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.

K2402 Avoid conditions with double negatives 

Although a property like customer.HasNoOrders make sense, avoid using it in a negative condition like this:

bool hasOrders = !customer.HasNoOrders;

Double negatives are more difficult to grasp than simple expressions, and people tend to read over the double negative easily.

K3403 Name assemblies after their contained namespace 

All DLLs should be named according to the pattern <Company>.<Component>.dll where <Company> refers to your company’s name and <Component> contains one or more dot-separated clauses. For example: Keous.Web.Controls.dll.

As an example, consider a group of classes organized under the namespace Keous.Web.Binding exposed by a certain assembly. According to this guideline, that assembly should be called Keous.Web.Binding.dll.

**Exception** If you decide to combine classes from multiple unrelated namespaces into one assembly, consider post fixing the assembly with Core, but do not use that suffix in the namespaces. For instance: Keous.Library.Core.dll.

K1404 Name a source file to the type it contains 

A source code file name must exactly match the name of the type it contains. Use Pascal casing for naming the file and don’t use underscores.

K1405 Limit the contents of a source code file to one type 

A source code file must exactly contain one and only one type.

**Exception** Nested types should, for obvious reasons, be part of the same file.

K3406 Name a source file to the logical function of the partial type 

When using partial types and allocating a part per file, name each file after the logical part that part plays. For example:

// In MyClass.cs

public partial class MyClass

{...}

// In MyClass.Designer.cs

public partial class MyClass

{...}

K1407 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:

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

Instead, do this

using System.Collections.Generic;

IList<string> list = new List<string>();

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

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

K1408 Don’t use "magic” numbers 

Don’t use literal values, either numeric or strings, in your code other than to define symbolic constants. For example:

public class Whatever

{

public static readonly Color PapayaWhip = new Color(0xFFEFD5);

public const int MaxNumberOfWheels = 18;

}

Strings intended for logging or tracing are exempt from this rule. Literals are allowed when their meaning is clear from the context, and not subject to future changes, For example:

mean = (a + b) / 2; // okay

WaitMilliseconds(waitTimeInSeconds \* 1000); // clear enough

If the value of one constant depends on the value of another, do attempt to make this explicit in the code.

public class SomeSpecialContainer

{

public const int MaxItems = 32;

public const int HighWaterMark = 3 \* MaxItems / 4; // at 75%

}

**Note** An enumeration can often be used for certain types of symbolic constants.

K1409 Only use var when the type is extremely obvious 

Only use var if the type is very obvious from the same statement and using it would improve readability. In other words using var is forbidden if the exact type is not written on the right side of the assignment. For example:

var repository = new RepositoryFactory.Get<IOrderRepository>();

var list = new ReadOnlyCollection<string>();

In both of the above examples it is clear what type to expect. For a more detailed rationale about the advantages and disadvantages of using var, read Eric Lippert’s [Uses and misuses of implicit typing](http://blogs.msdn.com/b/ericlippert/archive/2011/04/20/uses-and-misuses-of-implicit-typing.aspx).

So don't

var i = 3; // what type? int? uint? float?

var myfoo = MyFactoryMethod.Create("arg"); // Not obvious what base-class or

// interface to expect. Also difficult

// to refactor if you can't search for

// the class

**Exception** Obviously using var as the result of an anonymous LINQ query is perfectly fine:

var query = from order in orders

where order.Items > 10 and order.TotalValue > 1000

select new { order.Id, order.TotalValue };

K2410 Declare and initialize variables as late as possible 

Avoid the C and Visual Basic styles where all variables have to be defined at the beginning of a block, but rather define and initialize each variable at the point where it is needed.

K1411 Assign each variable in a separate statement 

Don’t use confusing constructs like the one below.

var result = someField = GetSomeMethod();

K1412 Always use Object and Collection Initializers instead of separate statements 

Instead of

var startInfo = new ProcessStartInfo(“myapp.exe”);

startInfo.StandardOutput = Console.Output;

startInfo.UseShellExecute = true;

Use [Object Initializers](http://msdn.microsoft.com/en-us/library/bb384062.aspx)

var startInfo = new ProcessStartInfo(“myapp.exe”)

{

StandardOutput = Console.Output,

UseShellExecute = true

};

Similarly, instead of

var countries = new List<string>();

countries.Add(“Netherlands”);

countries.Add(“United States”);

Use collection or [dictionary initializers](http://msdn.microsoft.com/en-us/library/bb531208.aspx)

var countries = new List<string> { “Canada”, “United States” };

K1413 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 != true) // also wrong

while (((condition == true) == true) == true) // where do you stop? while (condition) // OK

K2414 Don’t change a loop variable inside a for or foreach 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. Although this rule also applies to foreach loops, an enumerator will typically detect changes to the collection the foreach loop is iteration over.

for (int index = 0; index < 10; ++index)

{

if (*some condition*)

{

index = 11; // Wrong! Use ‘break’ or ‘continue’ instead.

}

}

K2415 Avoid nested loops 

A method that nests loops is more difficult to understand than one with only a single loop. In fact, in most cases having nested loops can be replaced with a much simpler LINQ query that uses the from keyword twice or more to join the data.

K1416 Always add a block after control structures such if, else, while, for, foreach and case 

Always succeed the keywords if, else, do, while, for and foreach, with opening and closing braces, even though the language does not require it. In other words, Braces are used around all statements, even singletons, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

if (condition) //AVOID! THIS OMITS THE BRACES {}!

statement;

Please note that this also avoids possible confusion in statements of the form:

if (b1) if (b2) Foo(); else Bar(); // which ‘if’ goes with the ‘else’?

// The right way:

if (b1)

{

if (b2)

{

Foo();

}

else

{

Bar();

}

}

K1417 Always add a default block after the last case in a switch statement 

Add a descriptive comment if the default block is supposed to be empty. Moreover, if that block is not supposed to be reached throw an InvalidOperationException to detect future changes that may fall through the existing cases. This ensures better code, because all paths the code can travel has been thought about.

void Foo(string answer)

{

switch (answer)

{

case "no":

Console.WriteLine("You answered with No");

break;

case "yes":

Console.WriteLine("You answered with Yes");

break;

default: // Not supposed to end up here.

throw new InvalidOperationException("Unexpected answer " + answer);

}

}

K2418 Finish every if-else-if statement with an else-part 

For example:

void Foo(string answer)

{

if (answer == "no")

{

Console.WriteLine("You answered with No");

}

else if (answer == "yes")

{

Console.WriteLine("You answered with Yes");

}

else

{

// What should happen when this point is reached? Ignored? // If not, throw an InvalidOperationException.

}

}

K2419 Be reluctant with multiple return statements 

One entry, one exit is a sound principle and keeps control flow readable. However, if the method is very small and complies with guideline [K1400](#K1400) then multiple return statements may actually improve readability over some central boolean flag that is updated at various points.

K2420 Don’t use if-else statements instead of a simple (conditional) assignment 

Express your intentions directly. For example, rather than

bool pos;

if (val > 0)

{

pos = true;

}

else

{

pos = false;

}

Write

bool pos = (val > 0); // initialization

Or instead of

string result;

if (someString != null)

{

result = someString;

}

else

{

result = “Unavailable”;

}

return result;

write

return someString ?? “Unavailable”;

K1421 Encapsulate complex expressions in a method or property 

Consider the following example:

if (member.HidesBaseClassMember

&& (member.NodeType != NodeType.InstanceInitializer))

{

// do something

}

In order to understand what this expression is about, you need to analyze its exact details and all the possible outcomes. Obviously, you could add an explanatory comment on top of it, but it is much better to replace this complex expression with a clearly named method:

if (NonConstructorMemberUsesNewKeyword(member))

{

// do something

}

private bool NonConstructorMemberUsesNewKeyword(Member member)

{

return (member.HidesBaseClassMember

&& (member.NodeType != NodeType.InstanceInitializer)

}

You still need to understand the expression if you are modifying it, but the calling code is now much easier to grasp.

K2422 Call the most overloaded method from other overloads 

This guideline only applies to overloads that are intended for providing optional arguments. Consider for example the following code snippet:

public class MyString

{

private string someText;

public MyString(string text)

{

this.someText = text;

}

public int IndexOf(string phrase)

{

return IndexOf(phrase, 0, someText.Length);

}

public int IndexOf(string phrase, int startIndex)

{

return IndexOf(phrase, startIndex, someText.Length - startIndex);

}

public virtual int IndexOf(string phrase, int startIndex, int count)

{

return someText.IndexOf(phrase, startIndex, count);

}

}

The class MyString provides three overloads for the IndexOf method, but two of them simply call the one with the most parameters. Notice that the same rule applies to class constructors; implement the most complete overload and call that one from the other overloads using the this() operator. Also notice that the parameters with the same name should appear in the same position in all overloads.

**Important** If you also want to allow derived classes to override these methods, define the most complete overload as a protected virtual method that is called by all overloads.

K1423 Only use optional arguments to replace overloads 

The only valid reason for using C# 4.0’s optional arguments is to replace the example from rule [AV1551](#K2422) with a single method like:

public virtual int IndexOf(string phrase, int startIndex = 0, int count = 0)

{

return someText.IndexOf(phrase, startIndex, count);

}

If the optional parameter is a reference type then it can only have a default value of null. But since strings, lists and collections should never be null according to rule [K1206](#K1206), you must use overloaded methods instead.

**Note** The default values of the optional parameters are stored at the caller side. As such, changing the default value without recompiling the calling code will not apply the new default value properly.

**Note** When an interface method defines an optional parameter, its default value is not considered during overload resolution unless you call the concrete class through the interface reference. See [this post](http://blogs.msdn.com/b/ericlippert/archive/2011/05/09/optional-argument-corner-cases-part-one.aspx) by Eric Lippert for more details.

K1424 Avoid using named arguments 

C# 4.0’s named arguments have been introduced to make it easier to call COM components that are known for offering tons of optional parameters. If you need named arguments to improve the readability of the call to a method, that method is probably doing too much and should be refactored.

The only exception where named arguments improve readability is when a constructor that yields a valid object is called like this:

Person person = new Person(

firstName: "John",

lastName: "Smith",

dateOfBirth: new DateTime(1970, 1, 1));

K1425 Don’t allow methods and constructors with more than three parameters 

If you end up with a method with more than three parameters, use a structure or class for passing multiple arguments such as explained in the [Specification](http://en.wikipedia.org/wiki/Specification_pattern) design pattern. In general, the fewer the number of parameters, the easier it is to understand the method. Additionally, unit testing a method with many parameters requires many scenarios to test.

Therefore, always avoid writing methods that take too many parameters. If there too many parameters, it means there are some relationships between those parameters which means they need to be grouped together in another place (like a class). For example:

public void LoginUser(string username, string password, IPAddress ipAddress, bool persist)

{

    // ...

}

Needs to be changed to:

public void LoginUser(UserLoginRequest request)

{

    // ...

}

K1426 Don’t use ref or out parameters 

They make code less understandable and might cause people to introduce bugs. Prefer returning compound objects instead.

K2427 Avoid methods that take a bool flag 

Consider the following method signature:

public Customer CreateCustomer(bool platinumLevel) { }

On first sight this signature seems perfectly fine, but when calling this method you will lose this purpose completely:

Customer customer = CreateCustomer(true);

Often, a method taking such a flag is doing more than one thing and needs to be refactored into two or more methods. An alternative solution is to replace the flag with an enumeration.

K3428 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.

K1429 Always check the result of an as operation 

If you use as to obtain a certain interface reference from an object, always ensure that this operation does not return null. Failure to do so may cause a NullReferenceException at a much later stage if the object did not implement that interface.

K1430 Don’t comment out code 

Never check-in code that is commented-out, but instead use a work item tracking system to keep track of some work to be done. Nobody knows what to do when they encounter a block of commented-out code. Was it temporarily disabled for testing purposes? Was it copied as an example? Should I delete it?

* 1. Naming Guidelines

K1500 Use US-English 

All type members, parameters and variables should be named using words from the American English language.

* Choose easily readable, preferably grammatically correct names. For example, HorizontalAlignment is more readable than AlignmentHorizontal.
* 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.

**Exception** In most projects, you will use words and phrases from your domain and names specific to your company. Visual Studio’s Static Code Analysis will perform a spelling check on all code, so you may need to add those terms to a Custom Code Analysis Dictionary.

K1501 Use proper casing for language elements 

|  |  |  |
| --- | --- | --- |
| **Language element** | **Casing** | **Example** |
| Class, Struct | Pascal | AppDomain |
| Interface | Pascal | IBusinessService |
| Enumeration type | Pascal | ErrorLevel |
| Enumeration values | Pascal | FatalError |
| Event | Pascal | Click |
| Private field | \_camel | \_listItem |
| Protected field | Pascal | MainPanel |
| Const field | Pascal | MaximumItems |
| Const variable | camel | maximumItems |
| Read-only static field | Pascal | RedValue |
| Variable | camel | listOfValues |
| Method | Pascal | ToString |
| Namespace | Pascal | System.Drawing |
| Parameter | camel | typeName |
| Type Parameter | Pascal | TView |
| Property | Pascal | BackColor |

K3502 Don’t include numbers in variables, parameters and type members 

In most cases they are a lazy excuse for not defining a clear and intention-revealing name.

K1503 Don’t prefix fields 

For example, don’t use g\_ or s\_ to distinguish static versus non-static fields. In general, a method in which it is difficult to distinguish local variables from member fields is too big. Examples of incorrect identifier names are: m\_currentUser, mUserName, m\_loginTime.

K2504 Don’t use abbreviations 

For example, use OnButtonClick rather than OnBtnClick. Avoid single character variable names, such as i or q. Use index or query instead.

**Exceptions** Use well-known abbreviations that are widely accepted or well-known within the domain you work. For instance, use UI instead of UserInterface.

K2505 Name a member, parameter or variable according its meaning and not its type 

* Use functional names. For example, GetLength is a better name than GetInt.
* Don’t use terms like Enum, Class or Struct in a name.
* Identifiers that refer to a collection type should have a plural name.

K2506 Name types using nouns, noun phrases or adjective phrases 

Bad examples include SearchExamination (a page for searching for examinations), Common (does not end with a noun, and does not explain its purpose) and SiteSecurity (although the name is technically okay, it does not say anything about its purpose). Good examples include BusinessBinder, SmartTextBox, or EditableSingleCustomer.

Don’t include terms like Utility or Helper in classes. Classes with a name like that are usually static classes and are introduced without considering the object-oriented principles (see also [K2105](#K2105)).

K2507 Name generic type parameters with descriptive names 

* Always prefix descriptive type parameter names with the letter T.
* Always use a descriptive name unless a single-letter name is completely self-explanatory and a longer name would not add value. Use the single letter T as the type parameter in that case.
* Consider indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to ISession may be called TSession.

K1508 Don’t repeat the name of a class or enumeration in its members 

class Employee

{

// Wrong!

static GetEmployee() {}

DeleteEmployee() {}

// Right

static Get() {...}

Delete() {...}

// Also correct

AddNewJob() {...}

RegisterForMeeting() {...}

}

K3509 Name members similarly to members of related .NET Framework classes 

.NET developers are already accustomed to the naming patterns the framework uses, so following this same pattern helps them find their way in your classes as well. For instance, if you define a class that behaves like a collection, provide members like Add, Remove and Count instead of AddItem, Delete or NumberOfItems.

K1510 Avoid short names or names that can be mistaken with other names 

Although technically correct, the following statement can be quite confusing.

bool b001 = (lo == l0) ? (I1 == 11) : (lOl != 101);

K2511 Properly name properties 

* Do name properties with nouns, noun phrases, or occasionally adjective phrases.
* Do name Boolean properties with an affirmative phrase. E.g. CanSeek instead of CantSeek.
* Consider prefixing Boolean properties with Is, Has, Can, Allows, or Supports.
* Consider giving a property the same name as its type. When you have a property that is strongly typed to an enumeration, the name of the property can be the same as the name of the enumeration. For example, if you have an enumeration named CacheLevel, a property that returns one of its values can also be named CacheLevel.

K2512 Name methods using verb-object pair 

Name methods using a verb-object pair such as ShowDialog. A good name should give a hint on the what of a member, and if possible, the why. Also, don’t include And in the name of the method. It implies that the method is doing more than one thing, which violates the single responsibility principle explained in [K1203](#K1203).

K3513 Name namespaces using names, layers, verbs and features 

For instance, the following namespaces are good examples of that guideline.

Keous.Commerce.Web

NHibernate.Extensibility

Microsoft.ServiceModel.WebApi

Microsoft.VisualStudio.Debugging

FluentAssertion.Primitives

Keous.Extensions

**Note** Never allow namespaces to contain the name of a type, but a noun in its plural form, e.g. Collections, is usually okay.

K2514 Use a verb or verb phrase to name an event 

Name events with a verb or a verb phrase. For example: Click, Deleted, Closing, Minimizing, and Arriving. For example, the declaration of the Search event may look like this:

public event EventHandler<SearchArgs> Search;

K3515 Use -ing and -ed to express pre-events and post-events 

For example, a close event that is raised before a window is closed would be called Closing and one that is raised after the window is closed would be called Closed. Don’t use Before or After prefixes or suffixes to indicate pre and post events.

Suppose you want to define events related to the deletion process of an object. Avoid defining the Deleting and Deleted events as BeginDelete and EndDelete. Define those events as follows:

* Deleting: Occurs just before the object is getting deleted
* Delete: Occurs when the object needs to be deleted by the event handler.
* Deleted: Occurs when the object is already deleted.

K3516 Prefix an event handler with On 

It is good practice to prefix the method that handles an event with On. For example, a method that handles the Closing event could be named OnClosing.

K3517 Use an underscore for irrelevant lambda parameters 

If you use a lambda statement, for instance, to subscribe to an event, and the actual parameters of the event are irrelevant, use the following convention to make that more explicit.

button.Click += (\_, \_\_) => HandleClick();

K2518 Group extension methods in a class suffixed with Extensions 

If the name of an extension method conflicts with another member or extension method, you must prefix the call with the class name. Having them in a dedicated class with the Extensions suffix improves readability.

K2519 Postfix asynchronous methods with Async or TaskAsync 

The general convention for methods that return Task or Task<TResult> is to post-fix them with Async, but if such a method already exist, use TaskAsync instead.

* 1. Performance Guidelines

K3600 Consider using Any() to determine whether an IEnumerable<T> is empty 

When a method or other member returns an IEnumerable<T> or other collection class that does not expose a Count property, use the Any() extension method rather than Count() to determine whether the collection contains items. If you do use Count(), you risk that iterating over the entire collection might have a significant impact (such as when it really is an IQueryable<T> to a persistent store).

**Note** If you return an IEnumerable<T> to prevent editing from outside the owner as explained in [K2205](#K2205) and you’re developing in .NET 4.5 or higher, consider the new read-only classes.

K3601 Only use async for low-intensive long-running activities 

The usage of async won’t automagically run something on a worker thread like Task.Run does. It just adds the necessary logic to allow releasing the current thread and marshal the result back on that same thread if a long-running asynchronies operation has completed. In other words, use async only for I/O bound operations.

K3602 Prefer Task.Run for CPU intensive activities 

If you do need to execute a CPU bound operation, use Task.Run to offload the work to a thread from the Thread Pool. Just don’t forget that you have to marshal the result back to your main thread manually.

K2603 Beware of mixing up await/async with Task.Wait 

await will not block the current thread but simply instruct to compiler to generate a state-machine. However, Task.Wait will block the thread and may even cause dead-locks (see [K2604](#K2604)).

K2604 Beware of async/await deadlocks in single-threaded environments 

Consider the following asynchronous method:

private async Task<string> GetDataAsync()

{

var result = await MyWebService.GetDataAsync();

return result.ToString();

}

Now when an ASP.NET MVC controller action does this:

public ActionResult ActionAsync()

{

var data = GetDataAsync().Result;

return View(data);

}

You’ll end up with a deadlock. Why? Because the Result property getter will block until the async operation has completed, but since an async method will automatically marshal the result back to the original thread and ASP.NET uses a single-threaded synchronization context, they’ll be waiting on each other. A similar problem can also happen on WPF, Silverlight or a Windows Store C#/XAML app. Read more about this [here](http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115163.aspx).

* 1. Framework Guidelines

K1700 Use C# type aliases instead of the types from the System namespace 

For instance, use object instead of Object, string instead of String, and int instead of Int32. These aliases have been introduced to make the primitive types a first class citizen of the C# language so use them accordingly.

K2701 Properly name properties, variables or fields referring to localized resources 

The guidelines in this topic apply to localizable resources such as error messages and menu text.

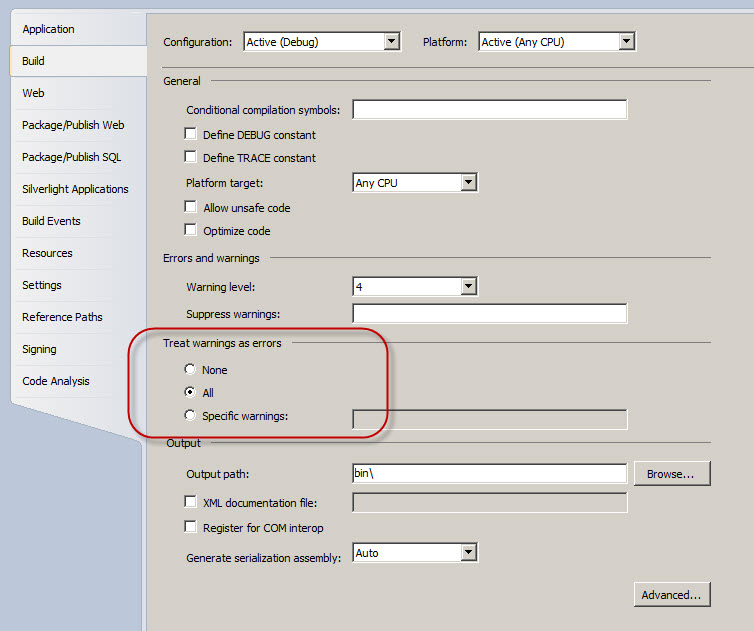
* Use Pascal casing in resource keys.
* Provide descriptive rather than short identifiers. Keep them concise where possible, but don’t sacrifice readability.
* Use only alphanumeric characters in naming resources.

K3702 Don’t hardcode strings that change based on the deployment 

Examples include connection strings, server addresses, etc. Use Resources, the ConnectionStrings property of the ConfigurationManager class, or the Settings class generated by Visual Studio. Maintain the actual values into the app.config or web.config (and most definitely not in a custom configuration store).

K1703 Build with the highest warning level and treat warnings as errors

Configure the development environment to use **Warning Level 4** for the C# compiler, and enable the option **Treat warnings as errors**. This allows the compiler to enforce the highest possible code quality.



K3704 Properly fill the attributes of the AssemblyInfo.cs file 

Ensure that the attributes for the company name, description, copyright statement, version, etc. are filled. One way to ensure that version and other fields that are common to all assemblies have the same values, is to move the corresponding attributes out of the AssemblyInfo.cs into a SolutionInfo.cs file that is shared by all projects within the solution.

K3705 Avoid LINQ for simple expressions 

Rather than

var query = from item in items where item.Length > 0;

Prefer using the extension methods from the System.Linq namespace.

var query = items.Where(i => i.Length > 0);

Since LINQ queries should be written out over multiple lines for readability, the second example is a bit more readable.

K2706 Use Lambda expressions instead of delegates 

Lambda expressions provide a much more elegant alternative for anonymous delegates. So instead of

Customer c = Array.Find(customers, delegate(Customer c)

{

return c.Name == “Tom”;

});

use a Lambda expression:

Customer c = Array.Find(customers, c => c.Name == “Tom”);

Or even better

Customer customer = customers.Where(c => c.Name == “Tom”);

K1707 Only use the dynamic keyword when talking to a dynamic object 

The dynamic keyword has been introduced for working with dynamic languages. Using it introduces a serious performance bottleneck because the compiler has to generate some complex Reflection code.

Use it only for calling methods or members of a dynamically created instance (using the Activator) class as an alternative to Type.GetProperty() and Type.GetMethod(), or for working with COM Interop types.

K1708 Favor async/await over the Task 

Using the new C# 5.0 keywords results in code that can still be read sequentially and also improves maintainability a lot, even if you need to chain multiple asynchronous operations. For example, rather than defining your method like this:

public Task<Data> GetDataAsync()

{

return MyWebService.FetchDataAsync()

.ContinueWith(t => new Data (t.Result));

}

define it like this:

public async Task<Data> GetDataAsync()

{

var result = await MyWebService.FetchDataAsync();

return new Data (result);

}

K1709 Use string.Empty instead of "" 

public void SomeMethod2()

{

    //Avoid

    string name = "";

    //Correct

    string name = string.Empty;

}

K1710 Do not use the this reference 

**Exception** Unless invoking another constructor from within a constructor:

//Example of proper use of ’this’

public class MyClass

{

    public MyClass(string message)

    { }

    public MyClass() : this("Hello")

    { }

}

K1711 Do not use the base keyword to access base class members 

**Exception** Unless you wish to resolve a conflict with a subclasses member of the same name or when invoking a base class constructor.

* 1. Documentation Guidelines

K2800 Avoid inline comments 

Putting in comments is not recommended since the code must be self-documented and clear. Comments usually only add noises to the code.

If you feel the need to explain a block of code using a comment, consider replacing that block with a method having a clear name.

**Note:** The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

K1801 Only write comments to explain complex algorithms or decisions 

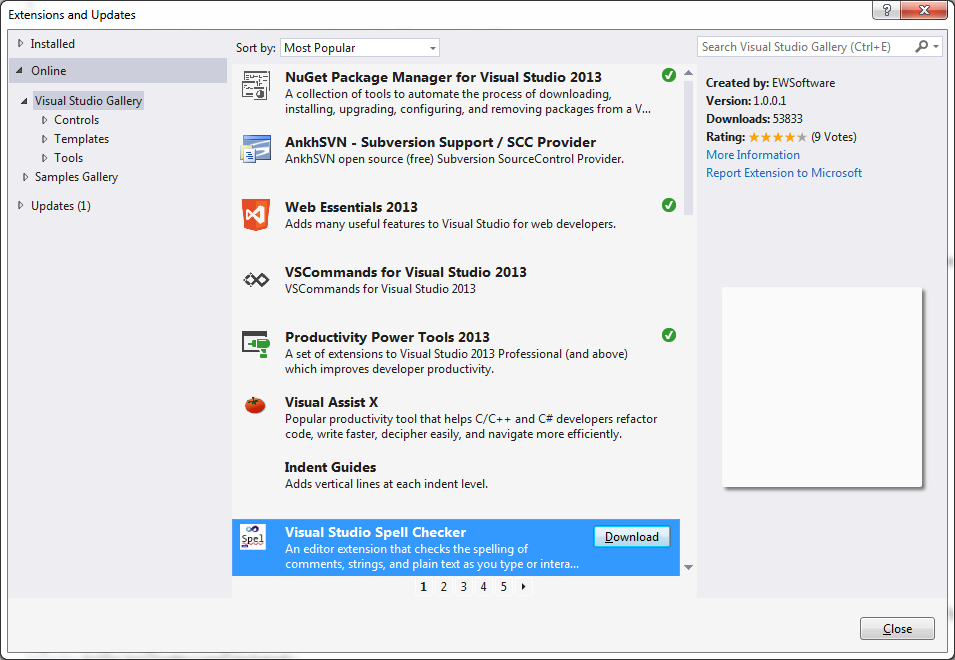
Try to focus comments on the *why* and *what* of a code block and not the *how*. Avoid explaining the statements in words, but instead help the reader understand why you chose a certain solution or algorithm and what you are trying to achieve. If applicable, also mention that you chose an alternative solution because you ran into a problem with the obvious solution.

K3802 Don’t use comments for tracking work to be done later 

Annotating a block of code or some work to be done using a TODO or similar comment may seem a reasonable way of tracking work-to-be-done. But in reality, nobody really searches for comments like that. Use a work item tracking system such as Team Foundation Server to keep track of left overs.

K1803 All comments must pass spell checking 

Misspelled comments indicate sloppy development. Some editor extensions like Visual Studio Spell Checker can checks the spelling of comments as you type them.



* 1. Layout Guidelines

K1900 Use a common layout 

* Keep the length of each line under 130 characters.
* Always put opening and closing braces on a new line. So Open brace “{” appears at the beginning of the next line indented to match the declaration statement and closing brace “}” starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the “}” should appear immediately after a whitespace before the “{“. Example:

class Sample : ISample

{

private int \_ivar1;

private int \_ivar2;

Sample(int ivar1, int ivar2)

{

\_ivar1 = ivar1;

\_ivar2 = ivar2;

}

private int EmptyMethod() { }

...

}

* Four spaces should be used as the unit of indentation so use an indentation of 4 whitespaces, and don’t use tabs.
* Each line should contain at most one statement. Example:

argv++; argc--; // AVOID!

* A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

return;

return myDisk.size();

return (size ? size : defaultSize);

K1901 Wrapping lines rules 

* The opening brace should be at the next line that follows the compound statement[[1]](#footnote-1); the closing brace should begin a line and be indented to the beginning of the opening brace.
* The enclosed statements should be indented one more level than the compound statement.
* Always indent object Initializers and initialize each property on a new line, so use a format like this:

var dto = new ConsumerDto()

{

Id = 123,

Name = “Microsoft”,

PartnerShip = PartnerShip.Gold,

}

* Always indent lambda statements and use a format like this:

methodThatTakesAnAction.Do(x =>

{

// do something like this

}

* Put the entire LINQ statement on one line, or start each keyword at the same indentation, like this:

var query = from product in products where product.Price > 10 select product;

or

var query =

from product in products

where product.Price > 10

select product;

* Start the LINQ statement with all the from expressions and don’t interweave them with where restrictions.

When an expression will not fit on a single line, break it according to these general principles:

* Break after a comma.
* Break before an operator.
* Prefer higher-level breaks to lower-level breaks.
* Align the new line with the beginning of the expression at the same level on the previous line.
* If the above rules lead to confusing code or to code that’s squished up against the right margin, just indent 8 spaces instead.

Here is one example of a breaking method call:

function(longExpression1, longExpression2, longExpression3,

longExpression4, longExpression5);

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

longName1 = longName2 \* (longName3 + longName4 - longName5)

+ 4 \* longname6; // PREFER

longName1 = longName2 \* (longName3 + longName4

- longName5) + 4 \* longname6; // AVOID

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

//CONVENTIONAL INDENTATION

SomeMethod(int anArg, object anotherArg, string yetAnotherArg,

object andStillAnother)

{

...

}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS

private static virtual HorkingLongMethodName(int anArg,

Object anotherArg, String yetAnotherArg,

Object andStillAnother)

{

...

}

Here are three acceptable ways to format ternary expressions:

alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta

: gamma;

alpha = (aLongBooleanExpression)

? beta

: gamma;

* In case of method chaining and fluent APIs, break and start the next method call on the next line at the same position as the previous call. Example:

Item item = \_context.Items

                 .Include(i => i.ItemInclusions)

                .Include(i => i.ItemSubstitutions)

                 .FirstOrDefault(i => i.ItemId == itemId);

If this conventional indentation shifts the second and third lines to the far right, instead it indents only 8 spaces. Example:

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS

IEnumerable<HorkingLongObjectName> items = \_context.HorkingLongObjects

.Include(i => i.ItemInclusions)

.Include(i => i.ItemSubstitutions)

  .FirstOrDefault(i => i.ItemId == itemId);

K1902 Blank Spaces rules 

Blank spaces should be used in the following circumstances:

* A keyword followed by a parenthesis should be separated by a space. Example:

while (true)

{

...

}

Following the same rule, keep one whitespace between keywords like if and the expression, but don’t add whitespaces after “(” and before “)” such as:

if (condition == null).

* A blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.
* A blank space should appear after commas in argument lists.
* All binary operators except “**.”** should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement (“--”) from their operands. Example:

a += c + d;

a = (a + b) / (c \* d);

while (d++ = s++)

{

n++;

}

prints("size is " + foo + "\n");

* The expressions in a for statement should be separated by blank spaces. Example:

for (expr1; expr2; expr3)

* Casts should be followed by a blank space. Examples:

MyMethod((byte) aNum, (object) x);

MyFunc((int) (cp + 5), ((int) (i + 3)) + 1);

K1903 Blank Lines rules 

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

* Between sections of a source file
* Between class and interface definitions

One blank line should always be used in the following circumstances:

* Between methods
* Between the local variables in a method and its first statement
* Before a block or single-line comment
* Between logical sections inside a method to improve readability

Add an empty line between multi-line statements, between members, after the closing parentheses and between unrelated code blocks.

K1904 Order and group namespaces according the company 

// Microsoft namespaces are first

using System;

using System.Collections;

using System.XML;

// Then any other namespaces in alphabetic order

using Keous.Business;

using Keous.Common;

using Telerik.WebControls;

using Telerik.Ajax;

K1905 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 you are reading a book. This prevents readers from having to browse up and down through the code file.

1. Private fields and constants
2. Public constants
3. Public read-only static fields
4. Constructors and the Finalizer
5. Events
6. Public Properties
7. Public Methods
8. Private Methods

K1906 #regions are forbidden 

Regions are a sign of code smell and forbidden. If you find yourself need to create regions in a class then your class is probably doing too much and need to be refactored. See [K1100](#K1100)

1. Compound statements are statements that contain lists of statements enclosed in braces

   “{ statements }”. [↑](#footnote-ref-1)