

Secure Coding Standards and Guidelines

Table of Contents

[2 Introduction 3](#_Toc423424123)

[2.1 What is this? 3](#_Toc423424124)

[2.2 Why are the Guidelines necessary? 3](#_Toc423424125)

[2.3 Basic Principles 3](#_Toc423424126)

[2.4 How do I get started? 4](#_Toc423424127)

[2.5 Key Design Principles 4](#_Toc423424128)

[3 Security Guidelines 5](#_Toc423424129)

[3.1 Input Validation 5](#_Toc423424130)

[3.1.1 Methods to accomplish this 6](#_Toc423424131)

[3.1.2 Other recommendations 6](#_Toc423424132)

[3.1.3 What do we do with things that fail validation? 7](#_Toc423424133)

[4 Class Design Guidelines 7](#_Toc423424134)

[5 Member Design Guidelines 11](#_Toc423424135)

[6 Miscellaneous Design Guidelines 12](#_Toc423424136)

[7 Maintainability Guidelines 15](#_Toc423424137)

[8 Naming Guidelines 26](#_Toc423424138)

[9 Performance Guidelines 30](#_Toc423424139)

[10 Framework Guidelines 31](#_Toc423424140)

[11 Documentation Guidelines 33](#_Toc423424141)

# Introduction

## What is this?

This document attempts to provide useful and pragmatic guidelines for programming at Software companies. Coding guidelines, or coding standards if you will, are documents consisting of rules and recommendations for coding in enterprise systems. They deal with code layout, naming guidelines, the proper use of frameworks, tips on writing useful comments and XML documentation, and often also include guidance on proper object-oriented design. Also as a primary focus this guide aims to foster Security consciousness and reinforce good design principles.

## Why are the Guidelines necessary?

To set development expectations for secure, maintainable and scalable systems by:

* Thinking defensively when it comes to internal and external threats to systems and sensitive data.
* Using proven methods and principles to ensure maintainability and scalability.

Complying with coding guidelines will help to ensure a high level of code quality for secure, risk managed systems that are agile as business needs change.

## Basic Principles

There are basic principles that over time have proven to be the cornerstone in modern development. These include:

* Separation of Concerns. Divide features with as little overlap as possible with an emphasis on decoupling and minimization of interaction points.
* Principle of Least Knowledge. An object should not know about the internal details of other objects.
* The Principle of Least Surprise (or Astonishment), which means that you should choose a solution that does include things people generally might not understand.
* Keep It Simple Stupid (a.k.a. KISS). Keep design simple and small. Complex designs are more likely to be misunderstood and are potentially more error prone.
* 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 in the future.
* Don’t Repeat Yourself (a.k.a. DRY), which requires you to rigorously remove duplication in your code base.

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

## How do I get started?

* Ask all developers to carefully read this document at least once. This will give them a sense of the kind of guidelines the document contains.
* Include the most critical coding guidelines on your Project Checklist and verify the remainder as part of your Peer Review.

## Key Design Principles

The key design principles if properly followed will help you create quality software using proven methods, minimize costs and maintenance requirements, promote scalability and help minimize security risks and exposure.

* Separation of concerns.
  + Divide applications into distinct features with minimal overlap.
  + Minimize communication surface areas
  + Decouple as much as possible
* Single responsibility principle.
  + Each module should address a single specific feature or function.
* Principle of least knowledge.
  + An object shouldn’t need to know the internals of another object.
* DRY principle (Don’t Repeat Yourself).
* Least Privilege Principle.
  + Only give the component or user the minimum access it/they require to accomplish the task.

# Security Guidelines

In the modern landscape of Information Systems it is imperative that every system design begin with an architecture that minimizes the attack surface area for both internal and external threats. An overarching concept is to keep it simple to reduce complexities for developers while still remaining resilient to attacks should they happen. A multi-layered approach that centers on secure defaults and **Principle of Least Privilege**, has been found to be very effective in industry studies and is an accepted best-practice.

Some design considerations:

* Where could attacks come from (internal/external/Trojans/worms/criminal entities)
* What is exposed, what is my attack surface? Only expose what is needed.
* If I was evil what could I do, what system features/processes could I exploit?
* Establish secure defaults: trusted = no, is authorized = false etc.
* Principle of Least Privilege: only give permission necessary to perform the task.
* Defense In Depth: Multiple layers of security controls i.e. at various levels from data, business, software and hardware.
* Separation of roles and job duties to ensure the chances of fraudulent, malicious and or abusive practices are mitigated.

How do we accomplish these design goals?

## Input Validation

If I was evil what tricks could I use to exploit the systems? Could I cause a buffer overflow? Can I use a SQL injection attack? Can I use a cross site scripting attack? Make sure the data is thoroughly validated at the client, server, module interfaces, then again through certain security controls at the database level.

### Methods to accomplish this

1. Html encoding strings to be shown on web pages/views.

Example:

public string GetEncodedText()

{

return HttpContext.Current.Server.HtmlEncode("<script>unsafe</script>");

}

1. A form post variable that is intended to be an integer should be checked with the System.Int32.TryParse method to verify the variable really is an integer.
2. Validate DataTypes
3. Validate Ranges
4. Validate Required Fields
5. Validate Min and Max lengths
6. Validate at unit boundaries that the integrity is not compromised.
7. Validate that we have an active/authenticated user or source.
8. Do not disable validateRequest in web.config or pages, this offers protection against Cross Site Scripting.
9. .Net 4.5 Framework offers AntiXssEncoder library for encoding input and is a recommended solution.

### Other recommendations

1. Always use parameterized queries: No Exceptions
2. Use enums when possible to user data from user is correct.
3. Use Enum.IsDefined to ensure the validity of the enum.
4. Apply Least Privilege Principle when querying database.
5. Use encryption for sensitive information.
6. Encrypt sensitive parts of web.configs using aspnet\_regiis -pe
7. Use HTTPS.
8. Turn off tracing.
9. Turn off debugging.
10. Use ErrorModule(standard error handling class)
11. Use ValidateAntiForgeryTokenfor MVC.
12. Remove the version header:

<httpRuntime enableVersionHeader=”false” />

1. Remove the Server Header:

HttpContext.Current.Response.Headers.Remove(“Server”)

1. Use Forms Authentication
2. Use cookies for persistence
3. User should always default to Not Authorized and be explicitly permitted to a resource.
4. Decorate controller methods with security attributes.
5. Always heed compiler warnings.
6. Always check developer tools for web facing apps to ensure there are no errors being swallowed by the browser.

### What do we do with things that fail validation?

In general, it depends on where it occurs and what it is. If for instance a page/view tries to submit <script>unsafe</script> we would not want to simply swallow that error and not report on it. Instead it should be redirected to an appropriate error handler that logs the issue, notifies the vested parties and prevents the user from entering this type of data. At a minimum all errors should be logged and notifications should occur. For production sites in IIS we have an ErrorModule that handles these logging and notification activities which is monitored and analyzed by vested parties.

Any specific vulnerabilities identified should be catalogued and added to developer unit testing as well as the Quality Assurance and User Acceptance team’s checklists for thorough coverage and mitigation of risk. This should be conveyed to management in order to properly assess the impact on current development efforts as well as existing code bases.

Any weaknesses discovered in testing or otherwise should be reported to management who will perform risk assessment, document the vulnerability in the aforementioned checklists and slate the task for remediation.

# Class Design Guidelines

A class or interface should have a single purpose. A class or interface should have a single purpose within the system it 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**, one of the **S.O.L.I.D.** principles.

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

Tip: B Use Design Patterns 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 its usage by making it immutable.

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.

An interface should be small and focused. Interfaces should have a clear name explaining 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**.

Use an interface to support multiple implementations, not a base class. If you want to expose an extension point to your class, expose it as an interface rather than a base class. It doesn’t force the users of that extension point to derive their implementations from a base-class that might have undesired behavior. It improves testability and allows them to use their own implementation. However, for their convenience you may implement an (abstract) default implementation that can serve as a starting point.

Use an interface to decouple classes from each other. Interfaces are a very effective mechanism for decoupling classes from each other and:

* 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 in a unit test.
* Using a dependency injection framework you can centralize the choice of which class is going to be used whenever a specific interface is requested.

Mark classes that only contain static members as static. The advantage of using a static class is that the compiler can make sure that no instance members are accidentally defined. The compiler will guarantee that instances of this class cannot be created and hence, relieves you of creating a private constructor such as was required in C# 1.0. Use a static class to contain methods that are not associated with a particular instance. For example:

public static class EuroConversion

{

public static Decimal FromUSD(Decimal inUsd) { ... }

public static Decimal ToUSD(Decimal inEuro) { ... }

}

Don’t hide inherited members with the new keyword. Not only does the new keyword break Polymorphism, 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 the following behavior which is not something you 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 the base class or through the derived class.

It should be possible to treat a derived object as if it were a base class object. In other words, it should be possible 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. This rule is also known as the **Liskov Substitution Principle**, one of the **S.O.L.I.D.** principles.

Don’t refer to derived classes from the base class. Having dependencies between a base class and its derivatives goes against proper object oriented design and prevents other developers from adding new derived classes without having to modify the base class.

Avoid exposing the objects an object depends on. If you find yourself writing code like this then you might be violating the **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 pattern does seem to violate this rule, but is in reality something else. It simply returns 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, then it is not considered a violation.

Avoid bidirectional dependencies. Having bidirectional dependencies between Classes 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** tend to occasionally involve bidirectional associations that model real-life associations. In those cases, make sure they are really necessary, but if they are, keep them in.

Classes should have state and behavior. The only exception to this rule are classes that are used to transfer data over a communication channel, also called Data Transfer Objects, or a class that wraps several parameters of a method. 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. Use the principles of object-orientation explained in this section and move the logic as close to the data it applies to.

# Member Design Guidelines

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, and vice versa.

Use a method instead of a property: If the operation is orders of magnitude slower than setting a field value. If the operation is a conversion, such as the Object.ToString method. If the operation returns a different result each time it is called, even if the parameters 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. Note: populating an internal cache or implementing lazy loading is a good exception.

Avoid 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.

A method or property should do only one thing. Similarly to what was previously stated, a method or property should do exactly one thing, and one thing only.

Use a public static readonly field to define predefined value objects. For example, consider a Color struct that stores a color internally as red, green, and blue components and this class has a constructor taking a numeric representation. This class may expose several predefined colors like this.

public struct Color

{

public static readonly Color Red = new Color(0xFF0000);

public static readonly Color Black = new Color(0x000000);

public static readonly Color White = new Color(0xFFFFFF);

public Color(int redGreenBlue)

{

// implementation

}

}

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. It’s very difficult to unit test a class like that without using a very intelligent and intrusive mocking framework such as TypeMock Isolator. Many see the HttpContext class as a source for a lot of ugly code. In fact, the testing guideline **Isolate the Ugly Stuff** often refers to this class.

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>.

String, list and collection properties should never return a null reference. Returning null can be unexpected by the caller. Always return an empty array and an empty string instead of a null reference. This also prevents cluttering your code base with additional checks for null.

Consider replacing properties using primitive types to use rich value objects. 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 creating 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.

# Miscellaneous Design Guidelines

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 situations occurs.

Provide a rich and meaningful exception message text. The message should explain the cause of the exception and clearly describe what needs to be done to avoid the exception. However, the security implications of the message should also be weighed when the message is returned to the user. Be mindful that an attacker originating internally or externally may try to exploit this and use the information returned.

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.

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.

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 {};

Use a protected virtual method to raise 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.

Important: 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.

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** and **Model-View-ViewModel** patterns.

Don’t pass null as the sender parameter 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 parameter should be null.

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

class SomeClass

{}

// Don't

class MyClass<T>

{

void SomeMethod(T t)

{

object temp = t;

SomeClass obj = (SomeClass) temp;

}

}

// Do

class MyClass<T> where T : SomeClass

{

void SomeMethod(T t)

{

SomeClass obj = t;

}

}

Don’t add extension methods to the same namespace as the extended class. Even though it may seem convenient to add extension methods related to the String class in the System namespace, this may cause conflicts with future versions of the .NET Framework.

Evaluate the result of a LINQ expression before returning it. Consider the following code snippet

public IEnumerable<GoldMember> GetGoldMemberCustomers()

{

const decimal GoldMemberThresholdInEuro = 1000000;

var 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.

# Maintainability Guidelines

Method names, variables etc. should be self-documenting: the naming conventions used should enable use of the code without any prior knowledge. This will make the source code easier to read and allow the developer to clearly understand the intent of the code without having to consult secondary documentation.

Methods should not exceed 7 statements. A method that requires more than 7 statements is 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.

Make all members private and types internal by default. To make a more conscious decision on which members to make available to other classes, explicitly set the scope of all new members to private and that of a new type to internal. Then carefully decide what to expose as a public member or type.

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.

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

All DLLs should be named according to the pattern <Project>.<Component>.dll where <Project> refers to your project’s name and <Component> contains one or more dot-separated clauses. For example TradePro.Web.Controls.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, TradePro.Consulting.Core.dll.

Name a source file to the type it contains. Also, use Pascal casing for naming the file and don’t use underscores.

Limit the contents of a source code file to one type. Exception Nested types should, for obvious reasons, be part of the same file.

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

{...}

Use using statements instead of fully qualified type names. Limit usage of fully qualified type names to prevent name clashing. For example:

Don’t:

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

Do:

using System.Collections.Generic;

var list = new List<string>();

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

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

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. For example, don’t:

public class SomeSpecialContainer

{

public const int MaxItems = 32;

public const int HighWaterMark = 24; // at 75%

}

However, do:

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.

Only use var when the type is very obvious. Only use var as the result of a LINQ query, or if the type is very obvious from the same statement and using it would improve readability.

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

Do:

var q = from order in orders where order.Items > 10 and order.TotalValue > 1000;

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

var list = new ReadOnlyCollection<string>();

In all of three above examples it is clear what type to expect.

Initialize variables at the point of declaration. 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. Exception: if a variable is set in multiple places in the code block then consider defining the var at the top of the method.

Favor Object and Collection Initializers over separate statements.

Instead of

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

startInfo.StandardOutput = Console.Output;

startInfo.UseShellExecute = true;

Use Object Initializers

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 Initializers

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

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

Use an enumeration instead of a list of strings if the list of values is finite. If a variable can have a limited set of constant string values, use an enumeration for defining the valid values. Using the enumeration instead of a constant string allows compile-time checking and prevents typos.

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.

}

}

Don’t use nested loops in a method. 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.

Add a block after all flow control keywords, even if it is empty. 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();

}

}

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);

}

}

Finish every if-else-if statement with an else-part. The intention of this rule, which applies to else-if constructs, is the same as the previous rule. 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.

}

}

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 other guidelines then multiple return statements may actually improve readability over some central Boolean flag that is updated at various points.

Don’t use selection statements instead of a simple assignment or initialization. Express your intentions directly. For example, rather than

bool pos;

if (val > 0)

{

pos = true;

}

else

{

pos = false;

}

Write:

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

ASI-1546 Prefer conditional statements instead of simple if-else constructs.

For example, instead of:

string result;

if (someString != null)

{

result = someString;

}

else

{

result = “Unavailable”;

}

return result;

Write:

return someString ?? “Unavailable”;

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.

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 arguments. 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.

Only use optional parameters to replace overloads. The only valid reason for using C# 4.0’s optional parameters is to replace the example from the previous rule 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 previously stated rules, 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 property.

Avoid using named parameters. C# 4.0’s named parameters have been introduced to make it easier to call COM components that are known for offering tons of optional parameters. If you need named parameters 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 parameters 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)

);

Avoid methods 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 parameters such as explained in the **Specification 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.

Don’t use ref or out parameters. Ref and out parameters make code less understandable and therefore may introduce bugs. Prefer returning compound objects instead.

Avoid methods that take a bool flag. A flag parameter based on a bool is not self-explanatory. 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.

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.

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.

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?

Consider abstracting an external dependency or 3rd party component. If your code relies on some kind of external class, service or UI control, consider wrapping that dependency in a lightweight wrapper that only exposes the members that are really used. Such a wrapper smoothens the changes required when replacing that dependency with another, but can also be used to hide any undesired behavior or bugs that you don’t have influence on. As stated earlier in this document, the intent is to minimize the communication points and attack surfaces.

# Naming Guidelines

Use proper US-English. All identifiers should be named using words from the American English language. All names should align themselves with the self-documenting or self-describing conventions.

* Choose easily readable identifier 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 identifiers 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.

Use proper casing for members.

|  |  |  |
| --- | --- | --- |
| Identifier | 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 |
|  |  |  |

Don’t include numbers in identifiers. Numbers in names of fields, variables or members are very rarely needed.

Don’t prefix member fields: Don’t use a prefix for field names. 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: \_currentUser, mUserName, m\_loginTime.

Don’t use abbreviations: Don’t use abbreviations or acronyms as parts of identifier names. 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.

Name an identifier according its meaning and not its type.

* Use semantically interesting names rather than language-specific keywords for type 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 an array or collection should have a plural name.

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.

Name generic type parameters with descriptive names. The following guidelines cover selecting the correct names for generic type parameters.

* Name generic type parameters with descriptive names, unless a single-letter name is completely self-explanatory and a descriptive name would not add value. For example: IDictionary is an example of an interface that follows this guideline.
* Use the letter T as the type parameter name for types with one single-letter type parameter.
* Prefix descriptive type parameter names with the letter T.
* Consider indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to ISession may be called TSession.

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() {...}

}

Name members similarly to members of .NET Framework classes. Stay close to the naming philosophy of the .NET Framework. Developers are already accustomed to the naming patterns .NET Framework classes use, so following this 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.

Avoid short names or names that can be mistaken with other names. Although technically allowed, the following statement is quite confusing.

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

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.

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**.

interface IEmployeeRepository

{

Employee[] First() { } // Wrong: What does first mean? How many?

Employee[] GetFirstFive() {} // Better

Employee[] GetFiveMostRecent(){} // Best: self-describing

void Add(Employee employee) {} // Although not using verb-object pair;

// the type name is clear enough

}

Name namespaces according to a well-defined pattern. All namespaces should be named according to the pattern:

(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]

For instance: TradePro.Models.MainMenu.

Note: Don’t use the same name for a namespace and a type in that namespace. For example, don’t use Debug for a namespace name and also provide a class named Debug in the same namespace.

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 SearchEventHandler Search;

Use -ing and -ed to express pre-events and post-events. Give event names a concept of before and after, using the present and past tense. 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.

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.

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.

# Performance Guidelines

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).

# Framework Guidelines

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, Exception When referring to static members of those types, it is custom to use the full CLS name, e.g. Int32.Parse() instead of int.Parse().

Properly name identifiers 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.

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. Use web.config/app.config to define resources and settings that change or may change.

Build with the highest warning level. 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.

Avoid suppressing specific compiler warnings.

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.

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.

Use Lambda expressions instead of delegates.

Lambda expressions have been introduced in C# 3.0 and 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

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

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.

# Documentation Guidelines

Write comments and documentation in US English.

Use XML tags for documenting types and members.

Document all public types and members of types using the built-in XML comment functionality of Visual Studio. Documenting your code allows Visual Studio to pop-up the documentation when your class is used somewhere else. Furthermore, by properly documenting your classes, tools can generate professionally looking class documentation.

Write XML documentation with the caller in mind.

Write the documentation of your class with the class user in mind. Assume the user will not have access to the source code and try to explain how to get the most out of the functionality of your class.

Write MSDN-style documentation.

Following the MSDN on-line help style and word choice helps the developer to find its way through your documentation more easily.

Avoid inline comments.

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.

Don’t use /\* \*/ for comments.

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.

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.

10 Layout Guidelines

Use a common layout.

* Keep the length of each line under 130 characters.
* Use an indentation of 4 whitespaces, and don’t use Tabs
* Keep one whitespace between keywords like if and the expression, but don’t add whitespaces after (and before) such as: if (condition == null).
* Add a whitespace around operators, like +, -, ==, etc.
* Always succeed the keywords if, else, do, while, for and foreach, with opening and closing parentheses, even though the language does not require it.
* Always put opening and closing parentheses on a new line.
* 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;

* Add braces around every comparison condition, but don’t add braces around a singular condition. For example:

if (!string.IsNullOrEmpty(str) && (str != “new”))

* Add an empty line after the closing parentheses, between unrelated code blocks and around the #region keyword.

Order and group namespaces alphabetically.

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 (in a region)

2. Public constants

3. Public read-only static fields

4. Constructors and the Finalizer

5. Events

6. Properties

7. Other members grouped in a functional manner.

8. Private properties

Private and protected methods should be placed behind the public member in calling order.

Be reluctant with #regions.

Regions can be helpful, but can also hide the main purpose of a class. Therefore, use #regions only for:

* Private fields and constants (preferably in a Private Definitions region).
* Nested classes
* Interface implementations (only if the interface is not the main purpose of that class)