Coding Guidelines
for C# 3.0 and C# 4.0

July 2011
Dennis Doomen

www.csharpcodingguidelines.com
www.avivasolutions.nl
www.dennisdoomen.net

1 Introduction

1.1 What is this?

This document attempts to provide useful and pragmatic guidelines for programming in C# 3.0 and C# 4.0 that we at Aviva Solutions already use in our day-to-day work. Coding guidelines, or coding standards if you will, are documents consisting of rules and recommendations on the use of C# in enterprise systems. They deal with code layout, naming guidelines, the proper use of the .NET Framework, tips on writing useful comments and XML documentation, and often also include guidance on proper object-oriented design.

Visual Studio's Static Code Analysis (a.k.a. FxCop) and StyleCop can automatically verify a majority of those rules and recommendations by analyzing the compiled assemblies. You can configure to do that at compile time or as part of a continuous or daily build. This document adds additional rules and recommendations and its companion site, www.csharpcodingguidelines.com provides a list of Code Analysis rules that are applicable for Line-of-Business applications and frameworks.

1.2 Why are guidelines necessary?

Because not every developer

- is aware of the potential pitfalls of certain constructions in C#.
- is introduced into certain conventions when using the .NET Framework (e.g. `IDisposable`)
- 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.

Although complying with coding guidelines may seem to appear as undesired overhead or may limit creativity, this approach has already proven its value for many years. Also beware of the fact that not all coding guidelines have a clear rationale. Some of them are simply choices we made at Aviva Solutions.

1.3 Basic Principles

There are many unexpected things I run into during my work as a consultant, each deserving at least one guideline. Unfortunately, I'm still trying 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 colleague 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 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.

1.4 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.
- Make sure there are always a few hard copies of the Quick Reference close at hand.
- Include the most critical coding guidelines on your Project Checklist and verify the remainder as part of your Peer Review.
- Decide which CA rules are applicable for your project and write these down somewhere, such as your TFS team site, or create a custom Visual Studio 2010 Rule Set. The companion site offers Visual Studio 2010 rule sets for both Line-of-Business applications and more generic code like frameworks and class libraries.
- Add a custom Code Analysis Dictionary containing your domain- or company-specific terms, names and concepts. If you don't, Static Analysis will report warnings for (parts of) phrases that are not part of its internal dictionary.
- Configure Visual Studio to verify the selected CA rules as part of the Release build. Then they won't interfere with normal developing and debugging activities, but still can be run by switching to the Release configuration.
- Add the verification of the CA rules part of your Continuous or Daily Build.
- Add a rule to your team that the first one who comes in the office in the morning will check the build for CA warning and will make sure somebody (not necessarily himself) solves it as soon as possible.
- Add an item to your project checklist to make sure all new code is verified against CA violations, or use the corresponding Check-in Policy if you want to prevent any code from violating CA rules at all.
- ReSharper has an intelligent code inspection engine that, with some configuration, already supports many aspects of the Coding Guidelines. It will automatically highlight any code that does not match the rules for naming members (e.g. Pascal or Camel casing), detect dead code, and many other things. One click of the mouse button (or the corresponding keyboard shortcut) is usually enough to fix it.
- ReSharper also has a File Structure window that shows an overview of the members of your class or interface and allows you to easily rearrange them using a simple drag-and-drop action.
- Using GhostDoc you can generate XML comments for any member using a keyboard shortcut. The beauty of it, is that it closely follows the MSDN-style of documentation. However, you have to be careful not to misuse this tool, and use it as a starter only.
- Consider reading the book Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin. It provides excellent guidance on writing elegant and simple code that is easy to maintain and to extend. His ideas have evolved into a new quality standard maintained by many well-known community members, and had a lot of influence on this document.

1.5 Why did you create it?

The idea started in 2002 when Vic Hartog (Philips Medical Systems) and I were assigned the task of writing up a coding standard for C# 1.0. Since then, I've regularly added, removed and changed rules based on experiences, feedback from the community and new tooling support such as offered by Visual Studio 2010.

Additionally, after reading Robert C. Martin's book Clean Code: A Handbook of Agile Software Craftsmanship, I became a big fan of his ideas and decided to include some of his smells and heuristics as guidelines. Notice though that this document is in no way a replacement for his book. I sincerely recommend that you read his book to gain a solid understanding of the rationale behind his recommendations.

I've also decided to include some design guidelines in addition to simple coding guidelines. They are too important to ignore and have a big influence in reaching high quality code.

1.6 Is this a coding standard?

The document does not state that projects must comply with these guidelines, neither does it say which guidelines are more important than others. However, we encourage projects to decide themselves what guidelines are important, what deviations a project will use, who is the consultant in case doubts arise, and what kind of layout must be used for source code. Obviously, you should make these decisions before starting the real coding work.

To help you in this decision, I've assigned a level of importance to each guideline:

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

❷ Strongly recommended guidelines

❸ Recommended guidelines that may not be applicable in all situations

In general, generated code should not need to comply with coding guidelines. However, if it is possible to modify the templates used for generation, try to make them generate code that complies as much as possible.

1.7 Feedback and disclaimer

This document has been compiled using many contributions from (former) colleagues, sources from the Internet, and many years of developing in C#. If you have questions, comments or suggestions, just let me know by sending me an email at dennis.doomen@avivasolutions.nl or tweet me at http://twitter.com/ddoomen. I will try to revise and republish this document with new insights, experiences and remarks on a regular basis. Notice though that it merely reflects my view on proper C# code so Aviva Solutions will not be liable for any direct or indirect damages caused by applying the guidelines of this document.

It is allowed to copy, adapt, and redistribute this document and its companion quick reference guide for non-commercial purposes or internal usage. However, you may not republish this document, or publish or distribute any adaptation of this document for commercial use without first obtaining express written approval from Aviva Solutions.

2 Class Design Guidelines

AV1000 A class or interface should have a single purpose ❶

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 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 it usage by making it immutable.

AV1001 Only create a constructor that returns a useful object ❸

There should be no need to set additional properties before the object can be used for whatever purpose it was designed.

AV1003 An interface should be small and focused ❷

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.

AV1004 Use an interface to support multiple implementations, not a base class ❸

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.

AV1005 Use an interface to decouple classes from each other ❷

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 which class is going to be used whenever a specific interface is requested.

AV1008 Mark classes that only contain static members as `static` ❸

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) { ... }
}
```

AV1010 Don't hide inherited members with the `new` keyword ❶

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.

AV1011 It should be possible to treat a derived object as if it were a base class object ❷

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.

AV1013 Don't refer to derived classes from the base class ❶

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.

AV1014 Avoid exposing the objects an object depends on ❷

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.

AV1020 Avoid bidirectional dependencies ❶

Having bidirectional dependencies between classes means that two classes know about each other public memers 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, I would make sure they are really necessary, but if they are, keep them in.

AV1025 Classes should have state and behavior ❶

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.

3 Member Design Guidelines

AV1100 Allow properties to be set in any order ❶

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.

AV1105 Use a method instead of a property ❸

- 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 that populating an internal cache or implementing lazy loading is a good exception.

AV1110 Avoid mutual exclusive properties ❶

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.

AV1115 A method or property should do only one thing ❶

Similarly to rule AV1000, a method or property should do exactly one thing, and one thing only.

AV1120 Use a `public static readonly` field to define predefined value objects ❸

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
   }
}
```

AV1125 Don't expose stateful objects through static members ❷

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.

AV1130 Return an IEnumerable<T> or ICollection<T> instead of a concrete collection class ❷

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>`.

AV1135 String, list and collection properties should never return a `null` reference ❶

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`.

AV1140 Consider replacing properties using primitive types to use rich value objects ❸

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.

4 Miscellaneous Design Guidelines

AV1200 Throw exceptions rather than returning some kind of status value ❷

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.

AV1202 Provide a rich and meaningful exception message text ❷

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

AV1205 Throw the most specific exception that is appropriate ❸

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

AV1210 Don't swallow errors by catching generic exceptions ❶

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.

AV1220 Always check an event handler delegate for `null` ❶

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 {};
```

AV1225 Use a protected virtual method to raise each event ❷

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.

AV1230 Consider providing property-changed events ❸

Consider providing events that are raised when certain properties are changed. Such an event should be named `*Property*Changed`, 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.

AV1235 Don't pass `null` as the sender parameter when raising an event ❶

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.

AV1240 Use generic constraints if applicable ❷

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;
   }
}
```

AV1245 Don't add extension methods to the same namespace as the extended class ❶

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.

AV1250 Evaluate the result of a LINQ expression before returning it ❶

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.

5 Maintainability Guidelines

AV1500 Methods should not exceed 7 statements ❶

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.

AV1501 Make all members private and types internal by default ❶

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.

AV1502 Avoid conditions with double negatives ❷

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.

AV1505 Name assemblies after their contained namespace ❸

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

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 `AvivaSolutions.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, `AvivaSolutions.Consulting.Core.dll`.

AV1506 Name a source file to the type it contains ❸

Also, use Pascal casing for naming the file and don't use underscores.

AV1507 Limit the contents of a source code file to one type ❸

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

AV1508 Name a source file to the logical function of the partial type ❸

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
{...}
```

AV1510 Use *using* statements instead of fully qualified type names ❸

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;
```

AV1515 Don't use "magic numbers" ❶

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%
}
```

but 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.

AV1520 Only use `var` when the type is very obvious ❶

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.

For a more detailed rationale about the advantages and disadvantages of using `var`, read Eric Lippert's Uses and misuses of implicit typing.

AV1521 Initialize variables at the point of declaration ❷

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.

AV1523 Favor Object and Collection Initializers over separate statements ❷

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" };
```

AV1525 Don't make explicit comparisons to `true` or `false` ❶

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
```

AV1526 Use an enumeration instead of a list of strings if the list of values is finite ❸

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.

AV1530 Don't change a loop variable inside a `for` or `foreach` loop ❷

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.
   }
}
```

AV1532 Don't use nested loops in a method ❷

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.

AV1535 Add a block after all flow control keywords, even if it is empty ❷

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();
   }
}
```

AV1536 Always add a default block after the last *case* in a *switch* statement ❶

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);
   }
}
```

AV1537 Finish every *if-else-if* statement with an *else*-part ❷

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.
   }
}
```

AV1540 Be reluctant with multiple return statements ❷

One entry, one exit is a sound principle and keeps control flow readable. However, if the method is very small and complies with guideline AV1500 then multiple return statements may actually improve readability over some central Boolean flag that is updated at various points.

AV1545 Don't use selection statements instead of a simple assignment or initialization ❷

Express your intentions directly. For example, rather than

```
bool pos;
if (val > 0)
{
   pos = true;
}
else
{
   pos = false;
}
```

write

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

AV1546 Prefer conditional statements instead of simple `if-else` constructs ❷

For example, instead of

```
string result;
```

```
if (someString != null)
{
   result = someString;
}
else
{
   result = "Unavailable";
}

return result;
```

write

```
return someString ?? "Unavailable";
```

AV1547 Encapsulate complex expressions in a method or property ❶

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.

AV1551 Call the most overloaded method from other overloads ❷

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.

AV1553 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 rule AV1551 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 AV1, 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 propery.

AV1555 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)
);
```

AV1561 Avoid methods with more than three parameters ❶

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.

AV1562 Don't use `ref` or `out` parameters ❶

`Ref` and `out` parameters make code less understandable and therefore may introduce bugs. Prefer returning compound objects instead.

AV1564 Avoid methods that take a bool flag ❷

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.

AV1668 Don't use parameters as temporary variables ❸

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.

AV1570 Always check the result of an `as` operation ❶

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.

AV1575 Don't comment-out code ❶

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?

AV1580 Consider abstracting an external dependency or 3rd party component ❷

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.

6 Naming Guidelines

AV1701 Use proper US-English ❶

All identifiers should be named using words from the American English language.

- 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.

AV1702 Use proper casing for members ❶

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`

AV1704 Don't include numbers in identifiers ❸

Numbers in names of fields, variables or members are very rarely needed. In fact, in most cases they are a lazy excuse for not defining a clear and intention-revealing name.

AV1705 Don't prefix member fields ❶

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`.

AV1706 Don't use abbreviations ❷

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`.

AV1707 Name an identifier according its meaning and not its type ❷

- 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.

AV1708 Name types using nouns, noun phrases or adjective phrases ❷

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.

AV1709 Name generic type parameters with descriptive names ❷

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`.

AV1710 Don't repeat the name of a class or enumeration in its members ❶

```
class Employee
{
   // Wrong!
   static GetEmployee() {}
   DeleteEmployee() {}

   // Right
   static Get() {...}
   Delete() {...}

   // Also correct.
   AddNewJob() {...}
   RegisterForMeeting() {...}
}
```

AV1711 Name members similarly to members of .NET Framework classes ❸

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 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`.

AV1712 Avoid short names or names that can be mistaken with other names ❶

Although technically allowed, the following statement is quite confusing.

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

AV1715 Properly name properties ❷

- 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`.

AV1720 Name methods using verb-object pair ❷

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
}
```

AV1725 Name namespaces according a well-defined pattern ❸

All namespaces should be named according to the pattern

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

For instance: `Microsoft.WindowsMobile.DirectX`.

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.

AV1735 Use a verb or verb phrase to name an event ❷

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;
```

AV1737 Use `-ing` and `-ed` to express pre-events and post-events ❸

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.

AV1738 Prefix an event handler with `On` ❸

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`.

AV1745 Group extension methods in a class suffixed with `Extensions` ❸

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.

7 Performance Guidelines

AV1800 Consider using Any() to determine whether an IEnumerable<T> is empty ❸

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).

8 Framework Guidelines

AV2201 Use C# type aliases instead of the types from the System namespace ❶

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()`.

AV2205 Properly name identifiers referring to localized resources ❸

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.

AV2207 Don't hardcode strings that change based on the deployment ❸

Examples include connection strings, server addresses, etc. Use `Resources`, the `ConnectionStrings` property of the `ConfigurationManager` class, or the `Settings` class generated by Visual Studio.

AV2210 Build with the highest warning level ❶

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.

AV2211 Avoid suppressing specific compiler warnings ❸

AV2215 Properly fill the attributes of the AssemblyInfo.cs file ❸

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.

AV2220 Avoid LINQ for simple expressions ❸

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.

AV2221 Use Lambda expressions instead of delegates ❷

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 an Lambda expression:

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

Or even better

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

AV2230 Only use the dynamic keyword when talking to a dynamic object ❶

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.

9 Documentation Guidelines

AV2301 Write comments and documentation in US English ❶

AV2305 Use XML tags for documenting types and members ❷

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.

AV2306 Write XML documentation with the caller in mind ❷

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.

AV2307 Write MSDN-style documentation ❸

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

Tip The tool GhostDoc can generate a starting point for documenting code with a shortcut key.

AV2310 Avoid inline comments ❷

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.

AV2315 Don't use /* */ for comments ❶

AV2316 Only write comments to explain complex algorithms or decisions ❶

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.

AV2318 Don't use comments for tracking work to be done later ❸

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

AV2400 Use a common layout ❶

- 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.
- Don't 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,
}
```

- Don't 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;
```

- 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 between multi-line statements, between members, after the closing parentheses, between unrelated code blocks, around the #region keyword, and between the `using` statements of different companies.

AV2402 Order and group namespaces according the company ❸

```
// Microsoft namespaces are first
using System;
using System.Collections;
using System.XML;

// Then any other namespaces in alphabetic order
using AvivaSolutions.Business;
using AvivaSolutions.Standard;

using Telerik.WebControls;
using Telerik.Ajax;
```

AV2406 Place members in a well-defined order ❶

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.

AV2407 Be reluctant with #regions ❶

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)

11 Important Resources

11.1 The companion website

This document is part of an effort to increase the consciousness with which C# developers do their daily job on a professional level. Therefor I've started a dedicated CodePlex site that can be easily found using the URL www.csharpcodingguidelines.com.

In addition to the most up to date version of this document, you'll find:

- A companion quick-reference sheet
- Visual Studio 2010 Rule Sets for different types of systems.
- ReSharper 5 layout configurations matching the rules in chapter 8.
- A place to start discussions on C# coding quality.

11.2 Useful links

In addition to the many links provided throughout this document, I'd like to recommend the following books, articles and sites for everyone interests in software quality,

Code Complete: A Practical Handbook of Software Construction (Steve McConnel)
One of the best books I've ever read. It deals with all aspects of software development, and even though the book was originally written in 2004, but you'll be surprised when you see how accurate it still is. I wrote a review in 2009 if you want to get a sense of its contents.

The Art of Agile Development (James Shore)
Another great all-encompassing trip through the many practices preached by processes like Scrum and Extreme Programming. If you're looking for a quick introduction with a pragmatic touch, make sure you read James' book.

Applying Domain Driven-Design and Patterns: With Examples in C# and .NET (Jimmy Nilsson)
The book that started my interest for both Domain Driven Design and Test Driven Development. It's one of those books that I wished I had read a few years earlier. It would have saved me from many mistakes.

Jeremy D. Miller's Blog
Although he is not that active anymore, in the last couple of years he has written some excellent blog posts on Test Driven Development, Design Patterns and design principles. I've learned a lot from his real-life and practical insights.

LINQ Framework Design Guidelines
A set of rules and recommendations that you should adhere to when creating your own implementations of IQueryable<T>.