TalentToolbox

The Innovative People Solution

Coding Guidelines

for C# 3.0, 4.0 and 5.0

Introduction

Who put this together?

There's no point in creating a whole document on coding standards specifically for Talent Toolbox. Coding guidelines should be pretty standard wherever you work.

So instead of creating our own, we adapted the guidelines made freely available by Dennis Doomen of Aviva Solutions at www.csharpcodingguidelines.com.

What is this?

This document attempts to provide guidelines (or coding standards if you like) for coding in C# 3.0, 4.0 or 5.0 that are both useful and pragmatic. Of course, if you create such a document you should practice what you preach.

ReSharper, Visual Studio's Static Code Analysis (which is also known as FxCop) and StyleCop can already automatically enforce a lot of coding and design rules by analyzing the compiled assemblies. You can configure it to do that at compile time or as part of a continuous or daily build. The companion site provides a list of code analysis rules depending on the type of code base you're dealing with. This document just provides an additional set of rules and recommendations that should help you end up with a better maintainable code base.

Why would you use this document?

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

- is aware that code is generally read 10 times more than it is changed;
- is aware of the potential pitfalls of certain constructions in C#;
- is up to speed on certain conventions when using the .NET Framework such as `IDisposable` or the deferred execution nature of LINQ;
- is aware of the impact of using (or neglecting to use) particular solutions on aspects like security, performance, multi-language support, etc;
- realizes that not every developer is as capable, skilled or experiences to understand elegant, but potentially very abstract solutions;

Basic principles

Not everything is covered in this document. It's already hefty and a lot to take in, but could be a lot worse. If you follow these principles you'll do well:

- The Principle of Least Surprise (or Astonishment), which means that you should choose a solution that does not include anything people might not understand or that put them 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 Gonna Need It (a.k.a. YAGNI), which tells you to create a solution for the problem at hand rather than the ones you think will happen later on. Since when can you predict the future?
- Don't Repeat Yourself (a.k.a. DRY), which encourages you to prevent duplication within a component, a source control repository or a bounded context, without forgetting the Rule of Three heuristic.
- 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.

Regardless of the elegance of somebody's solution, if it's too complex for the ordinary developer, exposes unusual behavior, or tries to solve many possible future issues, it is very likely the wrong solution and needs redesign. The worst response a developer can give you to these principles is: "But it works?".

How do you 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.
- Consider forking the original sources on GitHub and create your own internal version of the document.
- Decide which CA rules are applicable for your project and store these down somewhere, such as your source control system, or create a custom Visual Studio Rule Set. The companion site offers rule sets for both line-of-business applications and more generic code bases 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 an item to your project checklist to make sure all new code is verified against CA violations, or use something like Check-in Policy or a Git commit hook 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.

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 that may not be applicable in all situations

Class Design Guidelines

A class or interface should have a single purpose (TT1000) 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 SOLID 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.

Only create a constructor that returns a useful object (TT1001) ❸

There should be no need to set additional properties before the object can be used for whatever purpose it was designed. However, if you your constructor needs more than three parameters (which violates AV1561), your class might have too much responsibility (and violate AV1000).

An interface should be small and focused (TT1003) ❷

Interfaces should have a name that clearly explains the purpose or role of that interface within the system. Do not combine many vaguely related members on the same interface just because they were all on the same class. Separate the members based on the responsibility of those members so that callers only need to call or implement the interface related to a particular task. This rule is more commonly known as the Interface Segregation Principle.

Use an interface rather than a base class to support multiple implementations (TT1004) ❸

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

Use an interface to decouple classes from each other (TT1005) ❷

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

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

Avoid static classes (TT1008) ❸

With the exception of extension method containers static classes very often lead to badly designed code. They are also very difficult, if not impossible, to test in isolation unless you're willing to use some very hacky tools.

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

Don't hide inherited members with the new keyword (TT1010) 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 behavior that you would not normally expect from class hierarchies:

```
PocketBook pocketBook = new PocketBook();

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

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

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

It should be possible to treat a derived object as if it were a base class object (TT1011) ❷

In other words, you should be able to use a reference to an object of a derived class wherever a reference to its base class object is used without knowing the specific derived class. A very notorious example of a violation of this rule is throwing a NotImplementedException when overriding some of the base-class methods. A less subtle example is not honoring the behavior expected by the base-class.

Note This rule is also known as the Liskov Substitution Principle, one of the S.O.L.I.D. principles.

Don't refer to derived classes from the base class (TT1013) ❶

Having dependencies from a base class to its sub-classes goes against proper object-oriented design and might prevent other developers from adding new derived classes.

Avoid exposing the other objects an object depends on (TT1014) 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 it is simply returning itself so that method chaining is allowed.

Exception Inversion of Control or Dependency Injection frameworks often require you to expose a dependency as a public property. As long as this property is not used for anything else than dependency injection I would not consider it a violation.

Avoid bidirectional dependencies (TT1020) ❶

This means that two classes know about each other's public members or rely on each other's internal behavior. Refactoring or replacing one of those two classes requires changes on both parties and may involve a lot of unexpected work. The most obvious way of breaking that dependency is introducing an interface for one of the classes and using Dependency Injection.

Exception Domain models such as defined in Domain Driven Design 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.

Classes should have state and behavior (TT1025) ❶

In general, if you find a lot of data-only classes in your code base, you probably also have a few (static) classes with a lot of behavior (see TT1008). Use the principles of object-orientation explained in this section and move the logic as close to the data it applies to.

Exception The only exception to this rule are classes that are used to transfer data over a communication channel, also called Data Transfer Objects, or a class that wraps several parameters of a method.

Member Design Guidelines

Allow properties to be set in any order (TT1100) 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` or vice versa.

Use a method instead of a property (TT1105) ❸

- If the work is more expensive than setting a field value.
- If it represents a conversion such as the `Object.ToString` method.
- If it returns a different result each time it is called, even if the arguments didn't change. For example, the `NewGuid` method returns a different value each time it is called.
- If the operation causes a side effect such as changing some internal state not directly related the property (which violates the Command Query Separation principle).

Exception Populating an internal cache or implementing lazy-loading is a good exception.

Don't use mutual exclusive properties (TT1110) 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 increases the maintenance burden.

A method or property should do only one thing (TT1115) ❶

Similarly to rule TT1000, a method should have a single responsibility.

Don't expose stateful objects through static members (TT1125) 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. 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 (TT1130) 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>`.

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

Properties, methods and arguments representing strings or collections should never be `null` (TT1135) ❶

Returning `null` can be unexpected by the caller. Always return an empty collection or an empty string instead of a `null` reference. This also prevents cluttering your code base with additional checks for `null`, or even worse, `string.IsNotNullOrEmpty()`.

Define parameters as specific as possible (TT1137) ❷

If your member needs a specific piece of data, define parameters as specific as that and don't take a container object instead. For instance, consider a method that needs a connection string that is exposed through some central `IConfiguration` interface. Rather than taking a dependency on the entire configuration, just define a parameter for the connection string. This not only prevents unnecessary coupling, it also improved maintainability in the long run.

Note An easy trick to remember this guideline is the *Don't ship the truck if you only need a package*.

Consider using domain-specific value types rather than primitives (TT1140) 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.

Miscellaneous Design Guidelines

Throw exceptions rather than returning some kind of status value (TT1200) 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.

Provide a rich and meaningful exception message text (TT1202) ❷

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

Throw the most specific exception that is appropriate (TT1205) ❸

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 (TT1210) 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.

Properly handle exceptions in asynchronous code (TT1215) ❷

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

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

Always check an event handler delegate for `null` (TT1220) 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 Notify;

void RaiseNotifyEvent(NotifyEventArgs args)
{
    EventHandler 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 Notify = delegate {};
```

Use a protected virtual method to raise each event (TT1225) 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`.

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

Consider providing property-changed events (TT1230) ❸

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` argument when raising an event (TT1235) 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 argument should be `null`.

Use generic constraints if applicable (TT1240) ❷

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

```
class SomeClass
{}

// Don't
class MyClass
{
    void SomeMethod(T t)
    {
        object temp = t;
        SomeClass obj = (SomeClass) temp;
    }
}

// Do
class MyClass where T : SomeClass
{
    void SomeMethod(T t)
    {
        SomeClass obj = t;
    }
}
```

Evaluate the result of a LINQ expression before returning it (TT1250) ❶

Consider the following code snippet

```
public IEnumerable GetGoldMemberCustomers()
{
    const decimal GoldMemberThresholdInEuro = 1000000;

    var query =
        from customer in db.Customers
        where customer.Balance > GoldMemberThresholdInEuro
        select new GoldMember(customer.Name, customer.Balance);

    return query;
}
```

Since LINQ queries use deferred execution, returning `query` 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

Methods should not exceed 7 statements (TT1500) ❶

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

Make all members private and types internal by default (TT1501) ❶

To make a more conscious decision on which members to make available to other classes first restrict the scope as much as possible. Then carefully decide what to expose as a public member or type.

Avoid conditions with double negatives (TT1502) 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.

Name assemblies after their contained namespace (TT1505) ❸

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 `TalentToolbox.Web.Controls.dll`.

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

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

Name a source file to the type it contains (TT1506) ❸

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

Limit the contents of a source code file to one type (TT1507) ❸

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 (TT1508) 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
{...}
```

Use using statements instead of fully qualified type names (TT1510) ❸

Limit usage of fully qualified type names to prevent name clashing. For example, don't do this

```
var list = new System.Collections.Generic.List();
```

Instead, do this

```
using System.Collections.Generic;

var list = new List();
```

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 (TT1515) 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.

```
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 (TT1520) 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. So don't

```
var i = 3;                                  // what type? int? uint? float?
var myfoo = MyFactoryMethod.Create("arg");  // Not obvious what base-class or
                                            // interface to expect. Also difficult
                                            // to refactor if you can't search for
                                            // the class
```

Instead, use `var` like this.

```
var q = from order in orders where order.Items > 10 and order.TotalValue > 1000;
var repository = new RepositoryFactory.Get();
var list = new ReadOnlyCollection();
```

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.

Declare and initialize variables as late as possible (TT1521) 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.

Assign each variable in a separate statement (TT1522) ❶

Don't use confusing constructs like the one below.

```
var result = someField = GetSomeMethod();
```

Favor Object and Collection Initializers over separate statements (TT1523)

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();
countries.Add("Netherlands");
countries.Add("United States");
```

Use collection or dictionary initializers

```
var countries = new List { "Netherlands", "United States" };
```

Don't make explicit comparisons to `true` or `false` (TT1525) 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
```

Don't change a loop variable inside a for or foreach loop (TT1530) 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.
    }
}
```

Avoid nested loops (TT1532) 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.

Always add a block after keywords such if, else, while, for, foreach and case (TT1535) 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();
    }
}
```

Always add a `default` block after the last `case` in a `switch` statement (TT1536) 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);
    }
}
```

Finish every if-else-if statement with an else-part (TT1537) ❷

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 (TT1540) 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 TT1500 then multiple return statements may actually improve readability over some central boolean flag that is updated at various points.

Don't use if-else statements instead of a simple (conditional) assignment (TT1545) 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
```

Or 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 (TT1547) 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.

Call the most overloaded method from other overloads (TT1551) 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 parameters. Notice that the same rule applies to class constructors; implement the most complete overload and call that one from the other overloads using the `this()` operator. Also notice that the parameters with the same name should appear in the same position in all overloads.

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

Only use optional arguments to replace overloads (TT1553)

The only valid reason for using C# 4.0's optional arguments is to replace the example from rule TT1551 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 TT1235, you must use overloaded methods instead.

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

Note When an interface method defines an optional parameter, its default value is not considered during overload resolution unless you call the concrete class through the interface reference. See this post by Eric Lippert for more details.

Avoid using named arguments (TT1555)

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

Exception
The only exception where named arguments improve readability is when calling a method of some code base you don't control that has a `bool` parameter like this:

```
object[] myAttributes = type.GetCustomAttributes(typeof(MyAttribute), inherit: false);
```

Don't allow methods and constructors with more than three parameters (TT1561)

If you end up with a method with more than three parameters, use a structure or class for passing multiple arguments such as explained in the Specificationdesign 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 (TT1562) ❶

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

Avoid methods that take a bool flag (TT1564) ❷

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 (TT1568) 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.

Always check the result of an `as` operation (TT1570) 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.

Don't comment out code (TT1575) 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?

Naming Guidelines

Use US-English (TT1701) ❶

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

- Choose easily readable, preferably grammatically correct names. For example, `HorizontalAlignment` is more readable than `AlignmentHorizontal`.
- Favor readability over brevity. The property name `CanScrollHorizontally` is better than `ScrollableX` (an obscure reference to the X-axis).
- Avoid using names that conflict with keywords of widely used programming languages.

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

Use proper casing for language elements (TT1702) ❶

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

Don't include numbers in variables, parameters and type members (TT1704) ❸

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

Don't prefix fields (TT1705) ❶

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 (TT1706) ❷

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 a member, parameter or variable according its meaning and not its type (TT1707) ❷

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

Name types using nouns, noun phrases or adjective phrases (TT1708) 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 get introduced without considering the object-oriented principles (see also TT1008).

Name generic type parameters with descriptive names (TT1709) ❷

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

Don't repeat the name of a class or enumeration in its members (TT1710) ❶

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

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

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

Name members similarly to members of related .NET Framework classes (TT1711) ❸

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

Avoid short names or names that can be mistaken with other names (TT1712) ❶

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

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

Properly name properties (TT1715) 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`.

Name methods using verb-object pair (TT1720) 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 explained in TT1115.

Name namespaces using names, layers, verbs and features (TT1725) ❸

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

```
TalentToolbox.Commerce.Web
NHibernate.Extensibility
Microsoft.ServiceModel.WebApi
Microsoft.VisualStudio.Debugging
FluentAssertion.Primitives
CaliburnMicro.Extensions
```

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

Use a verb or verb phrase to name an event (TT1735) 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 EventHandler<SearchArgs> Search;
```

Use `-ing` and `-ed` to express pre-events and post-events (TT1737) ❸

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` (TT1738) 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`.

Use an underscore for irrelevant lambda parameters (TT1739) ❸

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

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

Group extension methods in a class suffixed with Extensions (TT1745) 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.

Post-fix asynchronous methods with `Async` of `TaskAsync` (TT1755) ❷

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

Performance Guidelines

Consider using `Any()` to determine whether an `IEnumerable<T>` is empty (TT1800) 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).

Note If you return an `IEnumerable<T>` to prevent editing from outside the owner as explained in TT1130 and you're developing in .NET 4.5 or higher, consider the new read-only classes.

Only use `async` for low-intensive long-running activities (TT1820) ❶

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

Prefer `Task.Run` for CPU-intensive activities (TT1825) ❶

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

Beware of mixing up `await`/`async` with `Task.Wait` (TT1830) ❶

`await` will not block the current thread but simply instruct to compiler to generate a state-machine. However, `Task.Wait` will block the thread and may even cause dead-locks (see TT1835).

Beware of `async`/`await` deadlocks in single-threaded environments (TT1835) ❶

Consider the following asynchronous method:

```
private async Task GetDataAsync()
{
    var result = await MyWebService.GetDataAsync();
    return result.ToString();
}
```

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

```
public ActionResult ActionAsync()
{
    var data = GetDataAsync().Result;

    return View(data);
}
```

You'll end up with a deadlock. Why? Because the `Result` property getter will block until the `async` operation has completed, but since an `async` method will automatically marshal the result back to the original thread and ASP.NET uses a single-threaded synchronization context, they'll be waiting on each other. A similar problem can also happen on WPF, Silverlight or a Windows Store C#/XAML app. Read more about this here.

Framework Guidelines

Use C# type aliases instead of the types from the `System` namespace (TT2201) 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()`.

Properly name properties, variables or fields referring to localized resources (TT2205) 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.

Don't hard-code strings that change based on the deployment (TT2207) 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. Maintain the actual values into the `app.config` or `web.config` (and most definitely not in a custom configuration store).

Build with the highest warning level (TT2210) 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.

Properly fill the attributes of the `AssemblyInfo.cs` file (TT2215) 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.

Avoid LINQ for simple expressions (TT2220) 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.

Use Lambda expressions instead of delegates (TT2221)

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

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

{

return c.Name == "Tom";

});

use a Lambda expression:

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

Or even better

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

Only use the `dynamic` keyword when talking to a dynamic object (TT2230) 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.

Favor `async`/`await` over the Task (TT2235) ❶

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

```
public Task GetDataAsync()
{
  return MyWebService.FetchDataAsync()
    .ContinueWith(t => new Data(t.Result));
}
```

define it like this:

```
public async Task GetDataAsync()
{
  var result = await MyWebService.FetchDataAsync();
  return new Data (result);
}
```

Tip Even if you need to target .NET Framework 4.0 you can use the `async` and `await` keywords. Simply install the Async Targeting Pack and of you go.

Documentation Guidelines

Write comments and documentation in US English (TT2301) ❶

Document all public, protected and internal types and members (TT2305) ❷

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 another developer in mind (TT2306) ❷

Write the documentation of your type with another developer in mind. Assume he or she will not have access to the source code and try to explain how to get the most out of the functionality of your type.

Write MSDN-style documentation (TT2307) 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 GhostDoccan generate a starting point for documenting code with a shortcut key.

Avoid inline comments (TT2310) 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.

Only write comments to explain complex algorithms or decisions (TT2316) 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.

Don't use comments for tracking work to be done later (TT2318) 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.

Layout Guidelines

Use a common layout (TT2400) 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;
```

- Start the LINQ statement with all the `from` expressions and don't interweave them with where restrictions.
- 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 root namespaces.

Order and group namespaces according the company (TT2402) ❸

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

// Then any other namespaces in alphabetic order
using TalentToolbox.Business;
using TalentToolbox.Standard;
using Telerik.WebControls;
using Telerik.Ajax;
```

Place members in a well-defined order (TT2406)

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. Factory Methods
5. Constructors and the Finalizer
6. Events
7. Public Properties
8. Other methods and private properties in calling order

Be reluctant with `#regions` (TT2407)

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)

Important Resources

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 2010/2012 Rule Sets for different types of systems.
- ReSharper layout configurations matching the rules in chapter 10.
- A place to have discussions on C# coding quality.

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.

Best Practices for c# async/await
The rationale and source of several of the new guidelines in this documented, written by Jon Wagner.