# StyleCop rules that are Disabled:

## SA1600 ElementsMustBeDocumented

When enabled, everything needs to be documented, which is usually overkill.

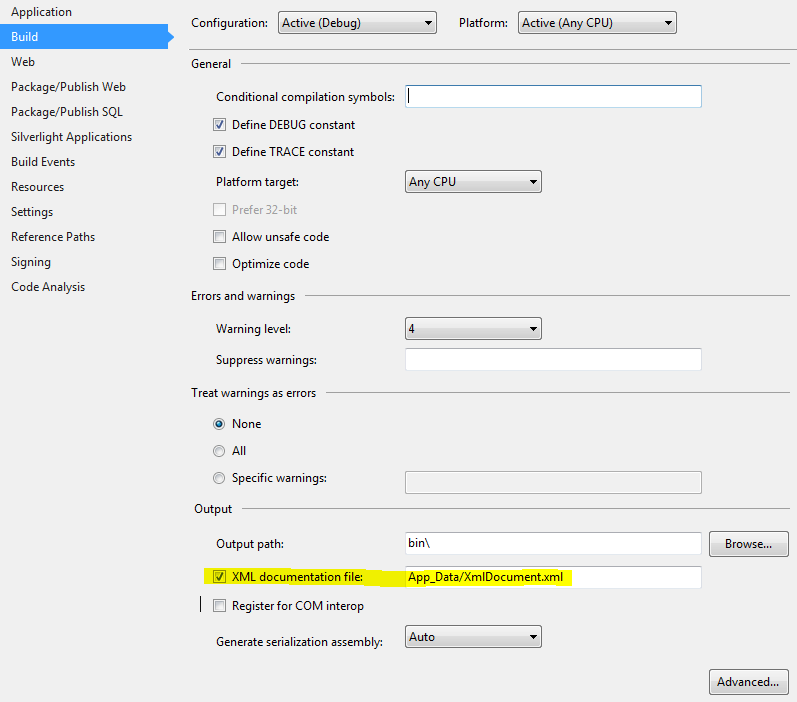
Basically let the target audience for your code decide what needs to be documented. The context where your code is being used in, results in how much must be documented. A framework class needs more documentation, than a Controller in ASP.NET MVC.

If needed, you can enable this rule for specific projects that are used as framework libraries. That will make sure it is well documented so that other developers can use it easily.

Also not every elements needs documentation, like for example fields (I assume these are always private). You can tell StyleCop to not the check the rule for these members (see <http://geekswithblogs.net/mapfel/archive/2009/11/12/136238.aspx>).

Still, when you are going to document an element, StyleCop will still enforce you to do it correctly. Also try to give examples in the documentation, how to use the code.

Keep in mind, that when an XML documentation file will be generated in the build, the Code Analysis will give warnings if\ public elements are not documented. *Should we disable this rule in Code Analysis?*



## File Headers (SA1633, SA1634, SA1635, SA1637, SA1638, SA1639, SA1640, SA1649)

File headers with copyright notices gives unneeded clutter in files. Legally, it makes no difference; copyright does not need to be declared explicitly.

It could be that for source code that you copied from the internet, which has a certain license, like GPL, that you want to mention the license in the file header. But you normally don’t want to copy this kind of code, with this kind of licenses, and use it in your projects.

## SA1503 CurlyBracketsMustNotBeOmitted

This rule is basically a good rule, but to have it forced on everything, is pushing it to the extreme. The following code is for example allowed:

public void Import(string resource, int limit)

{

if (resource != "products" && resource != "toners" && resource != "pricegroups")

throw new **NotImplementedException**("Resource is not implemented yet!");

if (limit > 100)

throw new ArgumentOutOfRangeException("limit", "Must be less or equal to 100!");

// the implementation of the method...

}

The rationale behind this is to save vertical space when doing multiple validation checks on a single argument and/or checks on many arguments. The logic in such a check is typically simple and concise and likewise for the exception that gets thrown.

I have never had an issue, nor seen an issue with a fellow developer, when adjusted if-statements to multi-line and then forgetting the curly brackets. It is an unwarranted fear that it prevents future code bugs and is often the argument used where ignorance is valued.

For other statements, besides the if-statement, there is usually no need to omit the curly brackets, because it normally doesn’t make the code any more readable.

## SA1200 UsingDirectiesMustBePlacedWithinNamespace

See the following link for the reasoning behind this rule (very edge case scenario): <http://stackoverflow.com/questions/125319/should-using-statements-be-inside-or-outside-the-namespace>

But as one answer summaries:

“No matter if you put the usings inside or outside the namespace declaration, there's always the possibility that someone later adds a new type with identical name to one of the namespaces which have higher priority.

Also, if a nested namespace has the same name as a type, it can cause problems.

It is always dangerous to move the usings from one location to another because the search hierarchy changes and another type may be found. Therefore, choose one convention and stick to it, so that you won't have to ever move usings.”

Almost every C# developer and all the Visual Studio's templates put the usings outside of the namespace, so we disable this rule. Enabling the rule would require to change every file generated by Visual Studio which is a too high impact, with basically no benefits.

## SA1101 PrefixLocalCallsWithThis & SA1126 PrefixCallsCorrectly

I have never had an issue, nor seen an issue with a fellow developer that repeatedly fights with code scope or writes code that is always buggy because of not using "this." explicitly. It is an unwarranted fear that "this." prevents future code bugs and is often the argument used where ignorance is valued.

The usage of “this.”, when used excessively or as a forced style requirement, is nothing more than a contrivance used under the guise that there is < 1% of developers that really do not understand code or what they are doing, and makes it painful for 99% who want to write easily readable and maintainable code.

Coders grow with experience “this.” is like asking someone to put training wheels on their bike as an adult because it is what they first had to use to learn how to ride a bike. And adult might fall off a bike 1 in 1,000 times they get on it, but that is no reason to force them to use training wheels.

As soon as you start typing, Intellisence will list the content available in the scope of where you are typing, "this." is not necessary to expose class members, and unless you are completely clueless to what you are coding for you should be able to easily find the item you need.

Even if you are completely clueless, use "this." to hint what is available, but don't leave it in code. There are also a slew of add-ons like Resharper that help to bring clarity to the scope and expose the contents of objects more efficiently. It is better to learn how to use the tools provided to you then to develop a bad habit that is hated by a large number of your co-workers.

Almost every C# developer and all the Visual Studio's templates don’t prefix calls with “this.”, so we disable this rule. Enabling the rule would require to change every file generated by Visual Studio which is a too high impact, with basically no benefits.

## SA1309 FieldNamesMustNotBeginWithUnderscore

This rule assumes that rule SA1101 & SA1126 is enabled, and that you should use “this.” Instead of using underscores to mark fields, but we disabled those rules (see comment elsewhere).

From convenience point of view, when Intellisence is used with “this.” It will give you all the members of a class, when used with an underscore it will give you only the fields, which is what you usually want.

When we looking at potential bugs, especially when we don’t use “this.”, we got a couple of potential naming collision, see the example below:

private int myVariable;

public void doSomething(int myVariable)

{

// This does not do what most people expect it to do.

// It does nothing and doesn't report any kind of error.

myVariable = myVariable;

}

public void doSomethingElse()

{

// This one hides the class level variable. If this is a

// big method someone else might not notice this.

int myVariable;

// This may not do what you were hoping it would do.

// Especially if you were trying to set the class-level

// variable instead of the method-level one.

myVariable = 5;

}

I've run into both of these cases/bugs on a number of occasions. To prevent these potential naming collision, we have four options:

1) Use a different name for private members. This is what I do. I precede the variable name with an underscore because it reminds me that this is local AND it should never be seen by users of my class anyway.

2) Use a different name for the argument. I hate this because the name you use will be visible to the user of your class through Intellisence. Some people do this and preface arguments with something like a lowercase "a" so the call ends up being something like:

public MyClass(int aMyValue) // uggh!

3) Call the argument something completely different. People will often try to abbreviate the argument name or come up with some other hack to make them different. You end up spending way too much time trying to come up with unique names.

4) Call them the same thing and make sure that you ALWAYS use "this." for all local member variables. This just causes hard to find bugs, see below (can you spot it? Resharper will find it for you ;-)):

class SomeClass

{

public string subscriptionMessage;

public string unsubscriptionMessage;

public string errorMessage;

public SomeClass(string subscriptionMessage, string unsubscriptoinMessage, string errorMessage)

{

this.subscriptionMessage = subscriptionMessage;

this.unsubscriptionMessage = unsubscriptionMessage;

this.errorMessage = errorMessage;

}

}

I personally prefer option 1. It is the easiest to follow and results in a better experience for those using my classes. They don't need to be bothered with the difference since I only use it on private members and all of my method signatures are easier to read. It is a simple rule that can always be followed.

Also a lot of C #developers use the underscore for fields, so we disable this rule.

# StyleCop rules that are Enabled:

## SA1124 DoNotUseRegions

The basic reasoning behind this rule is that it hides code. See also the first answer on the following question: <http://programmers.stackexchange.com/questions/53086/are-regions-an-antipattern-or-code-smell>

Some developers use regions to group fields, properties, methods, etc, even if there is only one method in it. Developers usually have large monitors, so by collapsing all (see short codes below), you should normally have the full class completely on your screen. If it doesn’t fit, it’s a smell that your class is maybe doing too much. Also the File Structure window of Resharper (Ctrl + Alt + F) can help in navigating.

CTRL + M + O will collapse all.  
CTRL + M + L will expand all.  
CTRL + M + P will expand all and disable outlining.  
CTRL + M + M will collapse/expand the current section.  
These options are also in the context menu under Outlining.

# StyleCop Validation:

If we want this to be a Check-in policy, there is a plugin:  
<http://visualstudiogallery.msdn.microsoft.com/5fb39f7a-3199-4bbd-94ca-0aaad00a94e6>

I would not advise to do this, because the code is not wrong. I would just run StyleCop on the build server and report the result (and try to solve them in time).