Code Snippets

A thorough explanation of everything code snippets in VB.NET and the presentation of a new collection of “Code Pattern” VB snippets based on the snippets available in C#.

Tags: Code Generation, Visual Studio, VB.NET

Today’s Menu  
- Introduction  
- Snippet Editors: What UIs are available to edit code snippets  
- Snippet XML Structure  
- VB.NET Default Snippets: The default code snippets that ship with Visual Studio  
- A more pragmatic approach: A collection of more commonly usable code snippets  
- Conclusions

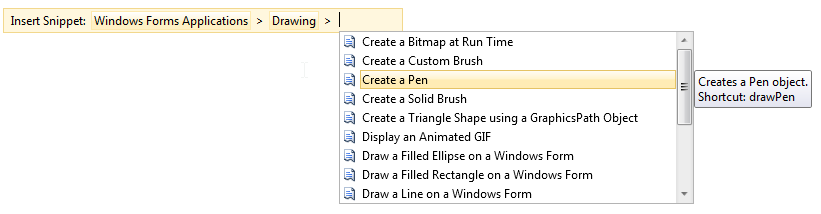
Downloads:  
- Additionally annotated and improved XSD validation schema.  
- A new VB.NET snippet collection based on the default C# snippets.

Introduction

I’ve been working on a VB.NET project for several years now and something I’ve always envied my C# colleagues for their built in [code snippets](http://msdn.microsoft.com/en-us/library/d60kx75h(v=vs.80).aspx) (<http://msdn.microsoft.com/en-us/library/d60kx75h(v=vs.80).aspx>). It’s not like we VB guys don’t get any, it’s that we have **too many**.

For those unfamiliar with code snippets, it is a feature introduced in Visual Studio 2005 that allows you to insert a block of code with just a few keystrokes. Many snippets are automatically installed with VS but a code snippet is just an XML file with a .snippet extension and Visual Studio makes it pretty easy to add your own snippets.

In the base snippet picker menu, which you can open either by right clicking in the code editor and choosing “Insert Snippet…” or by pressing CTRL K, X, there are snippets divided by categories such as “Application”, “Data”, “Fundamentals” and more.



In the screenshot I’ve first picked “Windows Forms Applications”, then “Drawing” and I am presented with a list of no less 16 snippets. All categories combined there are over 350 code snippets!

Having so many snippets makes it hard to find those that are truly useful. When I first started on the VB.NET project, I browsed the list for a while, threw my hands up in the air and never looked back.

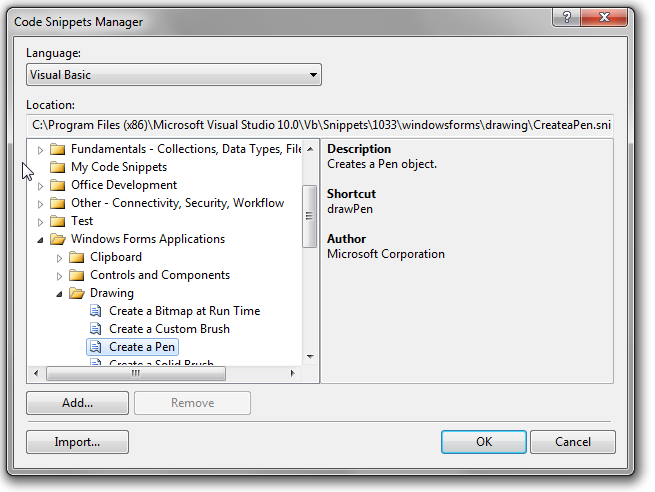
But, as we all know, generating code can save you lots of time and who wants to write the same code over and over again anyway… So I’ve now examined pretty much all the VB.NET snippets that shipped with Visual Studio and here are my findings.

We will first take a closer look at the available options for configuring, viewing and editing code snippets in and outside of Visual Studio. I’ll then continue with explaining the XML structure of the snippets and point out which elements work in VB.NET, and which you don’t need to bother with. Next is an overview of the default snippets in VB.NET and why I opted to write my own collection. The final part of this post contains the generated code and motivation of each snippet. But let’s find out what your options are when you want to use an editor first:

Snippet Software

Code Snippets Manager

Visual Studio ships with the “Code Snippets Manager”, to be found under “Tools” (If it’s not present, you need to add it manually as explained [here](http://prabathf.blogspot.com/2010/02/code-snippets-in-visual-studio-2010.html) (<http://prabathf.blogspot.com/2010/02/code-snippets-in-visual-studio-2010.html>)). The manager is required for changing the directories in which Visual Studio parses the snippets. It is however not possible to use it for actually changing the content of a snippet should you want to optimize the code it generates. Also the form is not resizable; you cannot see the code that will be generated, etc. Fortunately third party tools are available to address these issues.



Snippy

[Snippy](http://snippy.codeplex.com/) (<http://snippy.codeplex.com/>) is a standalone application for creating and editing code snippets. It does a pretty good job but the interface is not as slick as it could’ve been: You have to add/edit literals in a modal form one by one which slows the process down considerably. The last update was back in 2007 so I don’t think we can expect an update to address some of its usability issues…

Snippet Designer

[Snippet Designer](http://matthewmanela.com/projects/snippet-designer/) (<http://matthewmanela.com/projects/snippet-designer/>) is a Visual Studio plugin that provides a vastly superior experience for editing snippets when compared to snippy. Snippet properties can be changed with a PropertyGrid, the literals can be defined directly in a DataGrid and it even provides some additional contextmenu options inside the code editor. The last update was in June 2011 which makes it the most active snippet project I could find.

Snippet Editor

[Snippet Editor](http://snippeteditor.codeplex.com/) (<http://snippeteditor.codeplex.com/>) is another standalone application. While it doesn’t have the Visual Studio integration and the easy-to-use DataGrid to edit literals as the Snippet Designer has, it does have syntax highlighting for literals and a search box to find an existing snippet rapidly. There is a screencast explaining how to work with the Snippet editor [here](http://billmccarthy.com/Projects/Snippet_Editor/screencast.html) (<http://billmccarthy.com/Projects/Snippet_Editor/screencast.html>). The last update was in February 2009.

Snipper

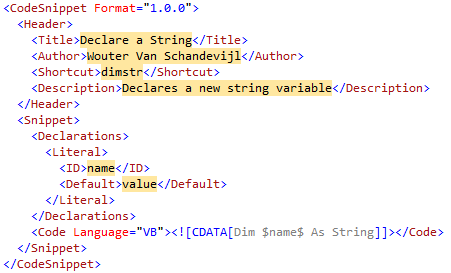
[Snipper](http://visualstudiogallery.msdn.microsoft.com/3439b095-03f5-4db7-ae3f-ba2cbe09c876/) (<http://visualstudiogallery.msdn.microsoft.com/3439b095-03f5-4db7-ae3f-ba2cbe09c876/>) is yet another standalone application. It has some (but not all) configuration for VB versus C# and also uses a modal screen to edit the literals and objects. Somehow the option to open an existing snippet is missing. The last update was in July 2011.

Visual Studio

Since snippets are merely XML files, you can also use the Visual Studio XML Editor to create or change your snippets. There is an XSD, located at “C:\...\Microsoft Visual Studio 10.0\Xml\Schemas\1033\snippetformat.xsd”, which will be automatically selected when opening a snippet file. To top it off, there is a snippet to create a snippet! The tooltip is “XML snippet for creating new XML snippets” but don’t let that fool you; you can also create snippets for VB.NET, C# and other languages that support them. Let’s find out how the generated XML looks like:

Snippet XML Structure

General structure



Header

The “Title” is the text shown in the code snippet picker and code snippets manager and is the only header node that is mandatory. Author and Description are pretty self-explanatory. They are both visible in the Code Snippets Manager. The description is also used as the tooltip in the Code Snippet Picker. If the shortcut node is set you can activate the snippet in the code editor by entering the text and pressing Tab. A shortcut is not required.

**Quick Tip**: When you are unsure about the exact shortcut, you can press the first letters followed by “?” and “Tab” and you get a menu listing all snippets that match the text before the question mark.

“SnippetTypes” defines how the snippet can be used. VB.NET only supports “Expansion” snippets which will insert the code at the cursor. Other options are “SurroundsWith” and “Refactoring” (not usable in custom scripts) both of which are used in C# only unfortunately. “Keywords” groups “Keyword” nodes which can be used by third parties. For example the Snippet Editor, a snippet editor which was discussed earlier, allows you to search your existing snippets based on these keywords. The last node is “HelpUrl” which doesn’t do much really.

References & Imports

“References” and “Imports” are code snippet features supported only by VB.NET. By adding a References node, it is possible to add a reference to an assembly to the current project automatically when inserting the snippet. The “Imports” node can then be used to add the correct namespace imports at the top of the source code file.

<Snippet>

<References>

<Reference><Assembly>System.Drawing.dll</Assembly></Reference>

</References>

<Imports>

<Import><Namespace>System.Drawing</Namespace></Import>

</Imports>  
 ...

</Snippet>

While automatically importing a namespace can be very convenient when used appropriately, having a new reference added to the project by merely inserting a code snippet might be a bit scary, especially when a new reference requires a special manual action when deploying.

Declarations

The “Declarations” node can contain “Literal” and “Object” children which are both used to indicate those parts of the generated code that can be changed once the snippet has been inserted into your code. They have an optional “Editable” attribute, which defaults to “true”, a feature that is mostly used together with the “Function” node in C# snippets as both are not supported by VB.NET (More on the “Function” node below in “Advantages of C#”). Expected child node is “ID”, which is the name that can later be used in the “Code” node to insert the Literal/Object. “Default” provides the default value of the substitutable part. It is marked as required in the validation schema but snippets still work when the node is absent. The final - also optional - node “Tooltip” does exactly what you think. Both “Object” and “Literal” have the same structure but an “Object” node has an additional required “Type” child. A .NET type is expected for the “Type” node, but its content is ignored by VB.NET and C#. The IDE does not enforce the “Type” so you can provide any other type (or even a literal) after insertion.

<Declarations>

<Object>

<ID>color</ID>

<Type>System.Drawing.Color</Type>

<ToolTip>Replace with the color of the pen.</ToolTip>

<Default>Color.Tomato</Default>

</Object>

<Literal>

<ID>penName</ID>

<Type>String</Type>

<ToolTip>Replace with the name of the pen object.</ToolTip>

<Default>tomatoPen</Default>

</Literal>

</Declarations>

Code

And finally we’ve arrived at the node where you can actually put the text that will be generated in Visual Studio:

<Code Language="VB" Kind="method body">

<![CDATA[**Dim $penName$ As New Pen($color$)**]]>

</Code>

“Language” is the only required attribute. Supported languages are VB, CSharp, VJSharp and XML. The “Kind” attribute is used to define the context in which the snippet can be inserted. The value “method decl” for example specifies that the snippet is an entire method declaration and can thus only be inserted inside a class or module. Unfortunately, the Visual Studio code editor does not limit the available snippets to the current context so a snippet with a “Kind” of “type decl” can still be inserted inside a method etc. The third and last attribute is “Delimiter”. Its value, of which only the first character is the actual delimiter, is used to surround the ID of Objects and Literals where the substitutable parts should be added in the code. The default value is $ as demonstrated in the example code above with $penName$ and $color$.

So what *does* work?

Both VB.NET and C#:  
Header: Title, Shortcut, Author and Description attributes.  
Snippet/Declarations: Literals with ID, Tooltip and Default attributes.  
Snippet/Code: The Language and Delimiter attributes.

C# only:  
Header: SnippetTypes (most notably the “SurroundsWith” value)  
Snippet/Declarations: Literals and Objects can use the Editable attribute and the Function node.  
Snippet/Code: Reserved literals $end$ and $selected$.

VB.NET only:  
Snippet/Imports: Automatically referencing and importing an assembly/namespace.

Works not or does nothing by default:  
Header: HelpURL, Keywords attributes.  
Snippet/Declarations: Objects with a Type attribute provide no additional benefit.  
Snippet/Code: The Kind attribute.

Advantages when using C#

The advantages of our C# colleagues :)

**Functions and non-editable fields**

A non-editable literal in C# is most conveniently used with the SimpleTypeName function:

<Literal Editable="false">

<ID>SystemAttributeTargets</ID>

<Function>SimpleTypeName(global::System.Attribute)</Function>

</Literal>

The SimpleTypeName function will figure out the correct namespace even when you are [using namespace aliases](http://msdn.microsoft.com/en-us/library/sf0df423(v=vs.80).aspx). Another available function is GenerateSwitchCases. It generates all the case statements for any enumeration. You can use this nifty feature by entering “switch [TAB]” and substituting the default value of $switch\_on$ with an Enum instance variable name (or with the name of an Enumeration Type) and pressing the down key. The last function, ClassName() is used for the constructor and finalizer C# snippets.

**SurroundsWith $selected$ literal**

With a SurroundsWith SnippetType you can insert code snippet surrounding the currently selected code. The selected text will be replaced with the reserved $selected$ literal.

**Quick Tip**: $selected$ can be used only once. If multiple $selected$ literals are present in the code node value, only the last one will be actually replaced. Any other $selected$ literal will be simply removed.

**The $end$ literal**

After the snippet has been inserted, the cursor will be placed wherever the $end$ keyword was present in the snippet code value. If there are other literals to be replaced, the cursor will move to the $end$ literal by pressing enter after all editable literals and objects have been replaced. The $end$ literal can be “faked” in VB.NET whenever no other literals are required for the snippet:

<Literal>

<ID>cursorHere</ID>

</Literal>

...

<Code Language="VB"><![CDATA[Debug.Assert($cursorHere)]]></Code>

VB.NET Default Snippets

Now we have a suitable editor ready and know how a snippet looks like, it is time to dive into the standard VB.NET code snippets. While scanning through the snippets, I quickly discovered that the VB team amassed so many snippets because many of the snippets provide little value for the professional VB.NET programmer. Many generate code that is only rarely required or code that you might not want to put in production.   
The default snippets per snippet menu group: The “Application” menu are snippets explaining how to use IsolatedStorage, Assemblies, Console and Deployment. They are mostly pretty straightforward mappings to the “My” namespace, stuff that is easily found using VS IntelliSense. The “Code Patterns” menu provides shortcuts to basic VB structures (loops, classes etc), they are closest to the default “Visual C#” snippets. The “Data” snippets are basically tutorials for learning ADO.NET, LINQ and VB.NET’s [Deep XML Support](http://msdn.microsoft.com/en-us/library/ms364068(v=vs.80).aspx#vb9overview_topic6) (<http://msdn.microsoft.com/en-us/library/ms364068(v=vs.80).aspx#vb9overview_topic6)>. “Fundamentals” are more tutorials and “My” namespace examples. There are even some snippets there that generate legacy VB6 code ☺. “Office Development” snippets explain how to automate Office products. The “Windows System” folder provides code samples on how to work with the EventLog, Registry, FileSystem, Processes etc. Most of these snippets/examples also use the My Namespace. “Windows Forms” contains examples of working with the Clipboard, Forms, GDI+ and basic WinForms Controls. Finally the “Other” menu contains examples of hashing, cryptography and device and network connectivity.

It seems to me as if the VB team has (mis?ab?)used the code snippets to help teach new programmers some simple “coding tricks” such as “loading a cursor from embedded resource” rather than providing often applicable shortcuts to common programming tasks as the C# snippets do.

To make things worse, it looks like many snippets are rushed, providing too little substitutable literals, not really following good coding practices (not disposing of resources most notably), adding [bad comments](http://www.codinghorror.com/blog/2008/07/coding-without-comments.html) (<http://www.codinghorror.com/blog/2008/07/coding-without-comments.html>) (for example “Iterate through a dictionary” followed by a for each statement) and the Microsoft.VisualBasic namespace gets often imported, even for snippets that don’t need the reference!  
For example, the drawPen snippet creates the following code:

C:\Users\Wouter\Desktop\Snippets\createapen2.png  
  
Obviously the “tomatoPen” should’ve been a literal. The snippet could also have been reduced to Dim tomatoPen As New System.Drawing.Pen(Color.Tomato). This might be the only statement in which VB.NET beats C# in code length so we better make use of it :). Finally, the System.Drawing namespace is automatically imported when using this snippet, so System.Drawing can also be dropped. Arguably a Using block should be used since a Pen implements IDisposable. ([Finish What You Start](http://pragprog.com/the-pragmatic-programmer/extracts/tips)!)

Using tomatoPen1 As New Pen(Color.Tomato)

End Using

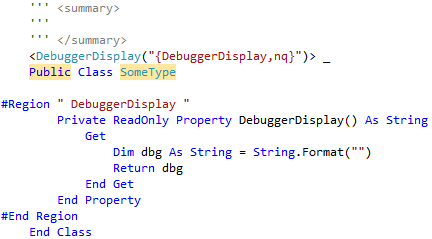
Should anyone be interested, the link to the updated drawPen snippet: ***Link tot the updated drawPen snippet***

A more pragmatic approach

While the snippets created by Microsoft might contain good “tutorials” on how to use the types in certain namespaces, they are not really helping improve my productivity so I’ve thrown away all default snippets and created a new collection. I started with translating the Visual C# snippets that are also applicable to VB.NET, improving them as I saw fit and then added some more snippets as I thought of other repetitive tasks. Most of these snippets have been customized for the project I’m currently working on but the downloads in this article have been kept pretty general. The snippets are listed per level of applicability: Namespace, Class and Method.

Namespace & Class

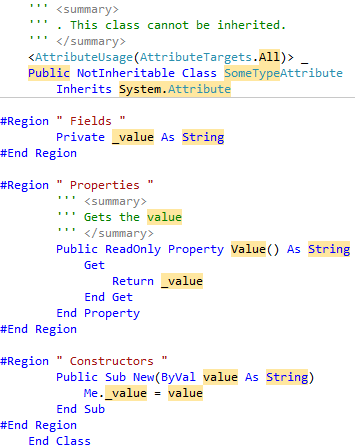
The Code snippets, applicable at the namespace and class levels, for inserting and enhancing different kinds of types:

**Generate a new class  
Download:** Default. Without DebuggerDisplay. With additional Regions.  
**Shortcut**: class  
**Code**:  


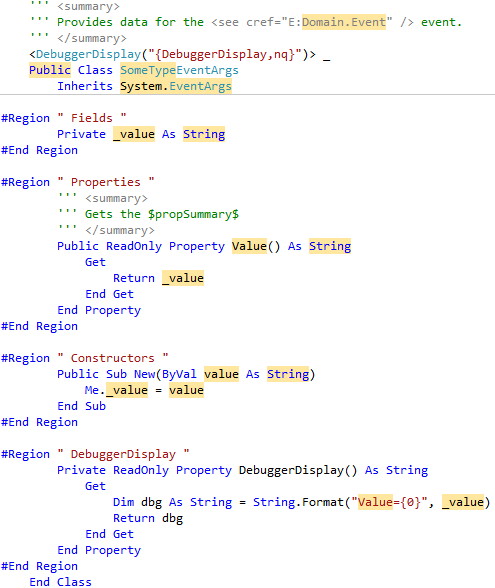
**Description**: Adds a new class with [the DebuggerDisplay attribute pattern](http://blogs.msdn.com/b/jaredpar/archive/2011/03/18/debuggerdisplay-attribute-best-practices.aspx) (<http://blogs.msdn.com/b/jaredpar/archive/2011/03/18/debuggerdisplay-attribute-best-practices.aspx>) added to the body.  
  
**Data Transfer Object  
Download:** With 1 Property. With 2. With 3. Without DebuggerDisplay: With 1 Property. With 2. With 3. **Shortcut**: dto, dto2 and dto3.  
**Code**:



**Generate a new Attribute class  
Download:** Default. **Shortcut**: attribute  
**Code**:



**Generate a new EventArgs class  
Download:** Default. Without DebuggerDisplay. **Shortcut**: evargs  
**Code**:

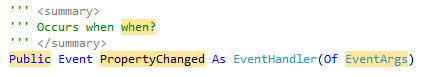


**Generate a new Exception class  
Download:** Default. Without DebuggerDisplay. Without Serializable. Without both. **Shortcut**: excep  
**Code**:

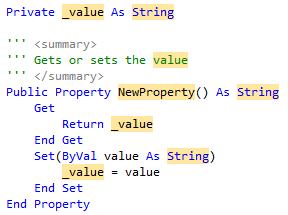


**DebuggerDisplay  
Download:** Adding the Attribute. Adding the Property. **Shortcut**: dda (DebuggerDisplayAttribute) and ddp (DebuggerDisplayProperty).  
**Description:** I’ve become a fan of adding the attribute to most/many of my classes. I use these snippets whenever I encounter a Type that doesn’t yet have it during debugging.

**Constructor  
Download:** Default. **Shortcut**: ctor  
 **Finalizer  
Download:** Default. **Shortcut**: finalize  
 **Event  
Download:** Default. **Shortcut**: event  
**Code**:

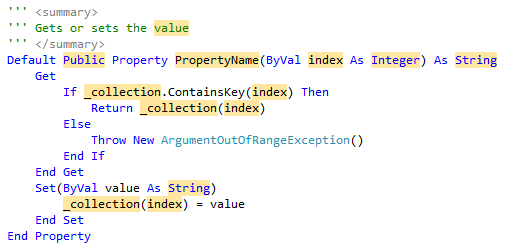


**Property  
Download:** Default. **Shortcut**: pop (“prop” is not possible as shortcut as it is automatically completed as the reserved “Property” keyword when pressing tab)  
**Code**:



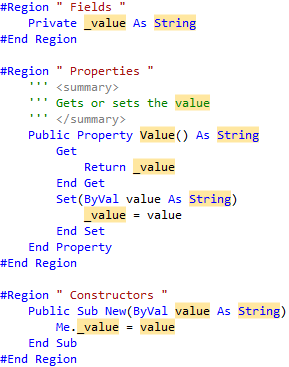
**Property with private setter  
Download:** Default. **Shortcut**: pwprop (Private Write Property)

**ReadOnly Property  
Download:** Default. **Shortcut**: rprop  
 **Default Property  
Download:** Default. **Shortcut**: defprop  
**Code**:



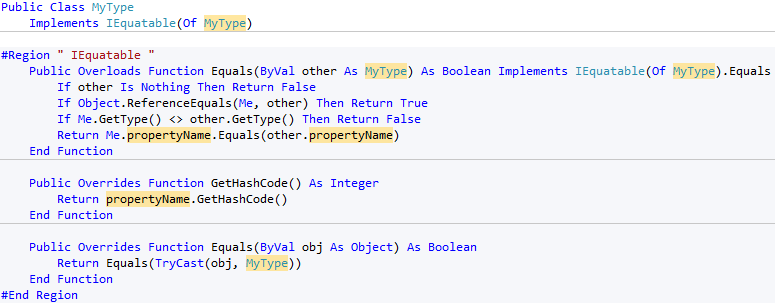
**ReadOnly Default Property  
Download:** Default. **Shortcut**: rdefprop

**Properties with constructor  
Download:** 1 Property. 2 Properties. 3 Properties.  **Shortcut**: fullprop, fullprop2 and fullprop3  
**Code**:



**Implement IDisposable  
Download:** Default. Without inheritance. **Shortcut**: dispose  
**Description**: Implements the IDisposable pattern for a class with unmanaged resources.

**Implement IEquatable  
Download:** Default **Shortcut**: equality  
**Description**: Implementing IEquatable is not nearly as easy as one would expect. Going over all the details would warrant a new blog entry. So I’ll keep this at two heads up: The GetType() check is required to maintain the symmetric property of Equals when derived classes are involved. $propertyName$ should be immutable for the GetHashCode() to work properly.  
**Code**:



Method

Snippets applicable within methods and properties to quickly add common code patterns.

**AndAlso and OrElse  
Download:** AndAlso and OrElse. **Shortcut**: ao and oe

**Console.WriteLine  
Download:** Default **Shortcut**: cw

**MessageBox.Show  
Download:** Default **Shortcut**: mbox  
**Description**: Will not automatically add a reference nor import for System.Windows.Forms.

**Code Patterns  
Download:** Default **Shortcuts**:

* If
* ifElse
* ifElseIf
* Select
* While
* For
* ForEach
* Using

**Assertions  
Download:** Assertion. NotNullAssertion. StringFormatAssertion. **Shortcut**: dast, dann and dasf

**Declare with TryCast  
Download:** TryCast. TryCast IfNotNull. **Shortcut**: dtc and dtcif

**TryCatch  
Download:** TryCatch. TryCatchFinally. TryFinally **Shortcut**: try, trycf and tryf

**String.Format  
Download:** Default. With Dim. **Shortcut**: strf and dsf

**Quick Tip**: Some code snippets don’t have ideal shortcuts. For example pop instead of prop for inserting a property. “dast” instead of “da” for Debug.Assert, “strf” instead of “sf” for String.Format etc. This is done to avoid conflicts with reserved keywords and .NET types as these take precedence in the code editor.

Conclusions

Code snippets are easily created, especially with the code-snippet-generating-snippet or by using an editor like the standalone application “Snippet Editor”. While many of the possibilities promised by the snippet XSD are being ignored by Visual Studio, code snippets can still help you be more productive by generating code for whatever repetitive task(s) you are doing, in C# or VB.NET.