Template Based Code Generator



[**Rudi Breedenraedt**](https://www.codeproject.com/script/Membership/View.aspx?mid=165729), 20 Nov 2007 [CPOL](http://www.codeproject.com/info/cpol10.aspx)

|  |  |
| --- | --- |
| https://codeproject.global.ssl.fastly.net/script/Ratings/Images/stars-fill-md.png  https://codeproject.global.ssl.fastly.net/script/Ratings/Images/stars-empty-md.png | 4.64 (31 votes) |
| https://www.codeproject.com/Articles/21162/Template-Based-Code-Generator | |  |  |  | | --- | --- | --- | | Rate: | [vote 1vote 2vote 3vote 4vote 5](https://www.codeproject.com/Articles/21162/Template-Based-Code-Generator) |  | |

A template based, command-line oriented .NET code generator

* [**Download demo project - 28.7 KB**](https://www.codeproject.com/KB/cs/T4BasedCodeGenerator/ArebisCGen12.zip)
* [**Download source - 57.9 KB**](https://www.codeproject.com/KB/cs/T4BasedCodeGenerator/ArebisCGenSource12.zip)
* [**Download sample - 30.9 KB**](https://www.codeproject.com/KB/cs/T4BasedCodeGenerator/CGenSample.zip)

Introduction

The hereby provided contribution is a command-line oriented template based code generator. Both source code and executable are made available for you to use, enhance and extend.

The code generator provides the following features:

* Customizable template syntax with default choice between T3-like template language (<% and %>delimiters for code, <%= %> for expressions, etc...) or T4-like template language (<# and #> delimiters for code, <#= #> for expressions, etc...).  
  Default is T3 like syntax.
* Support for both C# and VB.NET code in the templates
* Command-line oriented, arguments can be passed on the command-line or through a settings file
* Support for template include files
* Support for code behind files
* Templates can call each-other and pass parameters
* Support for debugging templates
* Can generate any textual output, such as programming code
* Customizable file writers, allowing to check out files from Team System Source Control or other source control systems or perform other actions.

Before we take a look at a sample, some thoughts about code generation.

Why Code Generation?

I have been using code generation for several years, and have seen it be an important key to the success of the projects. By generating code, sometimes over 20% of your projects, you make sure that whole layers of your implementation are consistently written and with no exception conform to the architectural requirements.

Code generation assures a high quality of code as the generated code contains no copy-paste errors or other errors often encountered in manually written code.

Code generation also provides you with a very high maintainability of the code. If you generated code based on a database structure, you only need to "press a button" to have the code again in sync whenever the database structure changes.

On the other hand, if you want to perform an architectural change, for instance have all your data objects implement INotifyPropertyChanged (in the System.ComponentModel namespace), you only need to change the generating template, then "press the button" again...

But for these benefits to come true, you must follow two important rules when using code generation on your project:

Rule 1: Make Sure You Can At All Times Regenerate All Generated Code

This implies that you should avoid 'interactive code generation' (where you choose a menu option in your IDE, then enter some data in a form and have a code file generated, or even worse, a piece of code). Interactive code generation can be a handy addition to your coding toolkit, such as snippets and other IDE features. It increases your typing rate, but not the maintainability of your code.

Because of rule 1, you should also never manually modify generated code. Therefore, the generated code must closely match your needs (see also rule 2) so that you don't need to change it.

You can however generate code that is adaptable and extendable. For instance, declare the generated methods virtual so you could still create a subclass and override methods, or declare your classes partial - this will allow you to extend them by simply adding an additional, non-generated file.

Rule 2: Make Sure You Generate Your Code Based on an Extendable Source

Because of rule 1, the generated code must closely match your needs. You must therefore make sure that all current and future requirements will be supported.

A common transgression of this rule is to generate data objects based on database table definitions. Although database table definitions are a good starting point to generate data objects from, it is not a place where you can indicate which generated property should be made "public" and which should be made "internal". Whenever I would have the requirement to specify the visibility modifiers of the generated properties, I would come into trouble.  
Good sources for code generation include:

* UML models, provided the modeling tool has extensibility features (custom attributes on model objects or support for UML profiles) and an API to read the model
* XML files
* Databases (not the database definition, but tables with data about what to generate)

As an alternative to generation based on database definitions, you could mix database definition data with information stored in 'meta'-tables stored in the database itself.

This brings us to the code generator available here. It is made with those two rules in mind.

To match the first rule, the generator does not interactively integrate within your IDE. Instead, it's a command-line tool and all arguments can be stored in a file. This way, you can rerun the code generation on your project with a single command on the command prompt!

As of the second rule, simple, the code generator makes no assumption whatsoever about the generation source. So it provides no default source to generate from, and you'll have to create and provide your own source.

Keep in mind that whatever source you choose, it should be a source you have full control over. You must be able to extend the information stored in the source to extend the capabilities of the code generation.

Making of the Generator

This article is not about the making of the code generator. However, the source code can be downloaded on this page, and to enlighten you on the code structure, here is some information about the code generator implementation.

At the core of the parser, you'll find an enhanced version of the Mixed Content Parser I presented in [another article](http://www.codeproject.com/useritems/MixedContentParser.asp).

The *Resources* folder of the Arebis.CodeGenerator project contains templates for the generated code (C# and VB.NET).

The whole generation is run in a separate AppDomain with ShadowCopyFiles enabled. This was needed to allow deletion of the generated assembly files. This could have been avoided by setting GenerateInMemory to true on the CompilerParameters passed to the CodeDomProvider used to compile the template, but then it seems the templates would not have the same debugging capabilities.

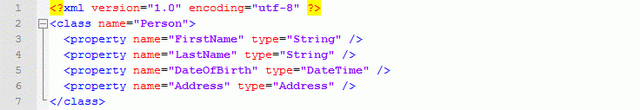
By adding line pragmas (#line in C#, or #ExternalSource in VB.NET) to the generated code, both compile time and runtime errors are reported by means of the original template source line numbers.

A Sample Project

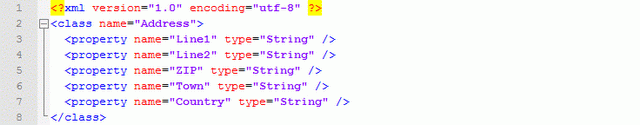
As a sample project, we will generate code based on information stored in XML files.

The Input Files

Take for instance the following XML file, as a definition for a Person class:

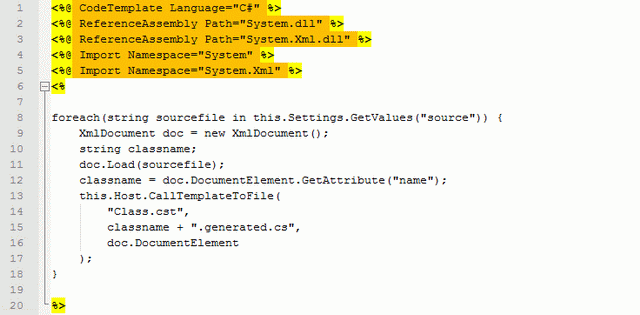


The referenced Address class could also be defined, for instance with the following XML:



First Template (BuildClasses.cst)

With "one press of a button", we want to be able to generate class definitions for those XML files. To achieve this, we need to write some templates. The first template that we write will read the XML files and ask another template to generate the classes:



We name this template *BuildClasses.cst*. (I like the extension CST for "C Sharp Template".)

Lines 1 to 5 contain directives. The first directive, CodeTemplate, is required and tells the generator the code in the template is written in C#.

ReferenceAssembly provides references to assemblies to be referenced. Provide either the full path of the assembly, a relative path if the assembly is located near the template, or just the filename if the assembly is in the GAC or in the same directory as the code generator itself. Note that the *mscorlib.dll*, *System.dll*, and *Arebis.CodeGeneration.dll* are referenced by default, however, duplicating the reference to *System.dll* does not harm.

The Import directive imports namespaces to the code, so that you don't need to specify full classnames.

The remainder of the template, lines 6 to 20, is a scriptlet — a piece of code executed inline. The code is written in C# as specified by the CodeTemplate directive.

At line 8, this.Settings is used. All settings provided to the code generator (both in a settings file or on the commandline) are available to the Settings property of the template. The property is of type System.Collections.Specialized.NameValueCollection. The interesting thing about the NameValueCollection is that it relates keys (strings) to either a single string, or an array of strings.

this.Host on line 13 gives access to an Arebis.CodeGeneration.IGenerationHost. The CallTemplateToFile() method allows to call another template, and have the output written to a separate file. Additional parameters can be passed to match parameters of the called template. In our case, we pass an XmlElement matching the document element of the XML file.

The IGenerationHost defines the following methods to call templates:

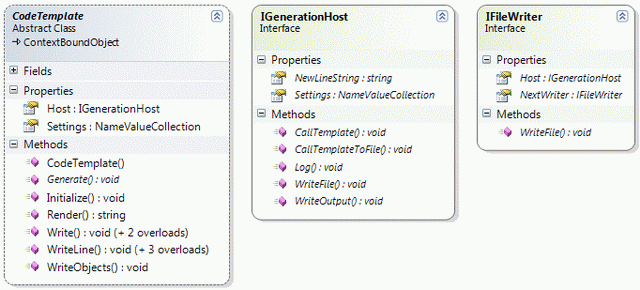
Hide   Copy Code

void CallTemplate(string templatefile, params object[] pars);

void CallTemplateToFile(string templatefile, string outputfile,

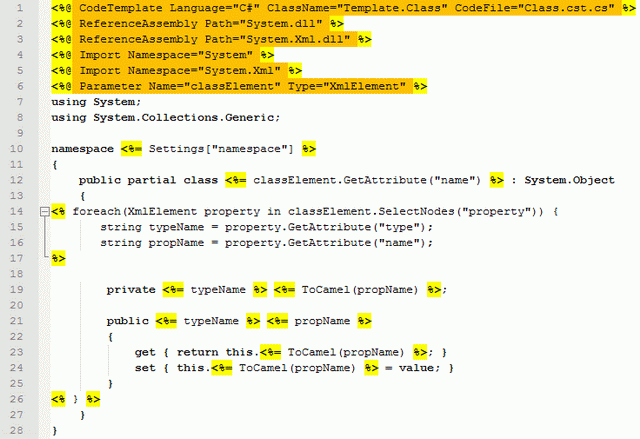
params object[] pars);

The template itself will be compiled to a class inheriting from CodeTemplate. An overview of the main classes and interfaces is defined in Arebis.CodeGeneration:



Second Template (Class.cst)

The second template, *Class.cst*, will generate individual class files. The template is the following:



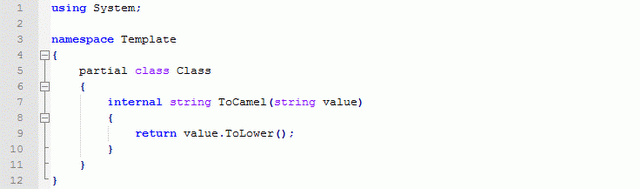
Again we start with directives. The CodeTemplate directive contains some additions (ClassName and CodeFile), more on this later on.

We have also a Parameter directive telling us this template requires one parameter, of type XmlElement, which will be accessed from the template code by the name classElement (as in line 12).

The remainder of the template is literal content, mixed with inline expressions (between <%= and %>), as well as 2 scriptlet parts, from line 14 to 17, and on line 26.

At line 19 we access a local method ToCamel(). This method does not exist by default. However, in the CodeTemplate directive, we have provided a CodeFile. The CodeFile is a partial class that will be used as part of the compiled template. Whenever the CodeFile directive attribute is used, you must also specify the ClassName attribute on the CodeTemplate directive (line 1), as the generation engine needs to give the generated class the exact same name as your class in the codefile.

The codefile *Class.cst.cs* provides a ToCamel() method on partial class names *Template.Class*. Therefore we define the *Class.cst.cs* file as follows:



Code (behind) files allow in making the code of the template easier to read as you can put complex logic away in the code behind file.

Running the Sample

The sample is almost ready to be run. The only thing we need to provide are the settings. The Code Generator command-line tool expects an argument being a settings file, and/or settings given on the command-line.

The easiest way to run the sample would be:

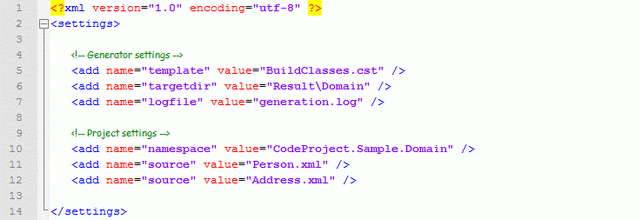
Hide   Copy Code

CGEN /template "BuildClasses.cst"

We provide the CGEN command-line tool with a value for the template setting. This is the only really required setting.

But as we use settings from within the templates, we need to provide values for those settings also. We need to provide a value for the source setting (used in the first template, line 8), and for the namespace setting (used in the second template, line 10).

In addition, we could provide values for the targetdir and the logfile settings. For information on those, and other settings, type CGEN /?. The settings file is now:

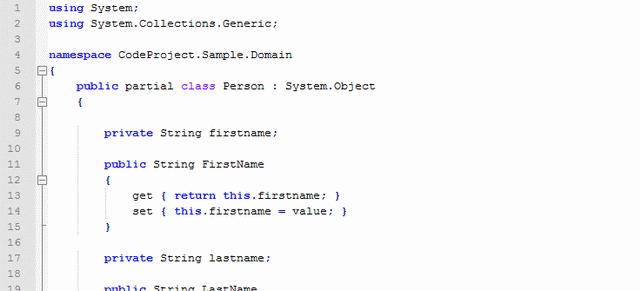


We can now run the code generator, passing it only the settings file:

Hide   Copy Code

CGEN BuildClasses.settings

The result, two files generated in the *Result\Domain* directory, of which one looks like:

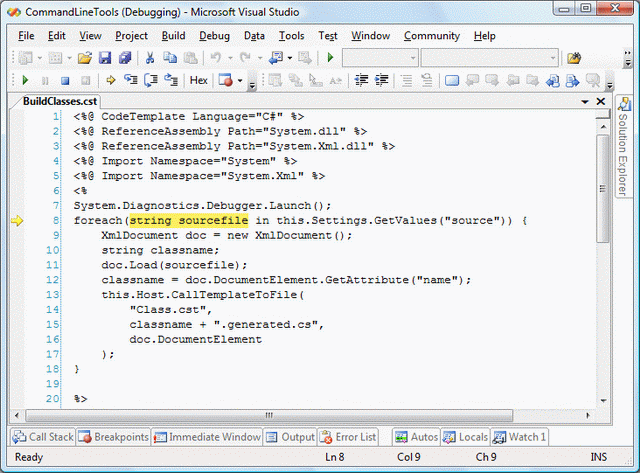


As you can see, the generated class is declared partial. This allows us to create a separate, non-generated file, to customize the behavior of the generated class.

Try the sample out yourself, it is available for download on this article. Download *CGenSample.zip*, extract it somewhere, and run the *RunSample.cmd* batch file to execute the sample. Then feel free to modify the templates or other files to see the effect, add syntax or runtime errors to see how errors are reported, or debug the execution of a template as described in the next paragraph.

Debugging Templates

By adding a call to System.Diagnostics.Debugger.Launch(), as done on the next screen (line 7), you can launch a debug session on the execution of your template.



Template Syntax

The following provides you information about the template syntax, including the list of directives and their attributes.

Directives

Directives contain information about the template, and about what is needed for the template to compile successfully. Directives provide information about the language of the template, required assembly references, namespaces to import, etc.

Templates must start with one single CodeTemplate directive, and can have additional directives. Although not mandatory, it is strongly advised to put all directive declarations in front of the template file.

Directives look like HTML tags (they have a name and usually have attributes), but are written between <%@ and %> markers.

**CodeTemplate Directive**

**Description**

Provides template declaration and meta data. Each template should have one and only one CodeTemplatedirective at the start of the file.

**Attributes**

|  |  |
| --- | --- |
| Language | The template code language. "C#" or "VB". If not specified, "C#" is assumed by default. |
| TargetLanguage | Optional. The target language, the language of the output. Can be any value. |
| AssemblyFile | Optional. The name of the assembly file to be generated. Could be used to reference this assembly from within other templates. |
| ClassName | Optional. Full name of the class to be created for this template. Setting is mandatory if using a CodeFile file. |
| CodeFile | Optional. Code behind filename containing a partial class to be completed by the template. |
| Inherits | Optional. Base class of the template class. Must be of type Arebis.CodeGeneration.CodeTemplate. |
| LinePragmas | Optional. Whether to output line pragmas and so provide debugging information in terms of the template. True by default. |
| Explicit | Optional. VB only. "On" or "Off", whether the Explicit option should be set on or off. Off by default. |
| Strict | Optional. VB only. "On" or "Off", whether the Strict option should be set on or off. Off by default. |
| Description | Optional. Free description of the template. |

**ReferenceAssembly Directive**

**Description**

References an assembly file and provides an absolute or relative filename path. The assembly should be present in the same directory as the template, a *bin* subdirectory, any directory passed to the referencepath setting or in the GAC.

**Attributes**

|  |  |
| --- | --- |
| Path | Mandatory. The path (absolute or relative) of the assembly file. |

**Notes**

Note that the assemblies *mscorlib.dll*, *System.dll* and *Arebis.CodeGeneration.dll* are automatically referenced.

**Import Directive**

**Description**

Imports a namespace in the template code source.

**Attributes**

|  |  |
| --- | --- |
| Namespace | Mandatory. Namespace to be imported. |
| Alias | Optional. An alias for the namespace imported. |

**Parameter Directive**

**Description**

Declares parameters of the template.

**Attributes**

|  |  |
| --- | --- |
| Name | Mandatory. Name of the parameter (must be a valid .NET identifier) |
| Type | Type of the parameter (System.Object is assumed by default) |

**CompileFile Directive**

**Description**

Provide additional files to be included in the template compilation. By default, the translated template and its eventual codebehind files are compiled. This directive allows for specifying additional files.

**Attributes**

|  |  |
| --- | --- |
| Path | Mandatory. Absolute or relative path to the file to include in the compilation of the template assembly. |

Expressions

Expressions are written in the language of the template (C# or VB), and are evaluated in place. Their result is converted to string and written to the output of the template.

Expressions are written between <%= and %> markers.

Code Blocks

Code blocks contain code that is executed in place. They can call methods, contain conditional (if) statements, loop definitions, etc. In fact they can contain any code that would be legal in a method of the compiled template.

Code blocks are written between <% and %> markers.

Function Blocks

Within templates, you can define one or more function blocks in which methods and other class-level elements can be placed. These are not executed in place, but are defined in the compiled template class and can contain methods called from within code blocks.

Function blocks are written between <%% and %%> markers.

Commenting Templates

Comments within templates can be written between <%-- and --%> markers.

Including Files

Files can be included in templates as if they were part of it. Include files can contain any content part valid in templates.

Include files are specified with the following notation:

Hide   Copy Code

<!--<span class="code-comment">#include file="filename"--></span>

The path is either absolute or relative to the template file.