tangible® T4 Editor User Guide

Product: tangible® T4 Editor 2010/2012 plus modeling tools

Version: 2.0.2

tangible® T4 Editor 2.0 is a Visual Studio 2010/2012 Extension that enables developers to write their own template based code generators based on a unified template language named T4 that is available in Visual Studio and standardized by Microsoft. Developers can generate code from databases, xml documents, UML models and many more sources using the T4 Text Templating Language. tangible® T4 Editor 2.0 provides syntax highlighting, Statement Completion and many more features useful for authoring and using T4 Text Templates.

Table of Contents

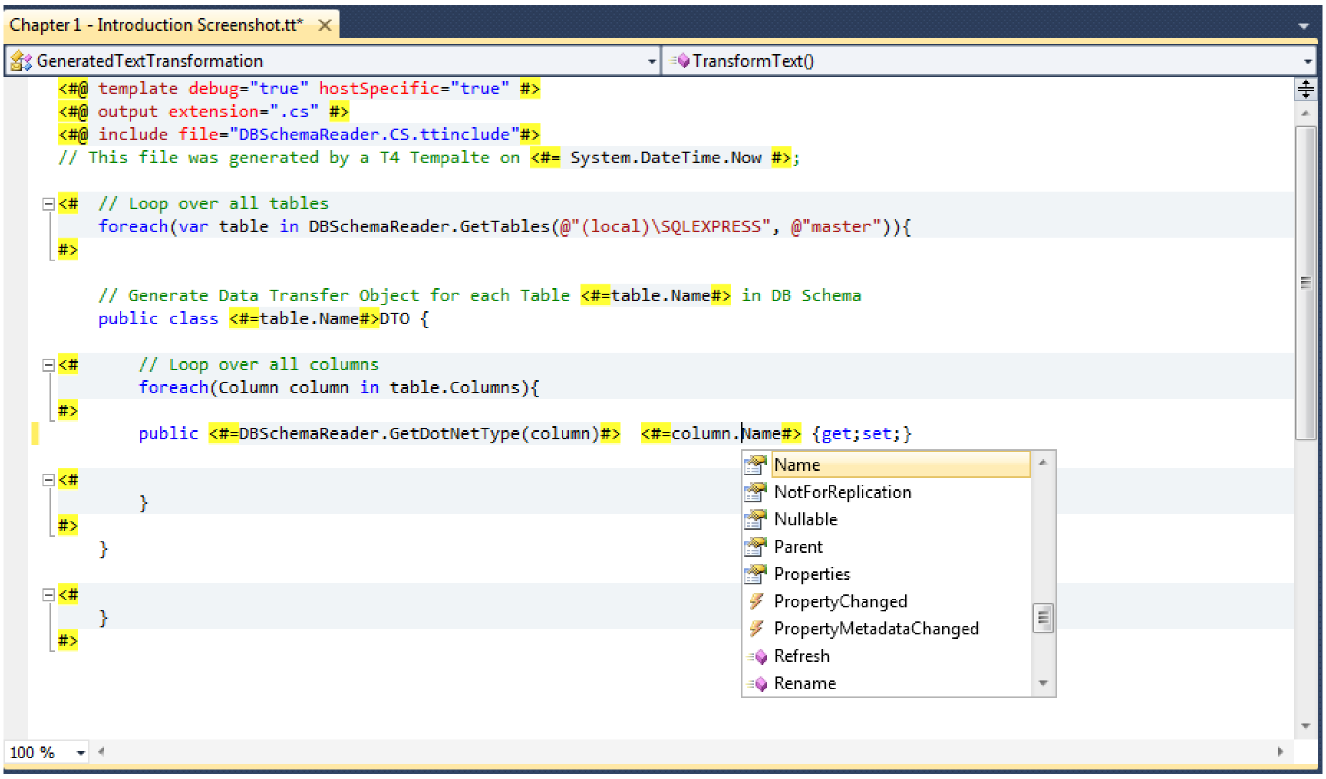
1. Introduction

tangible® T4 Editor 2.0 is a Visual Studio 2010/2012 Extension that enables developers to write their own template based code generators based on a unified template language named T4 that is available in Visual Studio and standardized by Microsoft.

Developers can generate code in several languages (including C#, XML) or other file formats from

- databases
- xml documents
- UML models
- existing code
- and many more data sources

using T4. The T4 text template language is looks pretty similar to ASP.NET and its control code is fully written in C# or Visual Basic which makes it a very productive environment for .NET developers.



```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ include file="DBSchemaReader.CS.ttinclude"#>
// This file was generated by a T4 Tempalte on <#= System.DateTime.Now #>;

<#  // Loop over all tables
    foreach(var table in DBSchemaReader.GetTables(@"(local)\SQLEXPRESS", @"master")){
#>

    // Generate Data Transfer Object for each Table <#=table.Name#> in DB Schema
    public class <#=table.Name#>DTO {

<#      // Loop over all columns
        foreach(Column column in table.Columns){
#>
        public <#=DBSchemaReader.GetDotNetType(column)#>  <#=column.Name#> {get;set;}
<#
        }
#>
    }
<#
    }
#>
```

tangible® T4 Editor 2.0 provides syntax highlighting for control code and output sections, statement completion*, debugging support* and many more features useful for authoring and using T4 Text Templates.

*) These features are limited in the free edition

2. Product Overview

tangible T4 Editor 2010/2012 Version 2.0 is a Visual Studio Extension that requires Visual Studio 2010 or VS 11 Beta to be installed in a Professional or higher edition.

2.1 Product Editions and Features

tangible T4 Editor is currently available in two Editions: Free and Professional. The Free Edition has fewer features but is pretty well accepted in the .NET community. The Professional Edition is valuable for template authors and provides excellent Statement Completion even for custom classes, debugging support and special Processing Directives for advanced code generation scenarios and much more.

A detailed comparison table between PRO and Free Edition can be found online at:
http://t4-editor.tangible-engineering.com/t4editor_features.html#Compare

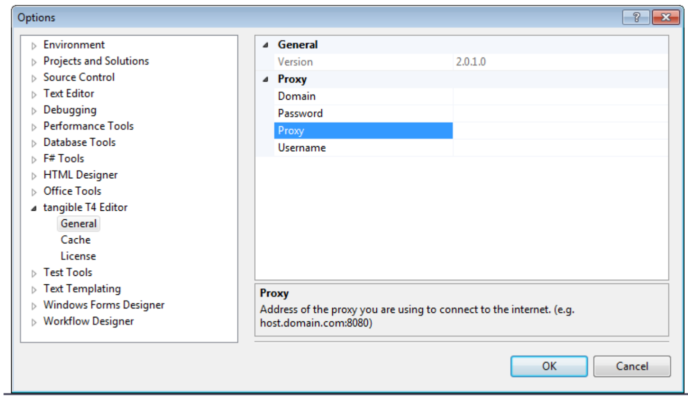
2.2 Installation

You can download the most current version from the product website at:
http://t4-editor.tangible-engineering.com/Download_T4Editor_Plus_ModelingTools.html.

After downloading the .msi-File please close all instanced of Microsoft Visual Studio. Then double click to install it under the user you want to use tangible T4 Editor later on. During the installation you might be prompted by security features of Windows Vista/7/8 to confirm the install – please confirm you want to install with elevated rights. This install will enable the tangible T4 Editor extension in Visual Studio for the user you installedas. If you need to enable the extension for another user on the system please follow the instructions here: http://t4-editor.tangible-engineering.com/forum/forum.aspx?g=posts&t=165

2.3 Configuring Internet Access

If you are using a proxy to connect to the internet please enter your proxy information in the General Settings. You can open these from Visual Studio Menu "Tools->Options" as shown in the screenshot below:

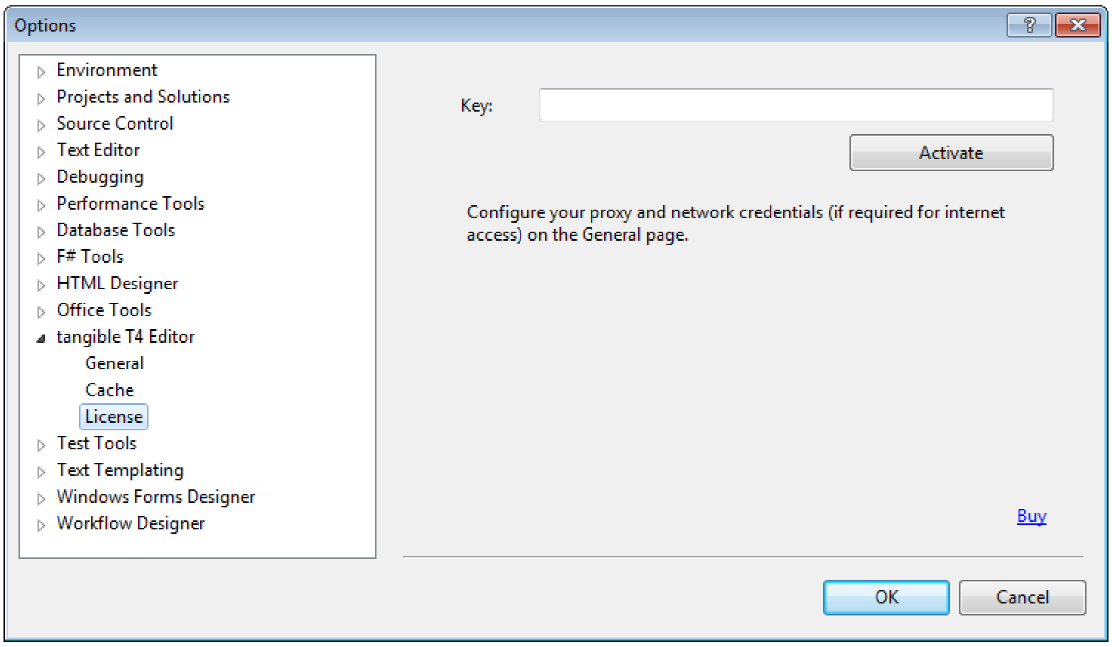


General
Version 2.0.1.0
Proxy
Domain
Password
Proxy
Username

Proxy
Address of the proxy you are using to connect to the internet. (e.g. host.domain.com:8080)

2.4 Activation of Pro-Edition Features

After purchasing the Pro-Edition from our web store http://t4-editor.tangible-engineering.com/buy.html you will receive an activation key. You can activate PRO-Features after you installed the software as outlined in the section above. To activate please start Visual Studio as Administrator via right click on the Visual Studio Icon and select start as Administrator, if you are running Windows Vista or higher. In Visual Studio open the "Tools" menu and select "Options" menu-entry. Find the "tangible T4 editor" page and enter your Activation Key on the "License" page in the "Key" textbox and press the Activate button. If you have trouble please make sure you configured proxy settings as discussed in the section above.

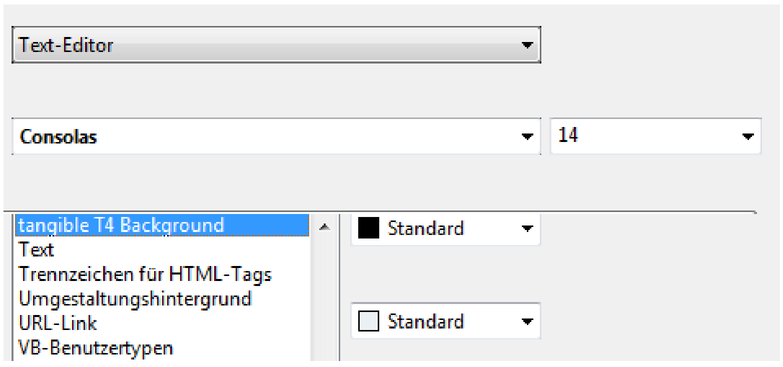


Configure your proxy and network credentials (if required for internet access) on the General page.

You can also return the license to the server, in order to activate on another machine.

2.5 Customizing fonts and colors

tangible T4 Editor picks up all fonts and colors as defined for the programming language in Visual Studio Tools->Options->Environment->Fonts and Colors Dialog. In addition it defines a background color called tangible T4 Background Color which defaults to a light cyan and can be changed as well there:

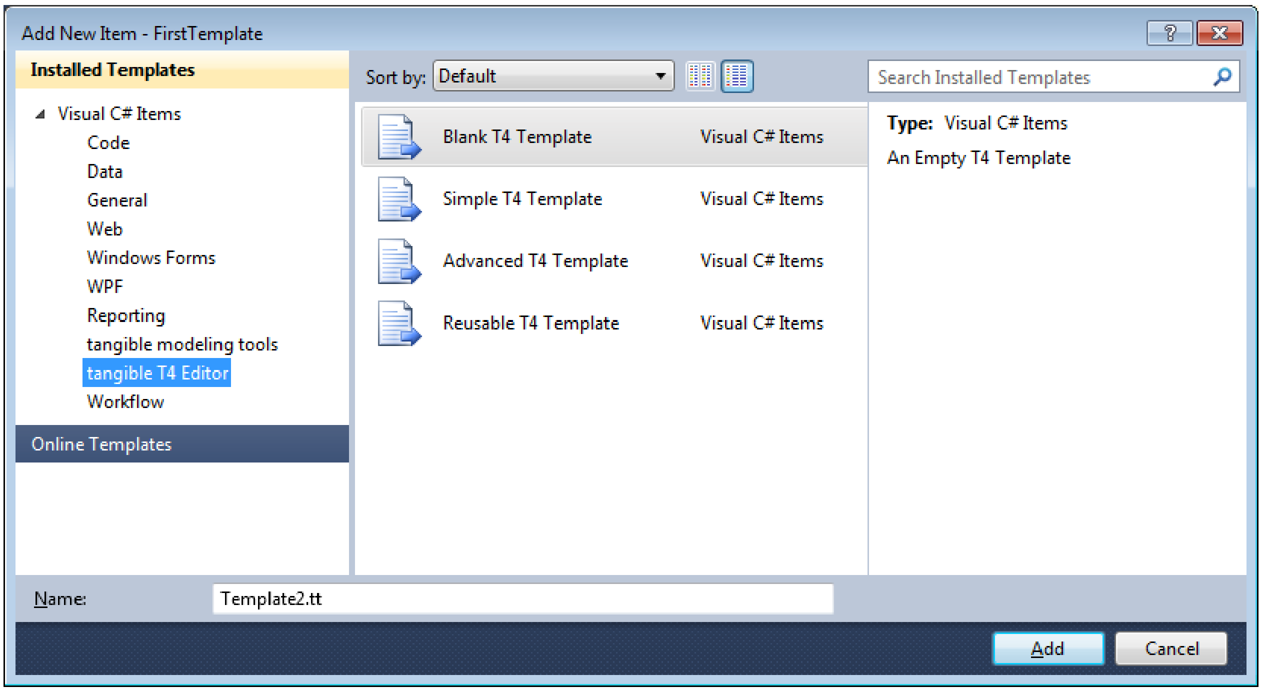


3. Getting Started with T4 Text Templating

In this section you will learn how to write T4 text templates and how you can debug T4 templates.

3.1 Writing your fist code generator using T4 Text Templates

Assuming you have installed tangible T4 Editor plus modeling tools as outlined above, you will find it easy to create T4 Text Templates with Visual Studio. First you startup Visual Studio and create a Sample Project or open an existing project. In the simplest case you would **first create a C# Console Project**, assuming your first template should generate C# code. After you created the C# Console Project you can add a T4 Text Template to it by **right clicking the project in the solution explorer and selecting "Add New Item"**. Select category "**tangible T4 Editor**" to find the template "**Blank T4 Template**" and add it to the Project.



Blank T4 Template	Visual C# Items
Simple T4 Template	Visual C# Items
Advanced T4 Template	Visual C# Items
Reusable T4 Template	Visual C# Items

Type: Visual C# Items
An Empty T4 Template

Name: Template2.tt

The contents of the template should look like this, note that the <# #> might be collapsed.

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core.dll" #>
<#@ Assembly Name="System.Windows.Forms.dll" #>
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>
<#


#>
```

The blank T4 Template already has some basic settings included, to get you started very fast. Let's discuss what is defined here, first:

The template directive `<#@ template debug="true" hostSpecific="true" #>` indicates that this file is a T4 Template and that it should allow debugging as well as access to the hosting process (e.g. Visual Studio) via "`this.Host`" variable. The next line contains an output directive `<#@ output extension=".cs" #>` which tells the T4 Engine that the file to generate should be a C# Code File. You can change the extension to any type of file you want to generate later on. In addition to these two basic directives there are two other important directives in the template.

Tip: Add debug and hostSpecific to your template directive to get the most of T4

```
<#@ template debug="true" hostSpecific="true" #>
```

Note you can also write templates in Visual Basic.NET and can also generate code in any language from them. However for the tutorial we decided to use C#. If you want to use VB you would need to use the following template directive instead: `<#@ template debug="true" hostSpecific="true" language="VB" #>`

To understand the `<#@ Assembly Name="System.Core.dll" #>` and `<#@ import namespace="System" #>` directive it is best to first be aware of the following fact: Each T4 Template is treated like a separate project in your Visual Studio Solution. Each T4 Template has its own assembly references – so it does NOT inherit any assembly references defined in your project. Furthermore each T4 Template needs to define which namespaces should be imported when typing short names for classes – e.g. DateTime instead of System.DateTime.

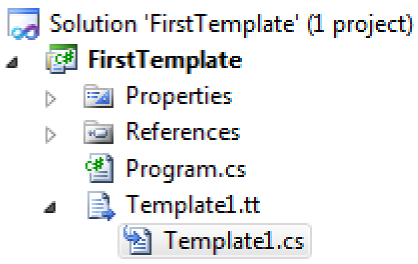
Q&A: Why do I need an Assembly directive like `<#@ Assembly Name="System.Core.dll" #>`

Each T4 Template is treated like a separate project in your Visual Studio Solution. Each T4 Template has its own assembly references – so it does NOT inherit any assembly references defined in your project. System.Core allows you to use System.Linq namespace.

Ok now let's write some lines of code:

```
<#@ templatedebug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core.dll" #>
<#@ Assembly Name="System.Windows.Forms.dll" #>
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>

class HelloT4 { // this is output code

          string GeneratedAt = "<#= System.DateTime.Now.ToString() #>"; // output code & evaluation


<#       // the following C# code is run when the template is saved or transformed
         if (System.DateTime.Now.Second % 2 == 0){
              this.WriteLine("        bool OddNumber=false;"); // this.WriteLine executed
          } else {
#>
              bool OddNumber=true; // output code
<#
          }
#>

} // this is output code
```

Now when you save the file you will see that a child file is being generated Template1.cs


Solution 'FirstTemplate' (1 project)
FirstTemplate
Properties
References
Program.cs
Template1.tt
Template1.cs

.

The contents of the Template1.cs file will look like similar to this:

```
class HelloT4 { // this is output code

       string GeneratedAt = "5/21/2012 7:55:44 AM"; // output code & evaluation

       bool OddNumber=false; // this.WriteLine executed

} // this is output code
```

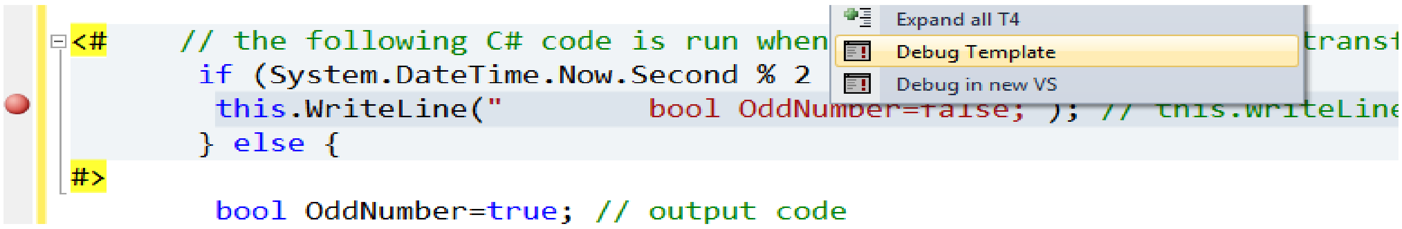
Note "OddNumber" will be true or false depending on the second you run the template. Basically what happens is that your template is compiled and the C# code is executed and inserted in the output code. T4 Templates have so much power because you can use ANY external DLL and ANY C# or VB.NET Code and run them during your transformation. We just looked at the tip of the iceberg there is more to learn in the next sections.

3.2 Debugging your first T4 Text Template

In the next step we will learn how to debug our template using the output preview feature of tangible T4 Editor PRO-Edition. In the file set a breakpoint in this line

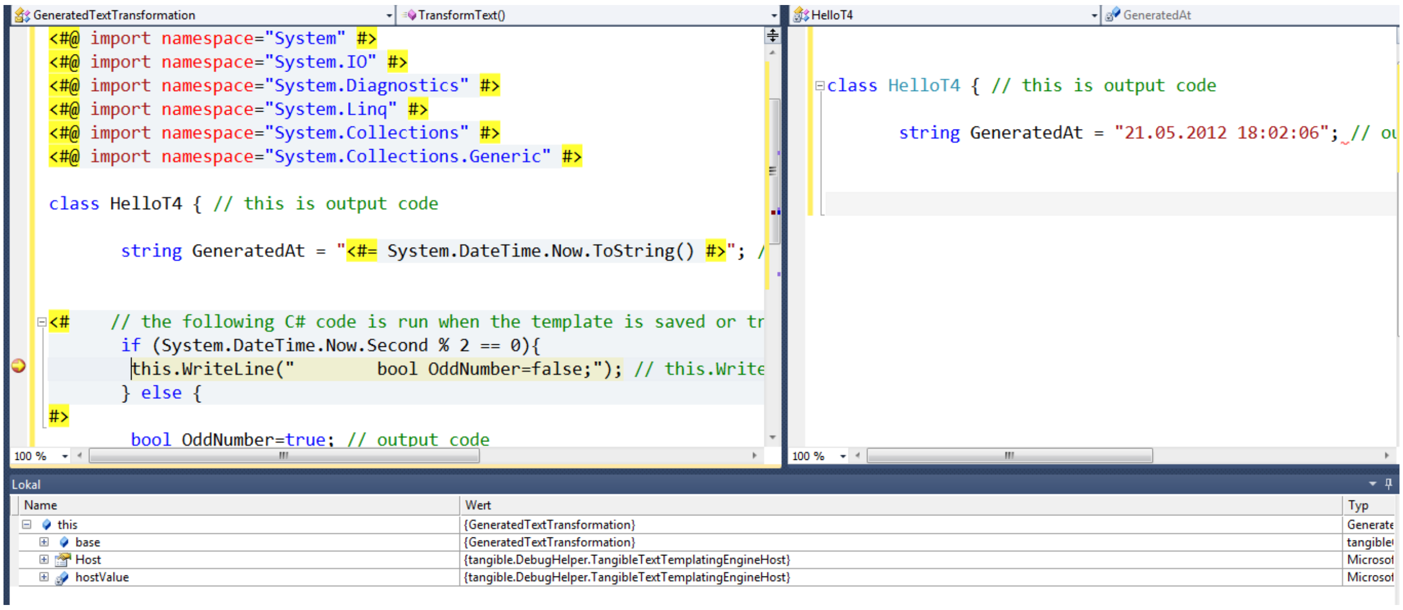
- `this.WriteLine("        bool OddNumber=false;"); // this.WriteLine executed`

Now right click on the T4 Editor pane to bring up the context menu and select the "Debug Template" command.



```
<#     // the following C# code is run when                              transf
        if (System.DateTime.Now.Second % 2
         this.WriteLine("        bool OddNumber=false;   ); // this.WriteLine
        } else {
#>
         bool OddNumber=true; // output code
```

Now you will see two things happen: first the debugger starts running and the template transform stops at the line with the breakpoint. You can inspect local variables defined in the template in the watch window. You can use F10 to step through the template code. Furthermore you will see that the output file is already there. You can dock it to the right side and see how the output grows while you execute more lines of the template.



```
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>

class HelloT4 { // this is output code

       string GeneratedAt = "<#= System.DateTime.Now.ToString() #>"; /


<#     // the following C# code is run when the template is saved or tr
        if (System.DateTime.Now.Second % 2 == 0){
        this.WriteLine("        bool OddNumber=false;"); // this.Write
        } else {
#>
         bool OddNumber=true; // output code
```

```
class HelloT4 { // this is output code

       string GeneratedAt = "21.05.2012 18:02:06"; // ou
```

Name	Wert	Typ
this	{GeneratedTextTransformation}	Generate
base	{GeneratedTextTransformation}	tangible
Host	{tangible.DebugHelper.TangibleTextTemplatingEngineHost}	Microsol
hostValue	{tangible.DebugHelper.TangibleTextTemplatingEngineHost}	Microsol

Note: There might be the case that lines are being hit multiple times in VS2010.

4. Understanding the T4 template language and template structure

T4 Text Templates consist of three parts (see also: http://msdn.microsoft.com/en-us/library/bb126478):

- **Directives** which control how the template is processed:
  e.g. <#@ assembly #> and <#@ imports #> and <#@ include #>. It is also possible to add custom directives like tangible T4 Editor provides. One example of this is the <#@ newappdomain #> directive that assures your template is always executed in a fresh AppDomain. More info: http://msdn.microsoft.com/en-us/library/bb126421.aspx

- **Text blocks** which contain text that is directly copied to the output. That is code the tangible T4 editor highlights (if the language is supported) but does not assign a background color to. It is also not possible to provide IntelliSense here.

- **Control blocks** – basically program code that controls the flow of the template or generates output text using variables and function:
  - Evaluation Control Blocks <#= myVariable #> which allow you to output variables
  - Code Blocks <# var myVariable="hello"; #> which allow you to write code
  - Class Blocks <#+ void x(){} #> which allow you to define functions or inner classes

All control blocks are colored with the T4 Background Color defined in the Visual Studio Fonts and Colors Dialog. In addition tangible T4 Editor provides IntelliSense, Coloring and Error Highlighting for all control blocks.
More info: http://msdn.microsoft.com/en-us/library/bb126545

Example:

```
<#@ template debug="true" hostSpecific="true" #>// template directive specify template features
<#@ assembly name="System.Windows.Forms"      #>// import directive enables use of assembly

var int i  = 0 ;  // this is output code it is not executed

<#
  // Control Code is executed when the template is transformed / e.g. when saved
  System.Windows.Forms.MessageBox.Show("Template Transform is running - Control Code is executed");

  WriteExtraCode();
#>

<#+  // class blocks allow to define methods and inner classes - mind the "+" sign

 void WriteExtraCode(){
    this.WriteLine(" // some extra code");
 }

#>

<#+
  // After a class block you need to have other class blocks so you need the "+" sign again here!
#>
```

4.1 Understanding the T4 Transformation Process for Design Time Templates

T4 Templates are represented as ".tt" or ".t4" files in Visual Studio. They have an assigned custom tool in the properties of the file when you inspect them in the **properties window** (keyboard shortcut F4). The default **custom tool** is named "**TextTemplatingFileGenerator**". This generator is used with so called **design time templates**, which are run every time you save the template or press the "Generate all templates" button (in the top right corner of the solution explorer). In this case the template is compiled to a dll, loaded in a separate application and code blocks are executed. The application domain used to run the code is typically reused up to 15 times. If you need a fresh AppDomain each time you should use the <#@ newappdomain #> directive which is explained later in this book. If you want to generate design time templates each time your build runs you will find another feature of tangible T4 Editor useful: The "**Transform on build**" feature of the tangible T4 Editor PRO Edition allows you to set the property **TransformOnBuild** in the property window for each Text Template file to true or false individually. This feature allows you to transform T4 files every time your solution is build.

4.2 Adding References to T4 Templates – Understanding T4 References

It can't be overstated that T4 Templates behave like very own projects. T4 Templates do NOT pick up the assembly references defined in the project they are contained in. Instead you are required to use the assembly directive to reference any assembly you want to use within your T4 template.

```
<#@ assembly name="System.Windows.Forms" #>
```

Also note that you need to use the `<#@ import #>` directive instead of "using (C#)" or "imports (VB)" for getting short hand names to classes in namespaces.

4.3 Using LINQ in Templates

One typical example where you need references and imports is when you intend to use LINQ in T4 Templates. You will need to add a reference to "System.Core" – A DLL you typically are not even aware of. See this example to learn how to use LINQ in T4:

```
<#@ output extension=".cs" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections.Generic" #>
<#
List<Person> persons = new List<Person>() { new Person(){ FirstName = "Tim" } };
var t = from a in persons.ToList()
join p in persons on a.FirstName equals p.FirstName into ps
where a.FirstName == ""
 group a by a.FirstName into tz
 select tz;
var q = (from p in persons where p.FirstName == "Tim" select new {NickName = p.FirstName});
var nickPlayer1 = q.ToList()[0].NickName;
#>
```

```
<#+
class Person
{
   public string FirstName { get; set; }
}
#>
```

4.4 Splitting up templates and including templates

It is possible to include other templates using the `<#@ include file="myTemplate.ttinclude" #>` directive. The template code in the other template with be included at the location this command is inserted into the including template. Note: Include files should NOT have a "Custom Tool" associated with them.

4.5 Understanding Host Interaction

Your template can access certain properties of the Host it is run in. Design Time T4 Templates typically run in Visual Studio and as such they can get access to some properties provided by Visual Studio.

Use the template directive with hostSpecific set to true `<#@ template debug="true" hostSpecific="true" #>` to enable access to "this.Host" a field that exposes useful features of the Host (Visual Studio). "this.Host" provides many useful properties and methods. Some of the most interesting are used in the sample below:

```
// template directive can enable access to host
<#@ template debug="true" hostSpecific="true" #>


// This code was generated by template located at full path:
<#= this.Host.TemplateFile #>

// ResolvePath resolves a relative path to an absolute
// based on the template location - note the file MUST exist!
<#= this.Host.ResolvePath("input.xml") #>
```

4.6 Visual Studio Automation from templates

It is also possible to access advanced functionality of Visual Studio. E.g. you can use COM-Automation which is nicely wrapped with .NET Wrappers to navigate your solution and change anything in your solution when execution the template. All you need is a pointer to EnvDTE.DTE Class which is the root of Visual Studio Object Model and gives you access to everything in Visual Studio, including Solution Names, Project Items, Files, Code Model, References and so on. Note: To debug these templates you must use "Debug in second VS Instance"-Feature.

Example on how to work with EnvDTE:

```
<#@ template hostSpecific="true"#>
<#@ assembly name="EnvDTE"#>
<#@ import namespace="Microsoft.VisualStudio.TextTemplating" >
<#
            var hostServiceProvider = Host as IServiceProvider;

            EnvDTE.DTE dte = null;

           if (hostServiceProvider != null) {

                    dte = hostServiceProvider. GetService(typeof(EnvDTE.DTE)) as EnvDTE.DTE;

           }
#>

// This Template is located in the following Solution
<#= System.IO.Path.GetFileNameWithoutExtension(dte.Solution.FullName)#>
```

4.7 Generating multiple output files

Using the mechanism presented in the Automating VS section it is also possible to generate multiple output files. You can find a helper template for doing this in the tangible T4 Template Gallery, which is available from the Context Menu of the Editor. It is available under Category: .ttinclude:cs->cs and called "TemplateFileManager V2.1". Assuming you have copied the contents of the template from the gallery to a file called "**TemplateFileManager.CS.ttinclude**" you could write an own template which generates multiple output files:

```
<#@ template hostSpecific="true" debug="true"#>
<#@ include file="TemplateFileManager.CS.ttinclude" #>
<#
  var fileTemplateManager = TemplateFileManager.Create(this);
  fileTemplateManager.StartNewFile("Hello World.txt");
#>
 Hello World Text goes to Hello World
<#
  fileTemplateManager.StartNewFile("Hello T4.txt");
#>
Hello T4 Text goes to Hello T4
<#
  fileTemplateManager.Process(); // Write the output via VS Automation to the project

  // Note:When debugging you need to use the Debug in second VS Instance Command because of VS Automation
#>
```

4.8 Understanding Pre-Processed T4 Templates (Runtime Templates)

There is a second type of templates available in VS2010 and higher. These templates are transformed to C# Source Code and are used by your application to generate code at runtime of your application. In this case your application is the host of the template and not visual studio. This type of template can be useful, when you need to generate data based on a structure and some input parameters. In order to enable this you need to set the Custom Tool of a T4 Template to "TextTemplatingFilePreprocessor". More info: http://msdn.microsoft.com/en-us/library/ee844259.aspx

Example on how to generate html in ASP.NET Code Behind using a T4 Template:

```
<#@ template debug="true" #>
<#@ output extension=".html" #>
<#@ Assembly Name="System.Core" #>
<#@ import namespace="System.Linq" #>

<#@ parameter name="Title" type="System.String"#>
<#@ parameter name="Author" type="System.String"#>
<#@ parameter name="Date" type="System.DateTime"#>
<#@ parameter name="Category" type="System.String"#>
<#@ parameter name="Tags"
            type="System.Collections.Generic.List<String>"#>

<div class="block_post" >
<h2><#= this.Title #></h2>                                    <div class="post_info">
  <ul>
   <li><p>Date: <a href="#"><#= this.Date #></a></p></li>
   <li><p>Posted by: <a href="#"><#= this.Author #></a></p></li>
   <li><p>Category: <a href="#"><#= this.Category #></a></p></li>
    <li><p>Tags:
<#
foreach(var tag in this.Tags) {
#>

<a href="#"><#= tag #></a><# if (this.Tags.First() != tag && this.Tags.Last() != tag) this.Write(",");#>

<#
}
#>
     </p></li>
   </ul>
  </div>
</div>
```

The runtime template can be tested using a Design Time Template:

```
<#@ template  debug="true"  #> <!-- You need to remove host specific here -->
<#@ output extension=".html" #>

<#
  this.Session.Add("Title", "tangible T4 Editor 2.0 released");
  this.Session.Add("Author", "Tim Fischer");
  this.Session.Add("Date", System.DateTime.Now);
  this.Session.Add("Category", "T4");
  this.Session.Add("Tags", new  System.Collections.Generic.List<string>(){"Tim Fischer"});

  this.Initialize();
```

```
#>
<!-- Note you must include the Pre-Processed Template after setting
     the Session and running initlialize -->
<#@ include file="BlogPost.tt" #>
```

As the preprocessed template becomes a C# class which will be compiled into your assembly, you can call it form your code. Here is an example call from a Code Behind in an ASP.NET Page:

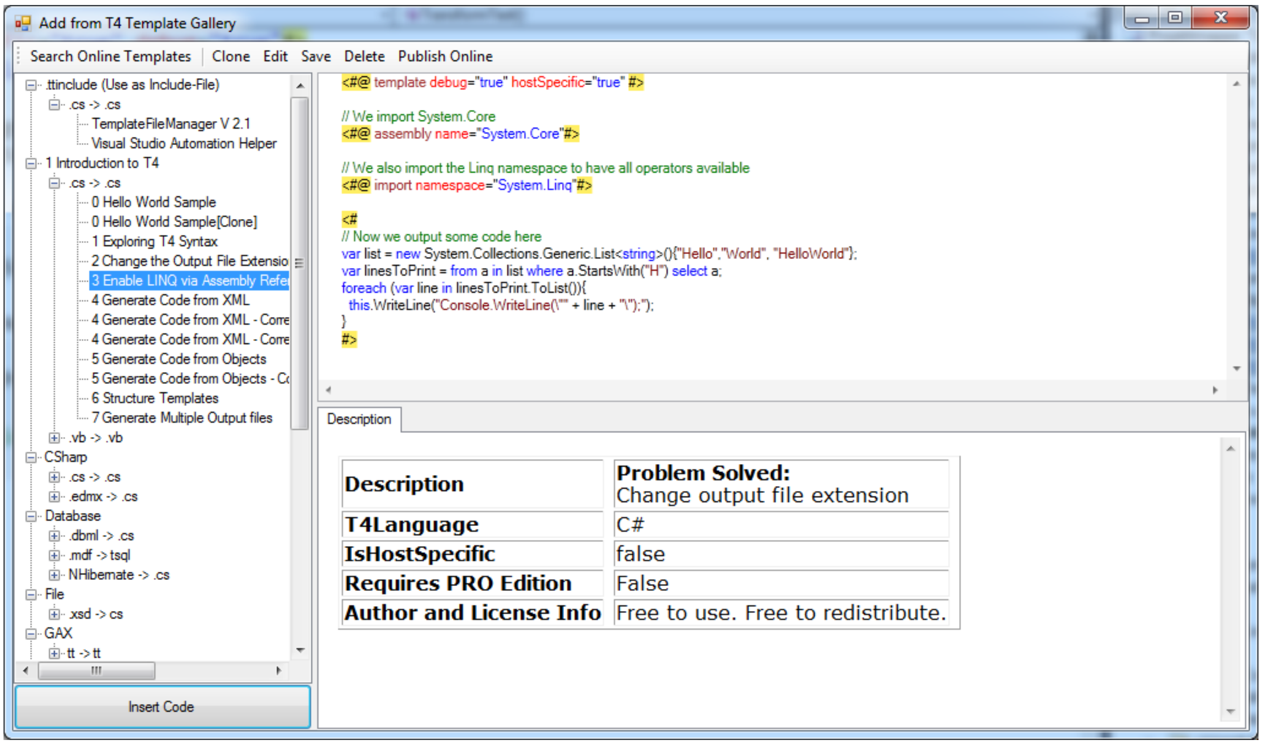
```
void PageLoad(){

       string html = "";
       foreach (var page in pages)
       {
                var blogPost = new blog.BlogPost(); // This is the template Class
                blogPost.Session = new System.Collections.Generic.Dictionary<string,object>();
                blogPost.Session.Add("Title", page.Title);
                blogPost.Session.Add("Author", page.Author);
                blogPost.Session.Add("Text", page.Content);
                blogPost.Session.Add("Date", page.PubDate);
                blogPost.Session.Add("Category", "T4");
                blogPost.Session.Add("Tags", page.Keywords);
                blogPost.Initialize(); // Sets the parameters from the session
                myControl.InnerHTML += blogPost.TransformText(); // runs the transform
       }

}
```

5. Special Features of tangible T4 Editor

This section covers all special features of tangible T4 Editor which were not fully covered before.

5.1 Template Gallery

tangible T4 Editor provides a Template Gallery with reusable templates. You can find it in the tangible T4 Menu in Visual Studio when a T4 File is open or in the context menu of the code editor:


Add from T4 Template Gallery
Search Online Templates
Clone
Edit
Save
Delete
Publish Online
.ttinclude (Use as Include-File)
.cs -> .cs
TemplateFileManager V 2.1
Visual Studio Automation Helper
1 Introduction to T4
.cs -> .cs
0 Hello World Sample
0 Hello World Sample[Clone]
1 Exploring T4 Syntax
2 Change the Output File Extensio
3 Enable LINQ via Assembly Refer
4 Generate Code from XML
4 Generate Code from XML - Corre
4 Generate Code from XML - Corre
5 Generate Code from Objects
5 Generate Code from Objects - C
6 Structure Templates
7 Generate Multiple Output files
.vb -> .vb
CSharp
.cs -> .cs
.edmx -> .cs
Database
.dbml -> .cs
.mdf -> tsql
NHibernate -> .cs
File
.xsd -> cs
GAX
.tt -> tt
Insert Code
Description

```
<#@ template debug="true" hostSpecific="true" #>

// We import System.Core
<#@ assembly name="System.Core"#>

// We also import the Linq namespace to have all operators available
<#@ import namespace="System.Linq"#>

<#
// Now we output some code here
var list = new System.Collections.Generic.List<string>(){"Hello","World", "HelloWorld"};
var linesToPrint = from a in list where a.StartsWith("H") select a;
foreach (var line in linesToPrint.ToList()){
  this.WriteLine("Console.WriteLine(\"" + line + "\");");
}
#>
```

Description	**Problem Solved:** Change output file extension
T4Language	C#
IsHostSpecific	false
Requires PRO Edition	False
Author and License Info	Free to use. Free to redistribute.

5.2 Modeling Tools

tangible T4 Editor provides 6 UML-Style Diagramming tools which you can use by adding a new tangible T4 Modeling Tools item from the Add new item dialog. You will find according templates to generate from in the in the Template Gallery.

5.3 Custom T4 Directives

tangible T4 Editor provides custom processors useful for advanced scenarios. This sections covers their syntax and some sample usage.

5.3.1 IntellisenseLanguage Directive Processor

```
Syntax: <#@ IntelliSenseLanguage processor="tangibleT4Editor" language="VB" #>
```

Use this directive in .ttinclude templates to give tangible T4 editor a hint what language the control code is written, so it can provide correct intellisense and highlighting. By default C# is used.

5.3.2 NewAppDomain Directive Processor

```
Syntax: <#@ NewAppDomain processor="tangibleT4Editor" #>
```

In some special cases T4 templates that use external libraries do not work every time they are executed. This is not due to T4 Editor but due to the fact that Visual Studio keeps the AppDomain to execute the template between executions. This new directive allows you do disable the reuse of AppDomains and assures any static caches in external types from dlls are cleared that way.

Assume the following sample in which your template uses a DLL that has caching included:

ClassLibrary1.dll - A Class Library Project or .NET Assembly that uses internal caching

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SomeLibraryWithInternalCaching
{
    public class SomeClassWithInternalCaching
    {
        // Assume this code is somewhere in your template or a DLL your
        // template referes to. Then you will need to assure your code
        // is always executed in a new AppDomain
        static string myCache = null;

        string GetSomeValueCached()
        {
            if (myCache == null)
            {
                myCache = "Retrieve Value " + new Random().Next();
            }
            return myCache;
        }
    }
}
```

The Template:

```
<#@ assembly name="$(ProjectDir)..\ClassLibrary1\bin\Debug\classlibrary1.dll" #>

<#
    SomeLibraryWithInternalCaching.SomeClassWithInternalCaching MyClass=new
SomeLibraryWithInternalCaching.SomeClassWithInternalCaching();
```

```
#>
//Issue a Caching Library returns always same value for 15 - 20 template generations

var GetCachedValue = <#= MyClass.GetSomeValueCached()#>;

var AppDomain_ID = <#= AppDomain.CurrentDomain.GetHashCode()      #>;
```

The Output of the generated code stays the same for 15-20 generations

```
//Issue a Caching Library returns always same value for 15 - 20 template generations

var GetCachedValue = Retrieve Value 1267574902;

var AppDomain_ID = 25560068;
```

To fix this issue include the `<#@ NewAppDomain processor="tangibleT4Editor" #>` directive in the template header. Then you will get a new Cached Base Value each time the template runs.

5.3.3 IncludeForIntelliSenseOnlyDirective

This directive can help you solving complex include issues.

```
<#@ IncludeForIntelliSenseOnly namespace="sample.ttinclude" processor="tangibleT4Editor"#>
```

Consider the following Problem:

```
Sample.tt
       ->Includes ->SampleInclude1.ttinclude
       -                    -> Includes SampleInnerInclude.include
                                       Which declares a class "HelloWorld";
       ->Includes ->SampleInclude2.ttinclude
       -                    -> Includes SampleInnerInclude.include
                                       Which declares a class "HelloWorld";
```

You will get the following error message when executing the template Sample.tt:

```
Compiled Transformation: The Type
"Microsoft.VisualStudio.TextTemplatingd6b30b2e677249449a5c99458b4d7c38.GeneratedTextTransformation"
already contains a definition for "HelloWorld"
```

Sample.tt

```
<#@ include file="SampleInclude1.ttinclude" #>
<#@ include file="SampleInclude2.ttinclude" #>
<#
       HelloWorld myWorld=new HelloWorld();

#>
```

SampleInclude1.ttinclude

```
<#@ include file="SampleInnerInclude.ttinclude" #>

<#
    var SomeCode= new HelloWorld();
#>
```

SampleInclude2.ttinclude

```
<#@ include file="SampleInnerInclude.ttinclude" #>

<#
    var SomeOtherCode= new HelloWorld();
#>
```

SampleInnerInclude.ttinclude

```
<#+
// Declare a new Class
public class HelloWorld{
}
#>
```

Solution:

Use IncludeForIntellisense Directive as follows:

```
Sample.tt
       ->Includes -> SampleInnerInclude.ttinclude
       ->Includes ->SampleInclude1.ttinclude
       -                    -> IncludesForIntelliSenseOnly SampleInnerInclude.include
                                          Which declares a class “HelloWorld”;
       ->Includes ->SampleInclude2.ttinclude
       -                    -> IncludesForIntelliSenseOnly SampleInnerInclude.include
                                          Which declares a class “HelloWorld”;
```

Sample.tt

```
<#@ include file="SampleInnerInclude.ttinclude" #>
<#@ include file="SampleInclude1.ttinclude" #>
<#@ include file="SampleInclude2.ttinclude" #>
<#
       HelloWorld myWorld=new HelloWorld();

#>
```

SampleInclude1.ttinclude

```
<#@ IncludeForIntelliSenseOnly file="SampleInnerInclude.ttinclude" processor=" tangibleT4Editor" #>

<#
    var SomeCode= new HelloWorld();
#>
```

SampleInclude2.ttinclude

```
<#@ IncludeForIntelliSenseOnly file="SampleInnerInclude.ttinclude" processor=" tangibleT4Editor" #>

<#
    var SomeOtherCode= new HelloWorld();
#>
```

SampleInnerInclude.ttinclude

```
<#+
// Declare a new Class
public class HelloWorld{
}
#>
```

5.3.4 AssemblyForIntelliSenseOnly

```
<#@ AssemblyForIntelliSenseOnly name="($ProjectDir)\obj\MyDLL.dll" processor="tangibleT4Editor"#>
```

This directive is especially useful in .ttinclude templates. If you need to have intellisense support for types of a certain assembly in your include template you cannot use the default include directive. This would result in an error when executing your main template – you would have duplicate includes. tangible T4 editor provides this directive, so you can have full intellisense even inside your include files. It will help a lot in splitting your large templates into few smaller reusable ones and keep your generation code clean.