## C# Based Template Transformation Engine

## This article, along with any associated source code and files, is licensed under [The Code Project Open License (CPOL)](http://www.codeproject.com/info/cpol10.aspx).

## **Introduction**

This drop-in function provides a simple template based text generation engine. It's named T44 after Microsoft’s T4.  T4 is a code generation tool that uses C#. It basically allows code to be encapsulated in special tags that can manipulate or insert text. If you have worked with classic ASP or T4 templates you're probably familiar with text template transformations. In classic ASP it was done like <body>Current time:<br /><%Response.Write Now()%></body>. In T4 you can do something like Current time:<<#= DateTime.Now #>. And finally in this project, T44, it can be done like Current time: [=DateTime.Now ]] or Current time: /\*=DateTime.Now :\*/.

Basically text transformations, also called dynamic text, allows the use of programming methods to modify text. Common uses might be for repeating text, filling in fields on an ASP page, showing a username or account code in an email, or to write 1 to 1000 on a webpage.

This project, named T44, is somewhat similar to Microsoft's T4 but much simpler. It similar because it uses encapsulated C# code to format text. Microsoft Visual Studio's T4 is much more powerful and this project is not meant to be replacement...at least not inside of Visual Studio!  There are a couple of issues when using T4 templates in a 3rd party applications.  The foremost is the licensing. The T4 DLL is not redistributable. There are funky ways around this by installing some MS packages that have the dll or installing visual studio express but that is messy.  This project by contrast is not even a DLL, it is a simple drop in function.  Another issue is T4 does not mesh well with many syntax highlighting and code completion projects. To work around this I created \\*: code-here :\*\ like commands in comments.

Here are some transformation examples.  The intermediate step just shows what gets executed to create the final output. This examples uses ]],[[,[=,[[! as for encapsulating code.

|  |  |  |
| --- | --- | --- |
| **Original** | **Intermediate Step** | **Final Output** |
| 1[[for(int i=0; i<9; i++)]]0 | Write(“1”); for(int i=0; i<9; i++)]] Write(“0”); | 1000000000 |
| Printed [=DateTime.Now]] | Write(“Printed ”); Write(DateTime.Now); | Printed 1/17/15 2:36 PM |
| [=i++]]. A [=i++]]. B | Write(i++);Write(“. A”);Write(i++); Write(“. B”); | 1. A 2. B |

Background

This function was built because of a need for a simple text template transformation engine for an AMD GCN assembly language project I am working on.  In assembly, a pre-compile, macro like feature is very useful – almost required.  Very often you might run into a situation like having to unroll a looped loop.  Since pure assembly languages does not support “for” or “while” like higher level languages, it is often left to the programmer to do these.  Working with and maintaining a few ugly text template lines of code is much better then writing ten assembly statements fifty times.

For example:

[[ for(int i = 0; i < 4; i++){ ]]

Add R[[=i+20], R4, [[=i]]; [[}]]

Would be transformed into...

Add R20, R4, 0;

Add R21, R4, 1;

Add R22, R4, 2;

Add R23, R4, 3;

Originally, I was planning on using Microsoft’s T4 but after some investigation I found out it required a DLL that was not redistributable.  It seemed pretty easy and fun to create a text transformation template engine so I set forth. The goal was to keep it as simple as possible because I might want to adopt it for different things in the future and if there was lots of junk then adjusting it would be difficult.

## **Using the code**

Just drop in the function or static class and then make a call to the function.

1. First, copy the T44 function into your application. Make the function public, private or internal as needed.
2. Select the formatting you wish to use by uncomment the style in the header. There are two formats:
   1. [[CODE]] ,  [[+CODE  and  [[!SKIP\_ME]]  - easier to read (recommended)
   2. /\*:CODE:\*/ ,  //:CODE  and  //!SKIP\_ME  - works code completion and syntax highlighting.
   3. Or, create your own.
3. Build some text (as a string) that needs to be converted. Use the following table for reference:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **“[[..]]” Style** | **“/\*:..:\*/” Style** | **comments** |
| **Code Block** | [[ code\_here ]] | /\*: code\_here :\*/ |  |
| **Code Line** | [[~code\_here | //: code\_here | terminates with line break |
| **Expression** | [[=variable]] | /\*= variable:\*/ | wraps var in write(..) |
| **Comment Block** | [[! comments ]] | /\*! comments :\*/ | excluded in final |
| **Comment Line** | (none) | //! comments | ends with line break |
| **IDE Only** | (none) | /\*\*/ IDE code /\*\*/ | dummy/filler IDE only code |

1. Call T44Expand(…) in your application.  It takes one string as a parameter and returns a string.  String myOutput = T44Expand(myInput);

## **How it works**

In a nutshell, the T44Expand function takes a string, converts that string into a program (step 1), compiles the program (step 2), and finally runs that program to collect its output (step 3).

Here is the entire function:

public static string T44Expand(string text)

{

// For [[CODE]] , [[~FULL\_LINE\_OF\_CODE and [[!SKIP\_ME]] style uncomment the next two lines

const string REG = @"(?<txt>.\*?)(?<type>\[\[!|\[\[~|\[\[|\[=)(?<code>.\*?)(\]\]|(?<=\[\[~[^\r\n]\*?)\r\n)";

const string NORM = @"[["; const string FULL = @"[[~"; const string EXPR = @"[="; const string TAIL = @"]]";

// For /\*:CODE:\*/ , //:FULL\_LINE\_OF\_CODE and //!SKIP\_ME style uncomment the next two lines

//const string REG = @"(?<txt>.\*?)((?<ide>/\\*\\*/.\*?/\\*\\*/)|(?<type>//!|/\\*:|/\\*=|//:|/\\*!)(?<code>.\*?)(:\\*/|(?<=(//:|//!)[^\r\n]\*)\r\n))";

//const string NORM = @"/\*:"; const string FULL = @"//:"; const string EXPR = @"/\*="; const string TAIL = @":\*/";

//////////////// Step 1 - Build the generator program ////////////////

System.Text.StringBuilder prog = new System.Text.StringBuilder();

prog.AppendLine(@"

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

class T44Class {

static StringBuilder sb = new StringBuilder();

public string Execute() {");

foreach (System.Text.RegularExpressions.Match m in

System.Text.RegularExpressions.Regex.Matches(text + NORM + TAIL, REG,

System.Text.RegularExpressions.RegexOptions.Singleline))

{

prog.Append(" Write(@\"" + m.Groups["txt"].Value.Replace("\"","\"\"") + "\"); ");

string txt = m.Groups["code"].Value;

switch (m.Groups["type"].Value)

{

case NORM: prog.Append(txt); break; // text to be added

case FULL: prog.AppendLine(txt); break;

case EXPR: prog.Append(" Write(" + txt + "); "); break;

}

}

prog.AppendLine(@"

return sb.ToString();}

static void Write<T>(T val) { sb.Append(val);}

static void Format(string format, params object[] args) { sb.AppendFormat(format,args);}

static void WriteLine(string val) { sb.AppendLine(val);}

static void WriteLine() { sb.AppendLine();}

}");

//////////////// Step 2 - Compile the generator program ////////////////

var res = (new Microsoft.CSharp.CSharpCodeProvider()).CompileAssemblyFromSource(

new System.CodeDom.Compiler.CompilerParameters()

{

GenerateInMemory = true, // note: this is not really "in memory"

ReferencedAssemblies = { "System.dll", "System.Core.dll" } // for linq

},

prog.ToString()

);

res.TempFiles.KeepFiles = false;

// Print any errors with the source code and line numbers

if (res.Errors.HasErrors)

{

int cnt = 0;

string errors = "There is one or more errors in the template code:\r\n";

foreach (System.CodeDom.Compiler.CompilerError err in res.Errors)

errors += "Line " + err.Line + " - " + err.ErrorText + "\n";

errors += "\r\n================== Source (for debugging) =====================\r\n";

return errors + System.Text.RegularExpressions.Regex.Replace(prog.ToString(), "\r\n",

m => { cnt++; return "\r\n" + cnt.ToString().PadLeft(4) + "| "; ; });

}

//////////////// Step 3 - Run the program to collect the output ////////////////

var type = res.CompiledAssembly.GetType("T44Class");

var obj = System.Activator.CreateInstance(type);

return (string)type.GetMethod("Execute").Invoke(obj, new object[] { });

}

**Step 1** –  The input text, with embedded C# commands, is fed through a Regular Expression to parse out the different sections.  The input text by nature is going to be in the format TEXT-CODE-TEXT-CODE… so we process each TEXT-CODE at a time. Here is the RegEx used for deciphering each TEXT-CODE:

(?<txt>.\*?)  <- This captures any normal text that will directly outputted with Write(“text here”).

(?<type>\[\[!|\[\[\~|\[\[|\[=)  <- This get the begin bracket and the type of it. It can be [[ , [= [[!.

(?<code>.\*?) <- This captures the code piece.

(\]\]|(?<=\[\[[^\r\n]\*?)\r\n)  <- This captures the closing bracket.

The goal is to convert the source text into a program so we can execute it.  For each <txt> we append an sb.Append(txt) where sb is a StringBuilder.  For each <code> we directly write the text – it is not wrapped in a sb.Append().   The beginning and ending brackets and anything that starts with a “[[!” are stripped out and not copied over.

In this first step, the program header and footer are also added.  In the header, we add some using statements, a class header, and function header. In the footer we add some useful functions like “Write(...)” and “WriteLine(...)”  and finally complete the class with a “}”.

One other item to note is that before we run the RegEx, a “[[]]” is appended at the end. (text + NORM + TAIL)  This is because the RegEx is looking TEXT-CODE chunks and this means we must end with a CODE. In this case it’s just an empty code “[[]]”.

**Step 2**– The program we built in step 1 is then compiled using the .Net CSharpCodeProvider.

GenerateInMemory does not save the file into RAM but rather a temporary folder.  TempFiles.KeepFiles = false must be set to ensure these files are cleaned up.

**Step 3** – In the last step we invoke the mini-program we generated and return its output.

Sample Input/Output

### **Sample Input**

This first example will write Hello World three times:

[[~ for(int i=0; i<3; i++){  
Hello World [[ Write(i.ToString()+"! "); }]]

[[! This comment will not be added to the output. ]]

Write() will print any bool, string, char, decimal, double, float, int...  
    A googol is 1[[ for(int i=0; i<100; i++) Write("0"); ]]

This will also write bool, string, char, decimal, double, float, int...  
    Hello at [=DateTime.Now]]!

This is the temporary executable that generates the exec:  
    [=System.Diagnostics.Process.GetCurrentProcess().MainModule.FileName]]

[[ for(int i=1; i<4; i++){ ]]  
    [="Hello " + i + " World"+ (i>1?"s!":"!") ]]  
    How are you? "[=i]]"  [="\r\n"]]  
[[ } ]]  
Bye!

### **Intermediate Stage**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

class T44Class {

static StringBuilder sb = new StringBuilder();

public string Execute() {

Write(@"

This first example will write Hello World three times:

"); for(int i=0; i<3; i++) Format("Hello World {0}! ",i); WriteLine();

Write(@"

Here is another way of doing the same thing:

"); for(int i=0; i<3; i++){

Write(@"Hello World "); Write(i.ToString()+"! "); } Write(@"

And, of course, you can just write it:

Hello World 0! Hello World 1! Hello World 2!

"); Write(@"

Write() will print any bool, string, char, decimal, double, float, int...

A googol is 1"); for(int i=0; i<100; i++) Write("0"); Write(@"

This will also write bool, string, char, decimal, double, float, int...

Hello at "); Write(DateTime.Now); Write(@"!

This is the temporary executable that generates the exec:

"); Write(System.Diagnostics.Process.GetCurrentProcess().MainModule.FileName); Write(@"

"); for(int i=1; i<4; i++){ Write(@"

"); Write("Hello " + i + " World"+ (i>1?"s!":"!") ); Write(@"

How are you? """); Write(i); Write(@""" "); Write("\r\n"); Write(@"

"); } Write(@"

Bye!

");

return sb.ToString();}

static void Write<T>(T val) { sb.Append(val);}

static void Format(string format, params object[] args) { sb.AppendFormat(format,args);}

static void WriteLine(string val) { sb.AppendLine(val);}

static void WriteLine() { sb.AppendLine();}

}

### **Final Output**

This first example will write Hello World three times:

Write() will print any bool, string, char, decimal, double, float, int...  
    A googol is 100000000000000000000000000000000000000000000000000000000000000000000000000000000000  
00000000000000000

This will also write bool, string, char, decimal, double, float, int...  
    Hello at 1/18/2015 1:55:50 PM!

This is the temporary executable that generates the exec:  
    C:\Projects\T44\_Text\_Template\_Transformation\bin\Debug\T44.vshost.exe

    Hello 1 World!  
    How are you? "1"

    Hello 2 Worlds!  
    How are you? "2"

    Hello 3 Worlds!  
    How are you? "3"  
Bye!

## **When not to use this code**

* **Security** – Since T44 compiles and runs commands (like a script) it has the potential to be abused. Be cautious of what or who might call this function and what permission levels the program is running in.
* **Not a replacement for T4 in Visual Studio**.  T4 is built into Visual Studio so use that.  It is also more feature rich, more commonly known, and easier to debug in newer versions of visual studio.
* **Avoid using templates if possible**. Be careful not to jump in and use text templates. They can be confusing for others and really make code complicated.  Make sure you need them first.  If the structure of the text template is always the same then just write the code.  For example, don’t use template transformation to do Current time: [=DateTime.Now ]] when "Current time:" + DateTime.Now.ToString() would suffice.   Also, the performance is not that great.

## **Points of Interest**

The most enjoyable part of the project was creating the language.  The main goal was for it to be simple and easy to read.  My first version used “#” for inline code but it was not as clean as I wanted.  After some experimentation I settled with “[[“ style comments.   After toying with “[[“ and other delimiters for a while, I noticed that it wreaked havoc with code completion and syntax highlighting.  After more experimentation, I eventually came up with using comments (/\*: ... :\*/ and //: ...) since any c-style editor would exclude these.   This worked well except in the instance when there needed to be dummy filler code for the editor.   Here is an example:

int myVar = /\*: for(int i=0; i<3; i++) Write(i) :\*/;   fails because the editor sees “int myVar = ;”

int myVar = /\*: for(int i=0; i<3; i++) Write(i) :\*/ /\* \*/1/\* \*/;   passes because the editor sees “int myVar = 1;”

## **Compatibility**

* no DLLs required
* no using statements needed
* works in both x64 and x86
* directly runnable in .Net 3.5, 4.0, 4.0 Client Profile, 4.5, and 4.51
* Also works in .Net 2.0, 3.0, 3.5 (Client Profile) if "System.Core.dll" and "using System.Linq;" are removed.
* Tested okay under Visual Studio 2010/2012/2013

## **Performance**

The included sample takes 85ms (i7 2nd Gen, 3.2Ghz, SSD).  Release, debug, and release without debugger all had similar performances.

Breakout:

* 1ms to generate code
* 76ms for compile
* 3ms execute program

## **History**

* Dec. 2015   - started
* 1-3-2015    - initial version
* 1-17-2015  - removed linq stuff (works better with pre .Net 3.5)