VS SDK Single File Generator Example Deep Dive

István Novák (DiveDeeper), Grepton Ltd.

September, 2008

# Introduction

Single file generators are a useful mechanism to support the development process by code generation. You can attach a single file generator to a source file through the files “Custom Tool” property. When you save the file the single file generator runs and creates source code by parsing the parent source file.

The most common reason of using single file generators is to help in repeatable and mechanical tasks, for example in using the same (and sometimes complex) code generation pattern for a (large) number of files. Let’s assume your source files can be XML structures each describing a business entity. You can use a single file generator to create the C# classes or VB Classes representing these entities.

One node in your XML structure can be represented with many code lines. By simply changing the information in the XML source files saves you a lot of work. You can even update the single file generator when your generation pattern changes.

This VS SDK sample demonstrates how to create a single file generator that parses an XML file and turns it into a C#, VB or J# source code file depending on the current project context.

By going through this deep dive you will get familiar with the following concepts:

* What is a single file generator and when should you use one?
* How to register a single file generator with Visual Studio?
* What is the pattern used in this example to build the single file generator?
* How to access VS context information in single file generators?
* What are the roles of CodeDom and CodeDomProvider in code generation?

To understand concepts treated here it is assumed that you are familiar with the build process of VSPackages. However, the Single File Generator is not a VSPackage, the same mechanisms are used to build and register it. To get more information about the package building process, please have a look at the Package Reference Sample (C# Reference.Package sample).

# Single File Generator Sample

Open the Microsoft Visual Studio 2008 SDK Browser and select the Samples tab. In the top middle list you can search for the “C# Example.SingleFileGenerator” sample (“VB Example.SingleFileGenerator” sample for VB). Please, use the “Open this sample in Visual Studio” link at the top right panel of the browser app to prepare the sample. The application opens in Visual Studio 2008.

## Scenario: Setting up and running the SingleFileGenerator

Rebuild the class library and start running it. However, **SingleFileGenerator** is not a VSPackage, during the build it has been registered with Visual Studio and the project is set up so that VS Experimental Hive starts.

Inside the new experimental hive instance of VS, create a new C# Class Library project and add an XML file to it; name it to **Types.xml**. Select the **Types.xml** file in the Solution Explorer and in the Properties window set its **CustomTool** property to **XmlClassGenerator**. At this point the file generator automatically starts generating C# /VB code from the **Types.xml** file. If you open the Error window you will find an error message there saying “XML document must contain a root level element”. This is due to the fact that we did not provide input information to generate code from.

Now, type the following code to the Types.xml file (or copy it from here):

<?xml version="1.0" encoding="utf-8" ?>

<Types xmlns="http://microsoft.com/XMLClassGeneratorSchema.xsd">

<Class name="MyClass" access="public">

<Constant name="MyInt" type="System.Int32" value="42"/>

<Variable name="\_MyField" access="private" type="System.String"/>

</Class>

</Types>

Save the file! The file generator now understands the schema of this file and generates a C# /VB code from it that is put into **Types.cs (Types.vb for VB sample)** file behind the **Types.xml** file as illustrated in Figure 1:

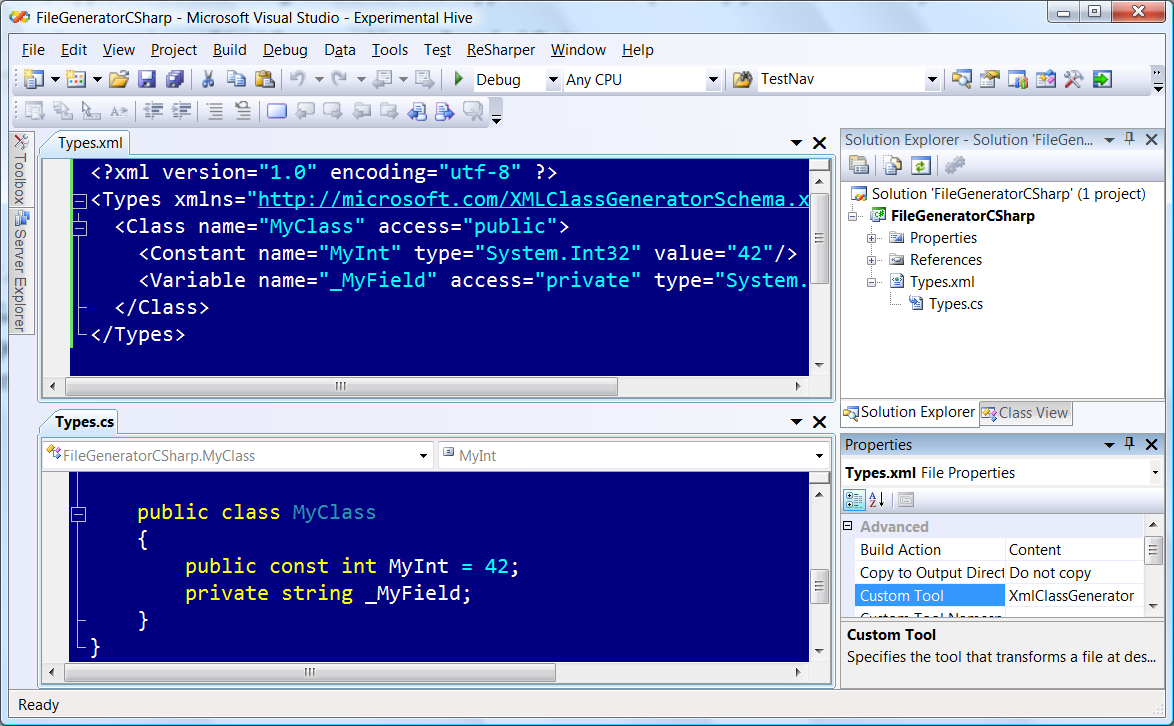


Figure 1: SingleFileGenerator has created the Types.cs file from Types.xml

This time when you open up the Error List window there will be no errors indicating that the **Types.xml** file has been successfully used to generate **Types.cs (Types.vb for VB sample)**.

Try the same scenario again but this time with a VB class library! At the end you will have a **Types.vb** file representing **MyClass** in VB.NET.

## The structure of the sample

The solution contains one C# class library named **GeneratorSample** that uses a few reference assemblies for VS interop starting with name “Microsoft.VisualStudio”. The project’s source files are the followings:

|  |  |
| --- | --- |
| Source file | Description |
| AssemblyInfo.cs/ AssemblyInfo.vb | Attributes decorating the assembly |
| ClassDiagram.cd | A class diagram summarizing project elements |
| CodeGeneratorRegistrationAttribute.cs/ CodeGeneratorRegistrationAttribute.vb | File for a registration attribute linked from the source code of VS SDK |
| Strings.resx | Various string resources used during code generation |
| BaseCodeGenerator.cs/ BaseCodeGenerator.vb | Abstract code generator implementing the **IVsSingleFileGenerator** interface |
| BaseCodeGeneratorWithSide.cs/ BaseCodeGeneratorWithSide.vb | Abstract code generator that can be sited in Visual Studio |
| XmlClassGenerator.cs/ XmlClassGenerator.vb | A single file generator creating code based on a well-formed XML file |
| XmlClassGeneratorSchema.xsd | Schema used to check the input XML file |
| SourceCodeGenerator.cs/ SourceCodeGenerator.vb | Class responsible for the generation of CodeDom-based source from the input XML infoset |

In the subsequent parts of this paper the focus is upon source files representing the essential elements of building up the single file editor. In code extracts within this deep dive I will omit or change comments to support better readability and remove using clauses, namespace declarations or other non-relevant elements.

## Scenario: Registering the custom tool

This VS SDK example demonstrates a project where the result of the compilation is not a VSPackage but a class library implementing a single file generator. This single file generator has to be registered with Visual Studio in order for VS to use it and for the single file generator to interact with VS—for example put notifications to the Error List window. So, the build process of this sample is set up so that the same build steps are executed for this class library as for VSPackages; the most important of them is running the **regpkg.exe** tool responsible to registering the single file generator. Actually, the sample contains an empty VSPackage but it does not contain any Package derived class.

The single file generator is implemented in the **XmlClassGenerator.cs** file (**XmlClassGenerator.vb for VB sample**). This file contains the information used to register the tool:

[ComVisible(true)]

[Guid("52B316AA-1997-4c81-9969-83604C09EEB4")]

[CodeGeneratorRegistration(typeof(XmlClassGenerator),

"C# XML Class Generator", vsContextGuids.vsContextGuidVCSProject,

GeneratesDesignTimeSource=true)]

[CodeGeneratorRegistration(typeof(XmlClassGenerator),

"VB XML Class Generator", vsContextGuids.vsContextGuidVBProject,

GeneratesDesignTimeSource = true)]

[CodeGeneratorRegistration(typeof(XmlClassGenerator),

"J# XML Class Generator", vsContextGuids.vsContextGuidVJSProject,

GeneratesDesignTimeSource = true)]

[ProvideObject(typeof(XmlClassGenerator))]

public class XmlClassGenerator : BaseCodeGeneratorWithSite { ... }

Our class is to be registered with Visual Studio, so it must be COM visible and have and explicit GUID. The **ProvideObject** attribute signs that the COM information about our object is registered under the appropriate VS registry key instead of globally under the HKEY\_CLASSES\_ROOT\CLSID section of the registry. The key is the three **CodeGeneratorRegistration** attribute instances that register our class for three context (C#, VB and J#) as a custom tool. The parameters of this attribute are the followings:

1. The first parameter specifies the type implementing the **IVsSingleFileGenerator** interface. This interface promotes a type to become a custom tool.
2. The second parameter gives a name to the code generator.
3. The third parameter (GUID) identifies the context where the generator would appear.

The **GeneratesDesignTimeSource** parameter indicates whether types from files produced by this custom tool are made available to visual designers. We set it true indicating availability.

The **CodeGeneratorRegistration** attribute (just like the elements of the custom project system) is not published in the VS SDK in binary form; instead, the related source files have been published. Our example refers back to the source code file in the VS SDK folder.

## Scenario: Building the Single File Generator

A single file generator is a COM component that implements the **IVsSingleFileGenerator** interface. Our **XmlClassGenerator** type does it through a number of inherited classes. In this scenario the following roles are undertaken by the participant classes:

|  |  |
| --- | --- |
| Class | Role |
| BaseCodeGenerator | This abstract class provides a simple implementation of the **IVsSingleFileGenerator** interface. |
| BaseCodeGeneratorWithSite | This abstract class derives from the **BaseCodeGenerator** class. This type also implements the **IObjectWithSite** interface and so can access not only the input source file but other information from VS (even other source files). |
| XmlClassGenerator | This class is derived from the **BaseCodeGeneratorWithSite** class and implements the class generation according to the information in the input file. Uses the static **SourceCodeGenerator** helper class. |

The pattern used here is the recommended one for your own single file generator. To understand how this pattern works, we look into details.

### BaseCodeGenerator and the IVsSingleFileGenerator interface

This interface provides only two methods that make a COM type a single file generator. To understand what these methods do, let’s see how they are implemented in the **BaseCodeGenerator** class:

**C#**

public abstract class BaseCodeGenerator : IVsSingleFileGenerator

{

// --- Non-relevant methods omitted

private IVsGeneratorProgress codeGeneratorProgress;

private string codeFileNameSpace = String.Empty;

private string codeFilePath = String.Empty;

int IVsSingleFileGenerator.DefaultExtension(

out string pbstrDefaultExtension)

{

pbstrDefaultExtension = GetDefaultExtension();

return VSConstants.S\_OK;

}

int IVsSingleFileGenerator.Generate(string wszInputFilePath,

string bstrInputFileContents,

string wszDefaultNamespace,

IntPtr[] rgbOutputFileContents,

out uint pcbOutput,

IVsGeneratorProgress pGenerateProgress)

{

codeFilePath = wszInputFilePath;

codeFileNameSpace = wszDefaultNamespace;

codeGeneratorProgress = pGenerateProgress;

byte[] bytes = GenerateCode(bstrInputFileContents);

if (bytes == null)

{

// --- This signals that GenerateCode() has failed.

rgbOutputFileContents = null;

pcbOutput = 0;

return VSConstants.E\_FAIL;

}

else

{

int outputLength = bytes.Length;

rgbOutputFileContents[0] = Marshal.AllocCoTaskMem(outputLength);

Marshal.Copy(bytes, 0, rgbOutputFileContents[0], outputLength);

pcbOutput = (uint)outputLength;

return VSConstants.S\_OK;

}

}

protected abstract string GetDefaultExtension();

protected abstract byte[] GenerateCode(string inputFileContent);

}

**VB.Net**

Public MustInherit Class BaseCodeGenerator

Implements IVsSingleFileGenerator

' --- Non-relevant methods omitted

Private codeGeneratorProgress As IVsGeneratorProgress

Private codeFileNameSpace As String = String.Empty

Private codeFilePath As String = String.Empty

Private Function

DefaultExtension(<System.Runtime.InteropServices.Out()> ByRef pbstrDefaultExtension As String) As Integer Implements IVsSingleFileGenerator.DefaultExtension

pbstrDefaultExtension = GetDefaultExtension()

Return VSConstants.S\_OK

End Function

Private Function Generate(ByVal wszInputFilePath As String, ByVal

bstrInputFileContents As String, ByVal wszDefaultNamespace As String, ByVal rgbOutputFileContents() As IntPtr, <System.Runtime.InteropServices.Out()> ByRef pcbOutput As UInteger, ByVal pGenerateProgress As IVsGeneratorProgress) As Integer Implements IVsSingleFileGenerator.Generate

codeFilePath = wszInputFilePath

codeFileNameSpace = wszDefaultNamespace

codeGeneratorProgress = pGenerateProgress

Dim bytes() As Byte = GenerateCode(bstrInputFileContents)

If bytes Is Nothing Then

' --- This signals that GenerateCode() has failed.

rgbOutputFileContents = Nothing

pcbOutput = 0

Return VSConstants.E\_FAIL

Else

Dim outputLength As Integer = bytes.Length

rgbOutputFileContents(0) = Marshal.AllocCoTaskMem(outputLength)

Marshal.Copy(bytes, 0, rgbOutputFileContents(0), outputLength)

pcbOutput = CUInt(outputLength)

Return VSConstants.S\_OK

End If

End Function

Protected MustOverride Function GetDefaultExtension() As String

Protected MustOverride Function GenerateCode(ByVal inputFileContent As String) As Byte()

End Class

(Please note, that this is not the full code of the class.)

The **DefaultExtension** method does what its name says: it delegates its task to the abstract **GetDafaultExtensions** method. The **Generate** method is responsible to make the “hard work”. This method accepts six parameters:

**wszInputFilePath** specifies the full path of the input file.

**bstrInputFileContents** is the contents of the input file. The project system converts text files to Unicode string.

**wszDefaultNamespace** is the value of the Custom Tool Namespace property of the input file. The custom tool can use this parameter on its own intention; generally it is used to set the namespace of a generated type.

**rgbOutputFileContents** represents an array of bytes to be written to the generated file. This byte stream needs special handling, look the related comments in the source code.

**pcbOutput** retrieves the count of bytes in the **rgbOutputFileContent** array.

**pGenerateProgress** is a reference to the **IVsGeneratorProgress** interface. The generator can report its progress to the project system through this reference.

This method inside uses the abstract **GenerateCode** accepting the file content; and retrieves the byte array representing the generated code.

### The BaseCodeGeneratorWithSite class

The code generator implemented in this sample can create code in the language of the current development context. For example, if the XML file is in a C# project, it generates .cs files; in case of a VB projects, it generates .vb file. It is not enough only to process the input file; our code generator must be able to access VS context information through VS services. The **BaseCodeGenerator** class is responsible to provide access to VS services by implementing the **IObjectWithSite** interface:

**C#**

public abstract class BaseCodeGeneratorWithSite :

BaseCodeGenerator, IObjectWithSite

{

private object site = null;

void VSOLE.IObjectWithSite.GetSite(ref Guid riid,

out IntPtr ppvSite)

{

if (site == null)

{

throw new COMException("object is not sited",

VSConstants.E\_FAIL);

}

IntPtr pUnknownPointer = Marshal.GetIUnknownForObject(site);

IntPtr intPointer = IntPtr.Zero;

Marshal.QueryInterface(pUnknownPointer, ref riid, out intPointer);

if (intPointer == IntPtr.Zero)

{

throw new COMException(

"site does not support requested interface",

VSConstants.E\_NOINTERFACE);

}

ppvSite = intPointer;

}

void VSOLE.IObjectWithSite.SetSite(object pUnkSite)

{

site = pUnkSite;

codeDomProvider = null;

serviceProvider = null;

}

}

**VB.Net**

Public MustInherit Class BaseCodeGeneratorWithSite

Inherits BaseCodeGenerator

Implements IObjectWithSite

Private site As Object = Nothing

Private Sub GetSite(ByRef riid As Guid,

<System.Runtime.InteropServices.Out()> ByRef ppvSite As IntPtr) Implements VSOLE.IObjectWithSite.GetSite

If site Is Nothing Then

Throw New COMException("object is not sited", VSConstants.E\_FAIL)

End If

Dim pUnknownPointer As IntPtr = Marshal.GetIUnknownForObject(site)

Dim intPointer As IntPtr = IntPtr.Zero

Marshal.QueryInterface(pUnknownPointer, riid, intPointer)

If intPointer = IntPtr.Zero Then

Throw New COMException("site does not support requested interface", VSConstants.E\_NOINTERFACE)

End If

ppvSite = intPointer

End Sub

Private Sub SetSite(ByVal pUnkSite As Object) Implements VSOLE.IObjectWithSite.SetSite

site = pUnkSite

codeDomProvider = Nothing

serviceProvider = Nothing

End Sub

End Class

**IObjectWithSite** contains two methods: **SetSite** to site the object and **GetSite** to obtain the current site of the object. Through the site we can access the service provider and so actually VS services:

**C#**

private ServiceProvider SiteServiceProvider

{

get

{

if (serviceProvider == null)

{

serviceProvider =

new ServiceProvider(site as VSOLE.IServiceProvider);

}

return serviceProvider;

}

}

**VB.Net**

Private ReadOnly Property SiteServiceProvider() As ServiceProvider

Get

If serviceProvider Is Nothing Then

serviceProvider = New ServiceProvider(TryCast(site, VSOLE.IServiceProvider))

End If

Return serviceProvider

End Get

End Property

The **BaseCodeGeneratorWithSite** class provides in important service for derived classes: it can retrieve the **CodeDomProvider** used and implements the **GetDefaultExtension** method of the custom tool. The class uses the current language context and so, using the appropriate CodeDom and file extension, the right language with the right extension can be used for code generation.

### The XmlClassGenerator class

The final custom tool class in the sample is the **XmlClassGenerator** class. This is a concrete class that turns the XML file it is associated with into generated source code.

The class overrides the **GenerateCode(string)** abstract method defined in the **BaseCodeGenerator**. With this method the class becomes a full-valued code generator. The implementation of **GenerateCode** uses the **SourceCodeGenerator** static helper. This is where the actual code generation happens: the XML is parsed and the corresponding CodeDom representation of the output is built up.

## Scenario: Accessing the current CodeDomProvider information

A great value of **SingleFileGenerator** is that it provides context-dependent code generation according to the current language project context. There are many ways to generate source code from an XML infoset. This sample uses the **CodeDom** which is a language-independent description. **CodeDom** is an abstract code model including namespaces, types, type members, statements, expressions etc. The greatness of using it is the fact that the same abstract model can be presented in several languages by applying the corresponding **CodeDomProvider**.

Here I show how **SingleViewEditor** accesses information to use the appropriate language model. The **BaseCodeGeneratorWithSite** class provides the key methods to it:

**C#**

public abstract class BaseCodeGeneratorWithSite :

BaseCodeGenerator, IObjectWithSite

{

// --- Non-relevant methods are omitted

protected object GetService(Type serviceType)

{

return SiteServiceProvider.GetService(serviceType);

}

protected virtual CodeDomProvider GetCodeProvider()

{

if (codeDomProvider == null)

{

IVSMDCodeDomProvider provider =

GetService(typeof(SVSMDCodeDomProvider))

as IVSMDCodeDomProvider;

if (provider != null)

{

codeDomProvider = provider.CodeDomProvider as CodeDomProvider;

}

else

{

codeDomProvider = CodeDomProvider.CreateProvider("C#");

}

}

return codeDomProvider;

}

}

**VB.Net**

Public MustInherit Class BaseCodeGeneratorWithSite

Inherits BaseCodeGenerator

Implements IObjectWithSite

' --- Non-relevant methods are omitted

Protected Function GetService(ByVal serviceType As Type) As Object

Return SiteServiceProvider.GetService(serviceType)

End Function

Protected Overridable Function GetCodeProvider() As CodeDomProvider

If codeDomProvider Is Nothing Then

Dim provider As IVSMDCodeDomProvider = TryCast(GetService(GetType(SVSMDCodeDomProvider)), IVSMDCodeDomProvider)

If provider IsNot Nothing Then

codeDomProvider = TryCast(provider.CodeDomProvider, CodeDomProvider)

Else

codeDomProvider = CodeDomProvider.CreateProvider("VisualBasic")

End If

End If

Return codeDomProvider

End Function

End Class

The **GetService** method uses the site information we stored in the **GetSite** method. Through this site we can obtain Visual Studio services. The **GetCodeProvider** method uses the **SVSMDCodeDomProvider** service to retrieve the provider according to the current language. If this method does not return a provider (for example no language specific solution is open) the method uses the provider related to the C# language.

The **GetCodeProvider** method is used in the **XmlClassGenerator** class’s **GenerateCode** method. The following extract illustrates how the provider is used there:

**C#**

protected override byte[] GenerateCode(string inputFileContent)

{

// --- Validating the XML file against the schema is omitted

CodeDomProvider provider = GetCodeProvider();

XmlDocument doc = new XmlDocument();

doc.LoadXml(inputFileContent);

CodeCompileUnit compileUnit =

SourceCodeGenerator.CreateCodeCompileUnit(doc, FileNameSpace);

using (StringWriter writer = new StringWriter(new StringBuilder()))

{

CodeGeneratorOptions options = new CodeGeneratorOptions();

options.BlankLinesBetweenMembers = false;

options.BracingStyle = "C";

provider.GenerateCodeFromCompileUnit(compileUnit, writer,

options);

writer.Flush();

// --- Converting to the result byte[] omitted

}

}

**VB.Net**

Protected Overrides Function GenerateCode(ByVal inputFileContent As String) As Byte()

' --- Validating the XML file against the schema is omitted

Dim provider As CodeDomProvider = GetCodeProvider()

Dim doc As New XmlDocument()

doc.LoadXml(inputFileContent)

Dim compileUnit As CodeCompileUnit = SourceCodeGenerator.CreateCodeCompileUnit(doc, FileNameSpace)

Using writer As New StringWriter(New StringBuilder())

Dim options As New CodeGeneratorOptions()

options.BlankLinesBetweenMembers = False

options.BracingStyle = "C"

provider.GenerateCodeFromCompileUnit(compileUnit, writer, options)

writer.Flush()

' --- Converting to the result byte[] omitted

End Using

End Function

(Please note, there are statements omitted from the code for clarity)

The hard work of code generation here is done by the **CreateCodeCompileUnit** of the **SourceCodeGenerator** class. You can observe, this method does not use the provider we obtained from VS. The provider is used “only” for creating the source code text from the **CodeCompileUnit**.

# Summary

Single file generators are great tools to reduce the efforts spent on repeatable and mechanic tasks, mainly when you have to create source code based on well-defined coding patterns. Your source file contains the input data for the code to generate, the single file generator contains the pattern logic to create the code based on the input data.

This sample implements a single file generator. In Visual Studio a single file generator is a COM object implementing the **IVsSingleFileGenerator** interface. Our single file generator is implemented in the **XmlClassGenerator** class and is able to create simple types from an XML definition in the language of the current project context.

Single file generators must be registered in order Visual Studio can find and use them. The registration process leverages on attributes decorating the custom tool class.

The implementation pattern used here (hierarchy of classes from **BaseCodeGenerator** to **XmlClassGenerator**) is a good starting point for your own generators.