RoslynDom and Friends – Just the Facts

See [this post](http://msmvps.com/blogs/kathleen/archive/2014/06/27/roadmap-for-roslyndom-codefirst-strong-typed-metadata-and-expansionfirst-templates.aspx) for the Roadmap of these projects

# RoslynDom

A wrapper for the .NET Compiler Platform – the roadmap has further plans

[Project on GitHub](https://github.com/KathleenDollard/RoslynDOM)

See the RoslynDomExampleTests project in the solution for the 20 things you’re most likely to do

Download via Visual Studio NuGet Package Manager if you want to play with that

# RoslynDom-Provider

By Jim Christopher

A PowerShell provider for Roslyn Dom

[Project on GitHub](https://github.com/beefarino/roslyndom-provider)

# CodeFirstMetadata

Strong-typed metadata from code-first (general sense, not Entity Framework sense)

[Project on GitHub](https://github.com/KathleenDollard/CodeFirstMetadata)

See the ConsoleRunT4Example project in the solution along with strong-typed files and T4 usage

Strong-typed Metadata

Your code is code and your code is data.

Metaprogramming opens up worlds where you care very much that your code is data. Editor enhancements open up worlds where you care very much that your code is data. Visualizations open up worlds where you care very much that your code is data. And I think that’s only the beginning.

There’s nothing really new about thinking of code as data. Your compiler does it, metaprogramming techniques do it, and delegates and functional programming do it.

So, let’s make your code data. Living breathing **strongly-typed** data. Strong typing means describing the code in terms of the underlying problem and providing this view as a first class citizen rather than a passing convenience.

## Describing the Underlying Problem

I’ll use logging as an example, because the simpler problem of PropertyChanged just happens to have an underlying problem of classes and properties, making it nearly impossible to think about with appropriate abstractions. Class/property/method is only interesting if the underlying problem is about classes, properties and methods.

The logging problem is not class/method – it’s log/log event. When you strongly type the metadata to classes that describe the problem being solved you can reason about code in a much more effective manner. Alternate examples would be classes that express a service, a UI, a stream or an input device like a machine.

I use EventSource for logging, but my metadata describes the problem in a more generalized way – it describes it as a SemanticLog. A SemanticLog looks like a class, and once you create metadata from it, you can create any logging system you want.

Your application has a handful of conceptual groups like this. Each conceptual group has a finite appropriate types of customization. Your application problem also has a small number of truly unique classes.

## Treating Metadata as a First Class Citizen

In the past, metadata has been a messy affair. The actual metadata description of the underlying patterns of your application have been sufficiently difficult to extract that you’ve had no reason to care,. Thus, tools like the compiler that treated your code as data simply created the data view it needed and tossed in out as rubbish when it was done.

The .NET Compiler Platform, Roslyn, stops throwing away its data view. It exposes it for us to play with.

## Usage Examples

I’m interested in strongly typed metadata to write templates for metaprogramming. I want these template to be independent of how you are running them – whether they are part of code generation, metaprogramming, a code refactoring or whatever. I also want these templates to be independent of how the metadata is loaded.

Strongly typed metadata works today in T4 templates. My CodeFirstMetadata project has examples.

I’m starting work on expansion first templates and there are many other ways to use strong-typed metadata - both for other metaprogramming techniques and completely different uses. One of the reasons I’m so excited about this project is to see what interesting things people do, once their code is in a strong-typed form. At the very least, I think it will be an approach to visualizations and ensuring your code follows expected patterns. It will be better at ensuring large scale patterns than code analysis rules. Whew! So much fun work to do!!!

# Strong-typed Metadata in a T4 Template

Here’s a sample of strong typing in a T4 template

<#@ template language="C#" inherits="CodeFirstT4CSharpBase<CodeFirstSemanticLog>" #>

<#@ assembly name="System.Core" #>

<#@ assembly name="$(SolutionDir)\CodeFirstEventSource\bin\Debug\CodeFirstSemanticLog.Dll" #>

<#@ assembly name="$(SolutionDir)\CodeFirstEventSource\bin\Debug\CodeFirstMetadataCommon.Dll" #>

<#@ assembly name="$(SolutionDir)\CodeFirstMetadataT4Support\bin\Debug\CodeFirstMetadataT4Support.Dll" #>

<#@ import namespace="System.Linq" #>

<#@ import namespace="CodeFirst.Common" #>

<#@ import namespace="CodeFirstMetadataTest.SemanticLog" #>

<#

OutputGenerationWarning();

…

#>

using System;

using System.Diagnostics.Tracing;

<# OutputNamespaceOpen(); #>

<# if (Meta.IncludesInterface) { #>

<#@ include file="EventSourceWithInterface.t4" #>

<# }

else

{ #>

<#@ include file="EventSourceNormal.t4" #>

<# } #>

<# OutputNamespaceClose(); #>

There’s some gunk at the top to add some assemblies and some using statements for the template itself. The important piece at the top is that the class created by this template is a generic type with a type argument – CodeFirstSemanticLog – that is a strong-typed metadata class. Thus the Meta property of the CodeFirstT4CSharpBase class ***is*** a SemanticLog class and understands concepts specific to the SemanticLog, like IncludesInterface. I’ve removed a few variable declarations that are specific to the included T4 files.

Code-first metadata

*This is “code first” in the general sense, not the specific sense of Entity Framework. This has nothing to do with Entity Framework at all, except that team showed us how valuable simple access to code-like metadata is.*

Code first is a powerful mechanism for expressing your metadata because code is the most concise way to express many things. There’s 60 years of evolution to todays’ computer languages being efficient in expressing explicit concepts based on a natural contextualization. You can’t get this in JSON, XML or other richer and less-opinionated formats.

Code first is just one approach to getting strong-typed metadata. The keys to the kingdom, the keys to your code, lie in expressing the underlying problems of your code in a strong-typed manner, which you can read about [here](http://msmvps.com/blogs/kathleen/archive/2014/06/27/strong-typed-metadata.aspx).

The problem is that the description of the problem is wrapped up with an enormous amount of ceremony about ***how*** to do ***what*** we’re trying to do. Let’s look at this in relation to metaprogramming where the goal is generally to reduce ceremony and

***Only write the code that only you can write***

In other words, don’t write any code that isn’t part of the minimum definition of the problem, divorced of all technology artifacts.

For example, you can create a SemanticLog definition that you can later output as an EventSource class, or any other kind of log output – even in a different language or on a different platform.

To do this, describe the SemanticLog in the simplest way possible, devoid of technology artifacts.

[SemanticLog()]

public class Normal

{

public void Message(string Message) {}

public void AccessByPrimaryKey(int PrimaryKey){}

}

Instead of the EventSource version:

using System;

using System.Diagnostics.Tracing;

namespace ConsoleRunT4Example

{

public sealed partial class Normal : EventSource

{

#region Standard class stuff

// Private constructor blocks direct instantiation of class

private Normal() {}

// Readonly access to cached, lazily created singleton instance

private static readonly Lazy<Normal> \_lazyLog =

new Lazy<Normal>(() => new Normal());

public static Normal Log

{

get { return \_lazyLog.Value; }

}

// Readonly access to private cached, lazily created singleton inner class instance

private static readonly Lazy<Normal> \_lazyInnerlog =

new Lazy<Normal>(() => new Normal());

private static Normal innerLog

{

get { return \_lazyInnerlog.Value; }

}

#endregion

#region Your trace event methods

[Event(1)]

public void Message(System.String Message)

{

if (IsEnabled()) WriteEvent(1, Message);

}

[Event(2)]

public void AccessByPrimaryKey(System.Int32 PrimaryKey)

{

if (IsEnabled()) WriteEvent(2, PrimaryKey);

}

#endregion

}

}

Writing less code because we are lazy is a noble goal. But the broader benefit here is that the first requires very little effort to understand and very little trust about whether the pattern is followed. The second requires much more effort to read the code and ensure that everything in the class is doing what’s expected. The meaning of the code requires that you know what an EventSource is.

Code-first allows you to just write the code that only you can write, and leave it to the system to create the rest of the code based on your minimal definition.

Creating Strong-typed Metadata Classes

This post is about an aspect of the CodeFirstMetadata library. You can find out more about this library and where to get it [here](http://msmvps.com/blogs/kathleen/archive/2014/06/27/roadmap-for-roslyndom-codefirst-strong-typed-metadata-and-expansionfirst-templates.aspx) and [here](http://msmvps.com/blogs/kathleen/archive/2014/06/27/roslyndom-and-friends-just-the-facts.aspx).

You can find out more about strong-typed metadata classes [in this post](http://msmvps.com/blogs/kathleen/archive/2014/06/27/strong-typed-metadata.aspx).

You can find out about code-first (generalized, not Entity Framework) [here](http://msmvps.com/blogs/kathleen/archive/2014/06/27/code-first-metadata.aspx).

This post talks about the two existing examples to explain how strong typing works in real code and to show how instances of these examples are created.

At present, in order to create a set of strong typed classes to solve a **new** problem you need to create a fairly messy set of classes. Feel free to ping me if you think you have a good problem or you want to extend the existing problems and I’ll help guide you. In the long run I want to automate that process, so I probably won’t document it until then.

Because part will be automated/generated, it comes in two parts. I’m currently combining them with inheritance, rather than partial classes, to make this code approachable for non-.NET programmers, and because virtual/override are simpler concepts.

These classes all derive from a common base class – CodeFirstMetadata<T> - to provide common features like naming. Below this are code element specific classes like CodeFirstMetadataClass<T> that help with the conversion. I may later replace this with a shallow hierarchy and interfaces, so don’t get dependent on this implementation.

For a semantic log, the class, the predictable part I’ll later generate looks like:

using System.Collections.Generic;

using CodeFirst.Common;

namespace CodeFirstMetadataTest.SemanticLog

{

// TODO: Generate this base class based on expected attributes

public abstract class CodeFirstSemanticLogBase : CodeFirstMetadataClass<CodeFirstSemanticLog>

{

public CodeFirstSemanticLogBase()

{

this.Events = new List<CodeFirstLogEvent>();

}

public virtual string Name { get; set; }

public virtual string LocalizationResources { get; set; }

public IEnumerable<CodeFirstLogEvent> Events { get; private set; }

}

}

The manual changes I’ve made, which are by far the most complex I’ve needed so far are:

using System.Linq;

using CodeFirst.Common;

namespace CodeFirstMetadataTest.SemanticLog

{

public class CodeFirstSemanticLog : CodeFirstSemanticLogBase

{

private string \_name;

public override string Name

{

get

{

if (string.IsNullOrWhiteSpace(\_name))

{ return Namespace.Replace(".", "-") + "-" + ClassName; }

return \_name;

}

set

{ \_name = value; }

}

public string UniqueName

{

get

{

var tempName = this.Name.SubstringAfterLast("-");

return tempName;

}

}

public bool IncludesInterface

{ get { return this.ImplementedInterfaces.Count() > 0; } }

public bool IsLocalized

{ get { return !string.IsNullOrWhiteSpace(this.LocalizationResources); } }

public override bool ValidateAndUpdateCore()

{

var isOk = base.ValidateAndUpdateCore();

if (isOk)

{ return CheckAndUpdateEventIds(); }

return false;

}

/// <summary>

/// This is a weird algorithm because it numbers implicit events from

/// the top, regardless of whether other events have event IDs. But

/// while I wouldn't have chosen this, I think it's important to match

/// EventSource implicit behavior exactly.

/// </summary>

private bool CheckAndUpdateEventIds()

{

var i = 0;

foreach (var evt in this.Events)

{

i++;

if (evt.EventId == 0) evt.EventId = i;

}

// PERF: The following is an O<n2> algorithm, probably a better way

var dupes = this.Events

.Where(x => this.Events

.Any(y => (y != x) && x.EventId == y.EventId));

return (dupes.Count() == 0);

}

}

}

EventSource, and presumably any other log system, requires a unique name, and I want to help you create that. Also, whether there is an interface and whether the class is localized have a significant impact on the template, so I simplify access to this information.

Loading strong-typed metadata is an opportunity for validation of the model. I use this to provide unique numeric ids to each of the log events, which are needed by EventSource and potentially other log mechanisms.

# Mapping Between Code-first and Strong-typed Metadata

A bunch of ugly Roslyn and reflection code maps between code-first and strong typed metadata. This is the code that drove creation of the RoslynDom library – directly hitting the .NET Compiler Platform/Roslyn API within this code was monstrous.

var cfNamespace = root.Namespaces.First();

var returnType = typeof(T);

var mapping = TargetMapping.DeriveMapping("root", "root", returnType.GetTypeInfo()) as TargetNamespaceMapping;

var mapper = new CodeFirstMapper();

var newObj = mapper.Map(mapping, cfNamespace);

* cfNamespace is the RolsynDom root
* T is the type to return – the strong-typed metadata
* mapping derived data about the mapping of the target– just create it as shown
* mapper is the class that does the hard work
* newObj is the new strong-typed metadata object

In the end, you have an object that is the strong-typed metadata for the initial code.

## OK, but how does that work?

For metaprogramming:

* I create a minimal description is in a file with a .cfcs extension
* I lie to Visual Studio and tell it that this is a C# file (Tools/Options/Text Editor/File Extensions) I get nice IntelliSense for most features (more work to be done later).
* MSBuild doesn’t see it as a C# file, so the .cfcs files are ignored as source in compilation
* Generation creates .g.cs files that are included in compilation

The intent is to have this automated as part of your normal development pipeline, through one or more mechanism – build, custom tools, VS extension/PowerShell. The pipeline part is not done yet, but you can grab the necessary pieces from the console application in the example.

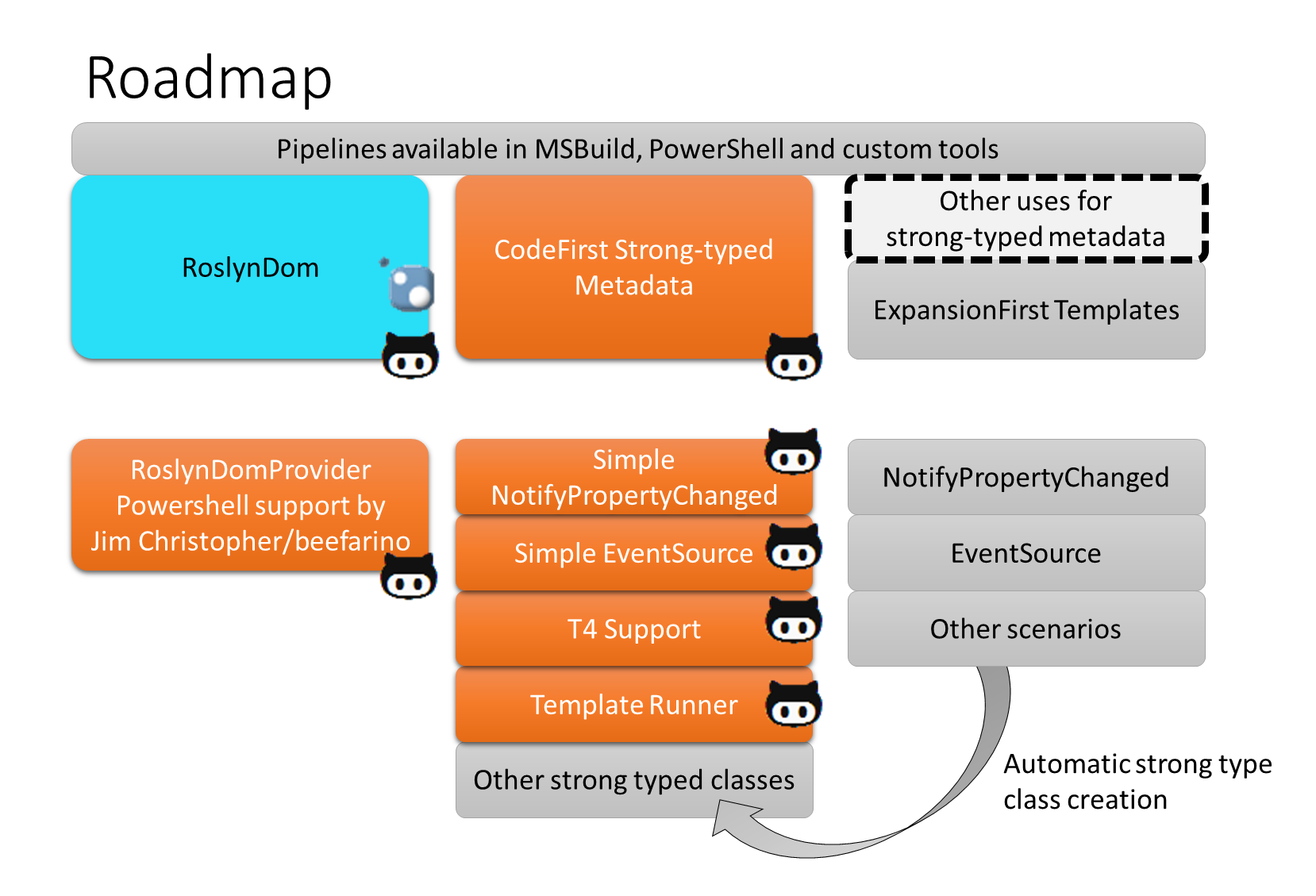
## Getting CodeFirstMetadata

You can [get this project on GitHub](https://github.com/KathleenDollard/CodeFirstMetadata). I’ll add this to NuGet when the samples are in a more accessible from your Visual Studio project.

Roadmap for RoslynDom, CodeFirst Strong-typed Metadata and ExpansionFirst Templates

I’ve been working on three interleaved projects RoslynDom, CodeFirst Strong-typed Metadata and ExpansionFirst Templates. Also, Jim Christopher (aka [beefarino](http://www.beefycode.com/)) built a PowerShell provider. This post is an overview of these projects and a roadmap of how they relate to each other.

You can find the short version here.



In the roadmap, blue indicates full (almost) test coverage and that the library has had more than one user, orange indicates preliminary released code, and grey indicates code that it’s really not ready to go and not yet available.

I’m working left to right, waiting to complete some features of the RoslynDom library until I have the full set of projects available in preliminary form.

# RoslynDom Library

.NET Compiler Services, or Roslyn, does exactly what it was intended to do, which is exactly what we want it to do. It’s a very good compiler, now released as open source, and exposing all of its internals. It’s great that we get access to the internal trees, but it’s not happy code for you and I to use – it’s compiler internals.

At the same time, these trees hold a wealth of information we want – it’s more complete information than reflection, holds design information like comments and XML documentation, and it’s available even when the source code doesn’t compile.

When you and I ask questions about our code, we ask simple things – what are the classes in this file? We don’t care about whitespace, or precisely how we defined namespaces. In fact, most of the time, we don’t even care about namespaces at all. And we certainly don’t care whether a piece of information is available in the syntactic or semantic tree or whether attributes were defined with this style or that style.

RoslynDom wraps the Roslyn compiler trees and exposes the information in a programmer friendly way. Goals include

* Easy access to the tree in the way(s) programmers think about code as a hierarchy
* Easy access to common information about the code as parameters
* Access to the applicable SyntaxNode when you need it
* Access to the applicable Symbol when you need it
* *Planned: Access to the full logical model - solution to smallest code detail*
  + Currently, file down to member
* *Planned: A kludged public annotation/design time attribute system until we get a real one*
  + Currently, attribute support only
* *Planned: Ability to morph and output changes*
  + Currently, readonly

## Getting RoslynDom

You can get [the source code on GitHub](https://github.com/KathleenDollard/RoslynDOM), and there’s a RoslynDomExampleTests project which shows how to do about 20 common things.

The project is also available via NuGet. It’s preliminary, use cautiously. Download with the Visual Studio NuGet package manager.

# RoslynDom-Provider

Jim Christopher created a PowerShell provider for RoslynDom. PowerShell providers allow you to access the underlying tree of information in the same way you access the file system. IOW, you can mount your source code as though it was a drive.

I’m really happy about the RoslynDom-Provider. It shows one way to use a .NET Compiler Platform/library to access the information that’s otherwise locked into the compiler trees. It’s also another way for you to find out about the amazing power of PowerShell providers. If you’re new to PowerShell, and you’re a Pluralsight subscriber, check out “Discovering PowerShell with Mark Minasi”. It uses Active Directory as the underlying problem and a few parts may be slow for a developer, but it will give you the gist of it. Follow up with Jim Christopher’s “Everyday PowerShell for Developers” and “PowerShell Gotchas.” If you’d rather read, there are a boatload of awesome books including [PowerShell Deep Dives](http://www.amazon.com/PowerShell-Deep-Dives-Jeffery-Hicks/dp/1617291315) and [Windows PowerShell for Developers](http://www.amazon.com/Windows-PowerShell-Developers-Douglas-Finke/dp/1449322700/ref=sr_1_1?s=books&ie=UTF8&qid=1403873238&sr=1-1&keywords=powershell+finke), and too many Internet sites for me to keep straight.

## Getting RoslynDomProvider

This [project is available on GitHub](https://github.com/beefarino/roslyndom-provider).

# Code-first Strong-typed Metadata

You can find out more about strong-typed metadata [here](http://msmvps.com/blogs/kathleen/archive/2014/06/27/strong-typed-metadata.aspx) and code-first strong-typed metadata [here](http://msmvps.com/blogs/kathleen/archive/2014/06/27/code-first-metadata.aspx).

As a first step, I have samples in runtime T4. These run from the command line at present. These templates inherit from a generic base class that has a property named Meta**.** This property is typed to the underlying strong-typed metadata item – in the samples either CodeFirstSemanticLog or CodeFirstClass. The EventSource template and problem is significantly more complex, but avoids some extra mind twisting with a strong-typed metadata class around a class. These templates are preliminary and do not handle all scenarios.

## Metaprogramming

While there are a couple of ways to solve a metaprogramming expansion or code first problem, I’ve settled on an alternate file extension. The code-first minimal description is in a file with a .cfcs extension. Because I lie to Visual Studio and tell it that this is a C# file (Tools/Options/Text Editor/File Extensions) I get nice IntelliSense for most features (more work to be done later). But because MSBuild doesn’t see it as a C# file, the .cfcs file is ignored as a source file in compilation.

Generation produces an actual source code file in a file with a .g.cs extension. This file becomes part of your project. This is the “real” code and you debug in this “real” code because it’s all the compiler and debugger know about. As a result

* You write is *the minimal code that only you can write*
* You understand your application through either the minimal or expanded code
* You easily recognize expanded code via a .g.cs extension
* You can place the minimal and expanded code side by side to understand the expansion
* You debug in real code
* You protect the generated code by allowing only the build server to check in these files

Again this happens because there are two clearly differentiated files in your project – the .cfcs file and the .g.cs file.

The intent is to have this automated as part of your normal development pipeline, through one or more mechanism – build, custom tools, VS extension/PowerShell. The pipeline part is not done yet, but you can grab the necessary pieces from the console application in the example.

## Getting CodeFirstMetadata

You can [get this project on GitHub](https://github.com/KathleenDollard/CodeFirstMetadata).

I’ll add this to NuGet when the samples are in a more accessible from your Visual Studio project.

# ExpansionFirst Templates

T4 has brought us a very long way. It, and CodeSmith have had the lion’s share of code generation templating in the .NET world for about a decade. I have enormous respect for people like [Gareth Jones](http://blogs.msdn.com/b/garethj/) who wrote it and kept it alive and [Oleg Sych](http://www.olegsych.com/) who taught so many people to use it. But it sucks, and it’s time to move on. Look for more upcoming on this – my current bits are so preliminary that I’ll wait to post.

# Summary

I look forward to sharing the unfinished pieces of this roadmap in the coming weeks and months.

I’d like to offer a special thanks to the folks in my April DevIntersection workshop. The challenges of explaining the .NET Compiler Platform/Roslyn pieces to you let me to take a step back and isolate those pieces from the rest of the work. While this put me way behind schedule, in the end I think it’s valuable both in simplifying the metaprogramming steps and in offering a wrapper for the .NET Compiler Platform/Roslyn.

Abandoned Text

EventSource is the .NET 4.5 class for tracing, generally semantic tracing. You create an instance of it, generally named Log, and call methods like StartMortgageCalculation() or AccessByPostalCode(string postalCode). These methods are called Events in the language of EventSource.

Thus, the strongly typed metadata for EventSource is a group of event sources, event sources, events, and event parameters – EventSourceGroup, EventSource, Event, and EventParameter classes respectively. These four classes describe the problem completely. All of the event sources in your application could be described as a set of instances these classes.