The CodeFirstMetadata project offers metadata management and a Dependency Injection based template runner that is capable of running both T4 preprocessed and Expansion First Templates. The CodeFirstMetadata project relies on RoslynDom to do the heavy lifting.

I expect this project, at least at this point in its evolution, to be of interest to people that want to build rich metaprogramming experiences that make it easy for people to solve specific problems. It is possible that it will later be a repository for common metadata structures.

Code Generation Overview

---

There are a number of ways to think about code generation. The straightforward approach that guides these projects is that metadata and custom code is combined with (or run through) one or more templates to create the code to output. Each template corresponds to a single output file, although templates may be made of reusable parts.

Metadata

---

The role of metadata in this generation approach is to be the simplest possible representation of the aspects of the application you are generating. The metadata allows you to understand the application in isolation from the technology used to create it.

There many approaches to metadata. It turns out one of the easiest approaches for programmers to use is code itself. While it’s easy to imagine using code as metadata via reflection – there are a couple problems with that approach. You have to be able to compiler the code, which is particularly problematic in systems where the metadata is part of the system it is generating. More fundamentally, reflections does not support all the possibilities for metadata creation – specifically excluding representational forms, discussed below.

So, instead of reflection, CodeFirstMetadata tackles the mapping problems in the different approaches to metadata.

Direct and Parallel Mapping

---

The common form, the one made popular by Microsoft’s Entity Framework is that the classes and properties of the metadata map to classes and properties in the metadata. I call this Direct Mapped Metadata because the logical format of the metadata maps directly to the metadata – in other words, the metadata contains an IEnumerable of classes and each of them contain an IEnumerable of properties.

Direct Mapping is the strictest case of Parallel Mapping. In the Entity Framework case, Direct Mapping is so close that attributes in the metadata reflect the technology. Direct mapping can also be looser.

An example of Direct Mapping is

```C#

public class Customer

{

public string FirstName{get; set;}

public string LastName{get; set;}

public int Id{get; set;}

public DateTime BirthDate{get; set;}

}

```

Parallel Mapping is nearly the same, aspects of the metadata parallel features in the metadata. A great example of Parallel Mapping is EventSource. The classes of your metadata directly map to the EventSource metadata element and the methods of the metadata directly map to the Event metadata elements. If there are more Event elements, there are more uniquely named methods in the class.

An example of parallel mapping is:

```C#

[SemanticLog()]

[UniqueName("MyUniqueName")]

public class Normal

{

public void Message(string Message) { }

[EventId(3)]

public void AccessByPrimaryKey(int PrimaryKey) { }

}

````

Certain problems are a good fit for Parallel or Direct Mapping. These problems where the using programmer is likely to think in terms of new classes, methods and properties.

Representational Mapping

---

The programmers using a metadata system are sometimes better served by an approach that guides them through the decisions and steps of metadata creation. A great example of this kind of problem is Visual Studio 2015 code analyzers and code fixes. In this case, each class contains the same methods and properties.

An example of representational mapping is:

```C#

public class IfElseBraceDiagnosticCodeFirst : DiagnosticAndCodeFixBase

{

public IfElseBraceDiagnosticCodeFirst()

{

Id = "KADGEN1001";

Description = "Needs braces";

MessageFormat = "{0} needs braces";

Category = "Style";

AddAnalyzer<IfStatementSyntax>(

syntaxKind: SyntaxKind.IfStatement,

condition: x => !x.Statement.IsKind(SyntaxKind.Block),

getLocation: x => x.IfKeyword.GetLocation(),

messageArg: "if statement");

AddCodeFix<IfStatementSyntax>(

makeNewNode: x => x.WithStatement(

SyntaxFactory.Block(x.Statement)));

AddAnalyzer<ElseClauseSyntax>(

syntaxKind: SyntaxKind.IfStatement,

condition: x => !x.Statement.IsKind(SyntaxKind.Block)

&& !x.Statement.IsKind(SyntaxKind.IfStatement),

getLocation: x => x.ElseKeyword.GetLocation(),

messageArg: "else statement");

AddCodeFix<ElseClauseSyntax>(

makeNewNode: GetNewElseNode);

}

private SyntaxNode GetNewElseNode(ElseClauseSyntax elseClause)

{

return elseClause.WithStatement(SyntaxFactory.Block(elseClause.Statement));

}

}

```

The programmer assigns properties and calls methods of the constructor. In Representational Mapping, no new classes or Properties are created, except for custom code.

There are no technical reasons that parallel/direct mapping couldn’t be used together. I just haven’t thought of an example, and therefore haven’t tested it.

Template Runner

---

Code generation has had a long history of mediocre to bad template runners. At the moment, I’ve focused only on the internals of a template runner that embraces CodeFirstMetadata and supports any template style that can be accessed via dependency injection. So, perhaps I have not improved the overall scenario – but this template runner can be incorporated in the future into MSBuild, PowerShell, a Visual Studio extension, and a custom tool that automatically runs when metadata files are changed.