# Notes on CodeFirstMetadata Representational Form

These are my notes so I don’t have to rethink this a dozen times.

I’ve tried three syntaxes for the Representational form. Two depend on a base class for analyzers, and the third creates an instance of a class. I want to focus on the syntax, not the base class, but the purpose of the base class or the class to instantiate is IntelliSense.

It is important to note that this is not code to run. It’s using C# to provide IntelliSense and syntax checking to metadata – including extra methods and properties that would be copied directly to the output as custom code.

In the end, I know I want to merge the notion of simplest and representational forms, so once I get my head around representational forms, the one that is easiest to merge will probably win. I’m not confident that there are scenarios that require both, it’s just I want to grabbing the part of the single that works right now, not an initial decision.

## Approach 1

A base class with properties set and methods called in a constructor. This was the first, simplest, and the one I keep returning to:

public class IfElseBraceDiagnosticCodeFirst : DiagnosticBase

{

public IfElseBraceDiagnosticCodeFirst()

{

Id = "KADGEN1001";

Description = "Needs braces";

MessageFormat = "{0} needs braces";

Category = "Style";

AddAnalyzer<IfStatementSyntax>(

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

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

messageArgs: "if statement");

AddCodeFix<IfStatementSyntax>(

makeNewNode: x => x.WithStatement(

SyntaxFactory.Block(x.Statement)));

AddAnalyzer<ElseClauseSyntax>(

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

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

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

messageArgs: "else statement");

AddCodeFix<IfStatementSyntax>(

makeNewNode: x => x.WithStatement(

SyntaxFactory.Block(x.Statement)));

}

// Methods used by the lambdas in the constructor appear here, or

// methods that can be directly used as delegates,

}

The base class for this much of the problem would have four properties and two methods, all of which are empty. The purpose is to create this class for interpretation by a Roslyn based tool. However, an interesting side effect is that there is a one to one mapping between the base class and the metadata – except for the extra methods.

## Approach 2

Peter Ritchie inspired me to think again about a fluent API. Every time I work with a Fluent API that I create I almost like it, then I just think there is too much extra. The creation of the dummy builder could happen anywhere, and IntelliSense is allowed. However, where does it make sense to put extra methods and properties for use by the lambdas?

var builder = new FluentDiagnosticBuilder()

.WithId("KADGEN1001")

.WithDescription("Needs braces")

.WithMessageFormat("{0} needs braces")

.WithCategory("Style")

.WithAnalyzer(new FluentAnalyzerBuilder<IfStatementSyntax>()

.WithCondition(x => !x.Statement.IsKind(SyntaxKind.Block))

.WithLocationGetter(x => x.IfKeyword.GetLocation())

.WithMessageArg("if statement"))

.WithCodeFix(new FluentCodeFixBuilder<IfStatementSyntax>()

.WithNewNodeMaker(x => x.WithStatement(

SyntaxFactory.Block(x.Statement))))

.WithAnalyzer(new FluentAnalyzerBuilder<ElseClauseSyntax>()

.WithCondition(x => !x.Statement.IsKind(SyntaxKind.Block))

.WithLocationGetter(x => x.ElseKeyword.GetLocation())

.WithMessageArg("else clause"))

.WithCodeFix(new FluentCodeFixBuilder<ElseClauseSyntax>()

.WithNewNodeMaker(x => x.WithStatement(

SyntaxFactory.Block(x.Statement))));

// Need a place to put methods, unless insisting that lambdas are always used, which I

// do not wish to do

## Approach 3

Since my dislike of Approach 2 seems related to the fact I’m trying to use a Fluent API in a derived class, I decided to try it in a separate file.

public class Anything

{

public void MakeDiagnostics()

{

var ifElseBraces = new DiagnosticBase();

ifElseBraces.Id = "KADGEN1001";

ifElseBraces.Description = "Needs braces";

ifElseBraces.MessageFormat = "{0} needs braces";

ifElseBraces.Category = "Style";

ifElseBraces.AddAnalyzer<IfStatementSyntax>(

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

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

messageArgs: "if statement");

ifElseBraces.AddCodeFix<IfStatementSyntax>(

makeNewNode: x => x.WithStatement(

SyntaxFactory.Block(x.Statement)));

ifElseBraces.AddAnalyzer<ElseClauseSyntax>(

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

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

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

messageArgs: "else statement");

ifElseBraces.AddCodeFix<IfStatementSyntax>(

makeNewNode: x => x.WithStatement(

SyntaxFactory.Block(x.Statement)));

}

}