The goal of RoslynDom is to present information about your code in the way **you think** about your code.

*A note on VB: I’m building out the C# version first, but I know VB very well and am designing to support later VB creation. If something is at odds with good C# support, I’ll cross that bridge when I get there.*

You can get RoslynDom on NuGet via the Package Manager in Visual Studio and [here](https://github.com/KathleenDollard/RoslynDOM) on GitHub. Keep in mind that it is an early experimental release.

RoslynDom celebrates the awesome .NET Compiler Platform, but also respects that the .NET Compiler Platform is built as a compiler, and you are not a compiler.

### Introduction

I started from the outside, highest level of code in a single file and am working inward - beginning with the structure and working inwards to statements and eventually expressions. Support for multiple files is coming – but not until I’ve completed work on statements.

By structural, I mean artifacts that organize your code - namespaces, classes, structures, etc. This post shows how to use RoslynDom to query code. Changing code is a different post. You can rather easily change the RoslynDom– outputting a new tree with your changes is currently buggy. In the meantime, most RoslynDom items expose the SyntaxNode it was created from, and where practical the corresponding ISymbol (SyntaxNode and ISymbol are part of the .NET Compiler Platform). You can use RoslynDom to get to the right location in your code, and then use .NET Compiler Platform techniques.

You can find more about the scenarios I wrote RoslynDom to support [here](http://msmvps.com/blogs/kathleen/archive/2014/06/27/roadmap-for-roslyndom-codefirst-strong-typed-metadata-and-expansionfirst-templates.aspx). If you have a tool idea and want me to make RoslynDom friendly to what you’re doing, let’s talk.

This post has walk-throughs of how you can use RoslynDom today. RoslynDom is a library to build tools from – it is not itself a tool. One tool that has been built on top of it is Jim Christopher’s [RoslynDom-Provider](https://github.com/beefarino/roslyndom-provider).

### Retrieving Namespaces

A namespace is a logical container. It’s orthogonal to the structure of your running application and tools like ObjectBrowser offer alternate physical (assembly/module) and logical (namespace) trees.

RoslynDom sets out to give access to your code the way you think about it, and you might think about it differently at different times. Both of these statements are true:

* A namespace is a dot delimited string attached to a class or other type to give it a more complete and hopefully unique name (in->out)
* A namespace is an identifier that you put at the top of a file that groups the contained code with related code in different files (in-out)

A namespace can be nested – the namespace System contains the namespace System.Diagnostics

The nesting of namespaces in code is entirely arbitrary – these code fragments are logically identical:

namespace RoslynDom

{

namespace Common.Test

{

public class Foo { }

}

}

namespace RoslynDom.Common.Test

{

public class Foo { }

}

The .NET Compiler Platform manages namespaces differently in the two trees. RoslynDom’s is committed to expressing code the way you think of it and you probably **don’t** think of your code in terms of different access mechanisms, each good for different things.

To access namespace information in RoslynDom, you first load your code. You can do this from a file, a source code string, a project document, or a SyntaxTree. For example:

IRoot root = RDomFactory.GetRootFromFile(@"..\..\TestFile.cs");

There are three properties regarding namespaces, one of which is still evolving (is the fully expanded view ever valuable to a person):

var nspaces1 = root.Namespaces;

var nspaces3 = root.NonemptyNamespaces;

These provide your namespaces as you wrote them and where the namespace is actually in use in this root. The following code would have two members in the Namespaces property and one member in the NonemptyNamespaces property:

namespace RoslynDom

{

namespace Common.Test

{

public class Foo { }

}

}

If you use the NonemptyNamespaces property, the behavior will not change if someone refactors this code to have a single or three namespace statements.

Now that you’ve seen RoslynDom in action, you may be able to predict how to retrieve using statements:

var usings = root.Usings;

Since namespaces and using directives can appear within other namespaces, both of these properties also appear on the RoslynDom INamespace interface.

### Retrieving Classes

The next step down the structural hierarchy is classes, structures and other types. These may appear at the root or in a namespace. You probably have a single namespace in your file and probably do not perceive your file as a nested structure of namespace(s) containing types. RoslynDom supports both approaches:

var nspace = root.Namespaces.First();

var classes = nspace.Classes;

var classes = root.RootClasses;

Again, I think the second will generally be more useful. Each class includes a property containing its namespace/fully qualified name.

### Retrieving Methods

RoslynDom reflects the four fundamental levels of code in .NET:

* Root attachable, which I call “stem members:” root, namespaces, using directives, classes, interfaces, structures, enums, and (in the future) delegates
* Type attachable, or type members: methods, properties, fields, (soon) enum values and (soon) events (constructors are currently a special case of a method, but waiting for a final understanding of primary constructors)
* Statements attachable to methods and property accessors
* Expressions that can be attached to statements and to fields as initializers (and now properties)

Remembering how you access namespaces, you can probably predict the code to access type members in RoslynDom:

IRoot root = RDomFactory.GetRootFromFile(@"..\..\TestFile.cs");

var class1 = root.Namespaces.Last().Classes.First();

var methods = class1.Methods;

var fields = class1.Fields;

var properties = class1.Properties;

Methods can have parameters:

var method = class1.Methods.First();

var parameters = method.Parameters;

Wow, that was easy!

### Retrieving information about items

Here’s some code

namespace Namespace2

{

public class FooClass

{

public string FooMethod(int bar1, string bar2)

{ }

public string FooProperty { get; set; }

}

}

Let’s say you want the name and type of a parameter:

var parameters = method.Parameters.ToArray();

Assert.AreEqual("bar1", parameters[0].Name);

Assert.AreEqual("Int32", parameters[0].Type.Name);

Except in the case of CLR types, you may not have access to the Reflection runtime type. To avoid taking a dependency on the .NET Compiler Platform, RoslynDom has its own type class. Alas, that’s another post.

Here is the set of features RoslynDom makes available for methods and parameters:

var method = class1.Methods.First();

var parameters = method.Parameters.ToArray();

Assert.AreEqual(2, parameters.Count());

Assert.AreEqual("FooMethod", method.Name);

Assert.AreEqual("String", method.ReturnType.Name);

Assert.AreEqual(AccessModifier.Public, method.AccessModifier );

Assert.IsFalse(method.IsAbstract);

Assert.IsFalse(method.IsExtensionMethod);

Assert.IsFalse(method.IsOverride);

Assert.IsFalse(method.IsSealed);

Assert.IsFalse(method.IsStatic);

Assert.IsFalse(method.IsVirtual);

Assert.IsFalse(method.IsVirtual);

Assert.AreEqual("bar1", parameters[0].Name);

Assert.AreEqual("Int32", parameters[0].Type.Name);

Assert.AreEqual(0, parameters[0].Ordinal);

Assert.AreEqual("bar2", parameters[1].Name);

Assert.AreEqual("String", parameters[1].Type.Name);

Assert.AreEqual(1, parameters[1].Ordinal);

Assert.IsFalse(parameters[1].IsOptional);

Assert.IsFalse(parameters[1].IsOut);

Assert.IsFalse(parameters[1].IsParamArray);

### Attributes

Attributes are another area where the .NET Compiler Platform syntax tree keeps track of arbitrary differences in how code is written – differences that you don’t think about when reading code. These two fragments of code have the same intent.

[SomeAttr, SomeAttr2]

struct Foo<T>

{ }

[SomeAttr]

[SomeAttr2]

struct Foo<T>

{ }

And that’s without even considering the optional parentheses on the attributes.

RoslynDom collapses these differences and has an Attributes property on every item that allows it in .NET:

var attributes = class1.Attributes;

Attributes have names. One way to use them is with LINQ expressions. This retrieves any attributes that is on a class and has a particular name:

var classAttributes = from x in root.RootClasses

from a in x.Attributes

where(a => a.Name == name)

select x;

A similar LINQ expression could return the matching, or non-matching classes.

Attributes may have values, which would be the parameters to the attributes. Since RoslynDom does not yet support multiple files, the attributes aren’t fully resolved and positional arguments are currently problematic.

If you have code like these attributes (used in a rather silly way):

[ExcludeFromCodeCoverage]

[EventSource(Name ="George")]

public class FooClass

{}

You can retrieve named arguments like this:

var class1 = root.Namespaces.Last().Classes.First();

var attributes = class1.Attributes.ToArray();

Assert.AreEqual(2, attributes.Count());

Assert.AreEqual("ExcludeFromCodeCoverage", attributes[0].Name);

Assert.AreEqual("EventSource", attributes[1].Name);

Assert.AreEqual("Name", attributes[1].AttributeValues.First().Name);

Assert.AreEqual("George", attributes[1].AttributeValues.First().Value);

Assert.AreEqual(LiteralKind.String, attributes[1].AttributeValues.First().ValueType);

### Summary

RoslynDom is in a preliminary stage, and I’d be happy to hear your thoughts. The goal of RoslynDom is to enhance the .NET Compiler Platform to make humans like you and me happy accessing the fantastic information the compiler is exposing!