RoslynDom: Structural Interrogation Walk-throughs

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.*

As a structural matter, I started from the outside and am working in. I started with the structure and am working in towards statements and perhaps eventually expressions. I’m clear on the value of the structural and statement layers, but the .NET Compiler Platform-Roslyn, has really nice support for spinning back and forth between the deeper complexity of expressions and you can do tasks like accessing identifiers without full creating an alternate DOM for expressions. Also, I started working with single files – multiple files are coming, but not until I’ve completed work on statements.

By structural, I mean artifacts that organize your code, not the code itself – namespaces, classes, structures, etc. This post shows ways to use that information to query and change code. This is a big ongoing project and at present it doesn’t make sense to change many aspects of code without the capacity to push those changes through the rest of your code. For this reason, RoslynDom exposes the attached syntax nodes and symbols (syntactic and semantic tree information).

I’m interested in your scenarios for using Roslyn information. Here’s mine… I want strongly typed metadata that’s created from code, which I get through the CodeFirstMetadata project. I’m using this for code generation, but my love hate relationship with T4 leads me to create a new style of template, which I call ExpansionFirstTemplates. The ideas behind these projects are independent – you could load strongly typed metadata anyway you want. But the current implementations are linked – the implementation of CodeFirstMetadata relies on RoslynDom and the implementation of ExpansionFirstTemplates relies on RoslynDom and CodeFirstMetadat.

This post has walk-throughs of how you can use RoslynDom today. It’s a library to build tools from. It is not itself a tool. One tool that has been built on top of it Jim Christopher’s RoslynDom-Provider. If you have a tool idea and want me to make RoslynDom friendly to what you’re doing, let’s talk.

# 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 the code you way you think about it, and sometimes you think about it differently at different times. All of these are true:

* A namespace is a dot delimited string attached to my class to give it a more complete and hopefully unique name (in->out)
* A namespace is a n identifier that I put at the top of a file that indicates the group code in this file belongs to (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 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. The syntax tree is committed to round-tripping your code, so reflects whatever you typed – these two fragments are different in the syntax tree. The semantic tree removes arbitrary artifacts to create a single semantic representation of your code. In the semantic tree, both versions are held in a third way:

namespace RoslynDom

{

namespace Common

{

namespace Test

{

public class Foo { }

}

}

}

The easiest way to traverse the namespaces in your project using the .NET Compiler Platform is probably by traversing the symbolic tree, such as in FAQ 9 in the Samples solution available as part of the .NET Compiler Platform API.

Part of RoslynDom’s commitment to expressing code the way you think of it is that you **don’t** think of your code in terms of three different access mechanisms with some things simpler in one approach than another – the namespace retrieval question is tedious or gnarly in the syntax tree for non-trivial scenarios.

To access information in RoslynDom, you first load your code, which you can do from a file, a source code string, or a SyntaxTree. You will be able to load your project when multi-file support is available.

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

Along with similar GetRootFromString, GetRootFromDocument and GetRootFromSyntaxTree methods.

The root has three properties regarding namespaces:

var nspaces1 = root.Namespaces;

var nspaces2 = root.AllChildNamespaces;

var nspaces3 = root.NonemptyNamespaces;

These provide your namespaces as you wrote them, fully expanded, and only where you actually used them. This could would have two members in the Namespaces property:

namespace RoslynDom

{

namespace Common.Test

{

public class Foo { }

}

}

It would have three members in the AllChildNamespaces property and one member in the NonemptyNamespaces property. I believe you will find the last the most useful.

Now that you know how to use RoslynDom, it’s pretty easy to predict how to retrieve using statements:

var usings = root.Usings;

# Retrieving Classes

The next step in the structural tree 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 contains a property containing it’s namespace/fully qualified name as expected.

# 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, which 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)

The cliff between types and statements is a doozy, as is the one between statements and expressions. DOM’s like the CodeDom are incomprehensibly difficult, and the .NET Compiler Platform has to achieve approximately the basic level of complexity internally to be a compiler. The .NET Compiler Platform helps you manage this complexity via a host of helper methods, including the SyntaxWalker and Rewriter classes.

RoslynDom doesn’t need to reflect the same level of complexity because humans, not compiler processes, and not round-tripping are the goals. RoslynDom manages change by direct alteration followed by outputting an updated tree when you’re ready to do something in .NET Compiler Platform World or Visual Studio Extensibility World.

Other than that background, 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 have parameters:

var method = class1.Methods.First();

var parameters = method.Parameters;

Wow, that was easy!

# Retrieving information about items

Each item in the RoslynDom is strongly typed. There is a set of interfaces because the majority of what code does is similar across a wide range of languages that include C#, Visual Basic, Java, and ECMA 6 JavaScript. At the very least, this allows better evaluation of differences. It would also be awesome to be able to have fragments of code written once and used across languages. Entire probably not. But fragments, maybe.

The hierarchy of these interfaces is a bit scary and you might want to skip that until I have a chance to write more about that interface style. Just flatten them, which is easiest to do in IntelliSense.

The interfaces are read-only. That’s intentional. You’ll cast for mutating items, but that’s a different post.

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:

[[ Show just parameter type and name ]]

[[ Say some more stuff ]]

[[ Show full list of available items on these ]]

[[ Final section on attributes ]]

[[ Review Public Annotations post ]]