RoslynDom Quick Start

This document is about an early version of RoslynDom, focusing mostly on working features, with notes on the impact of missing certain upcoming features. You can also see notes on missing features in GitHub issues.

For more information see these documents in the “Documents” folder on GitHub:

* See the RoslynDom Project Layout if you are curious about why there are five projects and the dependencies these projects have on the .NET Compiler Framework (Microsoft.CodeAnalysis), CSharp compiler (Microsoft.CodeAnalysis.CSharp) and Unity (Microsoft.Practices.Unity.\*)
* See the RoslynDom Design Overview for a discussion of how RoslynDom is built
* See the RoslynDom Extensibility if you’re interested in doing more with RoslynDom
* See the RoslynDom Roadmap.ppt for a vision of RoslynDom

# What is RoslynDom

RoslynDom is an alternative view of your code.

The most efficient, best way to express your code in ASCII is your code in your favorite language.

The most efficient, best way to organize your code for your compiler is within the compiler, and exposed as part of the .NET Compiler Platform, Roslyn.

Another, ephemeral expression of your code is the one in your head. This is the one that comes out in words in your meetings, and you have entire meetings without phrases like “angle bracket.”

RoslynDom models this third expression of code in memory which has several features:

* You can load existing code into the RoslynDom model and easily explore, navigate and analyze it. The RoslynDom model is language agnostic.
  + *This feature is currently affected by not yet having multi-file support*
* RoslynDom is mutable. You can alter your code in a load, alter, alter, alter, build output model. Since you can easily navigate your code, finding the location for change is easy
  + *This feature is currently affected by whitespace formatting and a lack of comment support*
  + *This feature is currently affected by not yet having semantic search and replace implemented*
* RoslynDom entirely isolates the language dependent load process from the model itself. At a simplistic level, when the VB factory is in place, you can load from C# and output to VB and vice versa.
* RoslynDom models can be created in any manner you desire. RoslynDom views can be created without loading code, and then brand new code created.

## The basic model

Code exists in the following hierarchy

* Root groups which are groups of files *(not yet implemented)*
* Roots, which are files or a single conceptual load unit
* Stem members – Namespaces and types that can contain be contained in roots
* Type members – nested types, methods, properties, etc. that can be contained in types
* Statements – code statements that are held primarily in methods and property accessors
* Expressions – sub parts of statements that return values

*Most major features, including most statements are complete, see GitHub issues.*

Expressions are currently handled via strings by design.

# Walkthrough 1: Load and check code

## Step 1: Load your code

Add a using statement for RoslynDom.CSharp.

Retrieve the singleton instance of the RDomCSharpFactory from the RDomCSharpFactory.Factory property and call the GetRootFromFile method to open a specific file:

var factory = RDomCSharpFactory.Factory;

var root = factory.GetRootFromFile(fileName);

NOTE: Other overloads support loading source code from strings or trees.

NOTE: You can iteratively work through the files in your project or solution. *This approach will be hampered because specifying references and multiple syntax trees for the underlying model isn’t yet supported.*

## Step 2: Check your code

Output your code to a string to test the output. You can do this by outputting to a new file and comparing the files:

var output = factory.BuildSyntax(root).ToString();

File.WriteAllText(outputFileName, output);

NOTE: *Whitespace will be ditzed due to a known problem.*

NOTE: *Type aliases are not preserved, known problem*

NOTE: *Features that aren’t yet implemented are listed in the GitHub issues. If you find more, tell me*

## Conclusion

You now know how to load and output code from RoslynDom

# Walkthrough 2: Navigate and interrogate code

## Step 1: Load and check code

Load and check your code as shown in Walkthrough 1.

## Step 2: Ask general questions about code

LINQ is your friend.

You’ll often find it convenient to make an array for easier sequential requests in testing.

Assigning intermediate values to variables in tests can help clarity

var factory = RDomCSharpFactory.Factory.GetRootFromFile(fileName);

Assert.AreEqual(1, root.Usings.Count());

Assert.AreEqual("System", root.Usings.First().Name);

Assert.AreEqual(1, root.Namespaces.Count());

Assert.AreEqual(1, root.RootClasses.Count());

var methods = root.RootClasses.First().Methods.ToArray();

Assert.AreEqual(0, methods[0].Parameters.Count());

Assert.AreEqual(1, methods[1].Parameters.Count());

Assert.AreEqual("x", methods[1].Parameters.First().Name);

## Step 3: Place a break point and query code

Place a breakpoint, run the test in debug mode and ask questions in the immediate window about the code. Sometimes you’ll have to use the Watch window because of the .NET Compiler Platform CTP behavior. Have fun!

## Step 4: Ask harder questions

Ensure RolsynDom.Common is included in the using statements.

Let’s say you’re concerned about unsigned ints variables in your code and want to examine their names. I don’t know why, I just had to make something up.

You can retrieve the RoslynDom entry with

var uintVars = root

.Descendants.OfType<IVariable>()

.Where(x => x.Type.Name.StartsWith("UInt"))

.Select(x => x.Name);

NOTE: Aliases are language specific, RoslynDom entries are agnostic so use the .NET name of the type. *Outputting non-aliased names is an open issue.*

As another example, say you want all the methods and variables where unsigned ints are used:

var uintCode = (from c in root.Descendants.OfType<IStatementContainer>()

from v in cl.Descendants.OfType<IVariable>()

where v.Type.Name.StartsWith("UInt")

select new

{

containerName = cl.Name,

variableName = v.Name

})

.ToArray();

# Walkthrough 3: Finding questionable implicit variable typing

I have a sin in code. I really like ignoring types when I am creating code and using var everywhere. This saves me time when I am creating code. However, it can result in code that’s less readable. I can accept a rule that implicit variable typing should only be used on object instantiation, strings, Int32 (int), and DateTime in VB.

This combination of selecting types based on the implemented interfaces, and examining additional properties, like types and names is very powerful in finding particular locations in code. What if you want to find all the implicitly typed variables that are not an object instantiation, an assignment of a string, or an assignment of an integer?

Since this is a complicated question, I’ll ask in steps, although you can certainly refactor this into a single statement if you prefer.

var implicitlyTyped = root

.Descendants.OfType<IDeclarationStatement>()

.Where(x => x.IsImplicitlyTyped);

var instantiations = implicitlyTyped

.Where(x => x.Initializer.ExpressionType == ExpressionType.ObjectCreation);

var literals = implicitlyTyped

.Where(x => x.Initializer.ExpressionType == ExpressionType.Literal &&

( x.Type.Name == "String"

|| x.Type.Name == "Int"

|| x.Type.Name == "DateTime" )// for VB

);

var candidates =implicitlyTyped.Except(instantiations).Except(literals);

## Step 6: Reporting

Once you get the information you’re interested in, you’ll probably want to output the information. Reporting information about RoslynDom requires that it is in its original, unchanged format because it relies on the original tree, rather than the proposed mutated tree. As long as you haven’t changed anything, it’s perfectly safe to access the underlying tree, but RoslynDom exposes this only as a raw object. You can use helper functions like these to access the information:

private string GetNewCode(IDom item)

{

var ret = new List<string>();

return RDomCSharp.Factory.BuildSyntax(item).ToString();

}

private string GetOldCode(IDom item)

{

var node = item.RawItem as SyntaxNode;

if (node == null)

{ return "<no syntax node>"; }

else

{

return node.ToFullString();

}

}

private LinePosition GetPosition(IDom item)

{

var node = item.RawItem as SyntaxNode;

if (node == null)

{ return default(LinePosition); }

else

{

var location = node.GetLocation();

var linePos = location.GetLineSpan().StartLinePosition;

return linePos;

}

}

private string GetFileName(IDom item)

{

var root = item.Ancestors.OfType<IRoot>().FirstOrDefault();

if (root != null)

{ return root.FilePath; }

else

{

var top = item.Ancestors.Last();

var node = top as SyntaxNode;

if (node == null)

{ return "<no file name>"; }

else

{ return node.SyntaxTree.FilePath; }

}

}

You can use these helper methods in LINQ:

var lineItems = from x in items

select new

{

fileName = GetFileName(x),

position = GetPosition(x),

code = GetNewCode(x)

};

And, you can iterate across the resulting enumerable of anonymous type like this:

var max = lineItems.Max(x => x.fileName.Length);

var format = "{0, -fMax} ({1,4},{2,4}) {3}".Replace("fMax", max.ToString());

foreach (var line in lineItems)

{

sb.AppendFormat(format, line.fileName, line.position.Line, line.position.Character, line.code);

sb.AppendLine();

}

return sb.ToString();

Resulting in the very pretty output

Walkthrough\_1\_code.cs(13, 16) RoslynDom.RDomDeclarationStatement : ret {String} var ret = lastName;

Walkthrough\_1\_code.cs(51, 16) RoslynDom.RDomDeclarationStatement : x3 {Int32} var x3 = x2;