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.

You can find the code for these quickstarts in the RoslynDomExampleTests NuGet package.

For more information see these documents in the “Documents” folder on GitHub (*Creation of these documents is currently in progress)*:

* 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 = RDomCSharp.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.*

Of course you can assign the factory property to a local variable or class field if you prefer.

RDomCSharp is the code that creates the language agnostic RoslynDom tree from C# code, and that can recreate C# code from the RoslynDom tree. You can create a RoslynDom tree from scratch as well. You will later be able to load from other languages, in particular VB.NET.

## 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);

## Conclusion

You now know how to load and output code from RoslynDom

# Walkthrough 2: Navigate and interrogate code

One of the major user scenarios intended for RoslynDom is to allow you to answer questions about your code. This is just a small sampling of the kinds of things you can do.

At present, RoslynDom supports structural features (classes, methods, etc) and statements. It does not support expressions because user stories with value aren’t yet clear.

## 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.

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());

Assigning intermediate values to variables in tests can help clarity

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);

The difference between Classes and RootClasses is that root classes include all classes under the root, regardless of namespace. Classes are only those directly under the root. Similar for Interfaces, Enums and Structures.

## 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

That might have been fun, but the real value from RoslynDom comes from asking complex questions. I’ll introduce LINQ in this walkthrough, and then show something I really wanted to accomplish in the next.

Ensure RolsynDom.Common and System.Linq are 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. The CSharp factory is responsible for straightening this out on output.

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 when I code. I really like ignoring types. When I write code I use var everywhere. This saves me time. But, I realize 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. *VB isn’t yet supported.*

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. I want to find all the implicitly typed variables that are not an object instantiation, assignments to literal strings, or assignments to integers?

Since this is a complicated question, I’ll ask in steps, although you can certainly refactor this into a single statement if you prefer. LINQ doesn’t evaluate until requested, so the piecewise creation is not a performance issue:

Find all implicitly typed local variables:

var implicitlyTyped = root

.Descendants.OfType<IDeclarationStatement>()

.Where(x => x.IsImplicitlyTyped);

Find all instantiations, because they’re OK:

var instantiations = implicitlyTyped

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

ExpressionType.ObjectCreation);

Find all string, integer (32 bit) and DateTime literals, because they’re OK:

var literals = implicitlyTyped

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

( x.Type.Name == "String"

|| x.Type.Name == "Int32"

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

);

Find all the implicitly types variables that aren’t instantiations or literals string/ints:

var candidates =implicitlyTyped

.Except(instantiations)

.Except(literals);

## Step 6: Reporting

The code discussed here is in the ReportCodeLines method.

Once you get the information you’re interested in, you’ll probably want to output it. Obviously in reporting, you’d like file, line and column positions. RoslynDom is an abstract tree that does not directly understand text. But it maintains, and can report, about the code it was created from by holding references to key aspects of the underlying .NET Compiler Platform (Roslyn) objects. As long as you haven’t changed the tree, these aspects remain correct.

If you change the tree, the only safe way to report positions of RoslynDom elements is to recreate the underlying syntax tree, and then reload that tree into RoslynDom – generally also searching again for the elements of interest.

Because we haven’t changed the tree since loading it, this isn’t a problem.

Create a SyntaxTree from part of the RoslynDom item:

private string GetNewCode(IDom item)

{

var ret = new List<string>();

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

}

Retrieve the original code that was used to create the RoslynDom element:

private string GetOldCode(IDom item)

{

var node = item.RawItem as SyntaxNode;

if (node == null)

{ return "<no syntax node>"; }

else

{

return node.ToFullString();

}

}

Retrieve the original code position:

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;

}

}

Retrieve the original code filename:

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 to create an IEnumerable of an anonymous type:

var lineItems = from x in items

select new

{

item = x,

fileName = GetFileName(x),

position = GetPosition(x),

code = GetNewCode(x)

};

I’ll use a string formatting trick to make pretty columnar output. I’ll first determine the length of each segment of the string output – such as the maximum file path length. I’ll replace dummy values in a format string, such as fMax, to create a custom format string for the sizes in this data:

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

var itemMax = lineItems.Max(

x => x.item.ToString().Trim().Length);

var lineMax = lineItems.Max(

x => x.position.Line.ToString().Trim().Length);

var format = "{0, -fMax}({1,lineMax},{2,3}) {3, -itemMax} {4}"

.Replace("fMax", filePathMax.ToString())

.Replace("itemMax", itemMax.ToString())

.Replace("lineMax", lineMax.ToString());

I can then iterate across the IEnumerable of anonymous type:

foreach (var line in lineItems)

{

sb.AppendFormat(format, line.fileName,

line.position.Line, line.position.Character,

line.item.ToString().Trim(), line.code);

sb.AppendLine();

}

return sb.ToString();

This results in nice output like (which would be nicer if Word didn’t wrap it):

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;

# Walkthrough 4: Fixing questionable implicit variable typing

What good would it be to find issues if you couldn’t fix them. But I’m tired, so I’m going to mostly let you figure out how this code works based on what you’ve already learned

[TestMethod]

public void Walkthrogh\_4\_Fix\_implicit\_variables\_of\_concern()

{

// Assumes Walkthrough\_3 passes

var root = RDomCSharp.Factory.GetRootFromFile(fileName);

var candidates = FindImplicitVariablesOfConcern(root);

foreach (var candidate in candidates)

{

candidate.IsImplicitlyTyped = false;

}

var output = RDomCSharp.Factory.BuildSyntax(root.RootClasses.First());

// For testing, force chhanges through secondary mechanism

var initialCode = File.ReadAllText(fileName);

var newCode = initialCode

.Replace("var ret = lastName;", "System.String ret = lastName;")

.Replace("var x3 = x2;", "System.Int32 x3 = x2;")

.SubstringAfter("Walkthrough\_1\_code\r\n{\r\n")

.SubstringBeforeLast("}")

;

Assert.AreEqual(newCode, output.ToFullString());

}

The only thing that’s required is to state that these declarations should not be implicitly typed by setting IsImpliciltyTyped to false for each candidate. The rest of the code is to create a test.

But this results in the rather ugly System.String declaration. That’s jarring in a file that uses the C# aliases. That fix is in the next test:

[TestMethod]

public void Walkthrogh\_4\_Fix\_non\_aliased()

{

// Assumes Walkthrough\_3 passes

var root = RDomCSharp.Factory.GetRootFromFile(fileName);

var candidates = FindImplicitVariablesOfConcern(root);

foreach (var candidate in candidates)

{

candidate.IsImplicitlyTyped = false;

candidate.Type.DisplayAlias = true;

}

var output = RDomCSharp.Factory.BuildSyntax(root.RootClasses.First());

// For testing, force chhanges through secondary mechanism

var initialCode = File.ReadAllText(fileName);

var newCode = initialCode

.Replace("var ret = lastName;", "string ret = lastName;")

.Replace("var x3 = x2;", "int x3 = x2;")

.SubstringAfter("Walkthrough\_1\_code\r\n{\r\n")

.SubstringBeforeLast("}")

;

Assert.AreEqual(newCode, output.ToFullString());

}

Here, in addition to setting IsImplicitlyTyped to false, I set DisplayAlias to true. Normally, this is set when the code is loaded based on whether the alias is used. Since you’re changing how the type is managed, you also have to request that you want to use the alias.