Why RoslynDom? Part 1

The .NET Compiler Platform, Roslyn, is a great piece of work. It’s a great compiler and it opens up the next generation of strongly typed compilers that take direct responsibility for supporting modern development. But you probably want a wrapper around it when using it for other purposes.

At least I did. I found it very, very challenging to do what I wanted to do with Roslyn. I was unable to build tools in a way that I felt was approachable for a multi-contributor open source project. Creating the Roslyn wrapper as an independent OSS library – useful to you and separate from my OSS tools – drove a significant part of the RoslynDom design.

RoslynDom is an alternate tree that wraps the .NET Compiler Platform syntax and semantic trees. RoslynDom is a tree designed for human coders. Part 2 of this post covers what RoslynDom does and how it enhances the .NET Compiler Platform. This post covers how and why RoslynDom differs dramatically from the .NET Compiler Platform.

There are a number of decisions that help make the .NET Compiler Platform a great compiler make it very challenging to work with for other purposes:

* The .NET Compiler Platform syntax tree is file based, while its semantic tree is (naturally) project or assembly based
* The .NET Compiler Platform class design is very shallow with little polymorphism and generally sealed
* The .NET Compiler Platform fully round trips source code – you can get back out exactly and completely the source code string you put in
* The .NET Compiler Platform describes code in exquisite detail
* The .NET Compiler Platform has an obsession with performance
* The .NET Compiler Platform syntax trees are language specific
* Almost everything in the .NET Compiler Platform is immutable
* The .NET Compiler Platform is not a compiler pipeline and deliberately postpones decisions like source code compiler annotations

# RoslynDom offers a unified tree

*The .NET Compiler Platform syntax tree is file based, while its semantic tree is (naturally) project or assembly based*

RoslynDom offers a unified vision of your code, without regard for where information is available. Things like naming is the best available from the current information.

RoslynDom does not attempt to solve all problems so maintains access to the underlying syntax and semantic trees in unmodified RoslynDom trees. Thus, even if the task you are performing requires the .NET Compiler Platform SyntaxNode or ISymbol, you might get it quicker through RoslynDom.

# Shallow hierarchy, sealed classes, and limited polymorphism

*The .NET Compiler Platform class design is very shallow with little polymorphism and generally sealed*

All RoslyDom classes support the IDom interface and most support the IDom<T> interface. All current implementations derive as low as possible in the RDomBase, RDomBase<T> (where T is the specific IDom derived interface)and RDomBase<T, TSyntax, TSymbol> hierarchy. This increases consistency and polymorphism.

To a significant degree, this is simply an arbitrary, opinionated decision. There are good arguments against deep trees and base class coupling for general application development. However, it seems highly unlikely that the things RoslynDom expresses – types, methods, attributes, etc. – will change very much. The tradeoff in simplicity of being able to interchange items is very effective for the kinds of problems where I anticipate people using the RoslynDom.

The interface hierarchy is currently complex, experimental and intended for experimental support. It is entirely free of .NET Compiler Platform artifacts. If you programmed only against these interfaces, you could create techniques that could work against many types of code bases, regardless of platform or language.

The interface hierarchy is complex because it expresses the commonalities between classes. You might want to explore this via the diagram in the source code – but the multi-layer design is likely to evolve. Try to write source code against IDom, IDom<T> and the leaves like IClass or IProperty.

# RoslynDom abandons full round-tripping

*The .NET Compiler Platform fully round trips source code – you can get back out exactly and completely the source code string you put in*

Support for full round-tripping means that logically equivalent code results in a different syntax tree. This occurs in multiple places, but two are currently significant in the RoslynDom design.

## Namespace nesting

These two source code fragments are semantically (logically) identical:

[[ namespace nested and non-nested sample]]

In this case, RoslynDom considers the difference significant, complex and potentially useful. To avoid losing this information, the RoslynDom tree maintains the namespace nesting of the input trees. The more likely approach of accessing the tree without regard for nesting is supported with an alternate set of properties on roots and namespaces:

* NonemptyNamespaces – the set of namespaces that actually contain types named via their compound name (the one you’d use in a using statement for the types)
* RootClasses – the set of classes in the root or namespace, regardless of namespace organization
* RootStructures – the set of classes in the root or namespace, regardless of namespace organization
* RootInterfaces – the set of classes in the root or namespace, regardless of namespace organization
* RootEnums – the set of classes in the root or namespace, regardless of namespace organization

I anticipate that these properties will be used more frequently than the properties that maintain the arbitrary namespace nesting and requires navigation.

## Attributes

These two source code fragments are semantically (logically) identical:

[[ two attribute approaches ]]

In this case, RoslynDom considers the difference insignificant. Except for recreating source code, there is no value in these different approaches. Therefore, RoslynDom collapses them and is opinionated in how attributes are expressed when source code is created. That will initially be attributes in separate brackets on different lines, but alternatives may later be provided in configuration for the creation outputter.

## SameIntent methods

RoslynDom explicitly embraces the concept of equivalent or semantically identical code. All classes have a SameIntent method that allows comparison of two items. The initial implementation handles common differences and it will evolve to handle more detail, possibly different levels of detail, and possibly identifier independent comparisons.

Initially false positives are possible. Please let me know which ones affect you as this is anticipated line of development.

# Exquisite detail

*The .NET Compiler Platform describes code in exquisite detail*

The .NET Compiler Platform is so detailed that semi-colons and brackets are first class citizens. These details, and whitespace, just don’t matter to most usage and add complexity.

RoslynDom’s goal is to provide adequate detail. While RoslynDom itself does not support exquisite detail, it generally provides access to the underlying SyntaxNode or SyntaxToken so you can access the detail important to you.

# High performance

*The .NET Compiler Platform has an obsession with performance*

What’s not to love about high performance code? The fact the .NET Compiler Platform, as a managed compiler, achieves performance parity with the older unmanaged compilers is one of its great accomplishments.

RoslynDom does not take this approach. I can imagine some scenarios where performance will be very important, but the current goal is maintainability and making it practical to have multiple contributors to the open source project. Performance issues aren’t expected, and will be addressed as they arise.

# Language specific

*The .NET Compiler Platform syntax trees are language specific*

The current implementation of RoslynDom is C# specific. That’s where I started. But the hope is to extend the RoslynDom implementation to Visual Basic and the interfaces potentially beyond Visual Basic and C#.

I’m particularly interested in being able to express method level code in JavaScript. This would make it easier to use common business logic. I’m also interested in F#, although the interface design may require significant change.

# Mutability

*Almost everything in the .NET Compiler Platform is immutable*

Immutability (the complete inability to make changes except by creating a new instance) has huge advantages, particularly in an async and multi-threaded world. It is also challenging for many tasks, even with the many helper methods that the .NET Compiler Platform supplies.

Mutability is not yet implemented in RoslynDom. The main scenario is loading a RoslynDom, making changes, then comparing or outputting the changes as a new .NET Compiler Platform SyntaxNode. The SyntaxNode can easily create source code.

# Pipeline and compiler annotations

*The .NET Compiler Platform is not a compiler pipeline and deliberately postpones decisions like source code compiler annotations*

The work I am doing with metaprogramming requires a pipeline and annotations. I use MSBuild for the pipeline and will write more about that in the future. I have written about public annotations [[here]]

# Summary

I love the .NET Compiler Platform. I needed to extend it to do the metaprogramming work I wanted to do. I felt these enhancements would be useful for people doing other things with the .NET Compiler Platform, so created my wrappers as an independent open source library you can find on GitHub and Nuget