RoslynDom Design

# Overview

There are six non-exclusive “levels” of items

* IRoot
* IStemMember
* ITypeMember
* IStatement
* IExpression
* IMisc

Four items can appear in two levels: class, interface, enum and structure.

There are five non-test assemblies

* RoslynDomUtillities (which should be renamed or combined with Common) which has general utilities and has no dependencies on the .NET Compiler Platform
* RoslynDomCommon (interfaces and enums) no dependencies on the .NET Compiler Platform
* RoslynDomRDomCommon dependency on language agnostic parts of .NET Compiler Platform and ONLY assembly with a reference to Unity
* RoslynDom language indepdent implementations of interfaces. Dependent on language agnostic parts of .NET Compiler Platform
* RoslynDomCSharpFactories Dependent on everything else, only route for loading or retrieving trees, and the ONLY assembly with a .NET Compiler Platform CSharp dependency (expect a VB version )

There are three entities (files, generally classes) for each conceptual units (like class or if statement)

* Interfaces – IDom… platform independent with caveats and often touched by end user
* Implementations – RDom… language independent and often touched by end-user
* Factories – RDom….Factory language dependent loader and outputter

There are some classes that overarch all entities

* Provider – IoC wrapper
* Corporation – RDomCorporation – single sealed class (multiple instantiations) that manages factories
* Workers
  + RDomWorker – language independent sealed class that delegates work to individual factories to simplify access and understands the work that workers can do and exposes the other two workers for extensibility
  + CreateFromWorker - language dependent helper class that allows code reuse for creating RoslynDom entities – overridden for some extensibility scenarios
  + BuildSyntaxWorker – language dependent helper class that allows code reuse for rebuilding syntax trees – overridden for some extensibility scenarios

# Loading multiple factories

Loading multiple factories has not been tested, and the design for retrieval is still under consideration. Options under consideration:

* Empty interface for each language – ICSharp, IVisualBasic, IVisualBasic6, etc.
  + Benefits: simplicity, extensibility and no need to instantiate unneeded objects
  + Drawbacks: Establishes universe of possibility without RoslynDom change, and may give implication that things are implemented that aren’t
* Property on each entity holding an integer, which happens to be an integer
  + Benefits: Does not establish the universe up front because new integers could be used (same as Keyword design in .NET EventSource)
  + Drawbacks: Requires instantiation of all entities to query for applicability. IOW, it happens after ResolveAll, not when filing IoC.

# Extensibility

## Change the way a common feature is loaded or unloaded

The plan is to derive from CreateFromWorker or BuildSyntaxWorker and ensure your assembly is in the executable directory. Derived classes are assumed to take precedence over default classes.

This is not yet implemented because of security concerns.

## Change the way an entity is loaded or unloaded

The plan is to derive from the specific factory and ensure your assembly is in the executable directory. Derived classes are assumed to take precedence over default classes.

This is not yet implemented because of security concerns.

## Add new features to an entity

If you’re willing to always access as the entity, not the interface, change the factory to create your alternate entity.

## Add or change features of an entity interface

Let’s talk about what I screwed up. This will particularly be an issue when I or someone else creates the Visual Basic factories.

## Add alternate platforms

In order to move forward, I tied the load/unload semantics very tightly to Roslyn. There’s no reason we can’t create a completely separate of load/unload that will use some entirely different mechanism.

The important scenarios I see for this are VB6, Java and JavaScript. If you’re interested in this, I’d really like to work with you. I don’t know what potholes there are, and I have interest in tweaking things for any of these scenarios that can work against the RDom implementations. JavaScript may require TypeScript/ECMA6.

If we can’t build these alternate platforms against the RDom implementations, I’d like to understand where the interfaces fall short. I believe this is interesting research.

Some languages will probably not be translatable into any of RoslynDom. I anticipate this will include TSQL and strict languages like Haskell. I don’t know about F# and would love to have the conversation – I suspect it may be partially doable, but have no value.